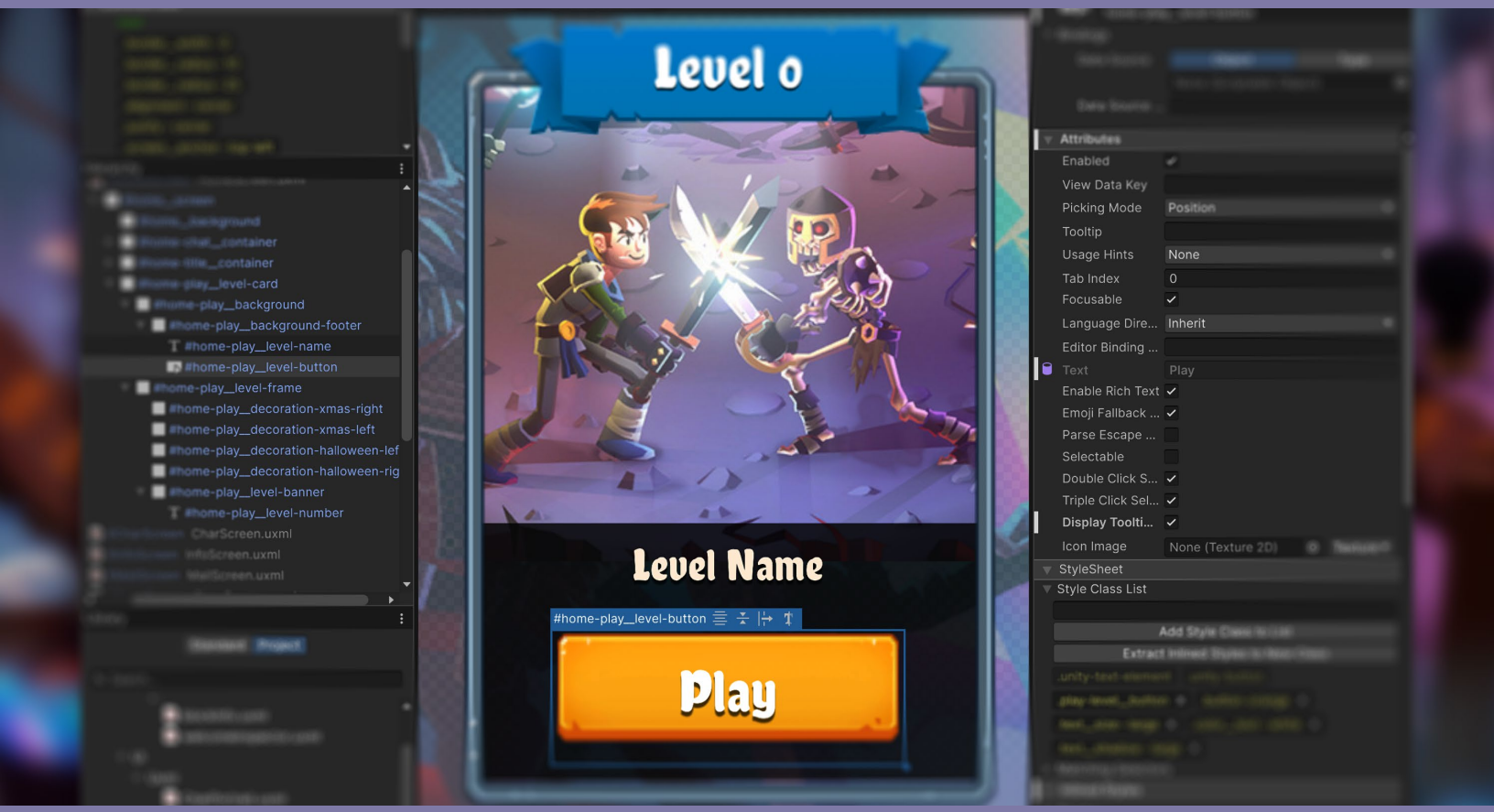
→ EBOOK

Unity 上級開発者向け UI Toolkit

(Unity 6 エディション)



Unity®

Contents

はじめに

ユーザーが意識することがない。それが最高のユーザーインターフェースです。

ユーザーインターフェース (UI) は、あらゆるゲームにおいて極めて重要な要素です。うまく作られている場合は、目立たず、アプリケーションへ丹念に織り込まれたものとなっています。しかし、うまく作られていない場合は、ユーザーを苛立たせ、ゲームプレイの体験を損なうものとなります。

優れた UI は、ゲームのビジュアルアイデンティティの延長線上にあるものです。現代のオーディエンスは、アプリケーションと一体化している直感的な UI を求めています。表示するのがキャラクターのバイタル統計データであれ、ゲーム世界での資金の状態であれ、インターフェースはプレイヤーを重要な情報へと導く入口となります。

UI が洗練されるにつれて、その根本の芸術性の追求についても磨かれていくものなのです。UI デザインを担うのは、主に 2 つのタイプのスペシャリストです。

UI アーティスト:デザイン、色、形状、タイポグラフィ、レイアウトの基本を熟知している存在です。UI アーティストは、ゲーム世界のターゲットオーディエンスに向けてデザインを行います。彼らのディテールへのこだわりは、"ピクセルパーフェクトな" UI を生み出す原動力となります。

UX デザイナー:ユーザーの動作とエンドユーザーの幅広いニーズを調査します。UX (ユーザー体験) デザイナーは、人とデジタル製品とが、いかなるインタラクションを行うのか、というところを取り扱います。その体験をできるだけ直感的で楽しいものにすることを目的として、ナビゲーションフローを構築します。

こういった役回りにおいては、他の 2D/3D アーティストやデザイナーも交えて密接に連携しています。この共同作業を通じて、さらに力のある、効果的な UI が生まれるのです。

もう 1 つの重要な役割として **UI プログラマー** があり、前述の 2 つの役割と緊密に連携します。UI プログラマーは、選ばれた技術スタックを使用して、UI デザインを機能的なインターフェースに取り込むためのプロセスやパイプラインを確立し、ゲームプレイコードを UI につなぎ、UI からゲームシステムにデータをフィードバックします。

前回の eBook Unity におけるユーザーインターフェースのデザインと実装 では、UI アーティストとデザイナーが 2 つの UI システムを使用して Unity でインターフェースを構築する方法を紹介しました。ゲームオブジェクトベースの古いシステムである Unity UI と、新しい UI Toolkit です。また、スタジオが UI をゼロからデザインし、アートをゲームにインポートする方法についても説明しました。前回のガイドは Unity 2021 LTS に基づいていました。

この新しい eBook では、Unity 6 の UI Toolkit に注目します。UI Toolkit は、ウェブ技術を参考にしたワークフローを備え、パフォーマンスと再利用性を最大化するよう設計されています。ウェブ開発経験のある UI デザイナーは直感的に使用できるでしょうし、UI プログラマーはゲーム制作における UI Toolkit の機能を明確に理解できます。このガイドはモジュール構成をとっていますので、各セクションを任意の順序で読むことができ、UI Toolkit の学習に役立つリファレンスとなっています。

それでは始めましょう。

主著者と貢献者

本ガイドと 2 つの UI Toolkit サンプルの主著者兼作成者は、3D およびビジュアルエフェクトの経験豊富なアーティスト、開発者、教育者である Wilmer Lin 氏です。

本ガイドと UI Toolkit サンプル –『Dragon Crashers』には、Unity のシニアコンテンツマーケティングマネージャーで、グラフィックデザイナーである Eduardo Oriz が大きく貢献しています。

また、本ガイドとサンプル QuizU には、Unity のコンテンツマーケティング管理担当シニアマネージャー Thomas Krogh-Jacobsen も貢献しています。

その他の Unity の貢献者

Camil Bouzidi、ソフトウェア開発者

Martin Côté、シニアグラフィック開発者

Hugo Bourret-Desmarais、シニアソフトウェア開発者

Benoit Dupuis、シニアテクニカルプロダクトマネージャー

Karl Jones、シニアソフトウェアエンジニア

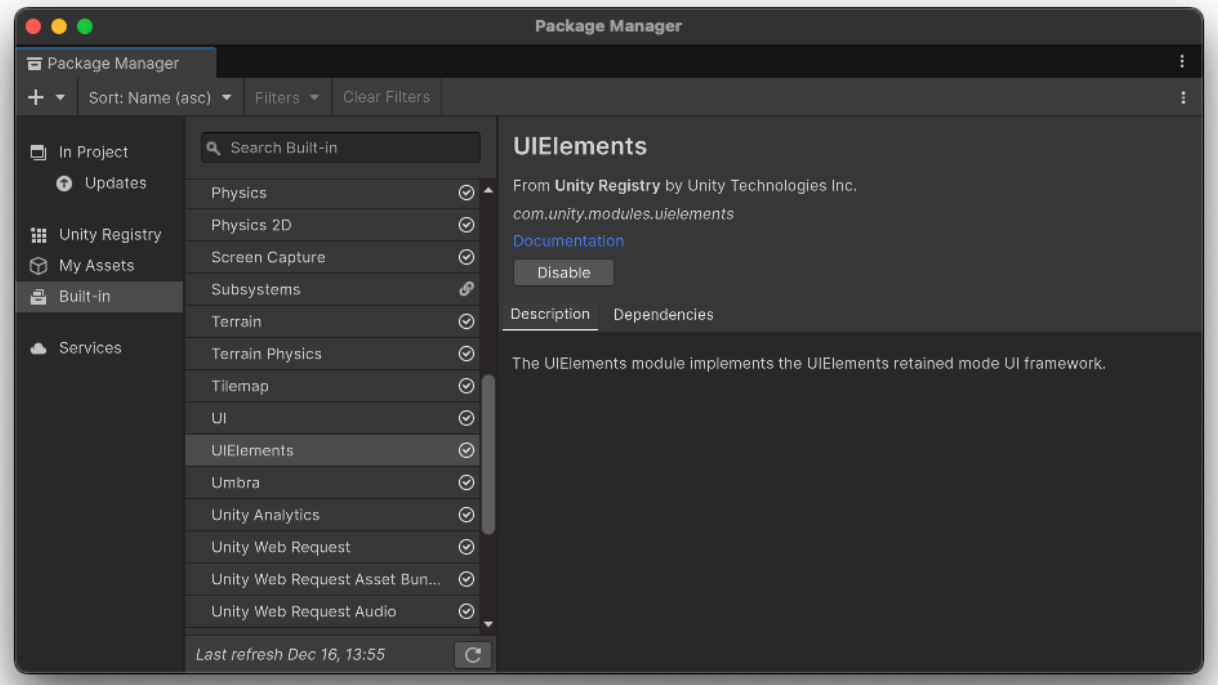
Antoine Lassauzay、スタッフソフトウェア開発者

Martin Paradis、スタッフソフトウェア開発者

Stefania Valoroso、マネージャー、製品デザイナー

UI Toolkit とサンプルプロジェクトのインストール

UI Toolkit は Unity 6 のコアプラットフォームに統合されているため、Unity 6 以降で使用するために別途パッケージをインストールする必要はありません。用意されているテンプレートの 1 つから新しいプロジェクトを開始すると、本ガイドの内容に沿って進められます。



UIElements は、UI Toolkit、UI Builder、およびそれらの機能の名前空間であり、これらはすべて Unity 6 に含まれている。

UI Toolkit の公式サンプル

この eBook では、主に次のサンプルを使用して、Unity プロジェクトの UI Toolkit 機能を説明していきます。各サンプルは、Unity Asset Store から無料でダウンロードできます。

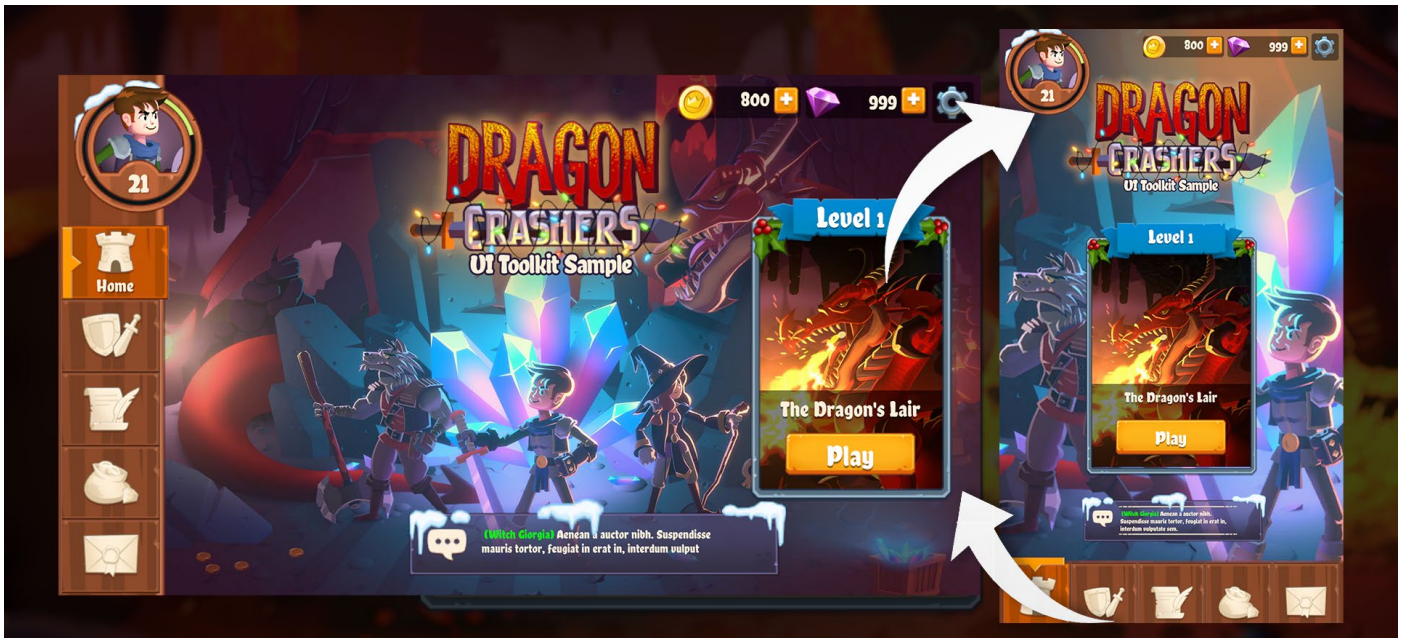
UI Toolkit サンプル –『Dragon Crashers』

このデモ は、2D プロジェクトのミニ RPG *Dragon Crashers* の一部をベースに、フロントエンドメニューシステムなどの豊富な機能を備えたインターフェースを実現するために、ランタイムで最新の UI Toolkit ワークフローを使用しています。

このデモは初心者向けのものではありません。UI の構造を調査し、デモを操作して具体的な実装を理解できる、経験豊富な Unity 開発者向けに作成されています。このデモは元々は Unity 2021 LTS 向けにリリースされ、その後 Unity 6 にアップデート されています。デモで詳しく学べるトピックをいくつか紹介します。

- プロジェクトの構造と命名規則
- テーマを使用して UI のバリエーションを作成し、縦向きと横向き用のサポートを追加
- タブ付きメニュー、インベントリ、メッセージ、カスタムコントロールなどの複雑な要素
- **SafeAreaAPI** を使用して、モバイル画面上のコンテンツをセーフエリア内に表示
- Localization を使用した多言語サポート
- UI コンポーネントとのデータの同期を簡略化するデータバインディング
- 一般的なカジュアルゲームインターフェースの実装方法の例

サンプルの 詳細な動画 は、アセットストアから ダウンロード できます。

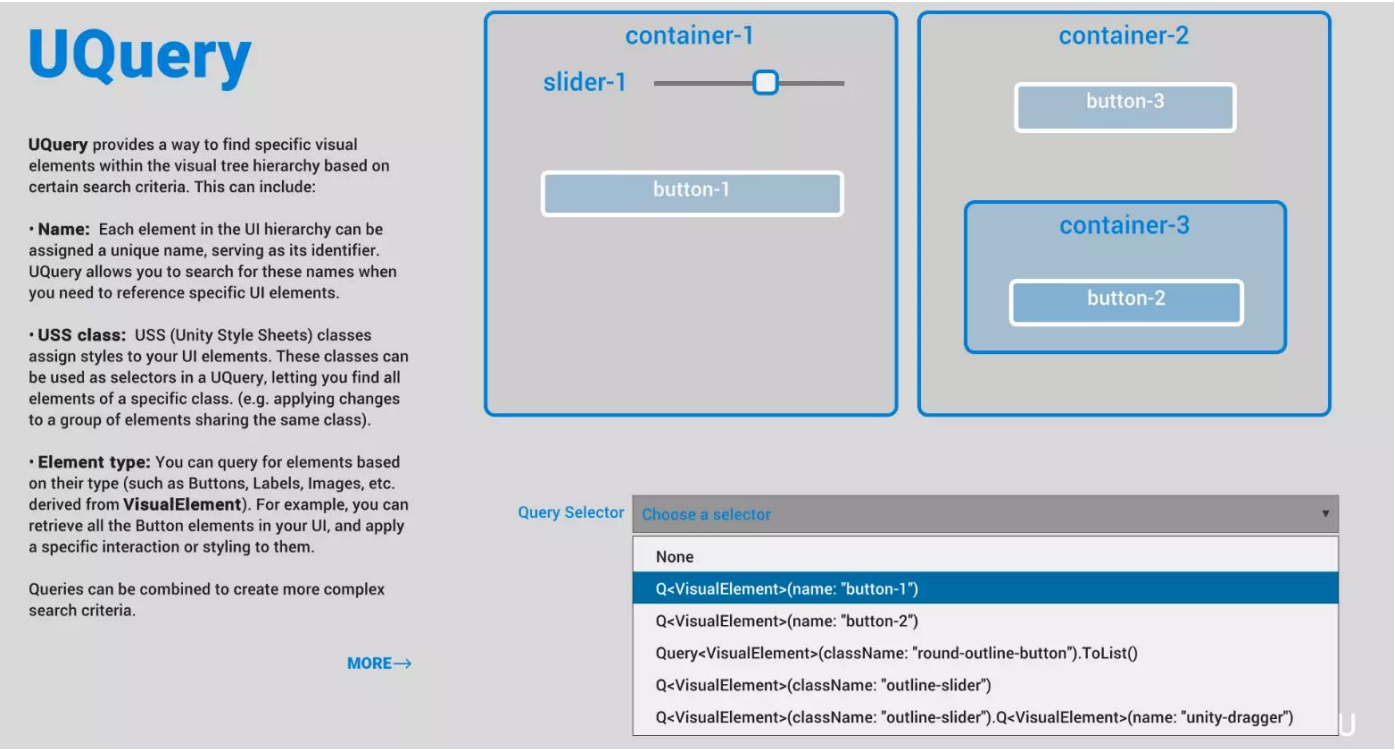


ホーム画面は横向きでも縦向きでも表示できます

QuizU

QuizU デモ では、Unity の UI Toolkit で構築した対話型のクイズゲームを紹介します。UI 開発者を対象としたこのプロジェクトでは、最新のゲームユーザーインターフェースを構築するための UI Toolkit ワークフロー、イベント駆動型アーキテクチャ、再利用可能なデザインパターンを取り上げています。このデモは以下の方法を紹介しています。

- UXML ファイルとネストされたビジュアルツリーを使用して、UI 要素を効率的に構築する。
- 対話的な要素がユーザー入力に反応できるよう、USS セレクターと擬似クラスを使用したスタイルルールを適用する。
- FlexBox の機能を使用して Flex ベースのレイアウトを作成し、応答性の高い UI 動作を実現する。
- セレクターを使用して UI 要素を動的に照会および変更する。
- イベント処理を再利用可能なクラスにカプセル化し、ドラッグやマルチタッチジェスチャなどのカスタムインタラクションを可能にする。
- イベントディスパッチによりイベントを段階的に処理し、伝播を管理する。
- USS transitions (USS 遷移) により、継続時間やイージングなどのプロパティを持つ UI 要素に滑らかなアニメーションやエフェクトを加える。



QuizU UI Toolkit デモのスクリーンショット

QuizU は元々は Unity 2022 LTS 用にリリースされましたが、Unity 6の新機能によって更新されました。プロジェクトには、カスタムコントロールの作成、データバインディングの設定、ローカライゼーションの実装に関するハウツーデモが用意されています。

プロジェクトはアセットストアから ダウンロード できます。

UI Toolkit の概要

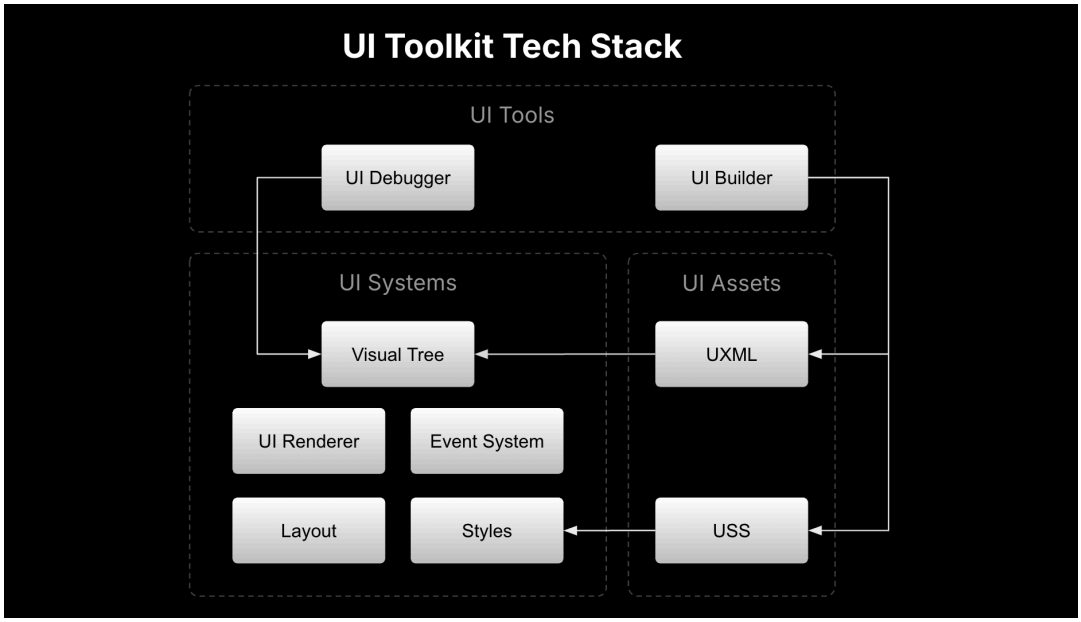
UI Toolkit は、従来の Unity UI (uGUI) や機能が古い IMGUI (エディター用ツール) よりも大きなメリットをもたらします。より現代的で柔軟性が高く、パフォーマンスを重視した代替手段を提供し、ほとんどのプロジェクトで優れたスケーラビリティを発揮します。また、エディターツールと、ランタイムのゲームやアプリケーションの両方に対応し、制作パイプライン全体をサポートします。

古い機能の UI システムと比べて、次のようなメリットがあります。

- **高速なイテレーション:**グローバルなスタイル管理とライブオーサリング機能により、作業とイテレーションを迅速化します。
- **レンダリングパフォーマンス:**レンダリングのヒントと動的テクスチャアトラスを使用して、ゲームのパフォーマンスをより詳細に制御できます。
- **コラボレーションの向上:**ロジック (C# コード)、UI 構造 (Unity XML (UXML)、ドキュメント)、スタイリング (Unity スタイルシート (USS) を使用) を分離することで、不一致を減らし、チームワークを向上させます。
- **再利用性:**プロジェクト内やプロジェクト間、そしてエディターとランタイム間でスタイルやウィジェットを共有し再利用できます。

UI Toolkit はウェブ技術からインスピレーションを得ており、ウェブアプリケーションに精通している開発者

には利点があります。HTML/XML やカスケーディングスタイルシート (CSS) などのマークアップ言語を使用するのが初めての方は、業界標準の強力なツールセットを習得する絶好の機会です。

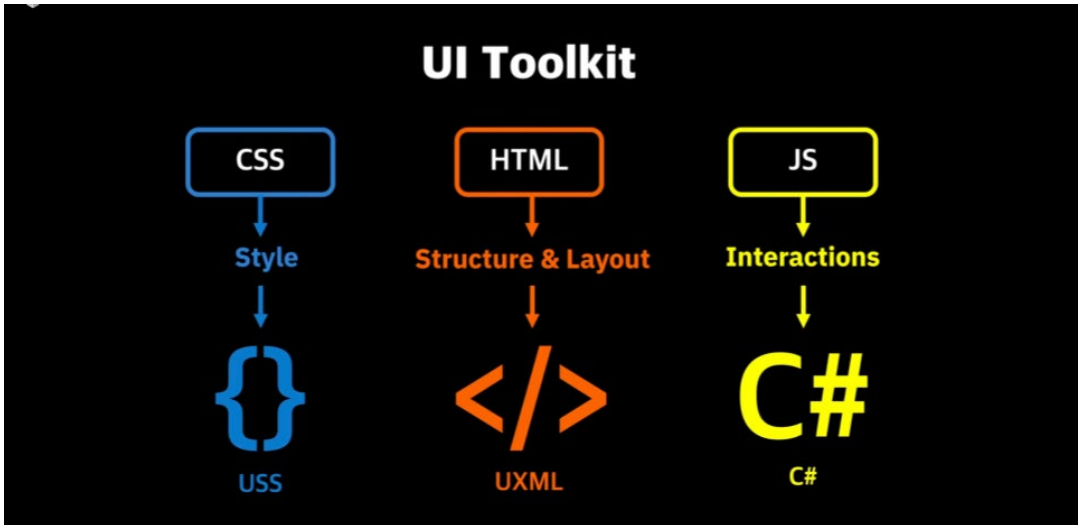


UI Toolkit インターフェースは基本的に UXML ファイルと USS ファイルで構成され、UI Toolkit システムによってレイアウトとスタイリングを作成します。

UI アセット

UI アセットは、UI を作成するためのビルディングブロックであり、UXML ファイルと USS ファイルで構成されます。UXML (Unity XML) は UI のコンテンツと構造を表し、HTML や XML などのマークアップ言語に似ています。

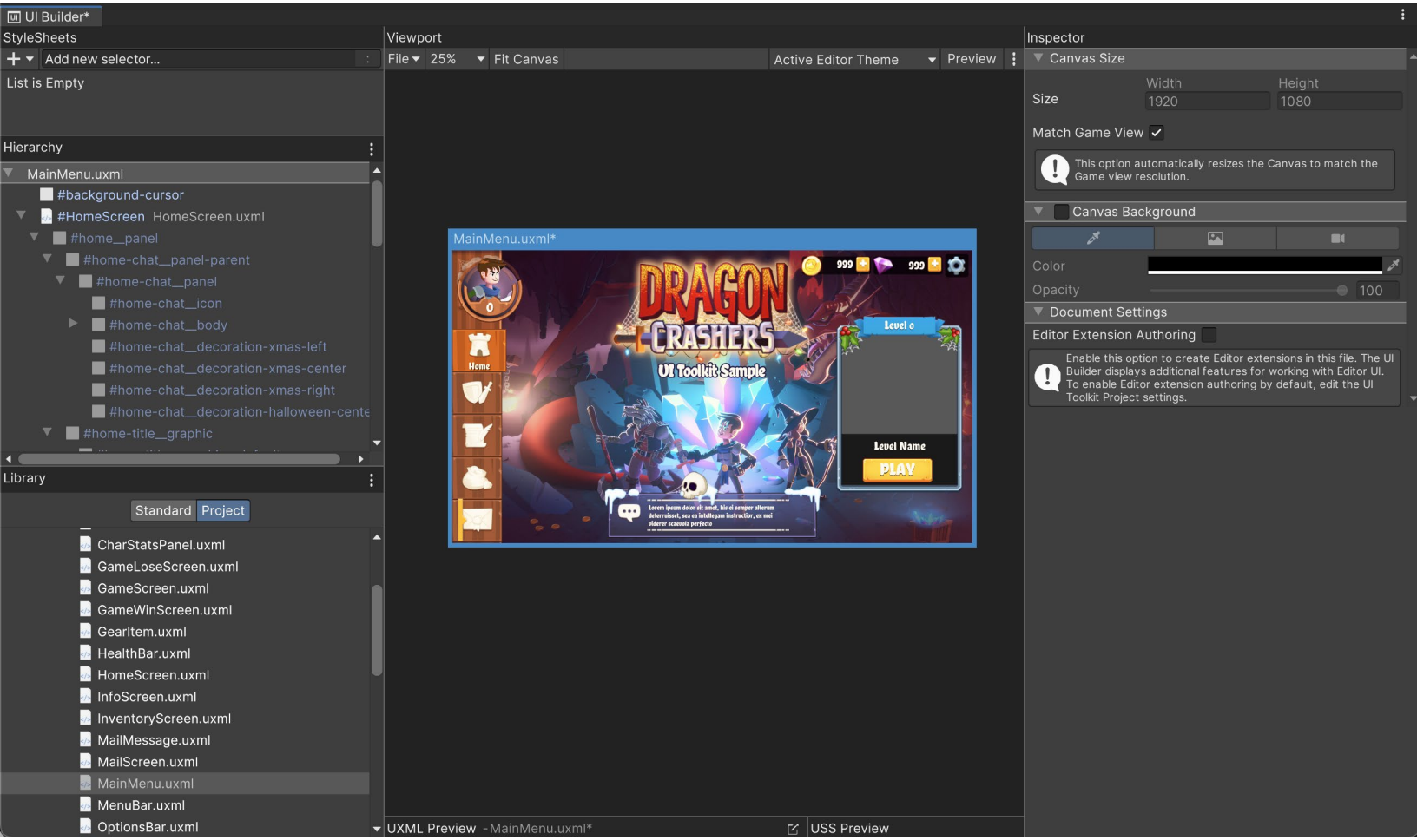
USS は、カスケーディングスタイルシート (CSS) からヒントを得ており、UI コンテンツの外観とスタイルを定義するために使用されます。本ガイドでは、UXML と USS の両方を使用しています。



UI Toolkit とウェブ技術の類似点

UI Builder

UI アセットは、任意の IDE でコードとして記述するか、UI Toolkit の UI Builder を使用して視覚的に作成できます。UI Builder のインターフェースを使用することで、アーティストやデザイナーは UI を視覚化しながら構築することができます。



グラフィックアセットとフォントアセットの準備

多くの UI デザインが Unity 外のデジタルコンテンツ制作 (DCC) アプリケーションを使用して作成されます。UI アーティストのスタイルや好みに応じて、Adobe Photoshop などのラスター描画アプリケーションやベクターベースのツールで UI をデザインできます。通常、UI グラフィックスの各部分は、透過性を備えたロスレスビットマップ画像 (PNG など) としてエクスポートされ、ランタイム時の効率化のために他の UI 要素とともにテクスチャアトラスに統合されます。

ベクターベースの DCC アプリケーションで作業する場合、UI Toolkit を使用するには、ベクターグラフィックスをラスター形式でエクスポートする必要があります。詳細については、ベクター画像のセクション を参照してください。

ビットマップ画像

Unity では、PNG、BMP、TIF、TGA、JPG、PSD など、最も一般的な画像フォーマットがサポートされています。これらの形式のファイルを Assets フォルダーに追加すると、Unity によって 3D プロジェクト用のテクスチャ 2D アセット、または 2D プロジェクト用のスプライトとしてインポートされます。テクスチャタイプは、インポート後に Inspector の Texture Type フィールドで変更できます。UI Toolkit は、UI ビットマップグラフィックス用に両方の形式をサポートしています。

テクスチャには画像のサイズと形式以外にそれほど多くの情報は含まれていませんが、スプライトには UI Toolkit で使用されるその他のプロパティが含まれています。

スプライト

スプライトとは 2D ゲーム開発用に準備されたテクスチャであり、Sprite Renderer コンポーネントによって使用されます。Unity の 2D スプライトは、アニメーション用にタイル化、リグ設定、スキン化したり、カスタムジオメトリを使用したり、2D ライティング用マップを追加したりできます。このセクションでは、UI Toolkit に関連する設定のみを取り上げます。2D グラフィックスの詳細については、『アーティスト向けの 2D ゲームアート、アニメーション、ライティング Unity 6 版』(https://unity.com/resources でまもなく入手可能) を参照してください。

ほとんどの UI グラフィックアセットは、Unity のワールドスケール (1 単位は 3D 空間の 1 立方メートルを表す) に従うのではなく、スクリーンスペースでレンダリングされます。UI Toolkit ではこれらのグラフィックスのスケールが管理されますが、スプライトの **PPU (1 単位あたりのピクセル数)** は UI 内のスプライトのサイズに影響します。例えば、スプライトの解像度をグリッド単位あたり 128 ピクセルにする場合は、PPU を 128 に設定します。

スプライトエディターには、青いハンドルでのトリミングや緑のハンドルでのスライスなど、グラフィックスを変更するためのツールが用意されています。これらのツールを使用すると、グラフィックスを タイル化できるようにしたり、スケーラブルな要素を作成する一般的な手法である 9 スライス を使用したりできます。

スプライトは、平らな長方形の 3D メッシュにマッピングされた 2D テクスチャです。デフォルトでは、インポート時の **Mesh Type** は**Tight** です。この設定では、スプライトの不透明 (透明でない) ピクセルのアウトラインにぴったり沿うようにメッシュを調整します。これにより、オーバードローが減少し、パフォーマンスが向上します。オーバードローは、GPU が 1 フレーム内で同じピクセルを複数回描画するときに、透明な領域が重なり合うことが原因で発生します。このメッシュは、スプライトエディターの Outline セクションで手動で調整および最適化できます。

スプライトモード は、スプライトアセットの Inspector から選択できる便利な機能です。以下のモードが用意されています。

- **Single:**これはデフォルトのモードで、画像には 1 つの画像要素のみが含まれます。
- **Multiple:** テクスチャソースファイルに同じ画像内の複数の要素が含まれている場合に選択します。そして、画像をさまざまなサブアセットに分割する方法を Unity が認識できるように、スプライトエディター で要素の位置を定義します。スライスされた各グラフィックスは、UI Toolkit で別々に使用できる個々のスプライトになります。
- **Polygon:**円形や正多角形の画像に最適なこのモードは、画像の形状にぴったり合ったアウトラインを設定し、アウトラインをより鮮明にするのに役立ちます。

レンダーテクスチャアセット

レンダーテクスチャは、カメラビューのスナップショットをテクスチャにしたもので、フレームごとに更新されます。これらは **Assets > Create > Rendering** で作成し、Output メニューの Camera コンポーネントから参照できます。そして、UI Toolkit でこれらのテクスチャを使用して、ミニマップ、キャラクター選択画面、UI に統合する必要があるその他のゲーム内ビジュアルなどの要素を表示できます。

レンダーテクスチャの例: *UI Toolkit サンプル:Dragon Crasher* では、キャラクタープレビューがレベルメーターに表示され、パーティクルエフェクトが UI ボタン上にレンダリングされます。

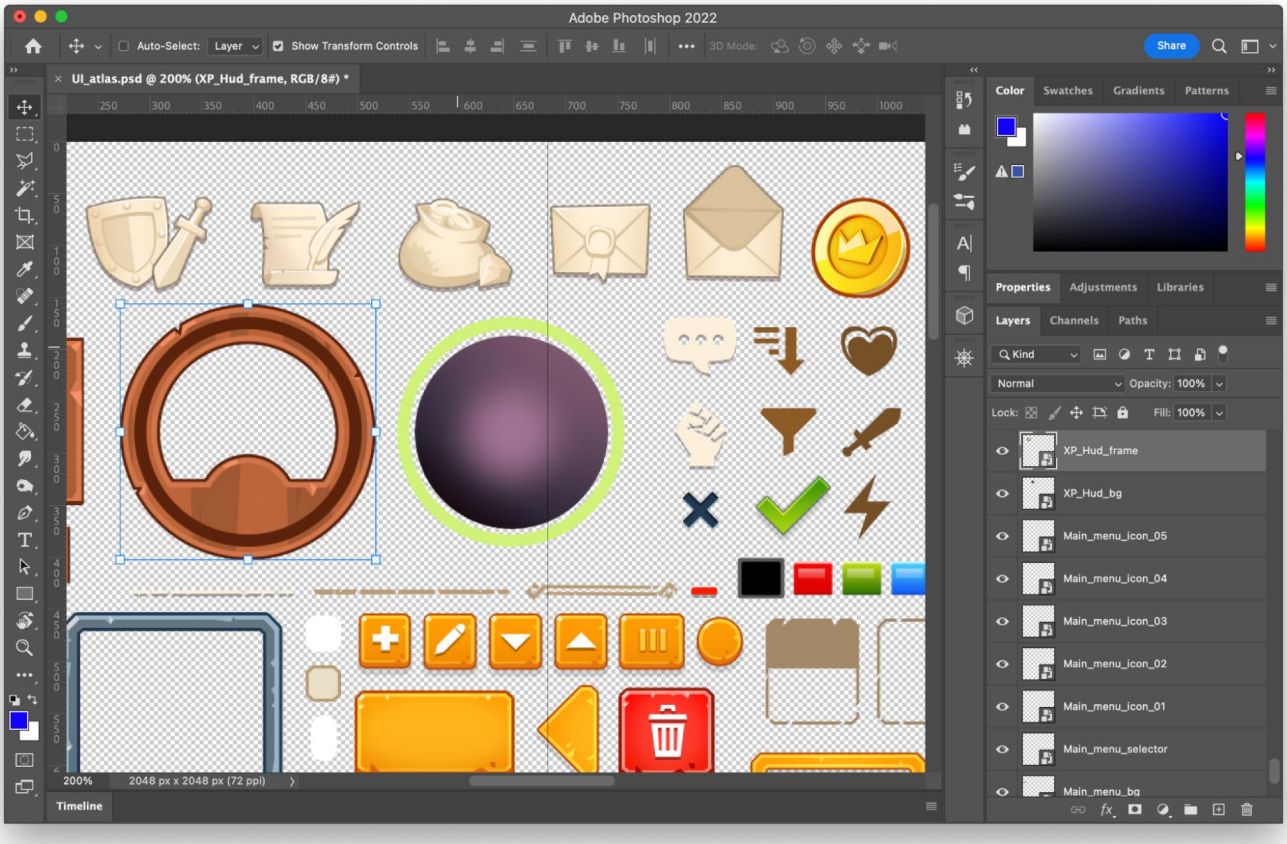
逆のユースケースとして、ゲーム要素内に UI を表示することもできます。例えば、アプリケーション内の 3D コンピューターモデルが、UI Toolkit で作成された機能的なインターフェースを表示しているとします。UI Toolkit キットのインターフェースをレンダーテクスチャにレンダリングし、Panel Settings と Camera で割り当てた後、3D モデルのマテリアルに適用できます。

レンダーテクスチャの設定と簡単なテスト

ただし、レンダーテクスチャは負荷が高いことに注意してください。使用は控え目にし、パフォーマンスを最適化するために、プロジェクトのプロファイリングを行ってください。他のアクティブなゲームプレイ要素がない全画面インターフェースの場合、この方法でエフェクトを追加しても、大きなパフォーマンス問題にはならないでしょう。

2D PSD Importer

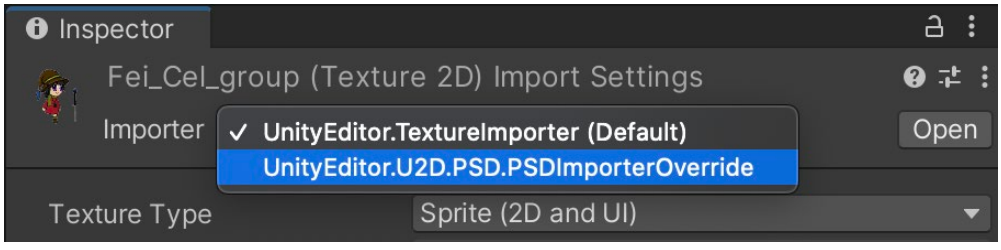
プロジェクトに 2D PSD Importer パッケージがインストールされていない場合、Unity は PSD (Adobe Photoshop のファイル) をフラットなテクスチャとしてインポートします。PSD ファイルは通常、複数の画像をレイヤーの状態で 1 つのファイルに保存するために使用されます。ほとんどの DCC ツールはこの形式でのエクスポートをサポートしています。



Photoshop での UI アセットの作成:通常、各要素には独自のレイヤーやグループがあるか、あるいはスマートオブジェクトです。スマートオブジェクトを使用すると、各要素を単独で制作し、後でメインドキュメントでサイズを変更した場合にも、その要素の元の解像度を保つことができます。

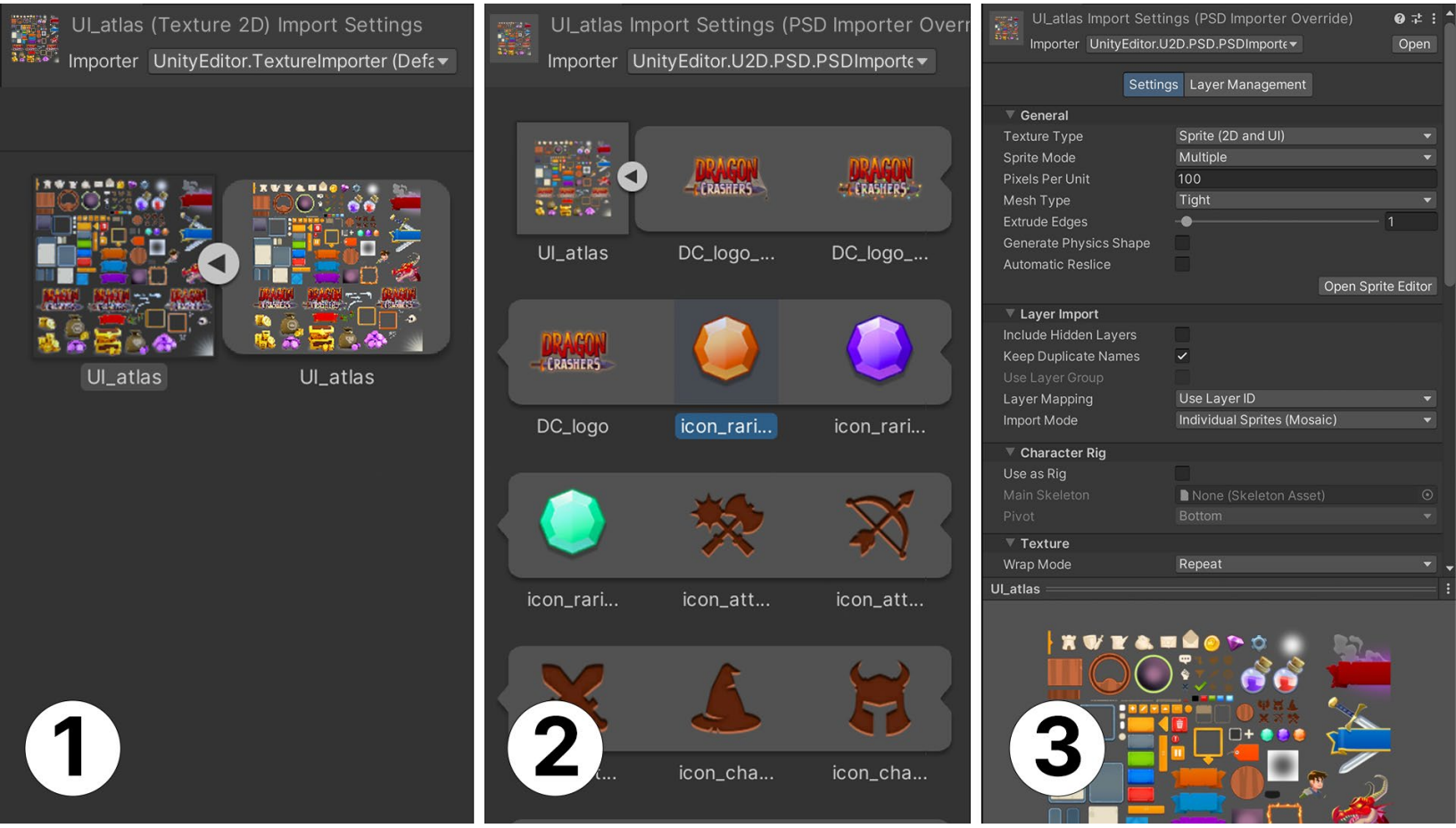
PSD ファイルは Unity に直接インポートできるため、ワークフローを簡素化することができます。各レイヤーを個別のファイルとしてエクスポートしたり、変更があるたびにそのプロセスを繰り返したりする必要はありません。

Package Manager から 2D PSD Importer パッケージをインストールした後、PSD ファイルが Inspector からインポートされていることを確認します。



Inspector で PSD Importer を選択して、ファイルを処理するオプションを確認します。

UI アセットを操作するときは、Inspector の**Character Rig** の下にある **Use as Rig** オプションの選択を解除します。この設定は 2D キャラクターのスケルタルアニメーションにのみ関係し、UI 要素には必要ありません。また、レイヤーをインポートするためのオプション (例: 非表示のレイヤーを破棄する、オブジェクトをレイヤー別にグループ化するなど) も見つかります。



PSD Importer に切り替えると、インポートオプションが増えます。

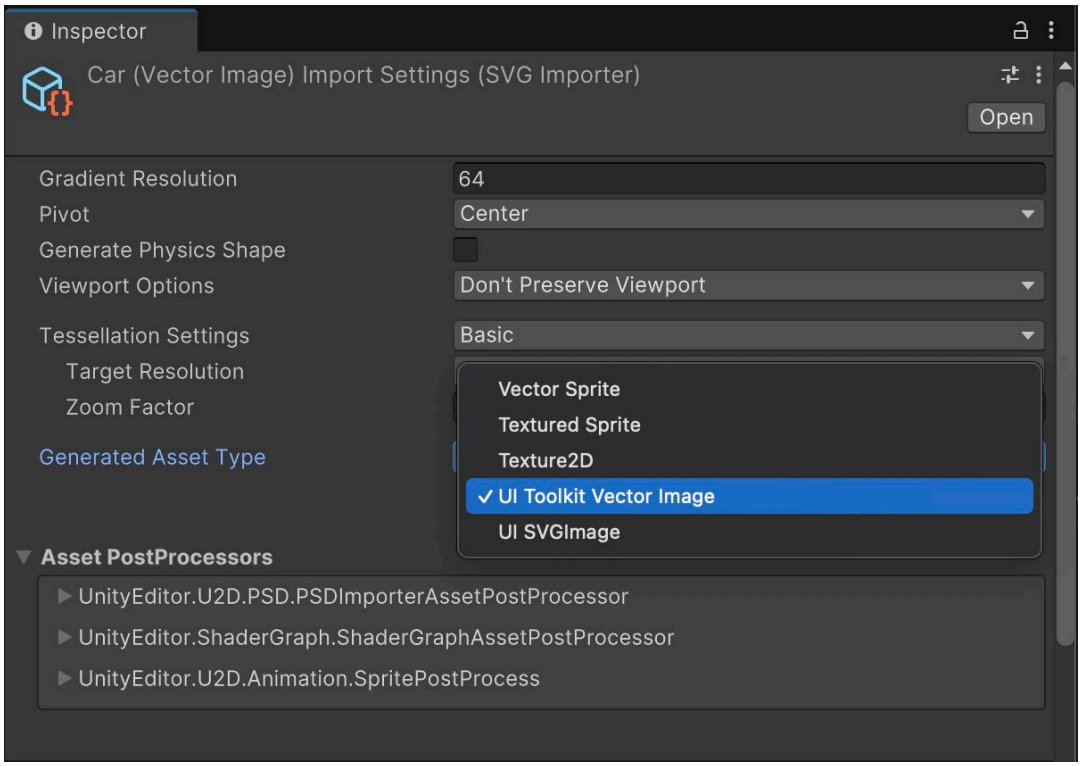
PSD から生成された Project ウィンドウのスプライトは、通常のスプライトとして使用できます。通常のスプライトアセットと同様に、スプライトエディターからスライス、アウトラインの変更、単位あたりピクセル数 (PPU) の変更を行うことができます。

ヒント:反復設計
Unity は、PSD ファイルに含まれているスプライトを、保存するたびに自動的に更新します。これにより、プレースホルダーを素早く作成し、ゲームビューで変更を確認しながら反復的に改良できます。ファイルを切り替えたり、チーム内の他の開発者からサポートを受けたりすることなく、コンテキスト内で作業を確認できるため、大幅な時間の節約になり、作業品質の向上につながります。

ベクター画像

本ガイドの執筆時点では、ベクター形式の対応については開発途上ですが、UI Builder で背景画像のオプションとして利用できます。ただし、現在 UI Toolkit で推奨される画像形式はラスター画像 (スプライトとテクスチャ) です。

この機能をテストするには、Vector Graphics パッケージが必要ですが、プレビュー段階のため Package Manager ではデフォルトで非表示になっています。ドキュメントの手順に従ってインストールしてください。このパッケージには、ベクターグラフィックスをポリゴンに変換するときのテッセレーションレベルを定義する設定が含まれています。**Generated Asset Type** の設定で、**UI Toolkit Vector Image** を選択して UI Toolkit で使用できるようにします。



Vector Graphics パッケージでは、SVG 画像を限られた容量で使用して、ゲームや UI でテストできます。

現在、SVG ファイルにはレンダリング時にポリゴン状にテッセレーションが適用されるため、ベクター画像のメリットが制限されています。拡大時に、ベクター画像らしい滑らかな曲線ではなく、ポリゴンエッジが目立つ場合があります。本ガイドの執筆時点では、UI Toolkit でアンチエイリアスはまだ有効になっていません。

ベクターサポートの完成バージョンでは、実際のベクター形状をネイティブにサポートできるようになる見込みであるため、Vector Graphics パッケージを用意する必要はありません。

Fonts

UI Toolkit は Font と FontAsset の両方をサポートしています。

- **Font:**後方互換性のために、TTF や OTF などの標準フォント形式がサポートされています。ただし、それらは裏で自動的に FontAsset に変換されます。
- **FontAsset:**これが推奨される形式です。元のフォントアセットを変更することなく、カーニングやベースラインなどを微調整できます。ゲームでよく見られるデザイン性の高いフォントの場合に役立ちます。
    - FontAsset では、文字セット、解像度、アトラス設定オプション など、アトラスの作成方法を詳細に制御することもできます。これらの設定は、文字種の多い言語をサポートする Unicode フォントで作業する場合には特に、メモリフットプリントを減らすのに役立ちます。

テクスチャパッカー

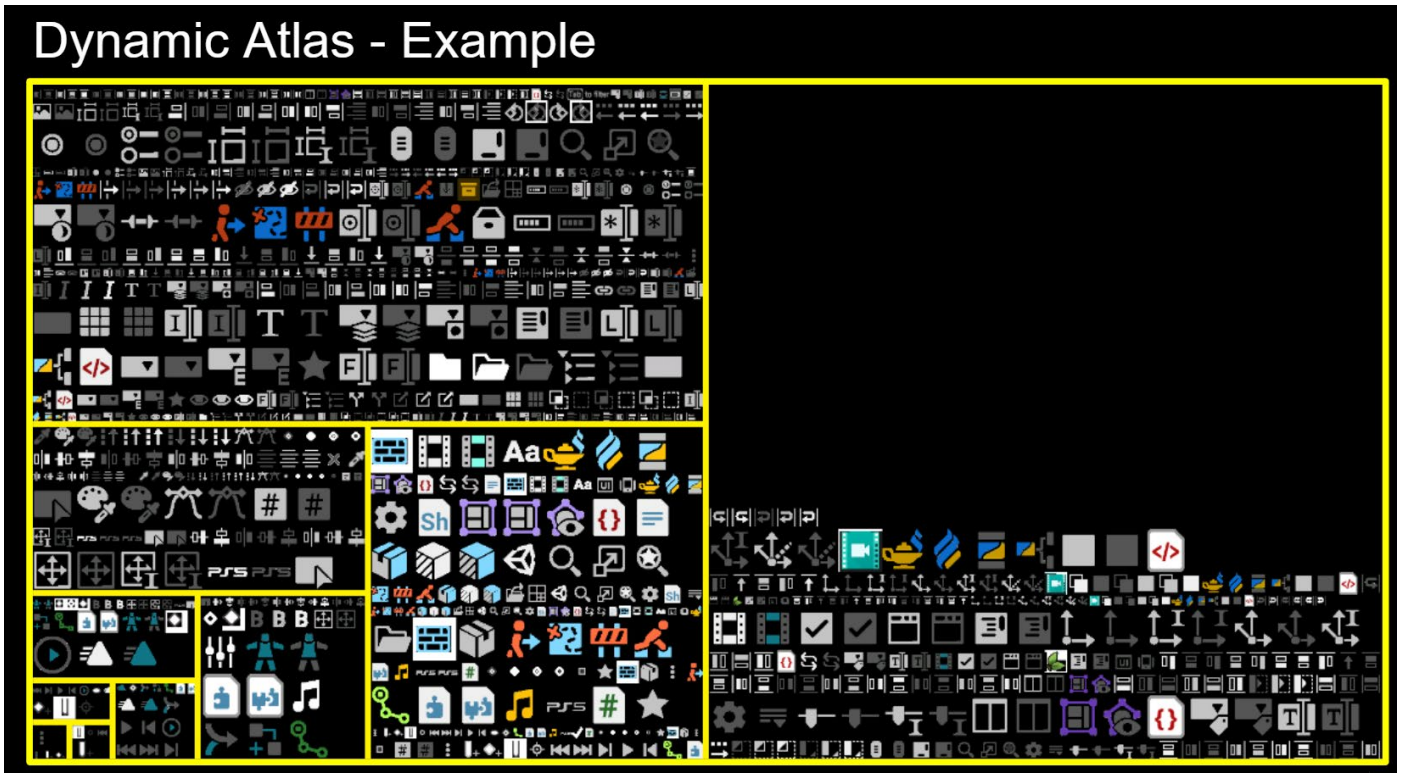
2D グラフィックスを同じテクスチャに統合することは、ドローコールを減らしてメモリ使用量を改善するための一般的な最適化手法です。UI Toolkit は、以下の 2 つのアトラスシステムをサポートしています。

スプライトアトラス

UI Toolkit サンプル – 『Dragon Crashers』の典型的なゲーム UI アトラス

スプライトアトラス は Unity の 2D ゲーム開発およびスプライト用のアトラスツールですが、UI グラフィックスにも使用できます。同じプロジェクトフォルダーにアセットが自動的にパックされ、スプライトのアトラス、法線マップ、マスクマップが作成されます。プラットフォーム固有のバリアントのサポート、高度な制御のための API も用意されています。スプライトアトラスは、アセットをパックするためにエディターで使用されるのが一般的であり、ランタイムでは使用されません。

動的アトラス



Unity エディターから生成され、Texture Atlas Viewer に表示された動的アトラス。アトラスはテクスチャの最大許容サイズに合わせて縦横ともに 2 の倍数で拡大されます。

UI グラフィックスがスプライトアトラスでパックされない場合、UI Toolkit の動的アトラスの機能により、プリパス中に自動的にパックされます。

ビジュアル要素内の参照画像は、UI Document の Panel Settings で定義された基準に従ってアトラス化されます。例えば、パックするテクスチャの最小サイズや最大サイズを定義したり、他のプロパティに基づいて画像をフィルタリングを行ったりできます。生成されたアトラスは、UI Toolkit Debugger 内の Texture Atlas Viewer でプレビューできます。

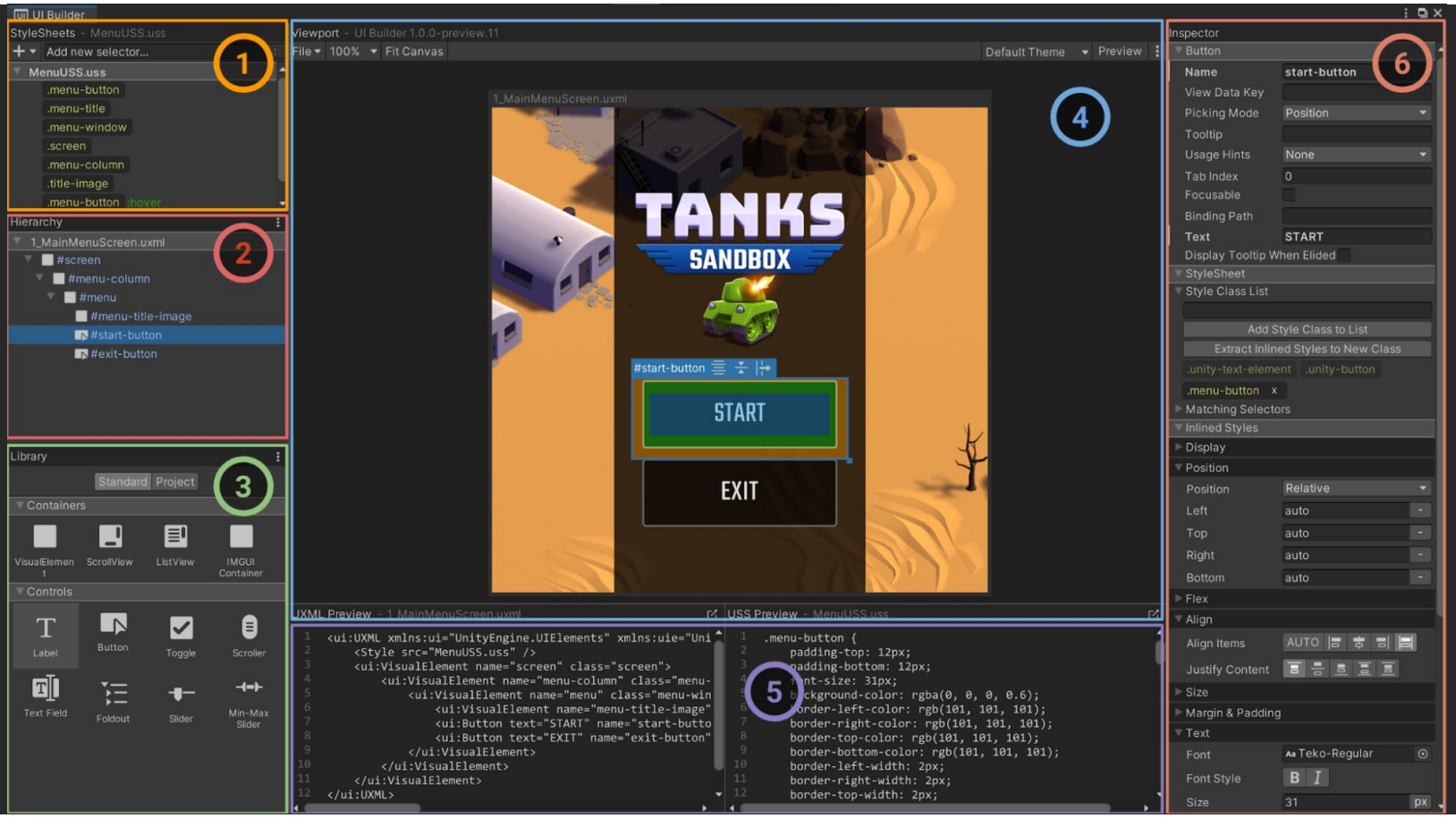
動的アトラスツールは、ランタイムとエディターの両方で機能し、プレイヤーのインベントリのように、動的に生成される UI 要素に役立ちます。

グラフィックスに関する一般的なベストプラクティスは次のとおりです。

- モックアップ画面の作成を開始したら、後で作業をやり直さずに済むように、描画ソフトウェアでターゲット解像度を最も高く設定してください。例えば、最大 4K のグラフィックスをサポートする場合は、それを最小動作解像度にします。
- ラスター画像を作成後に拡大することは避けてください。ピクセル化やぼやけが発生し、見た目の品質が低下する恐れがあります。代わりに、最初はサポートされている最高の解像度を設定し、後でグラフィックスアプリケーションからエクスポートする際に解像度を下げてください。

- ベクターグラフィックスを使用してデザインする場合、後でアセットのサイズを変更することはそれほど問題になりません。ただし、Reference Resolution を使用して、各アセットが正しい相対スケールになるようにします (例えば、要素のアウトラインの太さを一定に保つなど)。
- グラフィックアセットの解像度が低くなっている場合は、2D Enhancer と、スプライトエディター内の AI を利用したアップスケール機能を試してください。
- PSD を Unity に直接インポートし、2D PSD Importer を最大限に活用します。PSD ファイルを保存すると、レイヤーに対する変更が Unity に反映されます。Unity に PSD ファイルがある場合は、Version Control の恩恵を受けることもできます。
- インポートプロセスを自動化してください。グラフィックアセットを追加するたびにアセット設定を手動で変更することは避けてください。プリセット 機能を使用すると、1 つのアセットに適用した設定を保存して、特定のフォルダー内の同じ種類のすべてのアセットに自動的に適用できます。
- アセットに対するチェックの実行や、複数のアセットに対する設定の一括変更など、このプロセスをさらに自動化する必要がある場合は、Asset PostProcessor API を使用できます。

UI Builder



UI Builder には、Window > UI Toolkit > UI Builder メニューからアクセスできる。

UI Builder を使用すると、メインエディターに統合されたビジュアルのインターフェースで UXML や USS ファイルを作成、視覚化、変更できます。UI Builder の主な機能を見てみましょう。

1. **StyleSheets:**これにより、UXML ドキュメントや UI 要素間でスタイルを共有するレイアウトとスタイルの書式ルール (USS セレクター) を管理できます。
2. **Hierarchy:**シーンビューと同様に、UXML ドキュメント内のビジュアル要素の階層を表示します。
3. **Library:**ここには、ボタン、ラベル、スライダーなど、階層にすぐに加えることのできる定義済みまたはカスタムの制御が用意されています。ここから、現在の UXML に他の UXML (テンプレート) を追加することもできます。
4. **Viewport:**インターフェースの外観を示します。ギズモを使用してキャンバスで要素を直接編集できます。
5. **Code Previews:**UI Builder の裏側で作成されている UI Document (UXML) とスタイルシート (USS) のコードを表示します。適切に表示するには、ウィンドウのサイズを調整してください。
6. **Inspector:**選択した要素または USS セレクターの属性やスタイルのプロパティを変更するのに使用します。

ヒント: UI アセットの保存

UI Builder では、**Viewport** メニュー (**File > Save**) から変更内容を保存します。これにより、開いているすべての UXML ファイルと USS ファイルが保存されます。

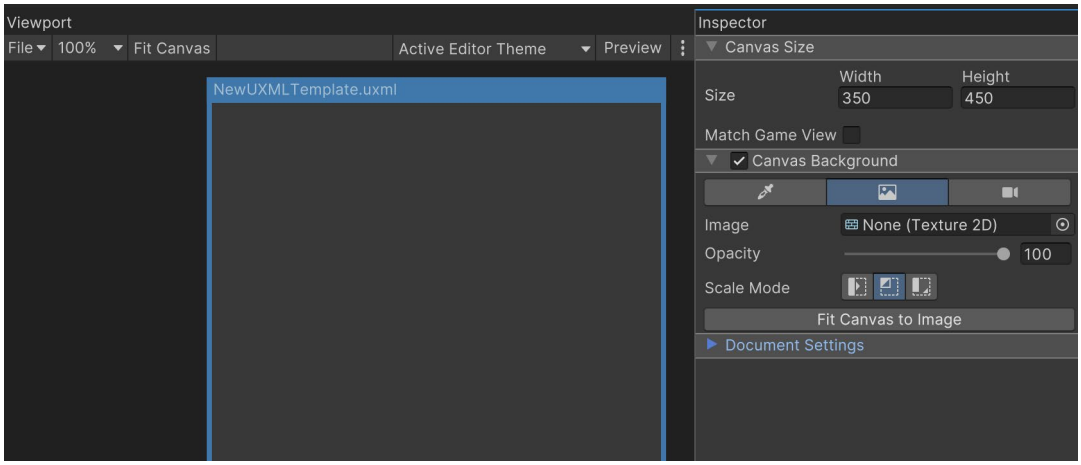
Unity UI とは異なり、UI Toolkit で変更を加えながらエディターでゲームを実行することができます。UI Builder のキャンバスヘッダーにあるファイル名の横に表示されるアスタリスク *は、保存されていない変更があることを示します。

Canvas Background

Canvas Background を有効にすることで、色や背景画像の上で、要素に対して視覚的にスタイリングが行いやすくなります。Hierarchy ペインで UXML ファイルを選択した後、最終的な UI インターフェースに近い Canvas Background を選択して、スタイルの変更を背景と併せて評価することができます。

Canvas Background には、いくつかのオプションがあります。

- **Background Color**:ゲーム環境の特定のシェードや色相を表します
- **Image**:背景として使用するスプライトやテクスチャの選択用です (モックアップ画面やリファレンスアートの再現に便利です)
- **Camera**:現在のゲームプレイを背景に表示し、実際のゲームのコンテキストで UI を確認できます



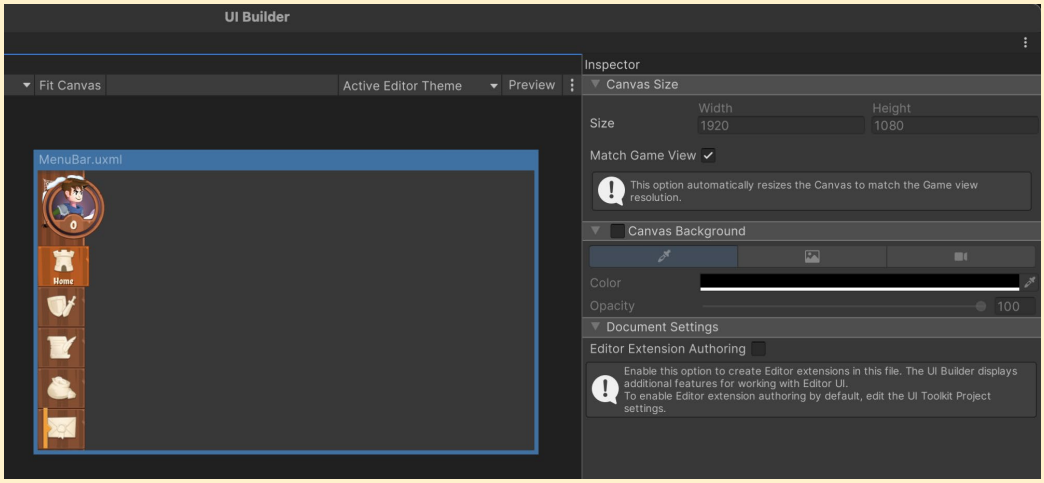
新しい UXML ドキュメントのキャンバス:Color と Image のオプションを使用して外観を調整します。

Viewport の設定

作業領域を移動する際には、ズームレベル (25% から 500% の間) を調整するか、現在の画面領域に合わせて自動的にズームを調整する **Fit Canvas** オプションを選択します。

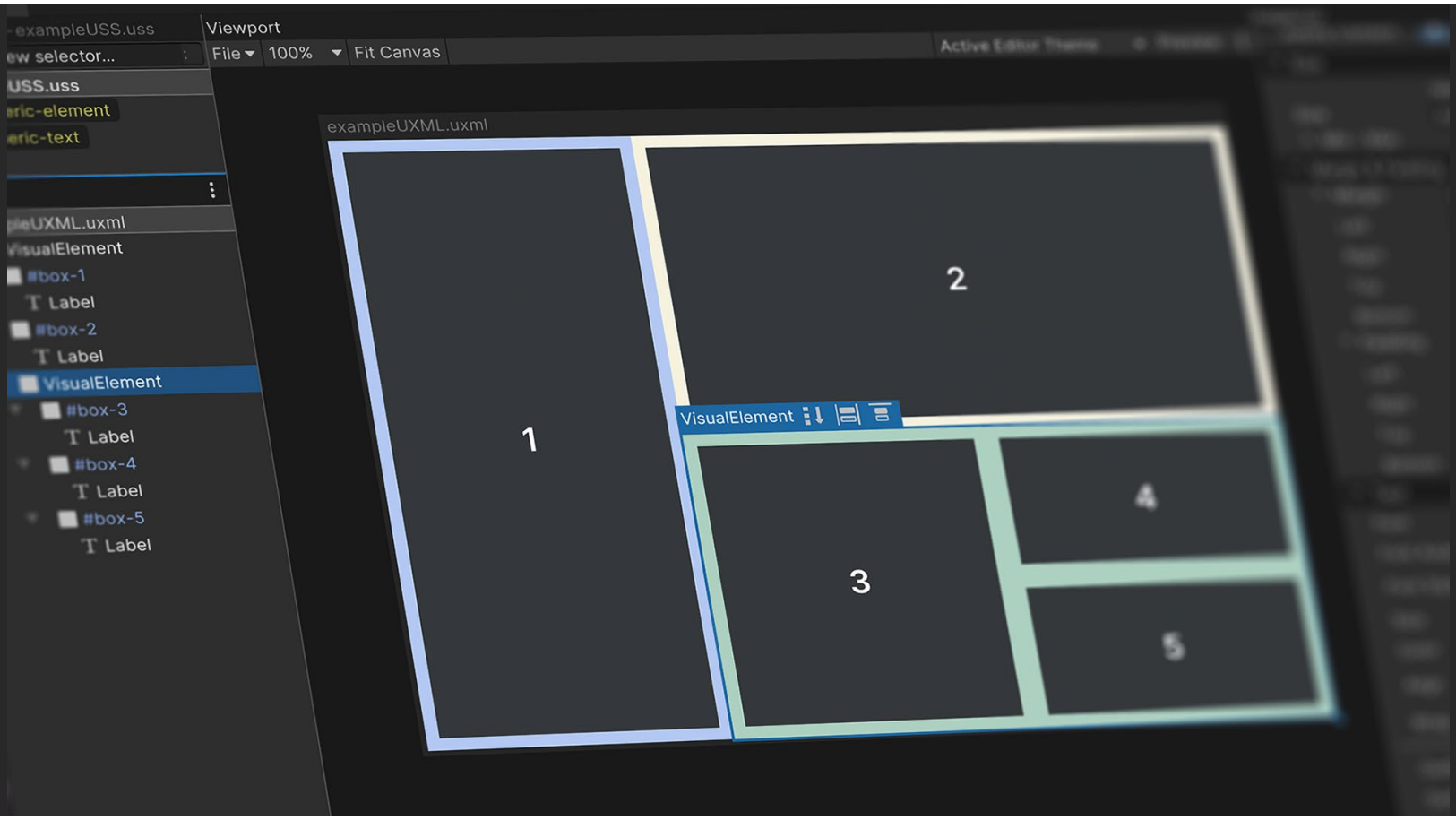
Preview を使用すると、選択した要素を誤って編集することなく、UI を視覚化することができます。アクティブな場合、Viewport は特定のマウスイベント (ホバリング、フォーカスなど) に適用されるスタイルも表示できます。

ヒント: ゲームビューとテーマの一致



ランタイム UI に近い表示にするには、Hierarchy で現在ロードされている UI Document (UXML) を選択し、**Match Game View** をオンにします。これにより、Viewport のサイズがプロジェクトの Reference Resolution に変更されます。このパラメーターを変更しても、変わるのは見た目のみで、UI ファイル自体には影響しないことを覚えておいてください。UI Builder では、プロジェクトで使用されるさまざまなテーマを事前に視覚化することもできます。この機能については、本ガイドで後ほど説明します。

レイアウト



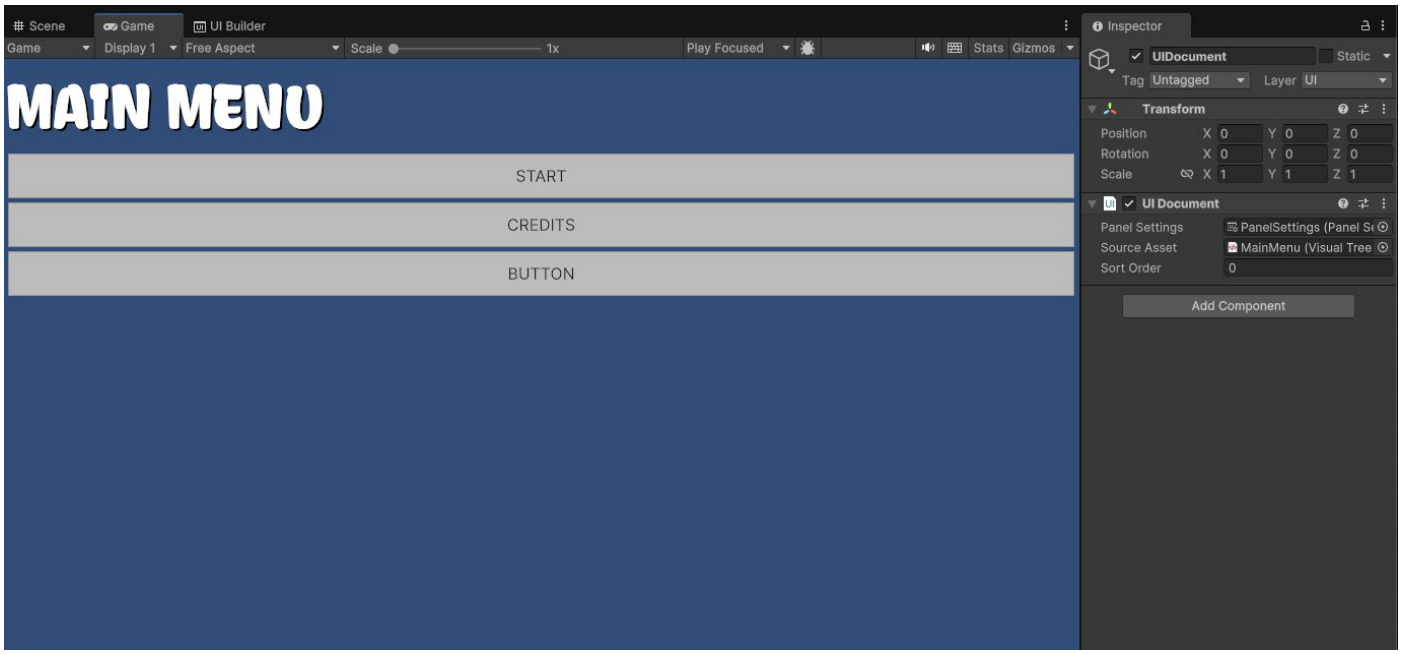
UI Builder にはレスポンシブレイアウトをデザインするために必要なツールがすべて用意されています。

このセクションでは、UI Builder でレイアウトを作成するための基本的な手順を説明します。

UI Builder は、デザイナーにとって使いやすい WYSIWYG ツールで、コードを書かずに UXML ファイルや USS ファイルを効率よく作成するのに役立ちます。チームによっては、コードで直接 UI を作成することを好む場合もありますが、UI Builder を使用すると、アーティストはクリエイティブにコントロールできるため、ワークフローを大幅に改善できます。UI Builder で変更を加えると、コードが自動的に生成され、UI Builder で作成するすべてのものを UXML や USS でコードとして直接実装できます。

レスポンシブレイアウトを効率的に構築できることが、UI Toolkit と UI Builder を使用する大きなメリットです。このようなレイアウトは、画面解像度や画面比率が異なる複数のプラットフォームを対象にしているゲームにとって必須の機能です。このセクションでは、UI Builder でレイアウトを作成するための基本的な手順を説明します。

以下に示されているのは、UI Builder の UXML Preview パネルにコードが表示された UXML ファイルです。UI Builder の、*File > New* でアセットを作成し、*Save As* で保存します。



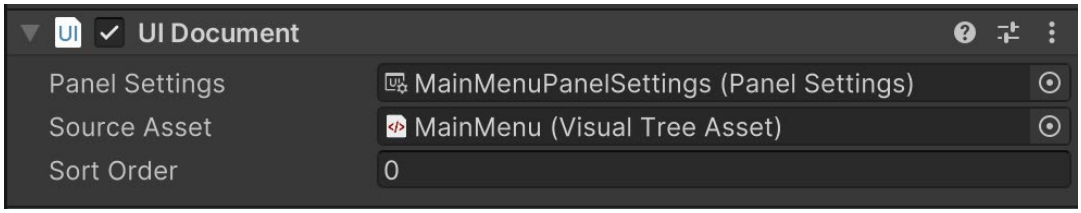
Code Preview ウィンドウの右上にあるアイコンをクリックすると、IDE で開きます。

この UXML コード例では、ビジュアル要素 が HTML に似たマークアップ言語 (開始ブラケットと終了ブラケットによる形式) で表されていることがわかります。例えば、開始ボタンの構文は次のとおりです。

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

コアランタイムコンポーネント

UI Toolkit の要素はシーンビューには表示されません。インターフェースは UI Builder で作成したとおりに表示できますが、ゲームビューではより正確なプレビューがターゲット解像度で表示されます。ゲームビューで UI をレンダリングするには、次のスクリーンショットに示すように、**Panel Settings** アセットと **Visual Tree** アセット (UXML) を持つ **UI Document** コンポーネントがゲームオブジェクトに必要です。



UI Document コンポーネントは、表示する UXML を定義するもので、デフォルトの Panel Settings アセットが含まれています。Sort Order フィールドは、同じ Panel Settings を使用する他の UI Document に対してこのドキュメントをどのように表示するかを決定します。

Inspector の**Add Component** メニューを使用してこのコンポーネントをゲームオブジェクトに追加するか、Hierarchy で右クリックして **UI Toolkit > UI Document** を選択すると、Panel Settings アセットが自動的に割り当てられます。

Panel Settings アセットは、UI Document コンポーネントをランタイムでインスタンス化する方法と可視化する方法を定義します。複数の Panel Settings アセットを指定し、UI ごとに異なるスタイルを設定することもできます。例えば、ゲームに HUD やミニマップが含まれている場合、こうした特別な UI ではそれぞれ独自の Panel Setting を指定することができます。

Assets > Create > UI Toolkit > Panel Settings Asset でアセットを作成します。このアセットはユーザーのルートプロジェクトフォルダーに追加され、その後ゲームオブジェクトの UI Document コンポーネントに適用できます。

レスポンシブレイアウト:Flexbox

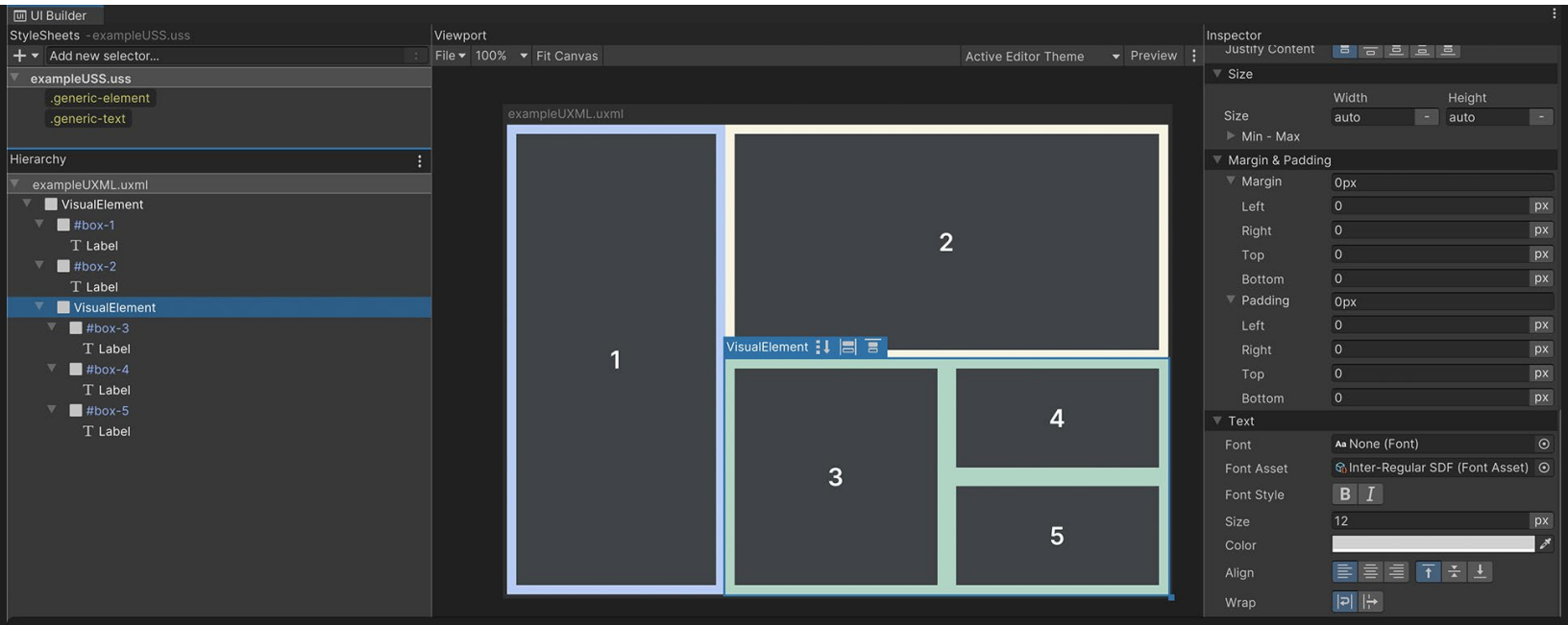
UI Toolkit は、Flexbox のサブセットを実装する HTML/CSS レイアウトエンジンである Yoga に基づいて、ビジュアル要素を配置します。Yoga と Flexbox に馴染みのない方の場合、UI Toolkit のレイアウトエンジンの背後にある原則を把握するのに、本章が役立ちます。

Flexbox（フレキシブルボックスレイアウト）は、アイテムを行または列に配置する手法です。Flexbox アーキテクチャは、きちんと整理された複雑なレイアウトを開発するのに便利です。以下のようなメリットがあります。

— **応答性の高い UI**:Flexbox により、すべてがボックスやコンテナのネットワークに整理されます。これらの要素を親と子にネストし、シンプルなルールを使用して画面上に空間的に配置することができます。子は親コンテナの変更に自動的に応答します。レスポンシブレイアウトは、画面のさまざまな解像度やサイズに対応し、より簡単に複数のプラットフォームを対象にすることができます。

— **複雑さの整理**:スタイルで、ビジュアル要素の美的価値をコントロールするシンプルなルールを定義します。1 つのスタイルを一度に何百万もの要素に適用できます。変更内容は UI 全体にすぐに反映されます。これにより、UI デザインが個々の要素の外観を作成することではなく、一貫した再利用可能なスタイルに集約されます。

— **ロジックとデザインの分離**:UI レイアウトとスタイルがコードから切り離されます。これにより、デザイナーと開発者が依存関係を壊すことなく、並行して作業を進めることができます。各ユーザーが得意なことに集中できます。



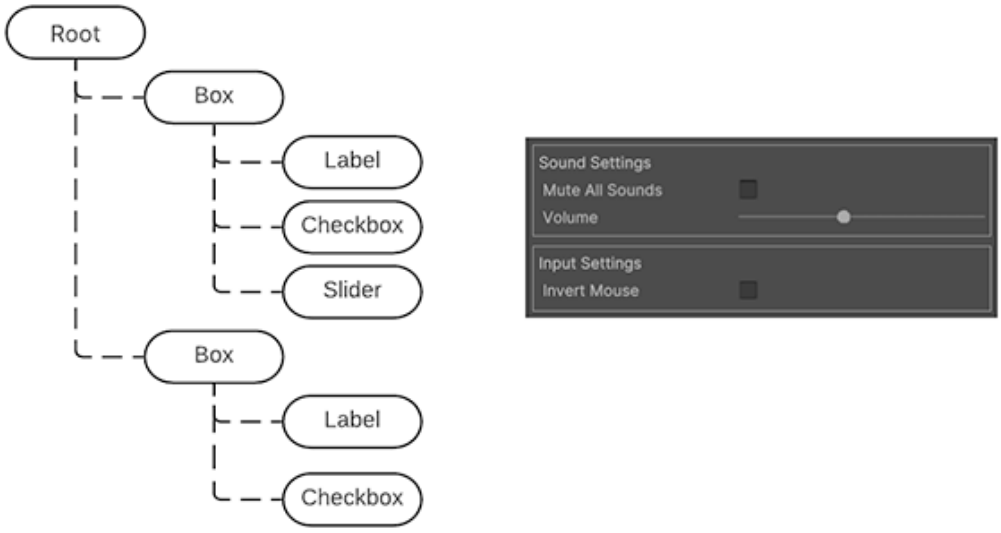

```
//Add logic that interacts with the UI controls in the `OnEnable` methods
private void OnEnable()
{
    // The UXML is already instantiated by the UIDocument component
    var uiDocument = GetComponent<UIDocument>();
    _box1 = uiDocument.rootVisualElement.Q<VisualElement>("box-1");
    _label =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

ロジックとデザインの分離:プログラマーがビジュアル要素を実際のゲームロジックに結び付ける一方、デザイナーはそれらに対するスタイルの定義に集中できます。

ビジュアル要素

UI Toolkit において、各インターフェースの基本となるビルディングブロックは、ビジュアル要素です。ビジュアル要素は、各 UI Toolkit 要素 (ボタン、画像、テキストなど) の基本クラスです。UI Toolkit はゲームオブジェクトに相当するものと考えてください。

1 つ以上のビジュアル要素で構成される **UI 階層** は、**ビジュアルツリー**と呼ばれます。



ビジュアルツリーの簡略化された UI 階層と右側の外観

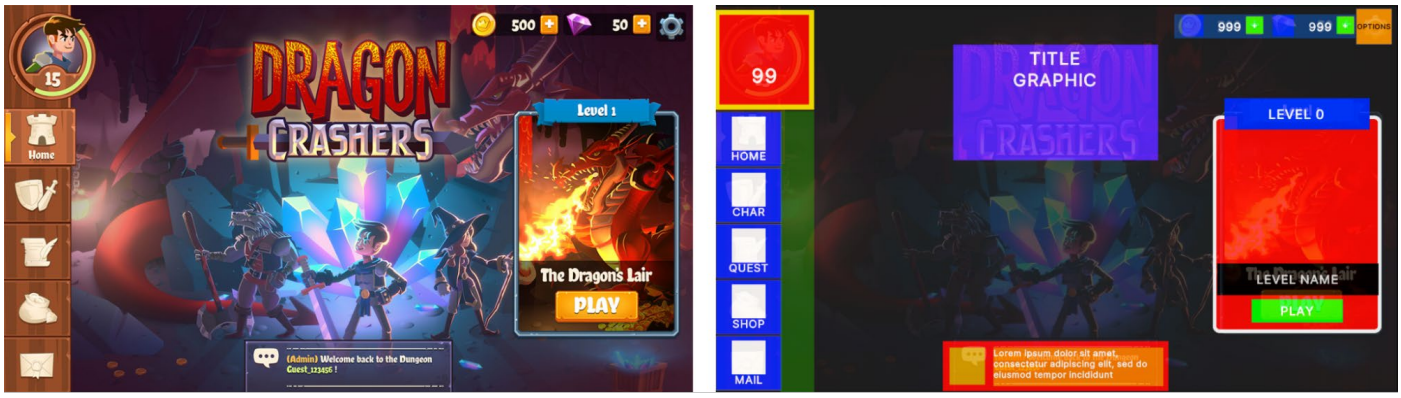
複数のビジュアル要素の組み合わせは UXML ファイルに格納されます。このファイルには、階層に関する情報と、そのスタイル (スタイルシートや USS を使用しない場合) 、ビジュアル要素のレイアウトが含まれます。

UI Toolkit についてさらに掘り下げる前に、**Flexbox レイアウト** の基礎を理解する必要があります。これは、UI Builder の基本のビジュアル要素を使用して実演することができるものです。

ビジュアル要素の配置

UI のモックアップを作成するときは、各画面を別個のビジュアル要素のグループとして扱います。画面を水平方向または垂直方向に積み重ねるボックスにどのように分割するかということと、情報を整理しておくための子ボックスが必要かどうかについて考えます。

下の例では、1 つの大きなビジュアル要素はコンテナ、メニューバー、メニューバーの左側の要素である可能性があります。各ボタンを表現するために、子のビジュアル要素を分離します。

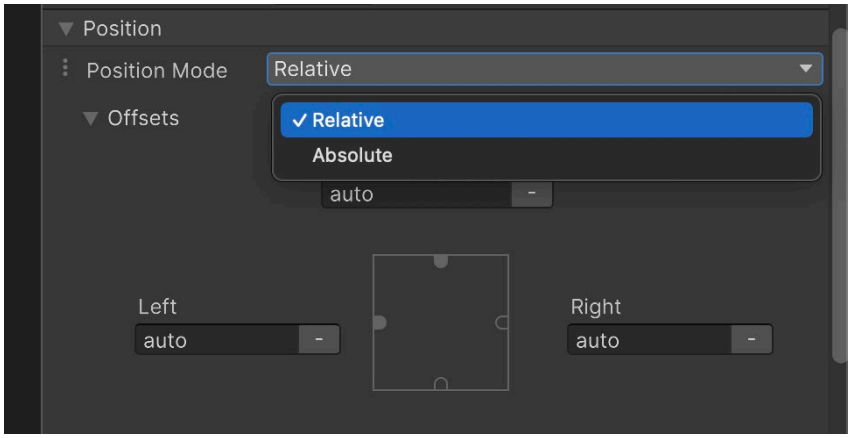


お勧めは、詳細なモックアップまたはワイヤーフレームを用意し (左)、UI Toolkit でデザインを再現するために要素を特定してブロック分けする (右) ことです。

UI Builder には、ビジュアル要素のために次の 2 つの配置 オプションがあります。

- **相対配置**:新しいビジュアル要素のデフォルト設定です。子要素は親コンテナの Flexbox のルールに従います。例えば、親要素の **Direction** が **Row**に設定されている場合、子のビジュアル要素は左から右へと配置されます。相対配置では、以下に基づいて要素が動的にサイズ変更および移動します。
    - **親要素のサイズまたはスタイルのルール**:**Padding** または **Align** > **Justify Content** で親要素の設定を変更した場合、その子要素はそれらの変更に従って自動的に調整されます。
    - **子要素独自のサイズとスタイルのルール**:子のビジュアル要素に独自の最小サイズまたは最大サイズが設定されている場合、レイアウトエンジンでもそれらの設定に従おうとします。

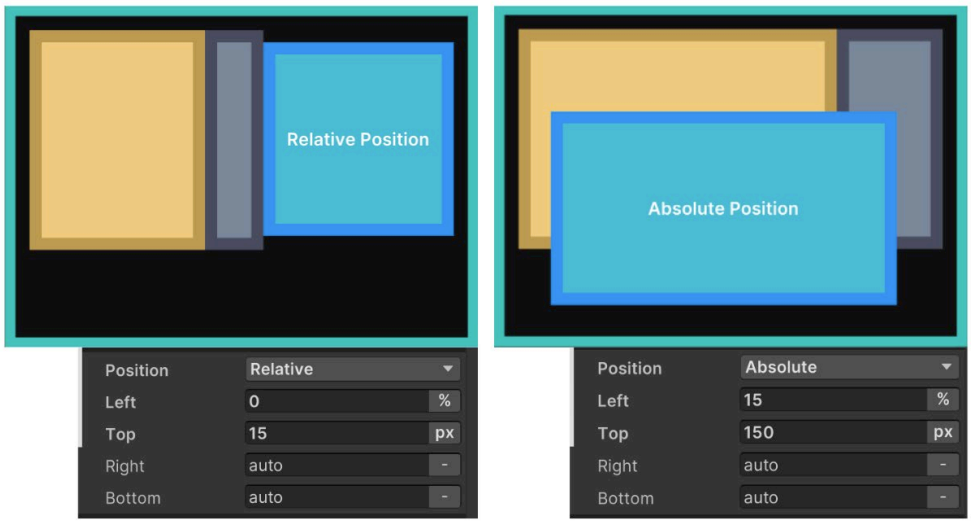
UI Toolkit は親と子の要素の間のあらゆる競合を処理します (例えば、子要素に親コンテナの幅よりも大きい最小幅が設定されていると、オーバーフローが発生します)。



任意のビジュアル要素で利用可能な配置モード

— **絶対配置**:ビジュアル要素の位置が親コンテナに固定されます。Unity UI のキャンバスでの動作に似ています。サイズルールや子要素に影響するルールは引き続き適用されますが、要素自体は **Grow**、**Shrink**、**Margins** などの Flex 設定を無視し、親コンテナの上にオーバーレイされます。

絶対配置の要素は、**Left**、**Top**、**Right**、**Bottom** の設定をアンカーとして使用します。例えば、Right と Bottom の値をゼロにすると、ボタンが親コンテナの右下に固定されます。



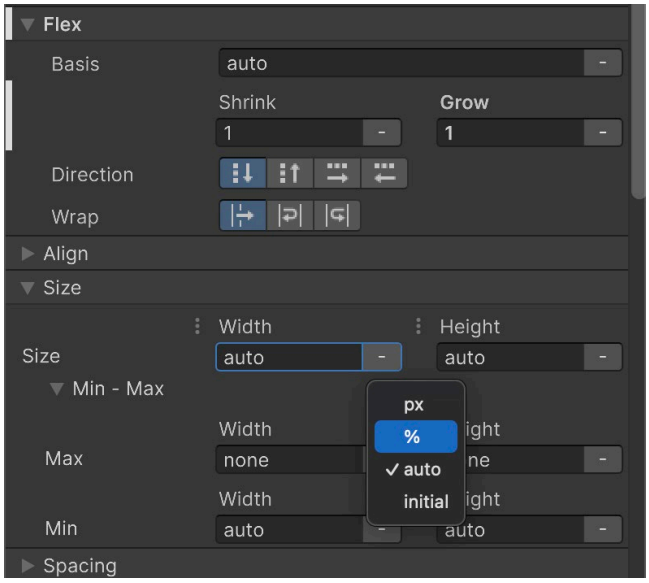
左では、青いビジュアル要素が相対配置に設定されており、親要素が Flex の設定として**Direction:Row** を使用しています。右は、青いビジュアル要素が絶対配置を使用し、親要素の Flexbox ルールを無視します。

永続的に表示される要素や、グループ化が入り組んでいる要素、多数の要素が含まれる要素の場合は、おそらく相対配置を使用することになるでしょう。

絶対配置は、一時的な UI (ポップアップウィンドウなど)、レイアウト構成に干渉しない装飾要素、ゲーム内の他の要素の位置に従う要素 (キャラクターの体力ゲージなど) に便利です。

Size の設定

ビジュアル要素は単なるコンテナにすぎません。Unity 6では、デフォルトの Grow 設定は 1 です。これは、コンテナ内の使用可能なすべてのスペースを使用することを意味します。それ以外の場合、特定のサイズにすでに設定されている他の子要素で埋まっていたり、具体的な **Width** や **Height** に設定したりしていない限り、空きスペースを埋めてしまうことはありません。



ビジュアル要素のサイズ設定

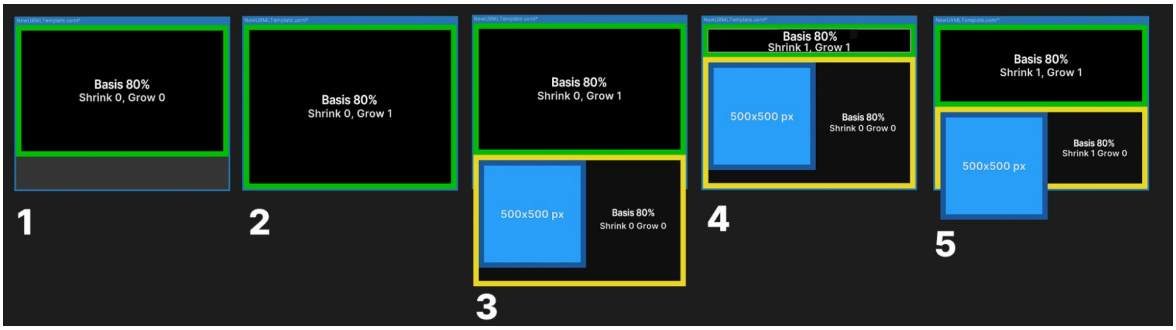
Width フィールド と Height フィールド で、要素のサイズを定義します。**Max Width** と **Max Height** で、拡大できる範囲に上限を設けます。同様に、**Min Width** と **Min Height** は、縮小できる範囲に下限を設けます。デフォルトの自動設定をオーバーライドして、ピクセル単位またはパーセント単位でサイズを定義できます。これらは、Flex の設定 (下) が空きスペースに基づいて要素のサイズをどれだけ変更できるかに影響します。

Flex の設定

Flex の設定は、相対配置を使用しているときに、要素のサイズに影響を及ぼすことがあります。まずその動作について理解するために、さまざまな要素を使用して実験することをお勧めします。

Basis は、Grow または Shrink に設定された割合に基づく処理が発生する前の、アイテムのデフォルトの Width と Height を指します。

- Grow が 1 に設定された場合、親要素内の空いている垂直方向または水平方向のすべてのスペースがこの要素で埋まります。0.5 に設定すると、使用可能なスペースの半分が使用されます。
- Grow が 0 に設定された場合、その要素は現在の Basis (サイズ) を超えては拡大しません。
- Shrink が 1 に設定された場合、親要素の空いているスペースに収まるように必要な分だけその要素が縮小されます。
- Shrink が 0 に設定された場合、その要素は縮小されず、必要な場合はオーバーフローします。



Basis、Grow、および Shrink の設定

上の例は、Basis が Grow オプションや Shrink オプションとどのように連動するかを示しています。

1. Basis が 80% の緑色の要素は、空いているスペースの 80 パーセントを占めます。
2. Grow を 1 に設定すると、その緑色の要素がスペース全体に拡大します。
3. 黄色の要素を追加すると、その要素がスペースをオーバーフローします。緑色の要素がスペースの 80 パーセントを占める状態に戻ります。
4. Shrink を 1 に設定すると、緑色の要素が黄色の要素に合わせて縮小されます。
5. ここでは、両方の要素の Shrink 値が 1 です。利用可能なスペースに収まるように等しく縮小されます。

ご覧のように、ピクセル単位でサイズが固定された要素 (3 から 5 の青色のボックス) は、Basis、Grow、Shrink の設定に反応しません。

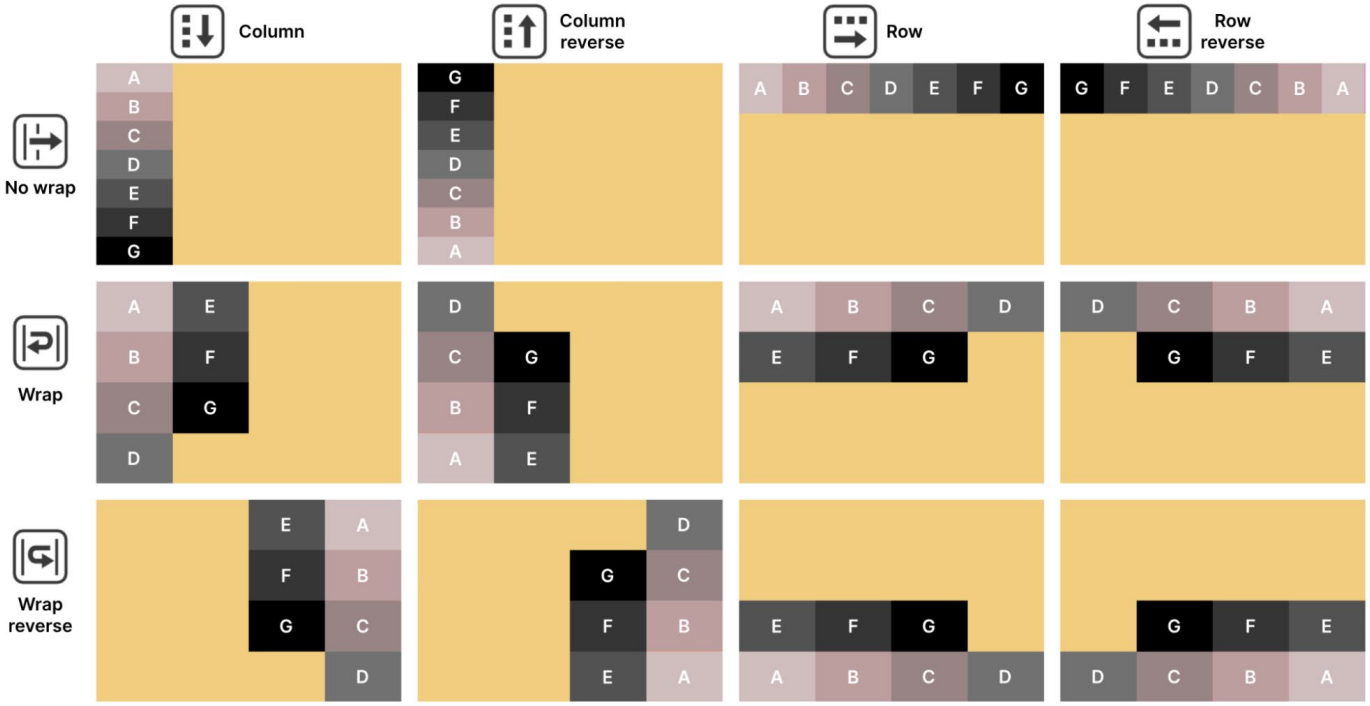
ヒント: ビジュアル要素のサイズの計算

レイアウトエンジンは、Size と Flexbox の設定を組み合わせて、相対配置を使用した場合に、各要素がどれほど大きく表示されるかを決定します。ビジュアル要素のサイズを計算する際に、以下のステップが実行されます。

1. レイアウトシステムが、Width と Height のプロパティに基づいて要素のサイズを計算します。
2. レイアウトエンジンが親コンテナ内に追加のスペースがあるか、対象の子要素が利用可能なスペースをすでに超えていないかを確認します。
3. 追加のスペースがある場合、レイアウトシステムは Flex/Grow 設定の値がゼロ以外である要素を探し、そのファクターに応じて追加のスペースを分配し、子要素を拡大します。
4. 子要素が利用可能なスペースを超えている場合、Flex/Shrink の値がゼロ以外である要素は適度に縮小されます。
5. そして、要素の最終的なサイズに影響を与える他のプロパティ (Min-Width、Flex-Basis など) も考慮されます。
6. レイアウトエンジンは、完成形となる、解決後のサイズを適用します。

Direction 設定は、子要素が親要素の内部でどのように配置されるかを定義します。Hierarchy メニューで高い位置にある子要素が最初に表示されます。Hierarchy の最後にある要素は最後に表示されます。

Wrap 設定で、複数の要素を 1 行または 1 列に収める (No Wrap) かどうかをレイアウトシステムに指示します。そうしない場合は、次の行または列に表示されます (Wrap または Wrap reverse) 。



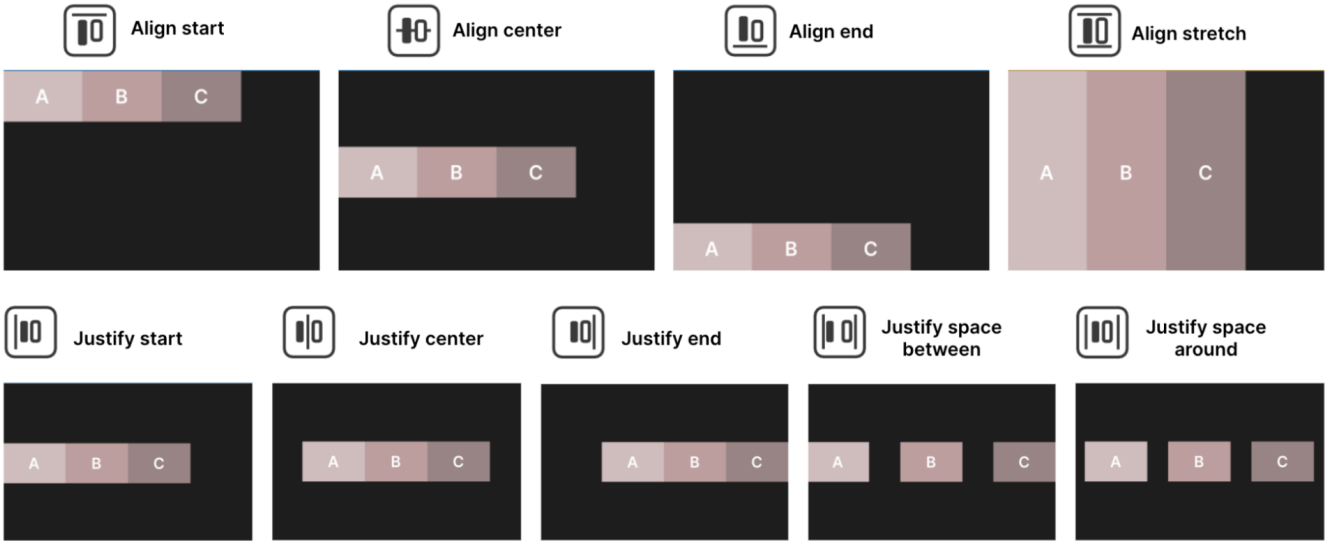
相対配置での Direction と Wrap のさまざまな組み合わせにおける UI Builder での親および子のビジュアル要素

Align の設定

Align の設定は、親要素に対して子要素がどのように整列するかを決めるものです。親要素で **Align > Align Items** を設定すると、子要素が開始位置揃え、中央揃え、終了位置揃えで整列します。ここでの選択は交差軸 (**Flex > Direction** の行または列と垂直) に影響します。

Stretch オプションも交差軸に沿って機能しますが、Size の Min 値と Max 値によって効果が制限される場合があります (これがデフォルトです)。一方で、**Auto** オプションは、レイアウトエンジンが他のパラメーターに基づいて他のオプションのいずれかを自動的に選択する可能性があることを示します。レイアウトをより細かくコントロールするためにはユーザーがオプションのどれかを選択し、Auto オプションは主に特殊なユースケースで使用することをお勧めします。

Align > Justify Content に移動して、レイアウトエンジンが親要素の中で子要素をどれくらいの間隔を空けて並べるかを定義します。要素を相互に隣接して並べることも、使用できるスペースを使用して広げる形で並べることも可能です。**Flex > Grow** と **Flex > Shrink** の設定がレイアウト結果に影響します。

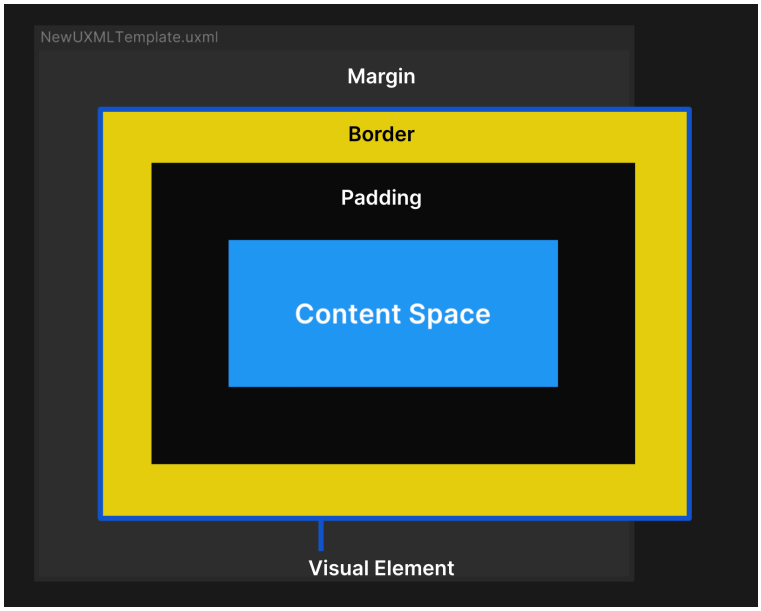


Direction を Row に設定した親要素に適用された Align 設定と Justify 設定。その他の配置とサイズのオプションが最終出力に影響する可能性があることに注意してください

Align Self オプションを使用すると、コンテナをフレックスレイアウトの中心、終了位置、または開始位置に揃えることができます。

Margin & Padding

Margin & Padding の設定で、ビジュアル要素とそのコンテンツの周囲の空きスペースを定義します。Unity では、下図のような標準の CSS ボックスモデルのバリエーションを使用します。



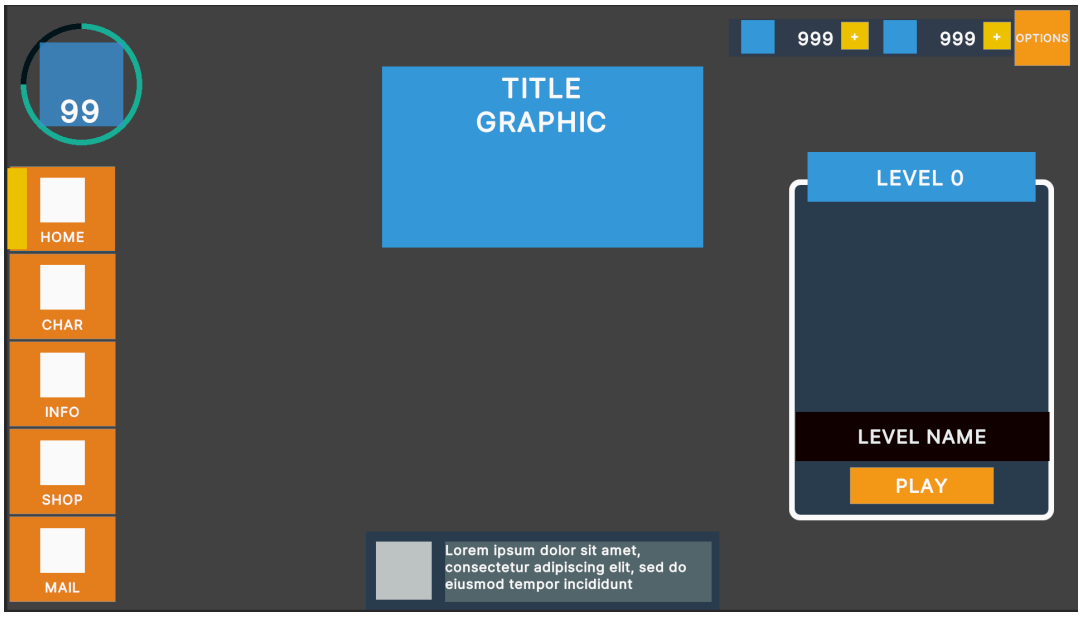
Size、Margin、Border、Padding の設定が定義された UI Builder のビジュアル要素。Width または Height が固定された要素がスペースに収まりきらない場合があります。

- **Content Space** には主なビジュアル要素 (テキスト、画像、コントロールなど) が保持されます。
- **Padding** は、Content Space の周囲の空いた領域 (ただし Border の内側) を定義します。
- **Border** は、Padding と Margin の間の境界を定義します。これには、色を付けたり、丸みを付けたりすることもできます。厚みが指定されている場合、その境界線は内側へ拡張します。
- **Margin** は Padding に似ていますが、Border の外側の領域を定義します。絶対配置の要素の場合、マージン設定は効果がありませんが、Position 設定を使用してアンカーポイントに関連付けて外部スペースを追加できます。

背景と画像

UI Toolkit では、あらゆるビジュアル要素を使用して画面上の画像を表示できます。背景プロパティを設定してテクスチャやスプライトを表示するだけです。

色で塗りつぶしたり画像を使用したりして、その要素の外観を変更することができます。これは、ワイヤーフレームを作成するのに便利です。対照的な明るい色を使用すると、隣り合った要素がどれくらい異なって見え、コンテナ内の変化に応答するかを把握することができます。



ワイヤーフレーム作成時には対比色を使用します。

変数または固定された測定単位

UI Builder には、要素の距離とサイズを定義する以下の 4 つのパラメーターがあります。

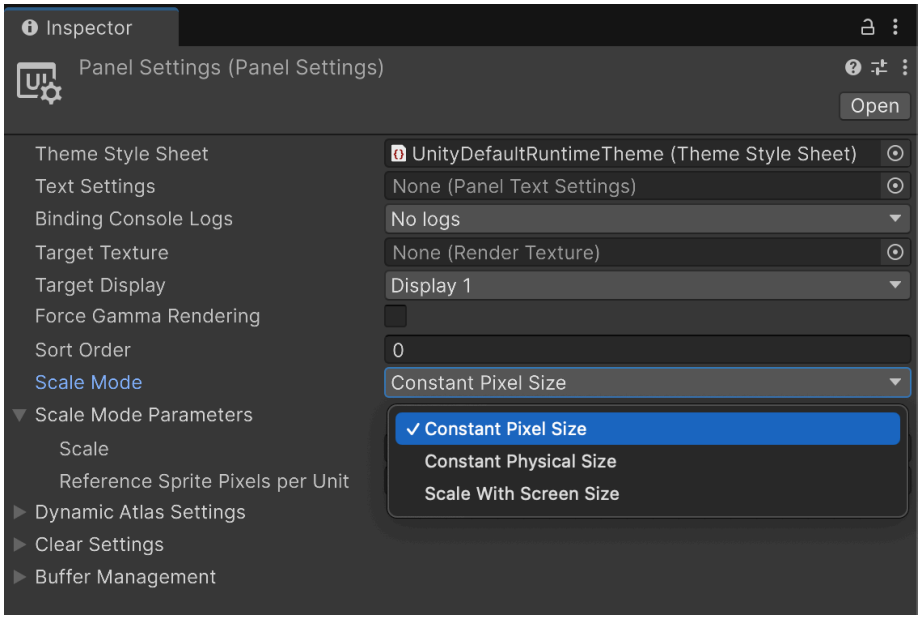
- **Auto**:サイズと位置のデフォルトのオプションです。レイアウトシステムは、親要素と子要素の両方の情報に基づいて、要素の値を計算します。
- **Percentage**:単位は要素のコンテナの割合と等しく、親の Width と Height によって動的に変化します。パーセンテージを使用すると、複数のフォーマットサイズを扱うときにスケーラビリティを確保できます。
- **Pixels**:このオプションは、要素のサイズを固定したい場合、例えば小さな要素をピクセル単位で最小サイズにし、常に読み取り可能にしたい場合に便利です。
- **Initial**:現在のスタイル設定値を無視し、プロパティをデフォルトの状態 (Unity のデフォルトのスタイルルール) に戻すオプションです。

デフォルトのサイズ設定と、ピクセルやパーセンテージで定義されたサイズの例

スケーリングルールを UI 全体に同時に適用する場合は、Panel Settings の **Scale Mode** パラメーターを使用します。

- **Constant Pixel Size:**このスケールモードでは、要素は画面サイズの影響を受けず、固定ピクセルサイズを維持します。スケールファクターを適用して要素サイズを拡大できます。
- **Constant Physical Size:**このモードでは、要素は複数の画面で同じ物理サイズを維持します。Reference DPI に基づいて UI がスケーリングされ、実際の画面 DPI が異なる場合はサイズが調整されます。

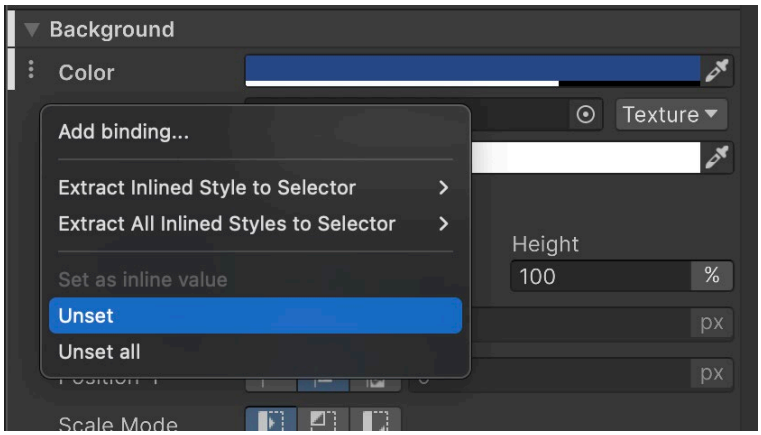
— **Scale with Screen Size:**このオプションにより、解像度に基づいて要素が動的にサイズ変更されます。**Screen Match Mode** では、スケーリングで幅、高さ、またはその組み合わせのどれを優先するかを指定します。**Reference Resolution** では、UI の基本サイズを設定します。Screen Match Mode が **Match Width or Height** に設定されている場合、UI システムが画面幅、画面高さ、またはその組み合わせに UI をスケーリングするかどうかが、**Match** 値によって制御されます。



UI Toolkit のパネル設定には、Unity UI のものと似た拡大縮小オプションがあります。

UI Builder でオーバーライドされたプロパティ

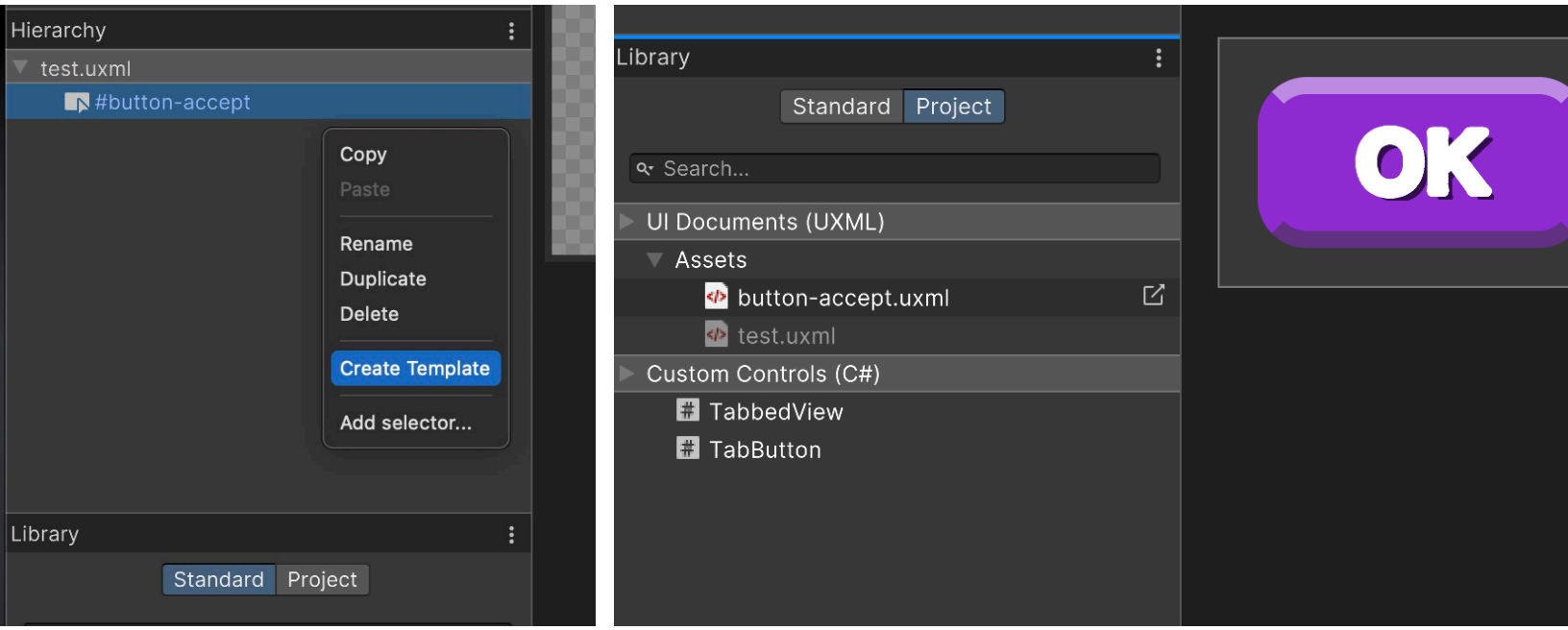
変更されたプロパティは、下のスクリーンショットに示すように、Inspector で太字とその左の白線により強調表示されます。これは、デフォルト値、または選択した UXML ファイルのスタイルシート (USS) のセレクターで設定した値をオーバーライドしていることを示します。この動作は "インラインスタイリング" とも呼ばれます。値を変更する必要がない場合は、変更を見つけやすく、管理しやすくするために、デフォルトの状態のままにしておくのが最善です。プロパティをデフォルト値にリセットするには、プロパティセクションの横にある縦ドット (:) メニューにあるオプションを使用できます。



このメニューから、変更した値をデフォルトの値またはセレクターの元の値に復元できます。

テンプレートとしての UXML

UXML ファイルはプレハブと同じように使用できます。例えば、インベントリ内で何度もスポーンする必要があるアイテムアイコンとカウント番号を含む UXML レイアウトのプロジェクトがあるとします。UXML を右クリックすると、テンプレートを作成するオプションが表示されます。テンプレートは、後で Hierarchy ペインのほかのビジュアル要素に加えたり、コードからインスタンス化したりできます。作成されたファイルは、Library と Project ウィンドウで確認できます。



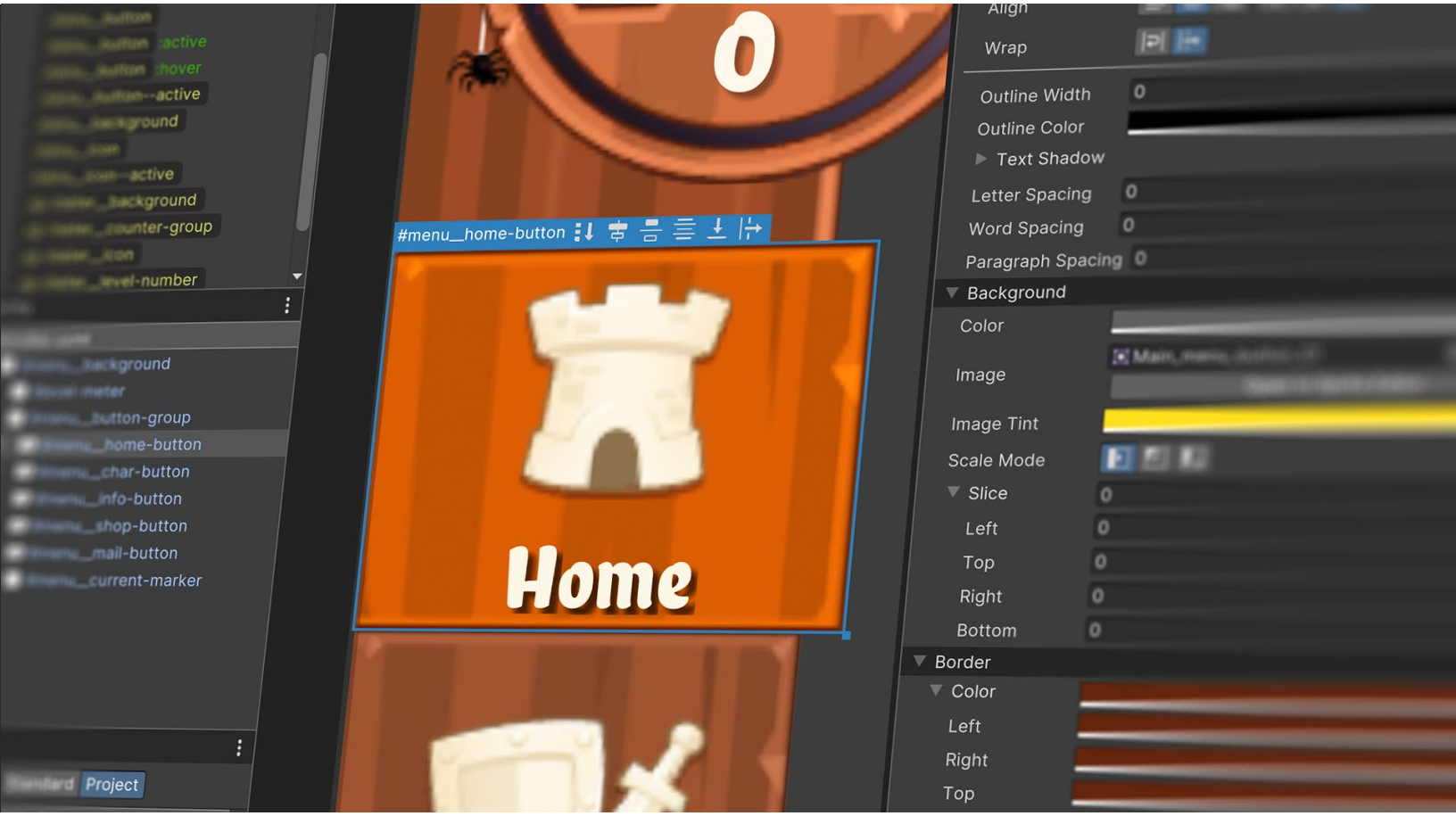
テンプレートは再利用可能な UXML であり、Library ペインの Project タブで利用できます。

その他のリソース

Flexbox レイアウトエンジンの詳細については、以下のリソースを参照してください。Flexbox と Yoga はウェブやアプリケーションの開発において標準となっているため、さまざまなリソースがオンラインで提供されています。

- ランタイムで UI Toolkit を使う:詳細を確認
- Yoga 公式ドキュメント
- CSS-Tricks の Flexbox ガイド

スタイリング



UI Builder でスタイルのプロパティを直接変更します

ビジュアル要素を含むワイヤーフレームレイアウトのモックアップを作成したら、スタイリングしたり、フォーマットのプロパティを再利用可能なスタイルに保存したりできます。スタイリングは、UI Toolkit の真の力が発揮される部分です。

ビジュアル要素にスタイルを追加するには、Unity スタイルシート (USS) ファイル (**Assets > Create > UI Toolkit > StyleSheet**) を使用するのが望ましい方法です。これはウェブの CSS ファイルに相当する Unity 用ファイルで、同様のルールベースの形式が使用されます。また、デザインプロセスの柔軟性を高めることで再利用が容易になり、大規模なプロジェクト全体にわたりスタイルの一貫性が保たれます。

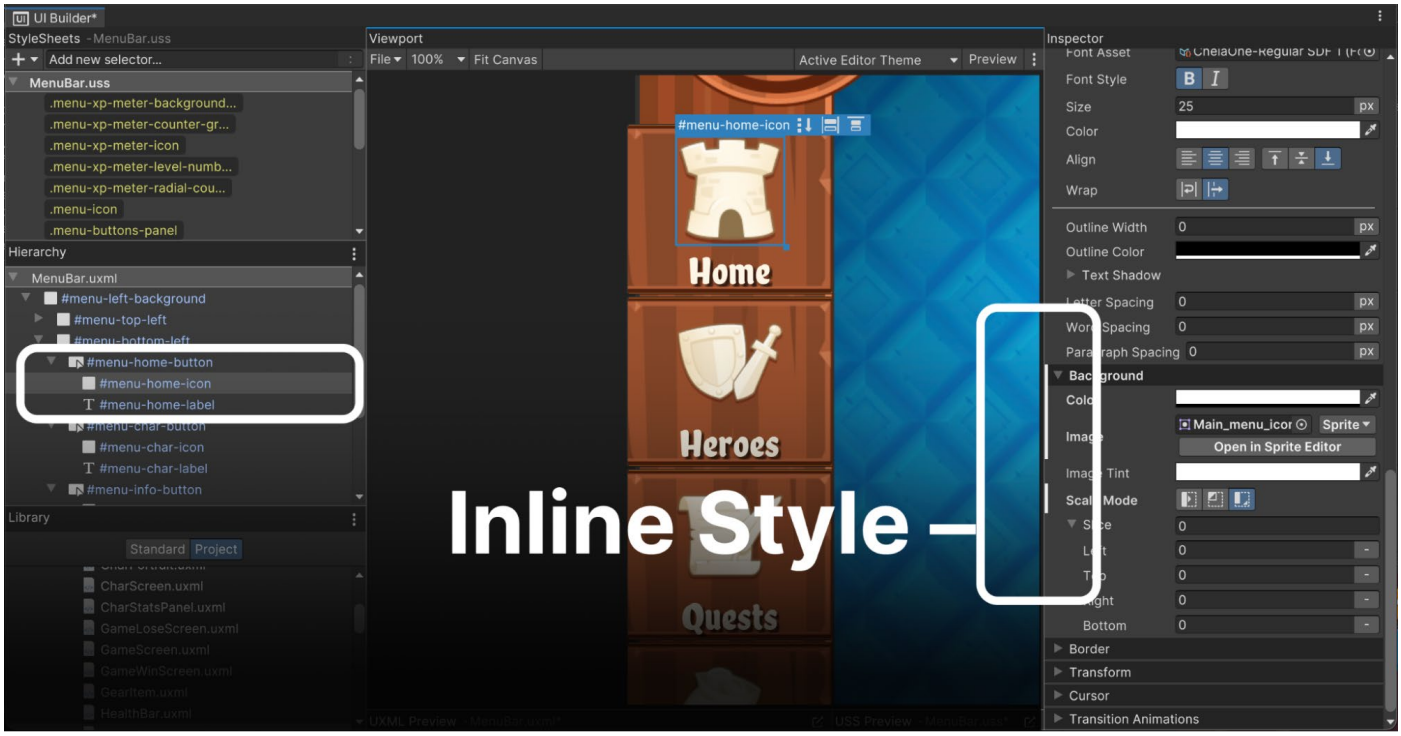
USS ファイルで、要素のサイズ、色、フォント、間隔、境界線、位置などを定義します。

USS セレクター

USS ファイルをまだ作成していない場合、スタイルの変更はすべてインラインスタイルとして UXML アセットに直接埋め込まれます。これらのインラインスタイルは、設定されている特定のビジュアル要素の外観に影響を与えますが、プロジェクトで再利用することはできません。

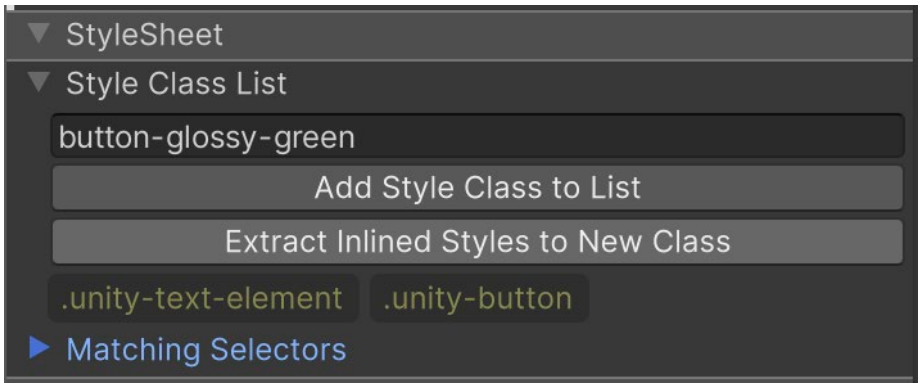
例えば、プロジェクトに何百ものボタンがある場合、個々のボタンのスタイルを更新するのは時間がかかり、非効率です。代わりに、USS でセレクターを定義できます。USS セレクター を使用すると、UI Builder のスタイルシートで UXML アセットの多くの要素にわたってスタイルを共有および適用できます。

既存のインラインスタイルをセレクターに変換



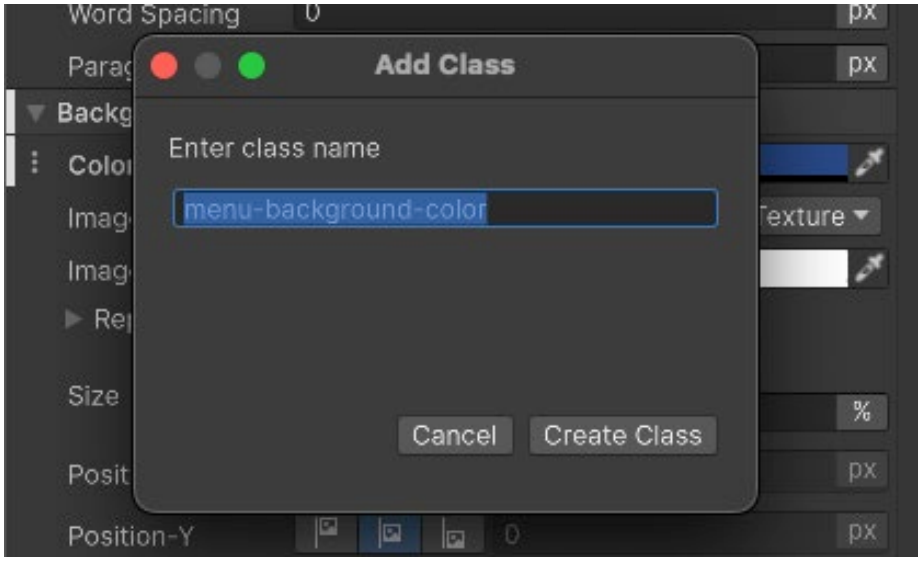
インラインスタイルはオーバーライドです。

Add Style Class to List ボタンを使用して、要素のインラインスタイルをすべて入力し、セレクターに変換します (黄色で "." で始まる)。このセレクターで、スタイリングプロパティが集中管理されるようになり、プロジェクト全体で一貫したスタイルを他のボタン (や他の要素) に適用できます。セレクターに対して行われた更新は、関連するすべての要素に自動的に反映されるので、プロセスがスケーラブルでメンテナンス性の高いものになります。



インラインスタイルのプロパティをすべてセレクターに抽出します

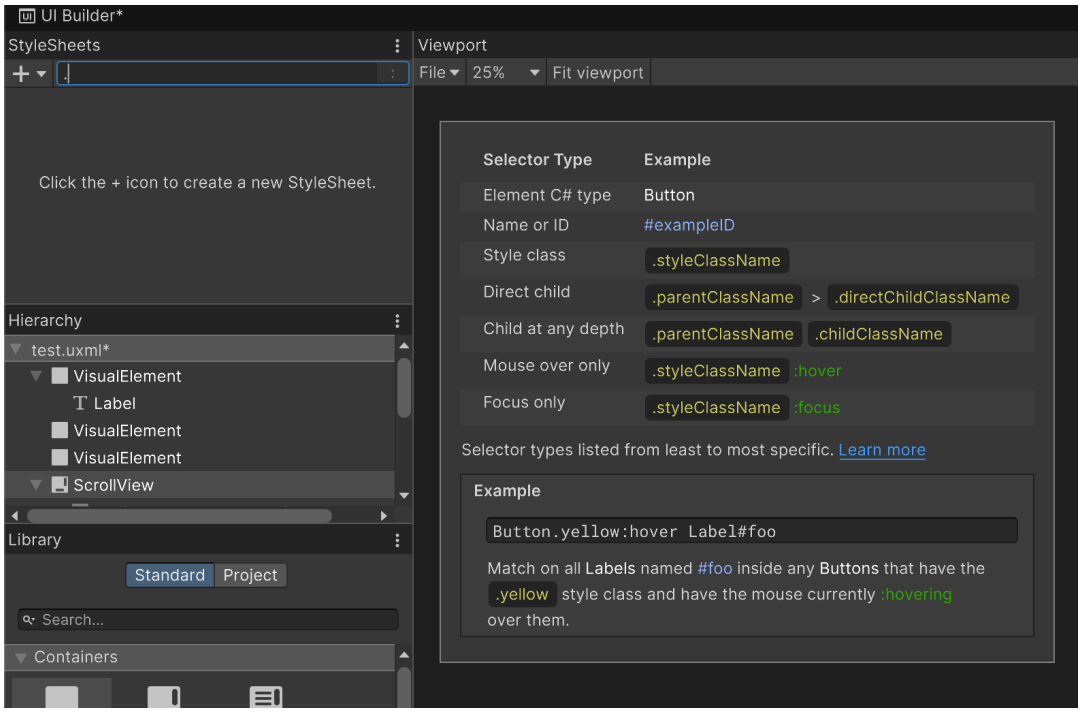
特定のインラインスタイルを新しいセレクターに抽出するには、プロパティの横にある縦ドット **(⋮)** をクリックし、**Extract Inlined Style to Selector / Add Class** を選択して、そのプロパティをセレクターに変換します。



プロパティのインラインスタイルをセレクターに抽出します

新しいセレクターの作成

セレクターは、指定された検索条件に一致する要素をビジュアルツリーから照会します。その後、UI Toolkit によって、一致するすべての要素にスタイルの変更が適用されます。新しいセレクターを追加するには、UI Builder の左上にある **Add new selector…** フィールドをクリックします。



新しいセレクター作成時の USS セレクターの参照

USS セレクターでは、次の方法でビジュアル要素を照合できます。

— **Element C# type**:これらのセレクターはタイプ (ボタン、ラベル、ListView など) ごとに機能します。セレクターは、Library パネルで使用可能なデフォルトのタイプ名と照合します。名前の前に特殊文字は使用しません。クラスセレクターは白色で表示されます。例えば、**Button は Button** タイプのすべての要素にスタイルを適用します

— **Name or ID**: これらのセレクターは、同じ名前のすべての要素にスタイルを適用できます。名前セレクターは先頭に ハッシュ "**#**" 記号が付き、青で表示されます。例えば、**#title** と指定すると、Hierarchy 内の**title** という*名前のすべての要素にそのスタイルが適用されます。*

注: UXML の名前属性は、HTML ID とは異なり、一意である必要はありません。これは、UI Toolkit が UXML テンプレートと再利用可能なコンポーネントをサポートしており、複数の要素で同じ名前とスタイルを共有できるためです。

— **Style class**: Style class セレクターは再利用可能なスタイルで、対応するクラス名を要素の Class List プロパティに追加することで、任意のビジュアル要素に適用できます。Style class セレクターは先頭にドット "**.**" が付き、黄色で表示されます。例えば、**.smallFont** を使用して Class List に smallFont を追加することで、任意の要素に特定のスタイルを適用できます。

— **Direct child:**照合基準の後に **>** を追加すると、その > 記号の後に書かれた 2 つ目の基準に一致する Direct child 要素だけが影響を受けます。例えば、セレクター **#title > Label** は、名前 **#title** の要素内にある任意の **Label** タイプにスタイルを適用します。その親の外部の Label や深い階層にある Label は影響を受けません。

— **Child at any depth:**これは前のセレクターと同じですが、この場合は 2 つ目の照合基準が親階層内の深さに関係なくすべての子に適用されます。

注: なるべくセレクターの範囲を広くしすぎないようにしてください (特に、* で終わるセレクターや、`.unity-button` などの汎用 Unity クラスを対象とするセレクター)。ビジュアルツリーの広い範囲を照会する場合、深い階層の子セレクターによってパフォーマンスが低下する可能性があります。

— **Pseudo-class:** Pseudo-classes (疑似クラス) を使用すると、ビジュアル要素がマウスオーバーされたときやフォーカスされたときなど、状態が変わるときにビジュアル要素に異なるスタイルを定義できます。疑似クラスはコロン **":"** で表され、これにより既存のセレクターが修正されます。

例えば、"ボタン **:focus**" セレクターは、フォーカスがあるすべてのボタン要素に特定のスタイルを適用します。そのため、疑似クラスはホバーエフェクトやフォーカスインジケーターなどの視覚的なフィードバックを追加するのに役立ちます。さらに、疑似クラスを USS アニメーションと組み合わせることで、滑らかなモーションと動的な遷移を導入できるので、ユーザー体験を強化できます。

使用可能な疑似クラスについては、こちらを参照してください。

ビジュアル要素が複数の セレクターに一致する場合、最も詳細度の高いセレクターが優先されます。

USS の詳細度の優先順位は次のとおりです。

1. **インラインスタイル**:要素に (UXML やコードなどにより) 直接適用されたスタイルが最も優先順位が高く、すべての USS セレクターをオーバーライドします。
2. **ID セレクター** (#id):最も具体的な USS セレクターで、一意の名前プロパティを持つ要素に適用されます。
3. **クラスセレクター** (.className):対応するクラスが **Class List** に追加された要素に適用されます。
4. **C# タイプセレクター** (ボタン、ラベルなど):指定したタイプのすべての要素に適用されます。

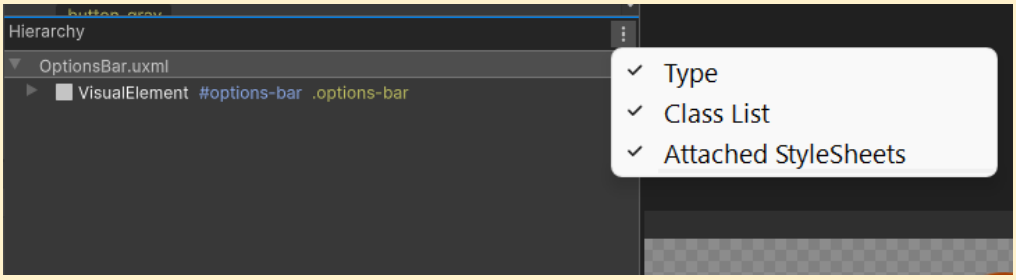
例えば、要素にインラインスタイルが設定され、かつ #title ID セレクターと一致する場合、インラインスタイルは ID セレクターをオーバーライドします。同様に、要素がクラスセレクター とタイプセレクターの両方に一致する場合は、クラスセレクターが優先されます。

複数のセレクターが同じプロパティをオーバーライドしようとしていて、その詳細度がすべて同じであれば、USS スタイルシートでの順序となり、リスト内で下にあるセレクターが優先されます。

セレクターの優先順位 については、ドキュメントで詳細を確認できます。

ヒント: Hierarchy に関する追加情報

Hierarchy ヘッダーの縦ドット (⋮) をクリックすると、UI 要素をさらに視覚化できます。



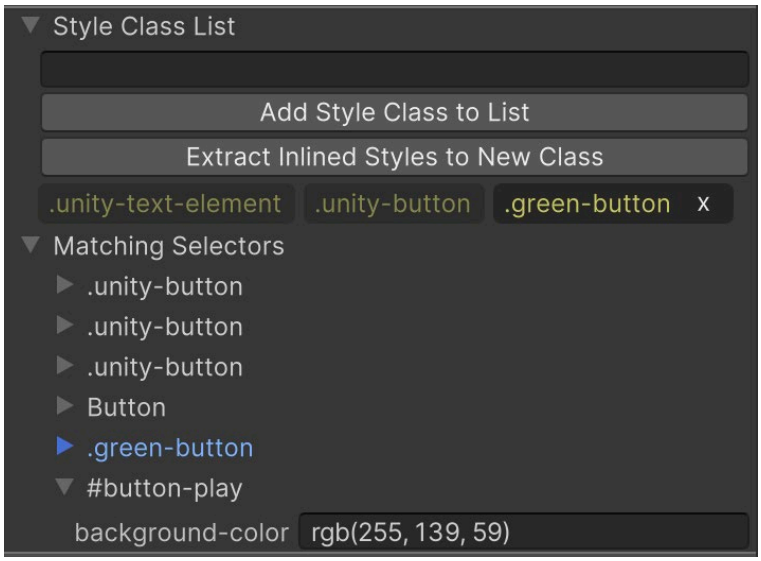
Hierarchy では、さまざまなセレクターでフィルタリングを行うことができます。

Hierarchy ペインでは、要素タイプの横に追加情報が表示されます。選択すると、**#options-bar** 名前セレクターと **options-bar** スタイルクラスセレクターが表示されます。

一部のセレクターには、プレフィックスとして **.unity-** が付いています。これらは、すべての要素に適用されるデフォルトのスタイルです。これらの値は、定義されたセレクターによってオーバーライドされます。

要素に割り当てられたセレクター

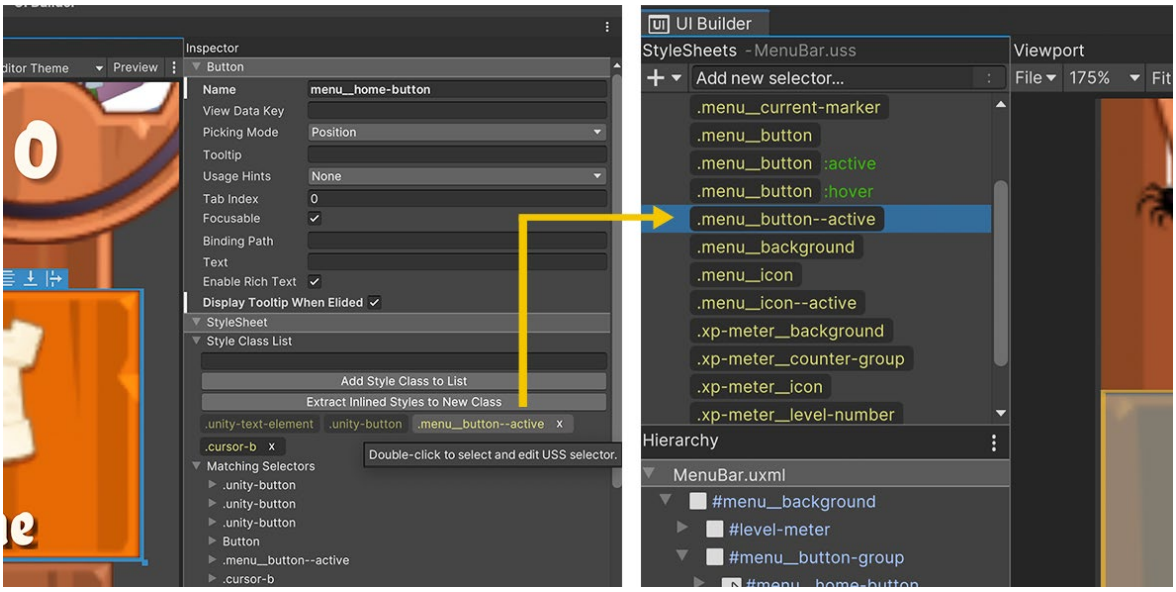
Inspector では、Hierarchy で選択した要素の一致するセレクターを視覚化することができます。リストの一番下にあるセレクターが優先されます。詳細を展開すると、どのスタイルパラメーターが変更されているかを確認できます。



Inspector では、選択されたビジュアル要素に一致したセレクターが表示されます。

セレクターの編集

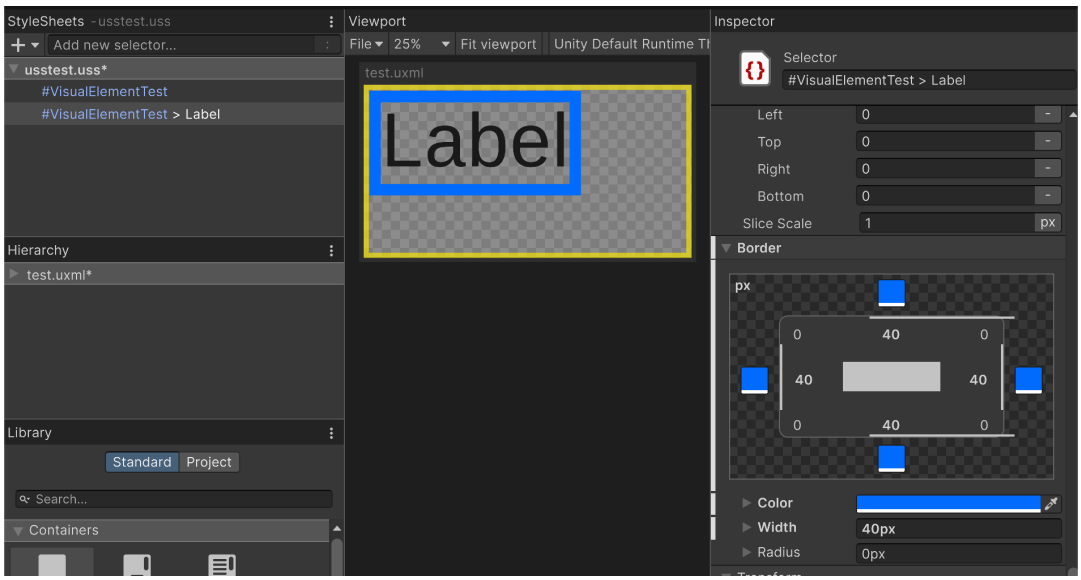
スタイルセレクターを変更する際は、Hierarchy のビジュアル要素ではなく、**Style Sheet** パネルで **Style Class** を選択してください。そうしないと、スタイルクラス自体ではなく、特定の要素のインラインスタイルが変更されます。



Inspector でスタイルクラスをダブルクリックして、スタイルクラスをアクティブにします。

要素の選択を解除し、代わりにスタイルセレクターを選択するには、Inspector でスタイルクラスをダブルクリックします。

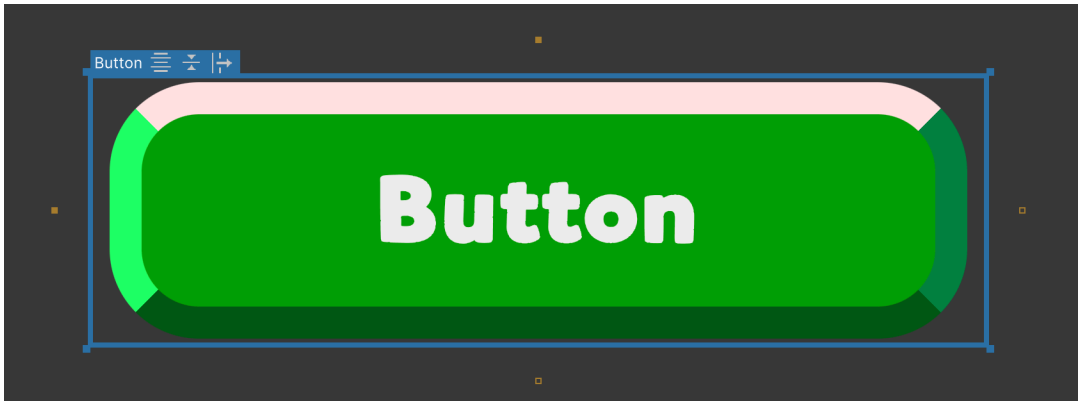
パラメーターを UXML でインラインスタイルとして直接変更したときと同様に、パラメーターを StyleSheets ペインで選択してセレクターで編集し、オーバーライドして変更できます。変更も太字で表示され、横に白線が表示されます。値の設定を解除するには、プロパティの横にある縦ドット (:) メニューから操作します。



USS セレクターの編集

さまざまなフォーマットオプションが用意されていて、要素やフォントの基本的な外観を変更することができます。UI Toolkit では、高度なスタイリング機能が提供されており、カスタムスプライトの必要性を減らすことができます。

UI Builder では、アウトライン、角丸、画像調整、境界線の色などを要素に簡単に追加できます。スタイリングでは、ベベル効果を含めたり、カーソル画像を変更できるようにすることもできます。



UI Toolkit には、追加のテクスチャを必要としないいくつかのスタイリング効果があります。

スタイルのオーバーライド

ルールは破られる宿命にあります。UI 要素にスタイルクラスを定義する際には、常に例外が存在します。

例えば、何百ものボタン要素があり、各ボタンに異なるアイコンがある場合、それぞれに新しいセレクターを作成する必要はありません。それではスタイルの再利用性 (利便性) が失われてしまいます。

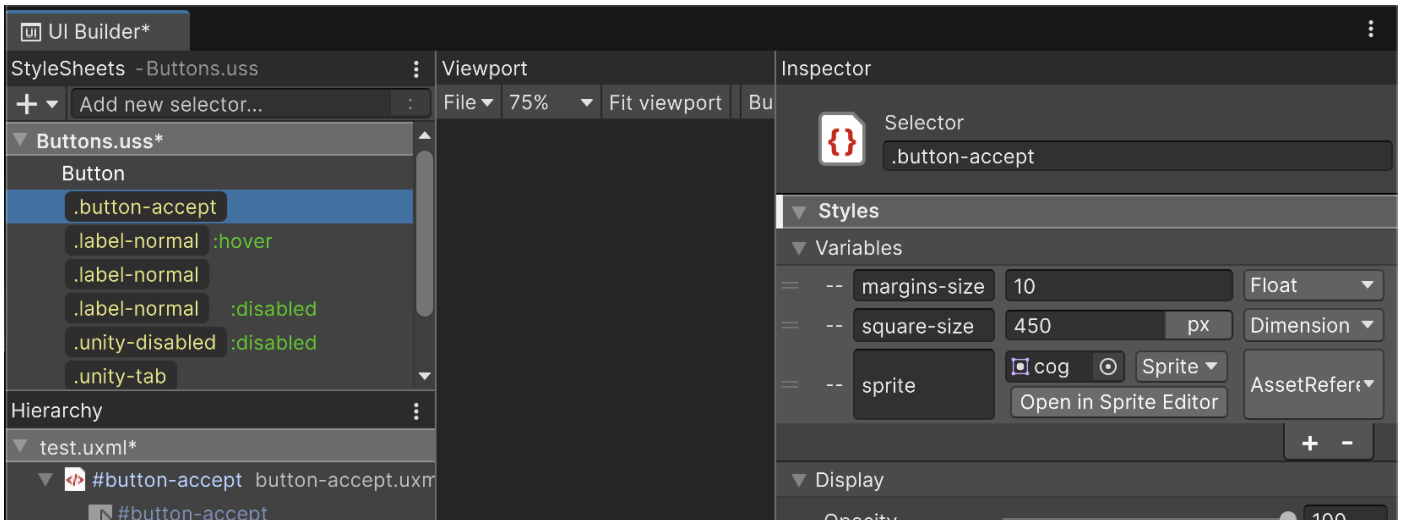
その代わりに、すべてのボタンに同じスタイルを適用し、それぞれのボタンごとに固有の部分をオーバーライドするのが合理的です (例えば、**Background > Image** をオーバーライドして、各ボタン要素に独自のアイコンを使用するなど)。これらのオーバーライドは、**インラインスタイル** プロパティと呼ばれます。

ヒント: インラインスタイルはセレクターより優先されます

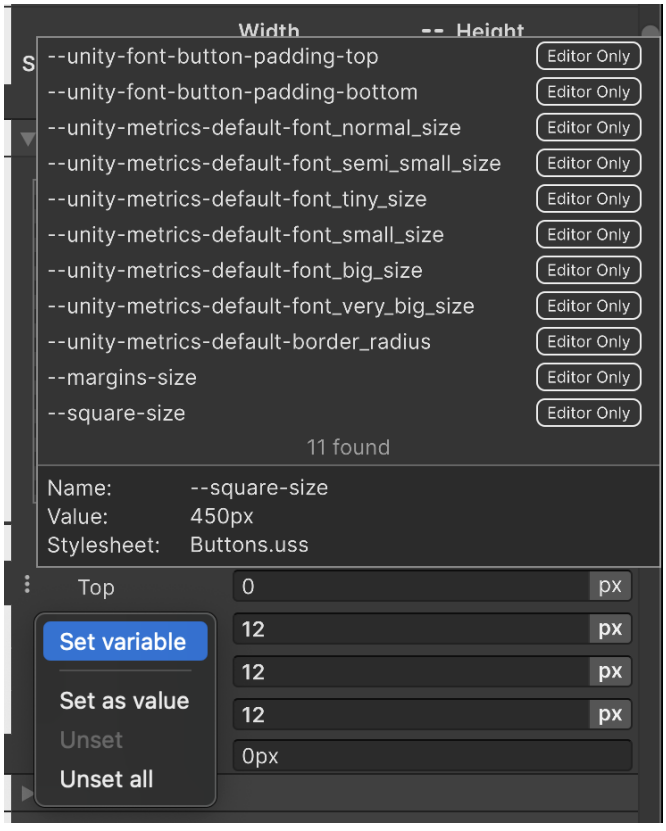
インラインスタイルは、常にセレクターより優先されます。したがって、セレクターが適用されているにも関わらずスタイルが更新されない理由がわからない場合は、要素を確認してオーバーライドが存在するかどうかを確認するとよいかもしれません。

USS 変数

USS 変数 を作成することで、異なるプロパティに同じ値を手動で設定する時間を節約できます。USS 変数を更新すると、その変数を使用するすべての USS プロパティが更新されます。Unity 6.1 では、これらの変数は UI Builder のエディターから設定することもできます。



USS セレクターの変数は Unity 6.1 の UI Builder で作成および編集できます



値を直接指定する代わりにプロパティに変数を設定します

Float、色、文字列、アセット参照 (背景画像用)、次元 (ピクセル、度、パーセンテージなど)、列挙型の変数を作成できます。変数にはセレクターレベルのスコープがあります。他のセレクターに存在する変数は使用できませんが、セレクター自体は必要な数の要素に適用できます。

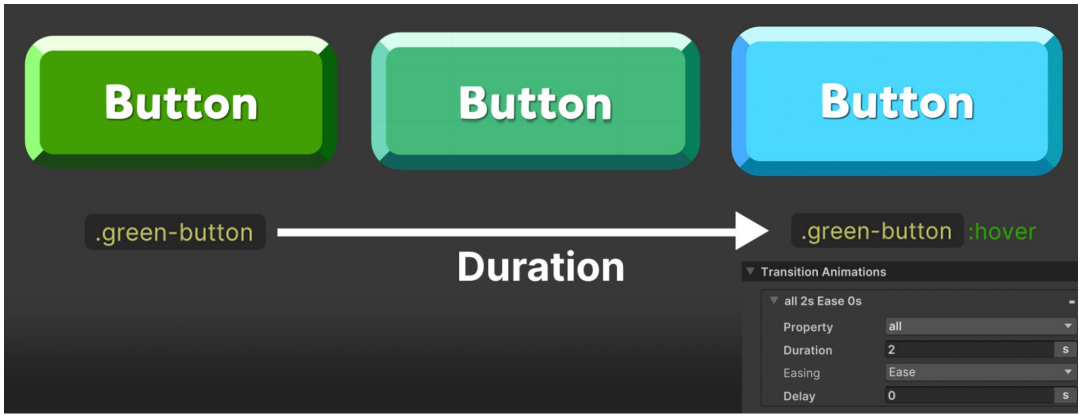
USS 遷移アニメーション

メニュー画面に遷移を加えることで、視覚的な洗練度とユーザー体験を大幅に高めることができます。UI Toolkit では、Inspector のTransition Animations プロパティを使用することで、比較的簡単に遷移を設定することができます。

アニメーションの Property、Duration、Easing、Delay を設定できます。一度設定すれば、関連するスタイルがランタイム中に変更されたときに、自動的に遷移が適用されます。

ボタンの疑似クラス間の遷移を考えてみましょう。クラスセレクター **.green-button** に対して疑似クラス **:hover** がある場合です。各スタイルには独自のサイズと色があります。

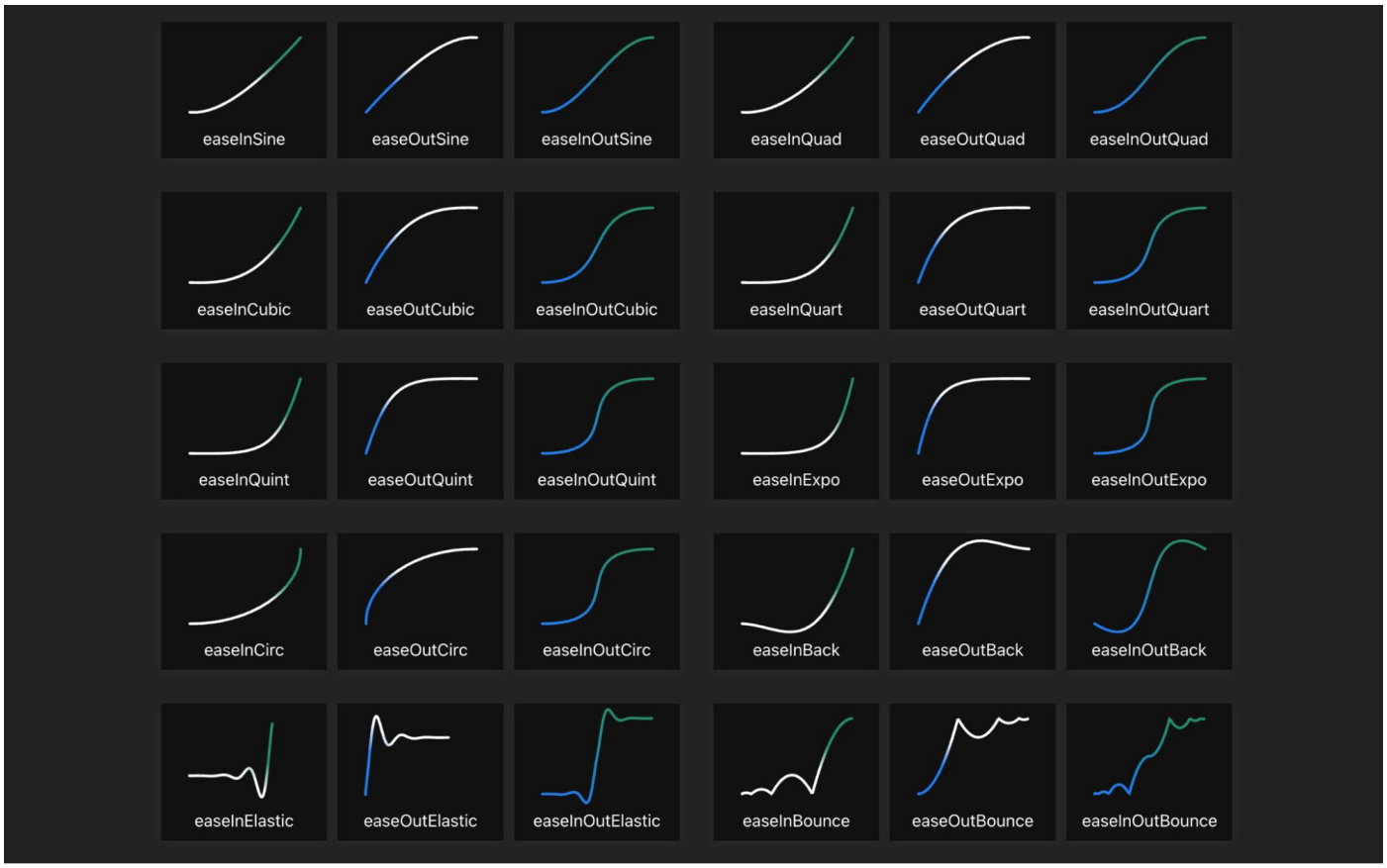
マウスのホバー状態での遷移を定義するには、**.green-button:hover** セレクターに、Inspector の下部にある Transition Animations を設定します。そうすると、ボタンがポインターの動きに合わせてアニメーション化されるようになります。



遷移のアニメーションを使用して、スタイル間を補間することができます。

遷移のアニメーションでは、スタイル間を補間するのに以下のオプションを使用できます。

- **Property**: どのプロパティを補間するかを決定します。デフォルト設定は **all** ですが、ドロップダウンリストから特定のプロパティを選択することもできます。上記の例では、**:hover** 状態は **Color プロパティ** と **Transform プロパティ** のみを変更します。アニメーション可能なプロパティの 完全なリストを参照ください。
- **Duration**:遷移の長さを秒またはミリ秒で表します。可視性を確保するには、Duration を 0 よりも大きな値に設定する必要があります。
- **Easing**:イージング機能は、アニメーションが時間とともにどのように進行するかを決定し、加速、減速、弾性などの自然な動きをシミュレートできます。イージング機能を使用することで、一定のスピードで動く基本的な線形補間と比較して、アニメーションの遷移が滑らかで自然に見えます。



利用可能な関数を視覚化したい場合は、こちらのチートシート を使用してください (視覚化の提供:https://easings.net/)。

- **Delay:**遷移を開始するまでの待機時間を、秒またはミリ秒で指定します。
- **Add Transition:**新しい状態の各プロパティを個別にアニメーション化できます。それぞれに異なる継続時間、遅延、イージング効果を指定できます。

  Add Transition ボタンをクリックすると、別の遷移のアニメーションを追加できます。これにより、オーバーラップした複数の遷移を同時にトリガーし、より機械的でない、自然な効果を得ることができます。

ヒント: 遷移イベント

アニメーション化されるビジュアル要素には、遷移イベント のコールバックを追加することができます。これにより、シーケンスやループなどの高度なワークフローをサポートすることができます。

以下は、一般的な遷移イベントと、それらがいつ送信されるかについての説明です。

- TransitionRunEvent:遷移が作成されたときに送信されます
- TransitionStartEvent:遷移の遅延フェーズが終了し、遷移が開始されると送信されます
- TransitionEndEvent:遷移の終了時に送信されます
- TransitionCancelEvent:遷移がキャンセルされたときに送信されます

USS 遷移の詳細については、ドキュメント を参照してください。

ビジュアル要素については、疑似クラス (:active、:inactive、:hover など) で独自のセレクターを設定できるので、その他のコードは必要ありません。擬似クラスによってスタイルの変更がトリガーされると、定義された遷移が変更を自動的にアニメーション化します。例: ホバー (:hover) やクリック (:active) でボタンが拡大縮小したり、ユーザーインタラクションやその他のイベントに応じて要素がフェードアウトしたり非表示になったりします。

擬似クラスは定義済みであり、独自に作成することはできません。

オンデマンドでのスタイルの切り替え

ゲーム内の他のイベントでも、UI 要素 API のメソッドを使用してコードのスタイルを変更できます。例えば、キャラクターのレア度に基づいて別のスタイルに変更するには、RemoveFromClassList メソッドと AddToClassList メソッドを使用できます。

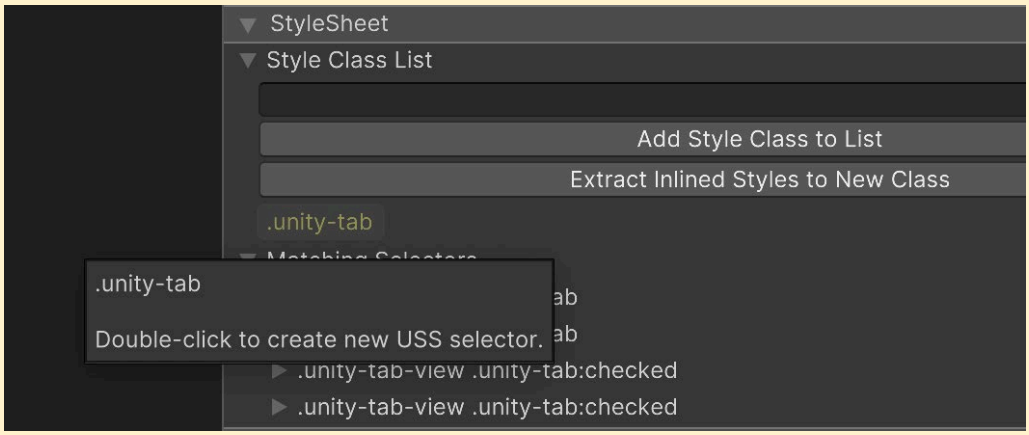
```
if (character.rarity == RarityType.Legendary)
{
   visualElement.RemoveFromClassList("common");
   visualElement.AddToClassList("legendary");
}
```

さらに、ビジュアル要素の有効状態に基づいて擬似クラス :active または :inactive をトリガーし、状態を変更したときに USS が遷移するように設定することもできます。そうすれば変更前と変更後の状態の表現ができるようになります。

*UI Toolkit サンプル –『Dragon Crashers』*のメニューバーボタンは PointerEventClick を使用して手動での遷移をトリガーします。

ヒント: Unity デフォルトセレクターのオーバーライド

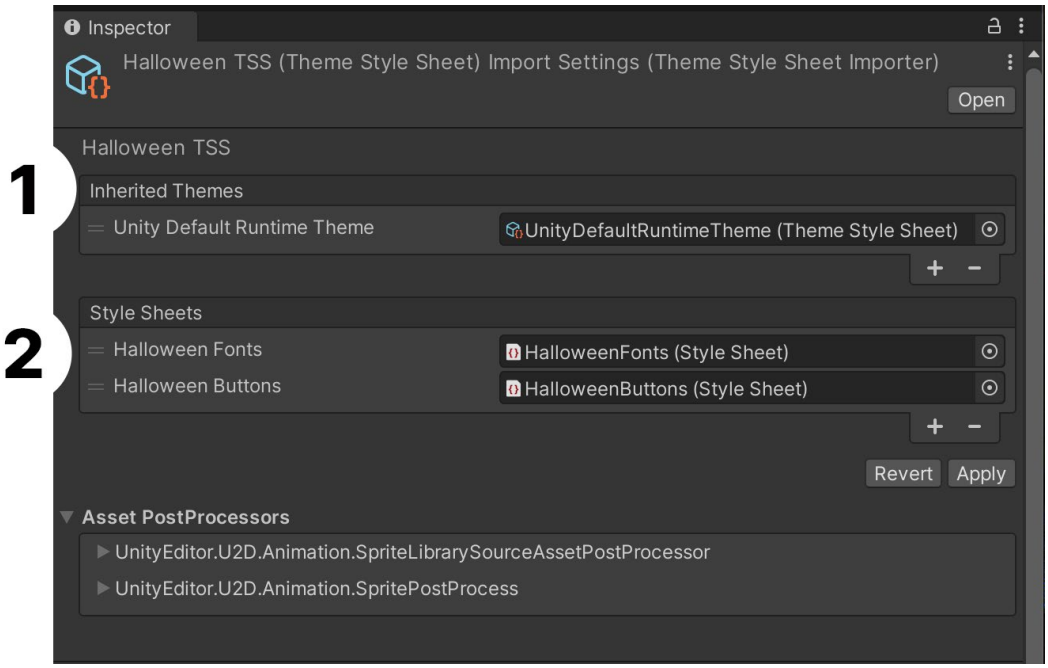
より複雑なビジュアル要素、例えばタブビューは、システムによって事前定義された子を持つ親要素で構成されます。これらの要素にコンテンツを追加すると、特定の動作をし、使用されているスタイルが無効化され、Unity で作成されたように見えます。使用中のセレクターをダブルクリックすることでデフォルトセレクターをオーバーライドし、コピーを作成してスタイルシートや USS で編集できます。



テーマ

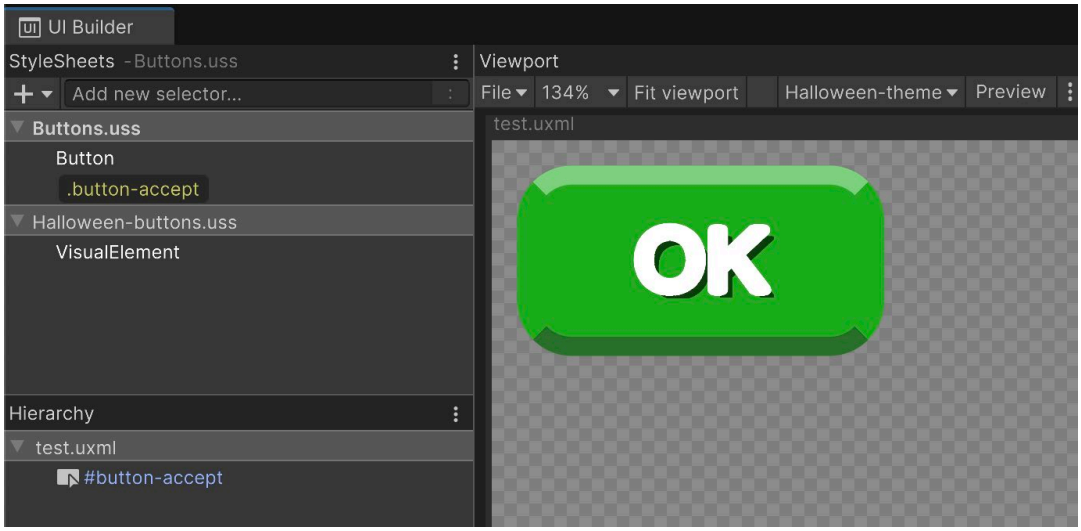
季節に合わせた UI バージョンを作成したい場合や、異なるカラースタイルを提供したりしたい場合には、Theme Style Sheets (TSS) を使用することでプロセスを簡素化できます。TSS を作成するには、**Create > UI Toolkit > TSS theme file の順に選択します。**

TSS ファイルは通常の USS ファイルと同様に機能するアセットファイルです。これは、USS セレクターだけでなく、プロパティや変数の設定も含んだ独自のカスタムテーマを定義するための出発点として利用できます。



ハロウィーンをテーマにしたこの UI 要素の例では、Halloween TSS はまず Unity Default Runtime TSS を継承し、次にテーマ固有の Fonts と Buttons のスタイルシートを追加しています。

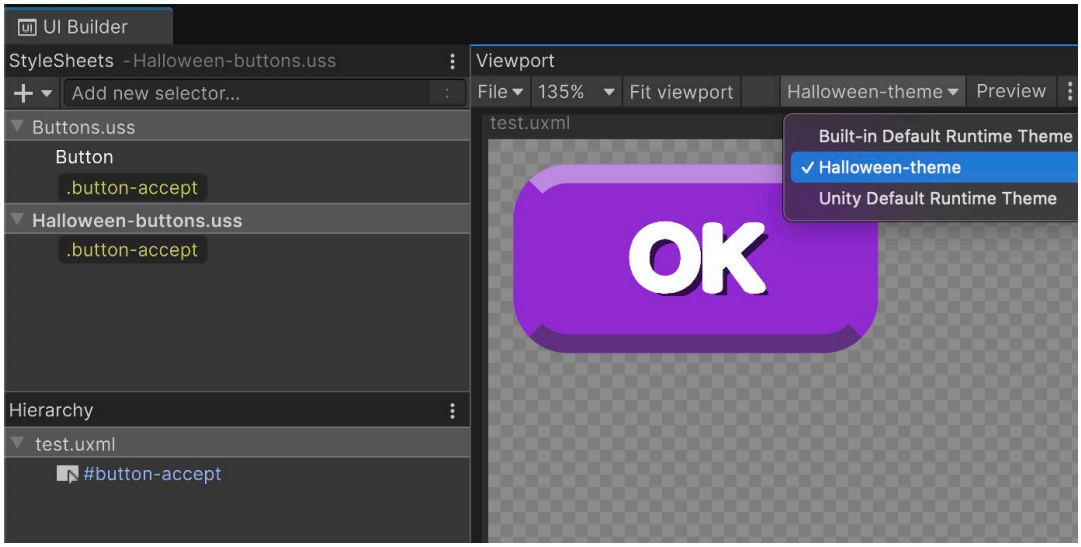
継承されたテーマとは、元のテーマと比較してセレクターが不足しているスタイルシートが新しいテーマに設定されている場合に、元のテーマのスタイルが適用されることを意味します。これにより、カスタマイズが容易になります。例えば、フォントのみを変更する新しいテーマを作成し、UI の残りの部分 (色、パディング、境界線など) は元のテーマに合わせたスタイルにすることができます。このアプローチは、ライト/ダークモードの実装、キャラクターごとの UI のカスタマイズ、ゲーム固有のイベントテーマ作成などのシナリオに便利です。



このスクリーンショットで示されているハロウィーンテーマの TSS は、**Halloween-buttons.uss** を使用していますが、ボタンで使用中のセレクター **.button-accept** に一致するセレクターがないため、元のテーマに適用されているものを使用します。

既存のテーマに基づいて新しいテーマを作成するワークフローは、次のとおりです。

1. 新しい TSS を作成し、継承するテーマと、この新しい TSS が使用する新しい USS ファイルを追加します。
2. **UI Builder > StyleSheets** で **Add Existing USS** をクリックし、新しいテーマが使用する USS を選択します。
3. 新しいテーマがオーバーライドするセレクターをコピーします。
4. それを新しいテーマが使用する USS に貼り付け、右クリックして **Set as Active USS** を選択します。
5. 新しい USS でセレクターを編集します。
6. あるテーマまたは別のテーマで使用されているスタイルは、UI Builder のドロップリストから確認できます。



UI Builder の Viewport で適用するテーマを選択します。

ランタイム時には、**Panel Settings Inspector** の **Theme Style Sheet** フィールドで新しいテーマを参照します。

命名規則

UI Toolkit では、ビジュアル要素と USS を文字列識別子を使用してクエリを実行する必要があるため、定義済みの一連の標準を使用することで、全体的にエラーが減り、コードが見やすくなります。

インターフェースを構成する UXML と USS アセットについては、開発チームも同じものを参照することになるので、ビジュアル要素とスタイルシートの両方の命名規則を標準化することが重要となります。命名規則は、UI Builder で階層を整理するのに役立ちます。また、コーディング規則やフォーマット規則を推測することがなくなり、一貫したコードベースの構築に役立ちます。

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

ビジュアル要素の名前は、それらへの参照をコードに格納するために使用されます。

万能なスタイルガイドはありません。自分のチームやプロジェクトに最適なものを選びましょう。ただし、できるだけ業界標準に近いものにすることをお勧めします。そのため、ビジュアル要素やスタイルシートの命名規則には、**Block Element Modifier (BEM)** を使用することを推奨します。BEM は、UI Toolkit が参考にした CSS や最新のウェブ開発に広く使用されています。

要素の名前に BEM スタイルを使えば、要素の役割、出現場所、および周りの他の要素との関係性が一目でわかります。BEM では、以下のように 3 つの主要コンポーネントで構成される命名規則を使用します。

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

以下に例を挙げます。

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

各名前の部分は、ラテン文字、数字、ダッシュで構成されます。また、各名前の部分はダブルアンダースコア `__` またはダブルダッシュ `--` で結合されています。3 つの部分について詳しく見ていきましょう。

- ブロック名 (`block-name`) は、ナビゲーションメニューやキャラクターの統計情報など、レイアウト内における大枠のコンポーネント、つまり明確で意味のある UI コンポーネントを表します。特定のブロックに固有ではない汎用ボタンの場合は、`button--small` のように省略してもかまいません。
- 要素 (`element-name`) は、ブロックの一部または子要素であり、セマンティックにブロックに結び付けられます。つまり、要素はコンテキストをブロックに依存しており、ブロックなしでは存在できません。そのため、`shop-button` の例は、navbar-menu ブロックに属する他のボタンとは異なるスタイルであることを示しています (例: `navbar-menu__shop-button` の `shop-button`)。

  新しい要素がコンストラクター内で子要素をインスタンス化する場合は、関連するクラスを子要素に割り当てます。例えば、`my-block__first-child` や `my-block__other-child` などです。

- 最後に、モディファイアはブロックや要素のバリエーションまたはステートを示します。例えば、ボタンが押されている、テキストボックスの項目が選択され強調されている、ショップボタンの小サイズバージョンである、などが該当します。これにより、コードを重複させることなく、さまざまなシナリオに簡単に適応できます。

BEM の命名例をさらにいくつか紹介します。

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

BEM のクラス名は自己記述的であるため、開発者がコンポーネントの構造と目的を理解しやすくなっています。そのため、階層構造を明確にし、プロジェクトの成長に合わせてスタイルを管理および更新するのに役立ちます。一般的な経験則として、簡潔さよりも読みやすさを重視します。一部の母音を省略して時間を節約することよりも、意味の明瞭さのほうが重要です。

これらの例では、ハイフン区切り (別称: ケバブケース) が使用されており、CSS の命名では一般的です。チームにとって最適な命名スキームをプロジェクトの早い段階で決定し、開発全体を通してそのスキームに従う必要があります。

CSS の命名規則については、こちらの記事や UI Toolkit のドキュメントを参照してください。

ヒント: UI Toolkit における命名規則

以下に示すのは、効果的な命名のためのガイドラインです。

- 短くてわかりやすい (曖昧でない名前) を付ける。名前は簡潔でありながら、UI における目的や役割が伝わる程度に説明的にします。
- `inventory__button--equipped` ではなく、`inventory__slot--equipped` にように、役割と関係を強調する名前を使用します。ボタンやラベルなどのタイプ名は、明確な場合は省略します。
- 変更される可能性のある名前/修飾子は避けます (例えば、色のスキームがまだ最終決定されていない場合は、"button-red" ではなく "button-quit" のようにします)。見た目に依存する名前ではなく、セマンティックな命名を使用しましょう。これにより、スタイルの細部が変更された場合でも、関連性のある名前を維持できます。
- これらの規則をアートアセットにも適用する (UI Toolkit インターフェースに関連付けられたスプライトやテクスチャなど)。コードとアセットの名前に一貫性を持たせることで、プロジェクト全体を通して明確な関係を維持し、整理整頓に役立ちます。
- 同じ要素を他のプロジェクトでも使用する場合は、既存のユーザークラス名との競合を避けるために、クラス名にプレフィックスを付けることを検討する。名前空間やプレフィックスを使用することで、他のプロジェクトやライブラリと統合する際の競合を防止できます。
- コンストラクター内で AddToClassList() を使用して、関連する USS クラスを要素インスタンスに加える。この方法により、要素のインスタンス化時に必要なクラスが加えられ、適切なスタイルが適用されるため、UI コードの一貫性と明瞭性が維持されます。

C# スタイルガイドの作成

あなたや、あなたのチームが、基盤となるコーディングプラクティスを改善し、プロジェクトをよりスケーラブルなものにしたいとお考えであれば、無料の eBook『*C# スタイルガイドの作成:スケーラブルでクリーンなコードの記述*』を参照してください。この eBook は、コードスタイルと命名規則を標準化するためのガイドとして活用いただけます。

e ブックをダウンロード

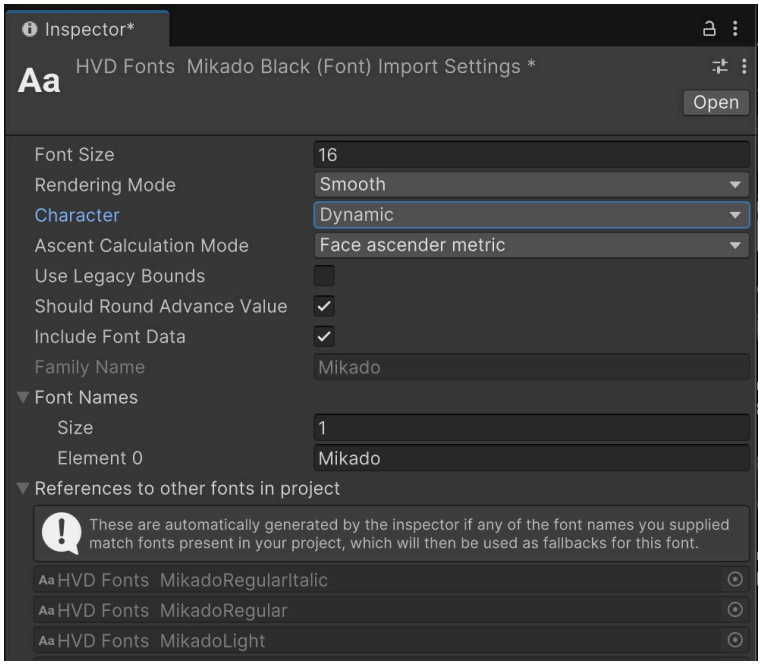
テキスト

UI Toolkit では、**TextCore** という、(古い機能の UI システムである Unity UI で使用されている) **TextMesh Pro** をベースとするフォントレンダリングテクノロジーが使用されています。TextCore は、高度なスタイリング機能を提供しており、さまざまなポイントサイズや解像度でテキストをクリーンにレンダリングすることができます。**符号付き距離フィールド** (SDF) フォントレンダリングを活用しており、変形や拡大が行われても見た目が鮮明なフォントアセットを生成することができます。TextCore のさまざまなレンダリングモードの詳細は、ドキュメント で確認できます。

フォントアセットの種類と、それらの使用目的について見ていきましょう。

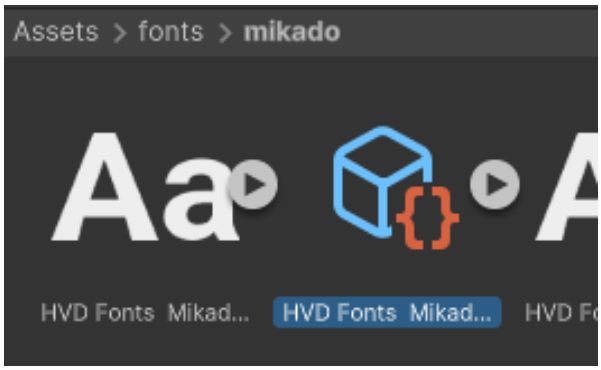
ソースフォントファイル

最も一般的なフォント形式である **TTF** ファイルと **OTF** ファイルは、Unity プロジェクトで使用する前に **フォントアセット** に変換する必要があります。フォントアセットは、文字のグリフ、フォントメトリクス、そしてサイズ、ウェイト、スタイルといったレンダリング設定などの、フォントをレンダリングするために必要なデータを含む Unity 固有のリソースです。インポートされたソースファイルには、各フォントファミリとそのレンダリングオプションの情報が示されます。



これらのインポートオプションの多くは、古い Unity UI のテキストシステムの名残で、将来のリリースで削除される予定です。Rendering Mode、Character、Include Font Data は、フォントアセットの生成に使用されます。

対応するフォントアセットを生成するには、ソースフォントファイルを選択し、Assets メニューを右クリックして、**Create > Text Core > Font Asset > SDF** (レンダリングモードを SDF とする場合) で生成します。



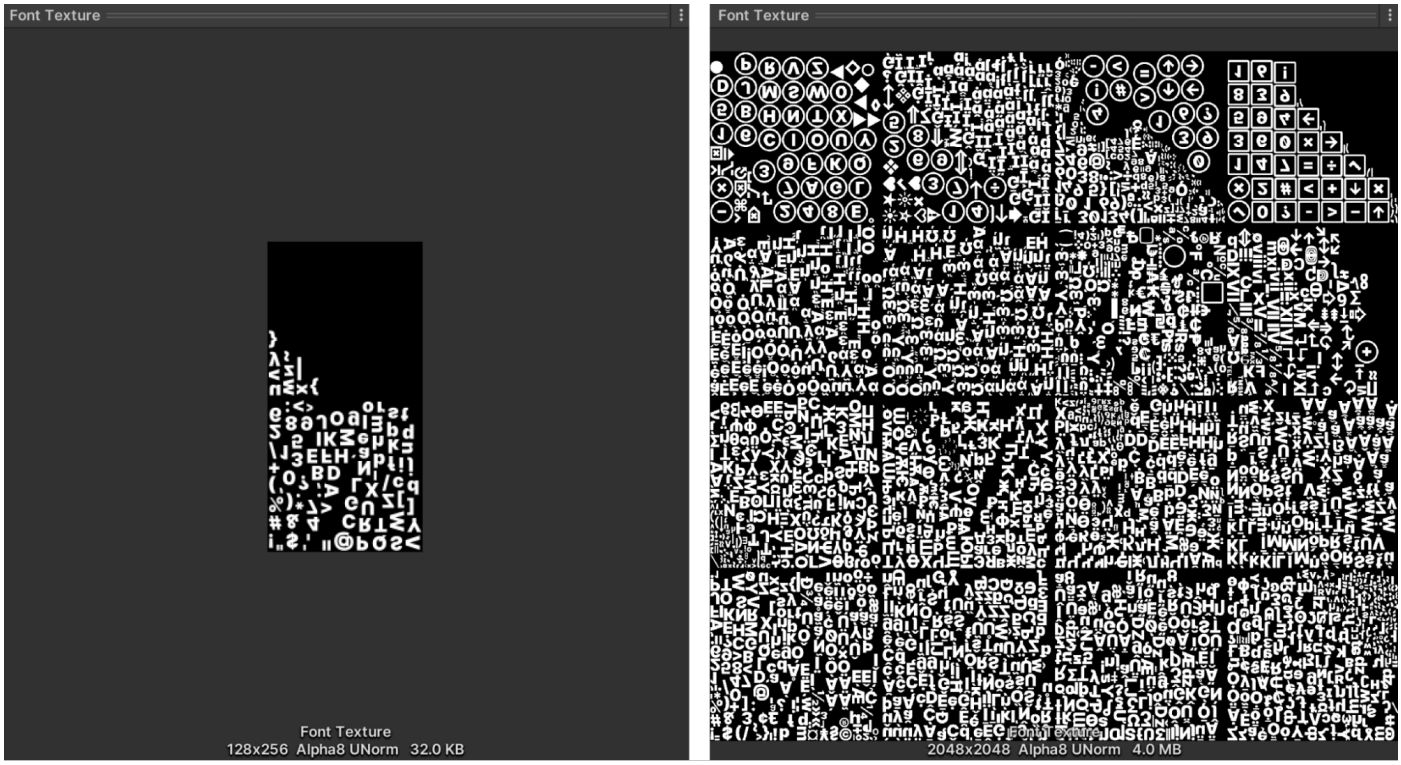
異なる UI システムでは、異なるフォントアセットが使用されます。

フォントアセットの設定

ソースフォントファイルの変換後は、フォントアセットを選択するとすべてのオプションが表示され、フォントの生成を完全に制御できます。主要なオプションをいくつか見てみましょう (フォントアセットの詳細については、ドキュメントを参照してください)。

— **Face Info:**デフォルトのソースフォントを調整する必要がある場合に、よりアプリケーションに適したパラメーターを調整するための間隔とスケーリングのオプション

- **Generation Settings**:ソースフォント、ソースフォントに複数のスタイルが含まれている場合のフォントアセットに使用するフォントフェイス、アトラス生成モード、レンダリングモードなどの基本的な設定
- **Atlas and Material:**生成されるマテリアルとテクスチャ。アトラスが静的か動的か、レンダリングモードがビットマップか SDF かを指定します。大規模な文字セットを持つ言語をサポートする場合に、生成されるアトラスのサイズの制御を提供します。
- **Font Weights:**ソースアセットにウェイトのバリエーションがない場合に、異なるフォントウェイトのシミュレーションを行います
- **Fallback Font Asset:**現在のフォントアセットに文字やグリフがない場合にフォールバックフォントを提供します
- **Character and Glyph Tables:**フォントアセットに含まれるすべての文字とグリフの詳細なリスト
- **Ligature table:**並んでいる 2 文字に使用するグリフを追加 (読みやすさとビジュアルフローを改善) します
- **Glyph Adjustments:**文字またはグリフごとにオーバーライドを定義します



ソースフォントとアトラスによりビルドサイズが大きくなることがあります。左側は ASCII 文字のアトラス、右側は完全な Unicode 文字セットのアトラスです。

Inspector の上部でフォントアセットを選択し、**Update Atlas Texture** ボタンをクリックすると、**Font Asset Creator** が現れます。ここでアトラスプロパティの生成と定義をすべて制御できます。

ヒント: パディングとアトラス解像度

Font Texture 内の文字については、文字間のパディング (ピクセル単位で指定) を設定して、文字を個別にレンダリングできるようにする必要があります。パディングによって、SDF グラデーションのためのスペースも確保されます。パディングが大きいほど、遷移が滑らかになり、高品質のレンダリングや、太いアウトラインなどの効果が可能になります。

ASCII 文字のみを使用している場合は、アトラス解像度が 512 x 512、パディングが 5 という設定で、ほとんどのフォントに十分対応できます。文字種が多いフォントの場合は、より大きな解像度や複数のアトラスが必要になる場合もあります。原則として、パディングのサイズについては、サンプリングサイズとの比率が 1:10 になることを目指してください。

フォントアセットバリアント

新しいフォントアトラスを作成することなく変更を加えたい場合は、**Create > Text Core > Font Asset Variant からフォントアセットバリアントを作成することができます。**このバリアントによって、フォントの行メトリクスの代替バージョンを保持することができます。

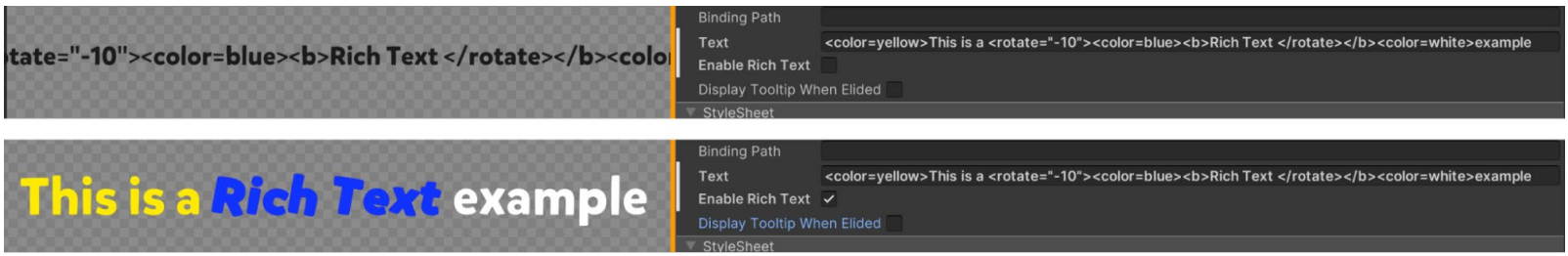
バリアントでは独自の Face Info 設定 (行の高さや下付き文字の位置など) が保持されますが、元のアトラスも参照されます。そのため、追加のテクスチャスペースを消費することなく、元のフォントアセットとは異なる独自のスタイリングを適用することができます。

リッチテキスト

リッチテキストタグ は、テキストフィールド内の補足的なタグを使ってテキストの外観やレイアウトを変更するものです。コードと UI Builder の両方のビジュアル要素で、リッチテキストタグを使用できます。このタグを使用すると、例えばユーザー名の外観をカスタマイズするために、ランタイム時にテキストをフォーマットできます。

リッチテキストタグを使用すると、テキストのプロパティやスタイリングを変更せずに、テキストの色や配置を変更することができます。これらのタグは、対話システムのテキストをフォーマットしたり、伝えたい内容を視覚的に強調したりする目的で使用できます。

Extra Settings に移動して、UI Builder のリッチテキスト機能を有効にします。有効にすると、テキスト (タグを含む) が適切にフォーマットされます。例えば、 タグとそれを閉じる タグとの間のテキストは、太字で表示されます。

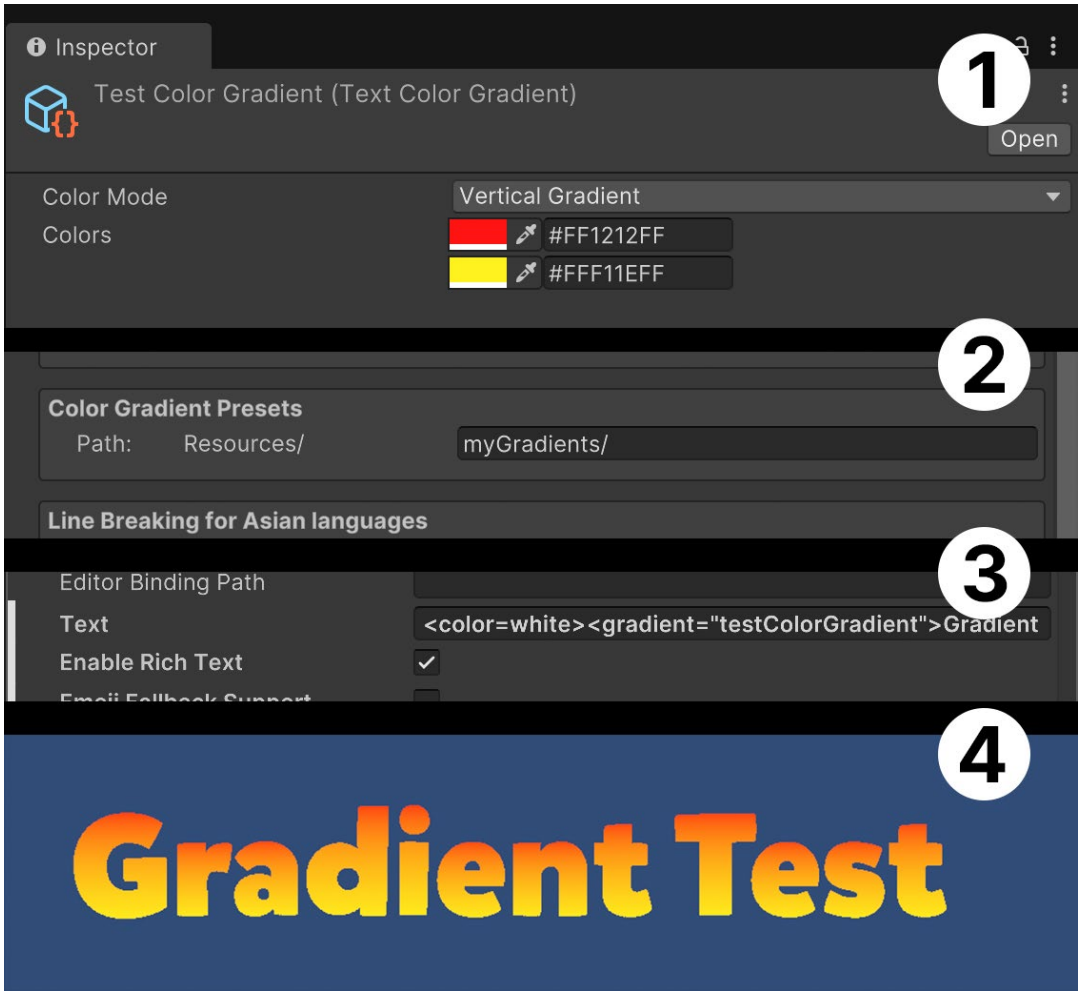


UI Builder でリッチテキストタグを有効にすると、タグによってビジュアルテキストのプロパティが変更されます。

利用可能なリッチテキストタグとパラメーターの 完全なリスト をご確認ください。

グラデーション

グラデーション はインターフェース全体にスタイルを設定します。UI Toolkit では、`<gradient>` タグで適用できます。以下のステップに従い、シンプルなグラデーションを作成します。

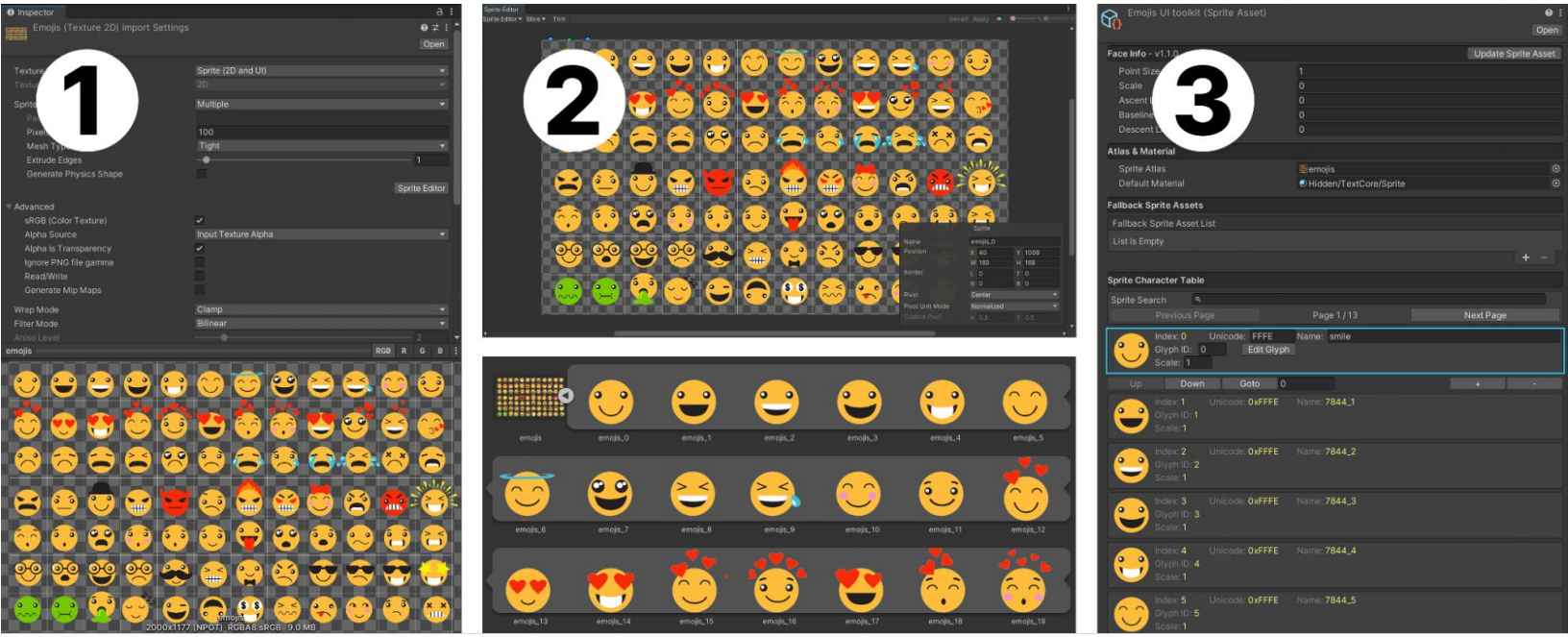


1. グラデーションカラーアセットは、**Create > Text Core > Gradient Color** で作成します。このファイルは、**Assets/Resources** または Resources のサブフォルダー内に配置してください。
2. Panel Settings から参照する Text Settings アセット を作成します。アセットで Color Gradient Presets を探し、アセットがあるフォルダーまたは Resources 内のサブフォルダーを指定します。
3. UI Builder 内に、リッチテキストタグ `<color=white><gradient="testColorGradient">Gradient Test</gradient></color>` を加えます。
4. このカラータグでフォントの色を白に復元することで、グラデーションが意図したとおりの外観になります。参照するグラデーションはステップ 1 で作成したアセット名と一致する必要があります。**Rich Text** が有効になっていることを確認します。
5. UI Builder 内またはゲームビューで変更が有効になっていることを確認できます。

スプライトアセットと絵文字

リッチテキストタグを使って絵文字のようなスプライトをテキストに含めることができます。それらを使用するには、グラデーションアセットに似たスプライトアセットを使用する必要があります。

複数のスプライトをインポートする場合は、ドローコールを減らすために、スプライトを単一のアトラスにパックします。スプライトアトラスには、対象のプラットフォームに適した解像度を設定するようにしてください。スプライトの解像度に関する詳細については、アセット準備のセクションに戻って確認してください。



スプライトアセットの一般的なユースケースとして、テキスト文字列に統合された絵文字やアイコンがあります。

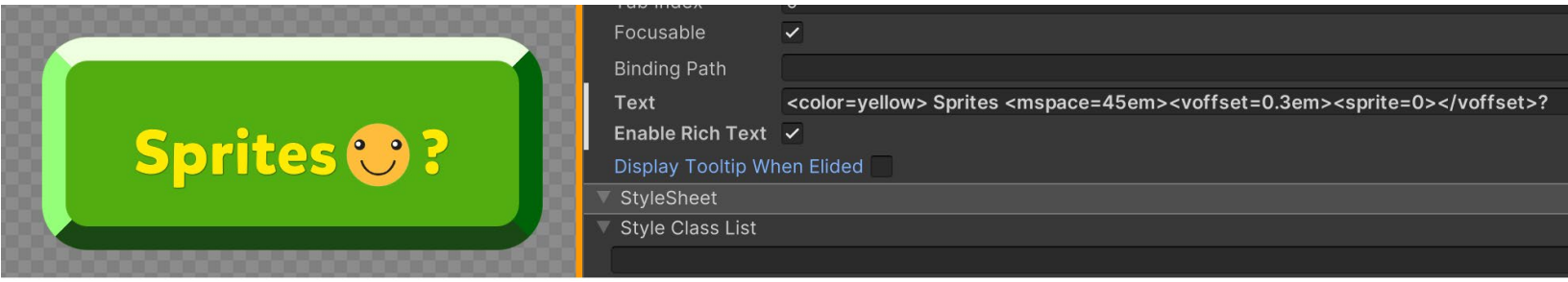
この目的のためにスプライトをインポートする手順は、以下のとおりです。

1. 絵文字やアイコンを含むスプライトまたは PSD ファイルをインポートします
2. 画像を複数のスプライトにスライスします。グラフィックアセットとフォントアセットの準備セクション で説明したように PSD ファイルを使用する場合は、このスライスは不要です。ファイルから **スプライトアセット** を生成します (**Create > Text Core > Sprite Asset** メニューを選択および使用します)。アセットが Assets/Resources またはサブフォルダーに配置されていることを確認します。
3. 新しいスプライトアセットの Face Info を調整し、各 "グリフ" の外観や名前をカスタマイズすることができます。ここでの変更によって、フォントアセットのデフォルトの Face Settings が置き換えられます。

注: この状況で **Update Sprite Asset** を実行すると、スプライトアセットがスプライトエディターでの最新の変更に対して同期されます。

このアセットを UI Toolkit で使用するには、グラデーションで行ったのと同じ手順に従う必要があります。

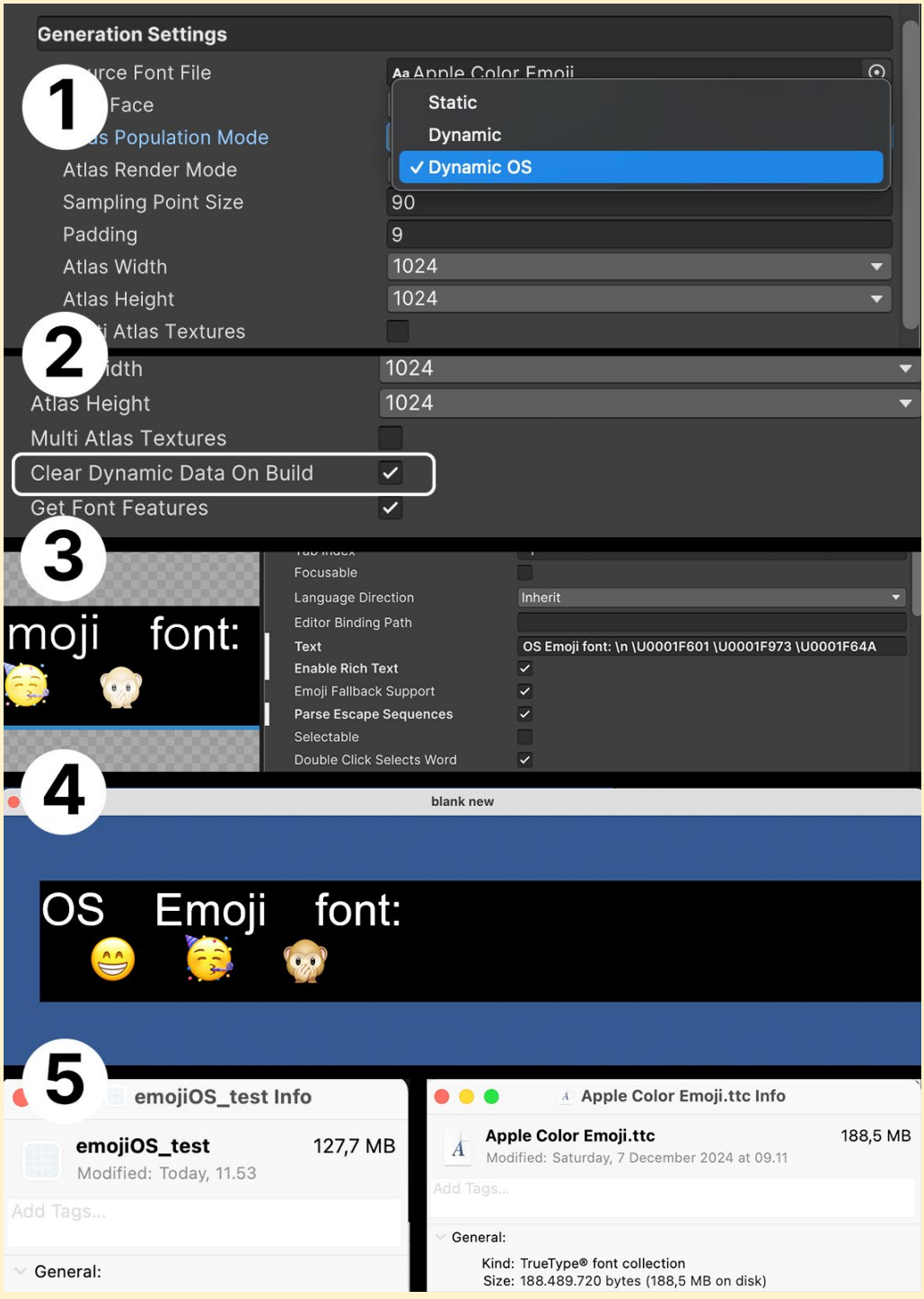
1. **UI Document** から **Panel Settings** を選択します。
2. **Text Settings** アセットを開きます (ない場合は作成します)。
3. Text Settings ファイルのファイルブラウザーを使用して、**スプライトアセット** にリンクします。保存して再生モードにすると、更新された設定が有効になります。
4. リッチテキストタグ (`<sprite index=0>` または `<sprite name="name">`) を使用して、スプライトを追加します。埋め込まれたスプライトには、他のテキストタグも適用されます。



リッチテキストタグ を使用して、UI Toolkit のテキストフィールドにスプライトアセットを追加します。**Enable Rich Text** オプションがオンになっていることを確認してください (上部)。

ヒント: OS の絵文字の使用

iOS や Android などの特定のランタイムプラットフォームをターゲットにしている場合は、プロジェクトにソースフォントを加える代わりに、システムにビルトインされている絵文字フォントを使用できます。これにより、メモリを節約し、アプリケーションに絵文字の大規模なコレクションをパッケージする必要がなくなります。また、Text Settings の Global Fallbacks にも適しています。

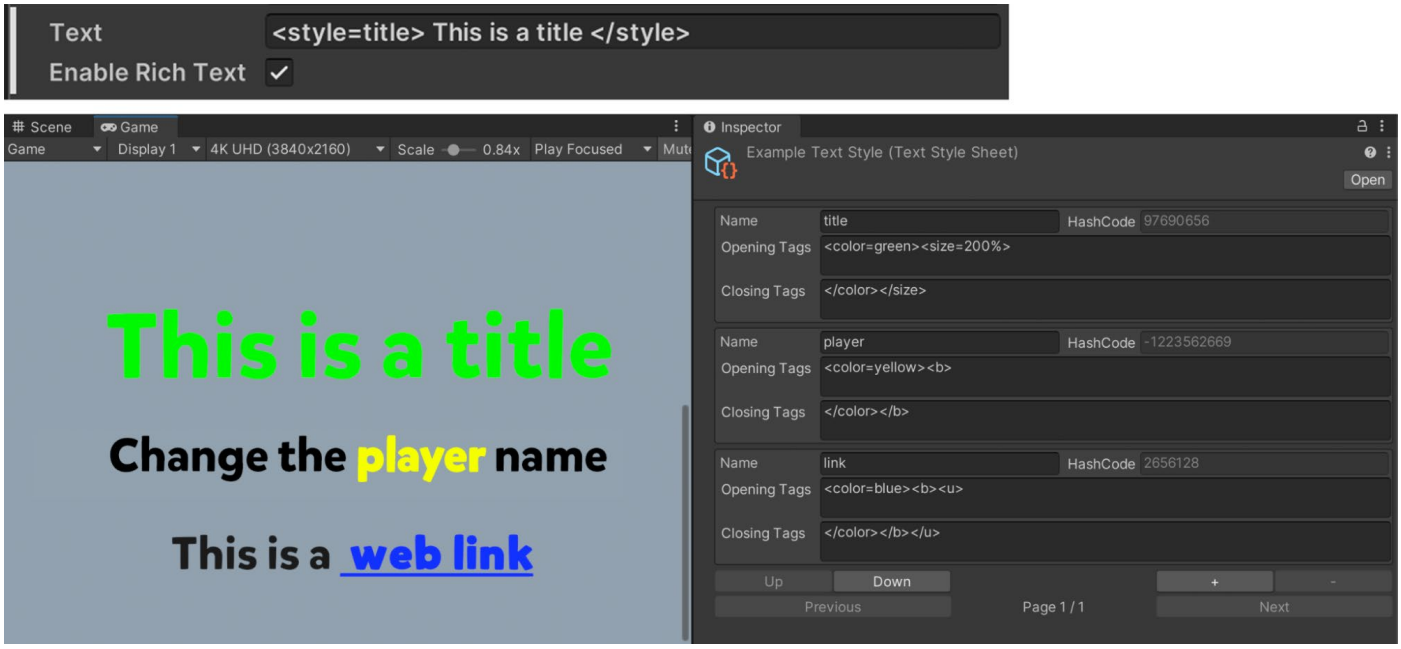


プロジェクトで OS の絵文字を使用する手順は、以下のとおりです。

1. ターゲットシステムが使用しているフォントからフォントアセットを作成します。iOS では、フォントを Apple Emoji (この例で使用)、Android では、Noto Color Emoji (現在はCOLRv0 のみサポート) と呼びます。フォントアセットのタイプが **Color** であることを確認してから、アトラスの生成モードを **Dynamic OS** に設定します。これにより、アセットにソースフォントを含める必要がなくなり、スペースの節約になります。
2. フォントアセットで **Clean Dynamic Data On Build** がオンになっていることを確認します
3. UI Builder で **Parse Escape Sequences** を有効にし、macOS や Windows の絵文字キーボードを使用するか、UTF 形式で目的の絵文字を入力します。例えば、\U0001F601 でスマイリーを入力できます。各絵文字の UTF は、フォントアセットの文字コード表で確認できます。
4. macOS で実行されるビルドでは、OS のフォントに従って絵文字が表示されます。
5. 試してみると、**ビルドサイズ**がスタンドアロンの絵文字フォントよりも小さく、プロジェクトに含まれていないにもかかわらず、適切な絵文字をレンダリングするために使用されていることがわかります。

テキストスタイルシート

アプリケーションで大量のテキストを扱う場合は、その書式設定を管理するための テキストスタイルシート を作成することを検討すると良いでしょう。そうすれば、<style> リッチテキストタグを使用してカスタムのテキストスタイルを作成することができます。これは、Create メニューから**Assets > Text Core > Text Stylesheet** を選択して実行できます。



再利用可能なテキストスタイルシート

テキストスタイルシートには、以下の利点があります。

— カスタムスタイルには、開始と終了のリッチテキストタグに加えて、前後のテキストも含めることができます。

— テキストスタイルシートは、リッチテキストの書式設定を直接変更する場合と比べて、便利に更新することができます。

— カスタムスタイルを使用すると、リッチテキストタグの数を減らすことができます。1 つのタグ <style= name> を使用するだけで、必要なすべてのスタイルを適用できます。

— これにより、テキストスタイルシート内の 1 つのリッチテキストタグを簡単に変更でき、複数の <style> タグを手動で変更するよりもエラーが起こりにくくなります。

データバインディング

本質的に、ユーザーインターフェースは、アプリケーションを動かすデータとプレイヤーの接続点です。ユーザーインターフェースは、ゲームの内部状態やロジックを把握、操作、関与するための主たる手段です。

プレイヤーは生データではなく、体力ゲージを見ます。アイテムリストを直接読み取るのではなく、ドラッグアンドドロップ式のインベントリを使用します。UI とデータのこの相互作用は、プロジェクトの組み立て方に影響を与えます。

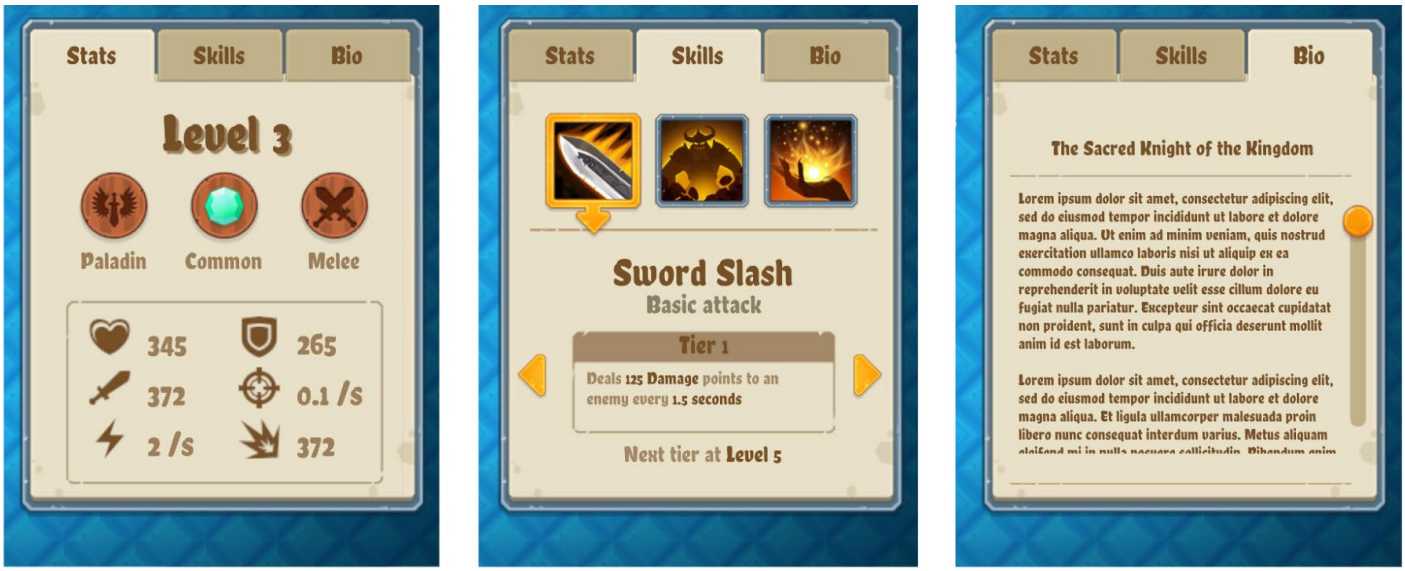
ゲームデータを反映する UI

*UI Toolkit サンプル –『Dragon Crashers』*のキャラクター統計ウィンドウを以下に示します。このユーザーインターフェースは、RPG 型ゲームの主要な属性を示しています。

View (ビュー) は UI そのものであり、プレイヤーが関わる部分です。タブ付きのコンテナはキャラクターの能力を整理し、ナビゲーションを容易にします。

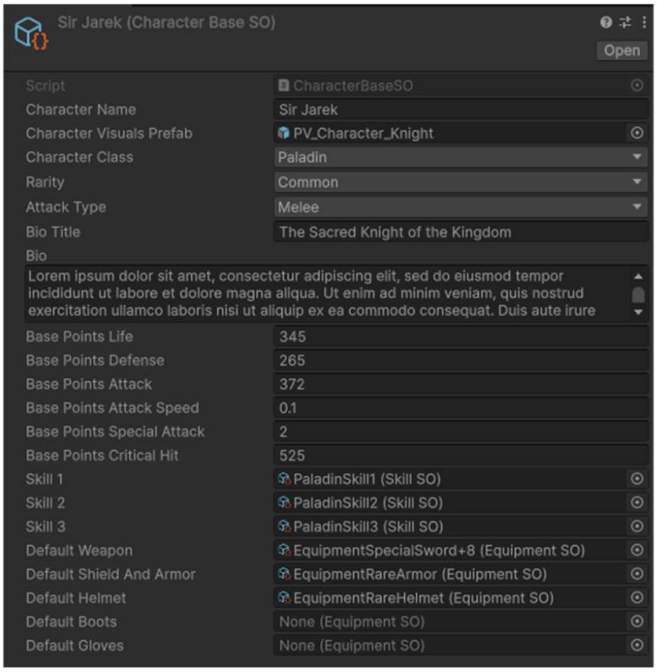
キャラクター統計ウィンドウはゲームデータを表します。

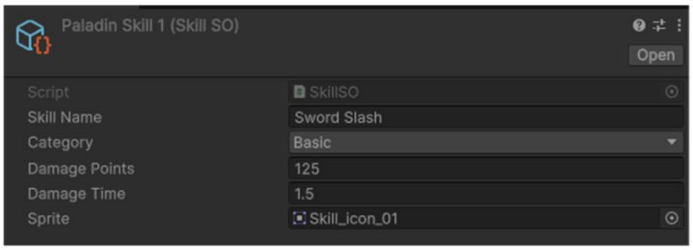
UI の裏側では、各キャラクターの統計情報を保存する ScriptableObject などの Model (モデル) 内にデータが存在します。



View (user interface)

Model (data)





ScriptableObject アセットにはキャラクターのデータが含まれます。

ビューとモデルの 関心の分離 は、UI アーキテクチャの中核となる原則です。基礎となるデータからビジュアルインターフェースを切り離すことで、コードの柔軟性と再利用性が高まり、管理しやすくなります。

ただし、分離した後でモデルをビューに接続するには、同期が必要になります。従来、これには直接的な更新またはイベント駆動型システムが使用されており、データが変更されたときにオブザーバーが UI を更新します。これらの同期操作は有効ですが、反復的な定型コードが必要になる可能性があります。

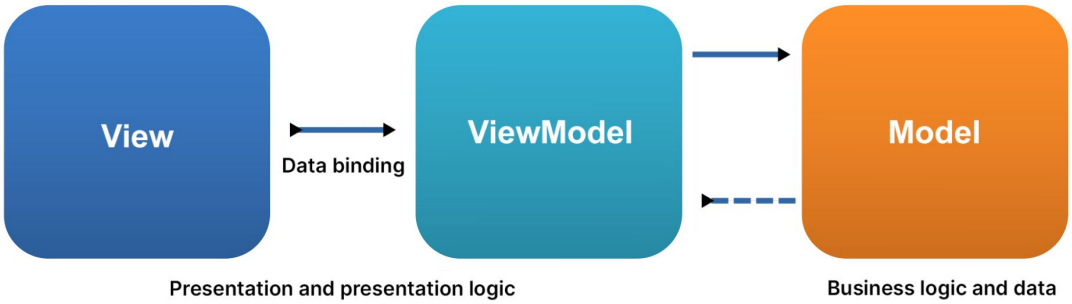
プロジェクトが大きくなるにつれて、こうしたシステムの管理は難しくなります。新しい要素や依存関係を追加するには、多くの場合、追加の更新ロジックやイベントハンドラーが必要です。スクリプトが乱雑で読みにくくなり、メンテナンスが難しくなります。

ランタイムデータバインディングの導入

Unity 6 のランタイムデータバインディングは、この問題に対する合理的なソリューションを提供します。アプリケーションのデータを UI 要素に直接リンクし、片方での変更が自動的にもう片方に反映されるようにします。

この モデルビュービューモデル (MVVM) アーキテクチャはビュー (View) とモデル (Model) の間にプレゼンテーションロジック (Presentation logic) のレイヤーを追加します。ビューモデル (ViewModel) は仲介役として機能し、ビュー用にフォーマットされたモデルからデータを公開します。

MVVM とその他のデザインパターンの詳細については、Unity eBook *デザインパターンと SOLID でコードをレベルアップする* を参照してください。



MVVM アーキテクチャ (出典:Wikipedia)

例えば、体力ゲージにプレイヤーの体力を自動的に表示したり、追加のスクリプトロジックや手動のイベント処理を必要とせずにスコアラベルをリアルタイムで更新したりできます。管理する必要がある同期ロジックが減ることで、プロジェクトをより効果的にスケールできます。

プロジェクトでどのように使用できるか、UI Toolkit のランタイムデータバインディングの例を探ってみましょう。

データバインディングのコンセプト

Unity 6 にはランタイムデータバインディングシステムが導入されており、UI 要素とアプリケーションデータをつなぐ構造化された方法を提供します。ビジュアル要素のプロパティをデータソースにバインドするには、DataBinding のインスタンスを作成します。

重要なコンセプトがいくつかあります。

- データソース:UI バインディングのデータを保持するオブジェクトです。
- データソースパス:データソース内のこのプロパティやフィールドが、UI 要素の接続先となります。
- バインディングモード:ソースと UI の間のデータフローを制御します。一方向と双方向のどちらでも可能です。

これらのパーツが連携してデータバインディングを作成します。さらに詳しく見ていきましょう。

データソースの準備

データソース は、UI バインディングのデータを保持するオブジェクトです。ScriptableObject、MonoBehaviour、カスタム C# オブジェクトなど、任意の C# オブジェクトがデータソースとして機能します。構造体をデータソースとして使用すると、軽量なメモリ割り当てとガベージコレクションの減少によりパフォーマンスを改善できます。データバインディングは、コードと Inspector のどちらでも設定できます。

このデモプロジェクトでは、Unity Inspector 内でデータをシリアル化するのが便利になるように、データソースとして ScriptableObject を使用します。

CreateProperty 属性の使用

バインディングのためにプロパティを公開するために、UI Toolkit はUnity Properties モジュールによって生成されたプロパティバッグ を使用します。これらは、データソース内のどのプロパティが UI バインディングにアクセスできるかを定義します。

プロパティをバインド可能にするには、CreateProperty 属性を使用します。これは、バインディングシステムのプロパティを明示的にマークします。一般的な設定パターンを以下に示します。

```
[SerializeField, DontCreateProperty]
int m_Value;

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

この例では、`m_Value` はシリアル化のために `SerializeField` 属性でマークされていますが、`DontCreateProperty` 属性によってバインディングから除外されています。

一方、`Value` プロパティは `CreateProperty` でマークされており、これによりバインディングシステムにアクセスできるようになります。この明確な分離により、モデルと UI の間のデータフローが管理されます。

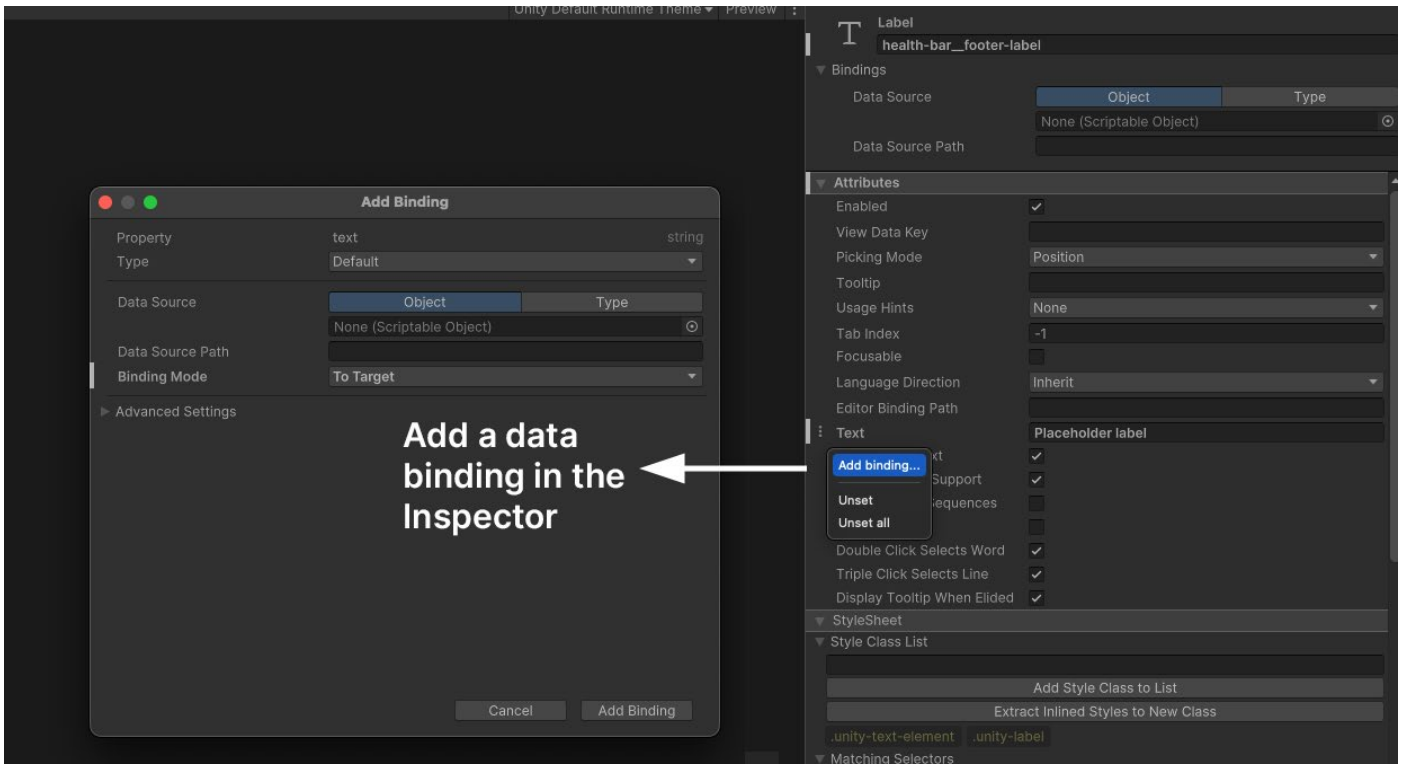
ランタイムデータバインディングは プロパティバッグ を使用して、特定の型のデータを効率的に走査および操作します。デフォルトでは、Unity は型への初回アクセス時にリフレクションを使用してプロパティバッグを生成するため、ランタイムのオーバーヘッドが少し高くなります。

これを回避するには、プロパティを定義するときに CreateProperty 属性 を使用します。これにより、コンパイル時にバインディングコードが生成され、ランタイムリフレクションが不要になり、パフォーマンスオーバーヘッドが削減されます。

データソースとパス

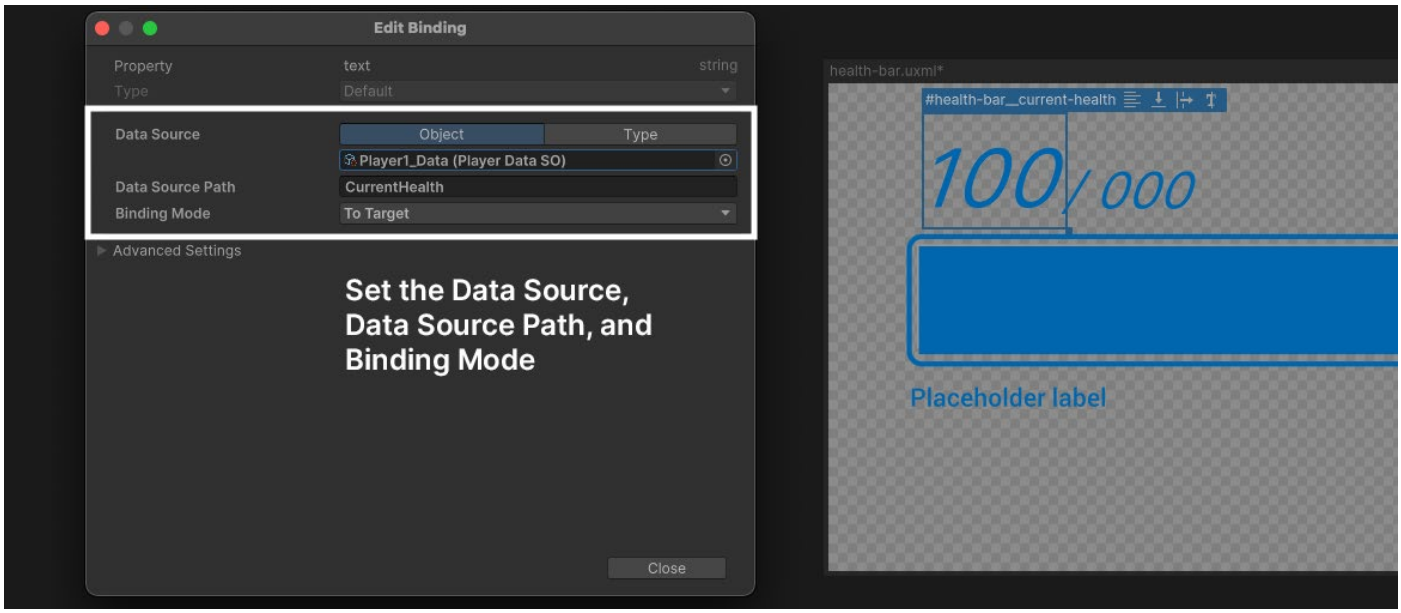
データソースの準備ができたら、それを UI にバインドできます。データソースパス は、UI 要素に結び付けるデータソース内のプロパティまたはフィールドを指定します。例えば、データソースに "体力" プロパティがある場合、パスは UXML または C# のバインディング設定を使用して、そのプロパティを直接示します。これを実際に見てみましょう。

UI Builder: Hierarchy の要素を選択し、Inspector に移動して、オプション **(⋮)** メニューから **Add Binding** オプションを使用します。



Inspector からバインディングを追加します。

次に、**Data Source** (PlayerDataSO ScriptableObject など) を割り当て、**Data Source Path** (CurrentHealth など) を指定します。



UI Builder で Data Source と Data Source Path を設定します。

UXML:UI Builder でデータバインディングを設定すると、対応する UXML が生成されます。テキストエディターでデータソースパスを手動で追加したり編集したりすることもできます。これは、バインディングを作成するコードブロックです。

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

C# の使用: ScriptableObject などのデータソースオブジェクトをスクリプトでインスタンス化または参照します。それをルート要素のdataSource プロパティに割り当てます。dataSourcePath を使用して、バインドする正確なプロパティを指定します。

以下は、スクリプトで dataSource と dataSourcePath のプロパティを設定する方法を示しています。これについては、後述の C# でのデータバインディング設定 のセクションで詳しく説明します。

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

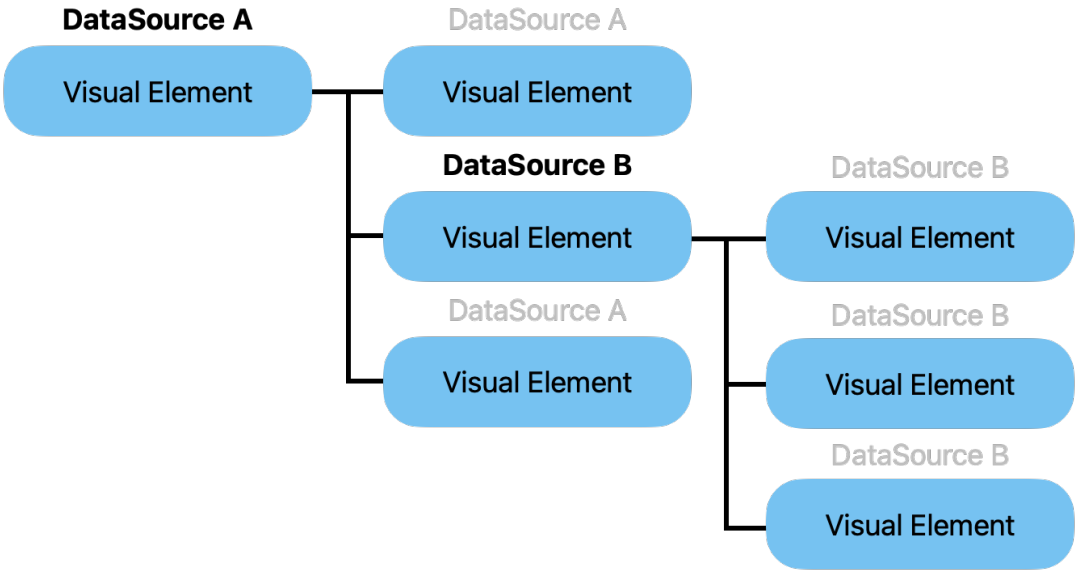
注: 同じ UI 要素にデータバインディングを定義している場合、競合が発生する可能性があります。混乱を避けるには:

- **UI Builder/UXML バインディングを使用します。**ランタイム調整を必要としない静的データまたはデフォルトデータ設定の場合です。
- **C# バインディングを使用します。**動的な更新やゲームプレイ中にデータソースを変更する必要がある場合です。

UI Builder/UXML でバインディングの一部を設定し、ランタイム時にバインディングを完了することもできます。コンテキストの詳細については、後述の 未解決データバインディングのワークフロー のセクションを参照してください。

データソースの継承

ビジュアル要素は、新しいデータソースが明示的に割り当てられない限り、親要素のデータソースを自動的に継承します。例えば、ルート要素にデータソースがある場合、すべての子要素がデフォルトでそのデータソースを使用します。この図はその動作を示しています。



子要素は親データソースをオーバーライドできます。

親要素にデータソースがある場合、子要素は自動的にそれを継承します。UI Builder では、子の Data Source フィールドには親のデータソースが事前に入力されていますが、必要に応じてオーバーライドできます。

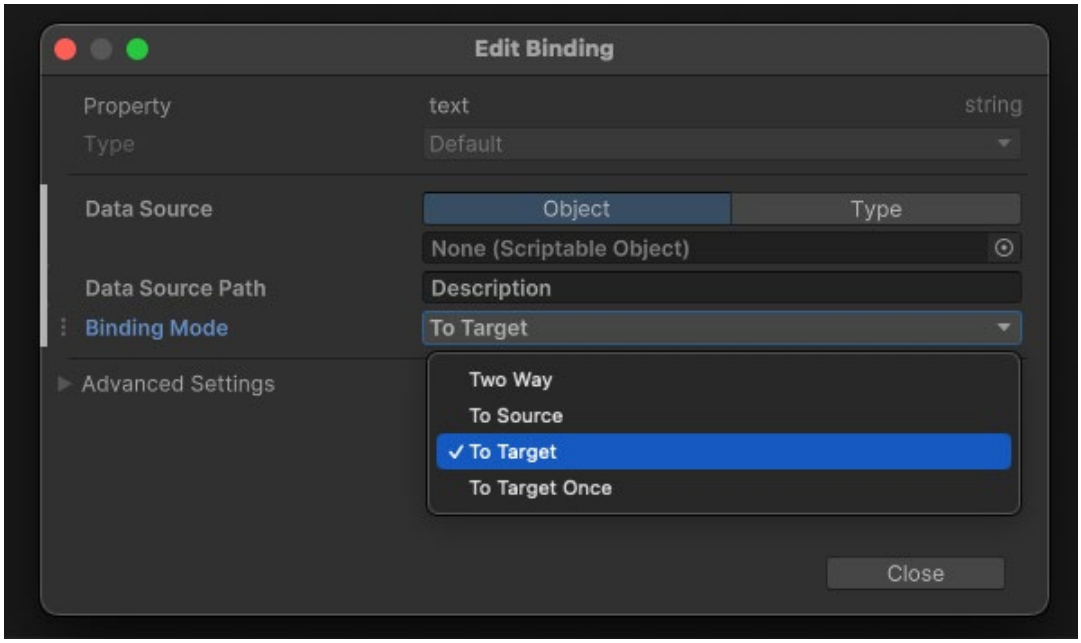
次の例に示すように、C# で作業するときにも同じ継承ロジックが適用されます。

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

// ルート要素にデータソースを割り当てる
root.dataSource = parentData;

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

// 子の継承データソースをオーバーライド
child.dataSource = childData;

root.Add(child);
```

ここでは、子が親をオーバーライドし、独立したデータソースを提供しています。

バインディングモード

バインディングモード で、データソースと UI の間のデータフローを制御します。



UI Builder のバインディングモードを使用すると、データソースと UI の間のデータフローを制御できます。

以下のオプションは UI Builder と C# API に表示されます。

- **TwoWay (Default):**変更は、データソースから UI 、および UI からデータソース、の両方に反映されます。スライダーやテキストフィールドなど、ユーザーがデータを変更できる対話型の要素に使用します。
- **ToTarget:**データはデータソースから UI へのみ流れます。読み取り専用の UI 要素に使用します。
- **ToSource:**データは UI からデータソースへのみ流れます。現在の値を最初に表示する必要のない入力に便利です。
- **ToTargetOnce:**データソースから UI へデータが 1 度だけ流れ、データソースでのその後の変更は追跡されません。

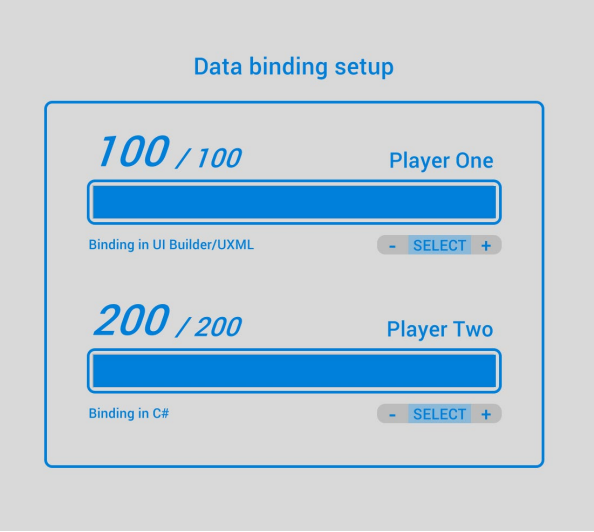
例: 体力ゲージのデータバインディング

実際の例で、UI Toolkit で基本的なデータバインディングを作成する方法を見てみましょう。デモシーンの例です。プレイヤーの体力に基づいて動的に更新されるシンプルな体力ゲージです。

デモシーン
次の例は、*QuizU サンプルプロジェクト* に含まれている **Data Binding** のハウツーデモにあります。

ランタイム時にアクセスするには、メインメニューに移動して**Demos > Data Binding** を選択するか、ブートローダーを無効にした (**Quiz > Don't Load Bootstrap Scene on Play**) 直後に **DataBindingDemo** シーンを直接ロードします。

このデモシーンには 2 つの体力ゲージが含まれています。1 つは UI Builder を使用して UXML で作成されたバインディングで、もう 1 つは C# で作成されたバインディングです。



体力ゲージはプレイヤーデータを表します。

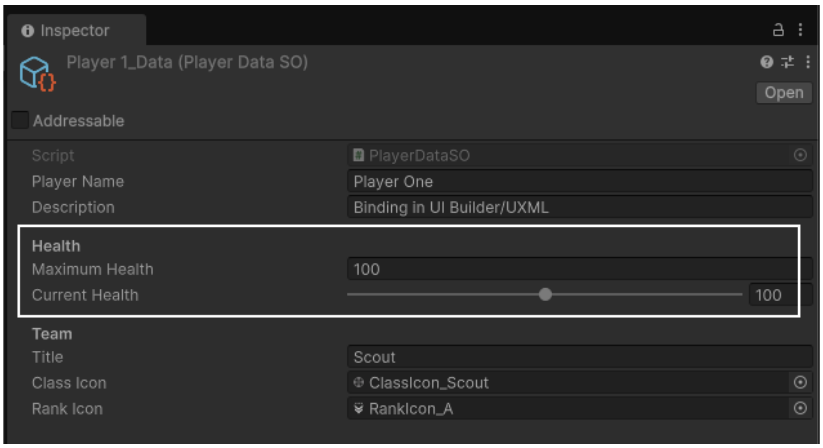
データソースの準備

サンプルプロジェクトには、PlayerDataSO ScriptableObject に格納されているプレイヤー情報と統計が含まれています。PlayerDataSO の関連プロパティは CreateProperty 属性でマークされ、バインディングに使用できます。

各体力ゲージは、プレイヤー名や体力値など、PlayerDataSO のデータのサブセットのみを表します。以下は、クラスのプロパティと関連フィールドの一部を示しています。

```
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;
```

UI では、PlayerName、CurrentHealth、MaximumHealth などの特定のデータパスを使用して、この情報を画面上に視覚的に表示します。



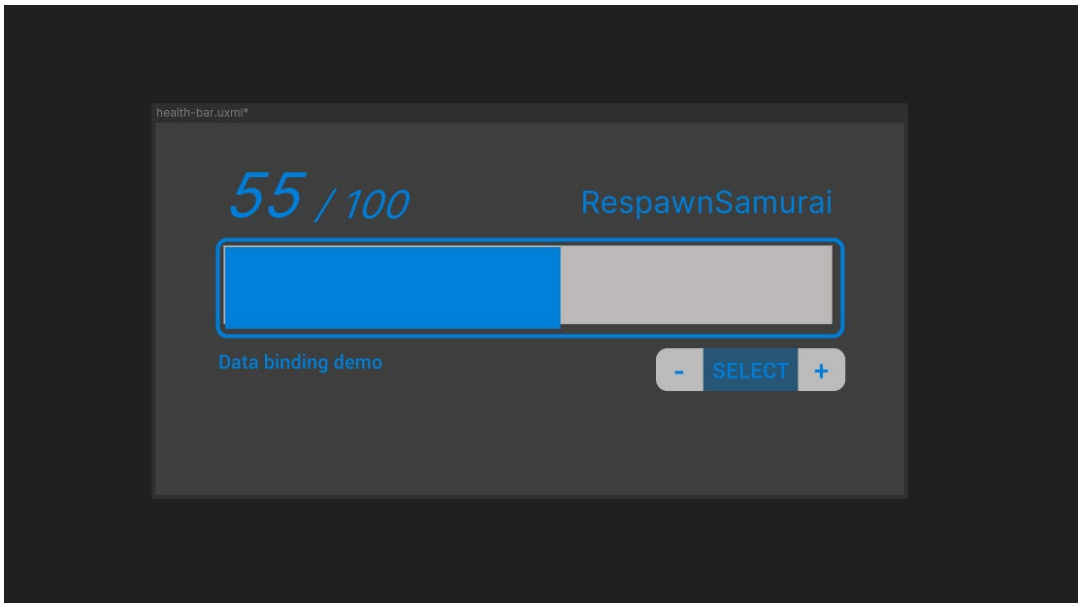
データソースには PlayerDataSO ScriptableObject の体力データが含まれています。

UI Builder/UXML でのデータバインディング

UI Builder は、UI 要素をデータにバインドする視覚的で対話的な方法を提供します。これは、デザイン中心のワークフローを好む UI アーティストや、設定中にリアルタイムのフィードバックのメリットを享受できる開発者に最適です。また、データバインディングを初めて使用する方に役立つ学習ツールとしても機能します。

デモシーンでは、Player One の体力ゲージのデータバインディングはすべて UI Builder で設定されています。これには以下が含まれます。

- **ルート要素の選択**:体力ゲージを含む階層内でルート要素を選択します。この例では、最上位のコンテナは **demo_container-uxml** 要素です。
- **データソースの割り当て**:Inspector の Data Binding セクションで、データソースを ScriptableObject アセットに設定します。データソースが割り当てられ、すべての子要素に反映されます。
- **データソースパスの定義**:個々の UI 要素を ScriptableObject のそれぞれのプロパティにリンクするデータソースパスを指定します (例えば、`PlayerDataSO.PlayerName`)。



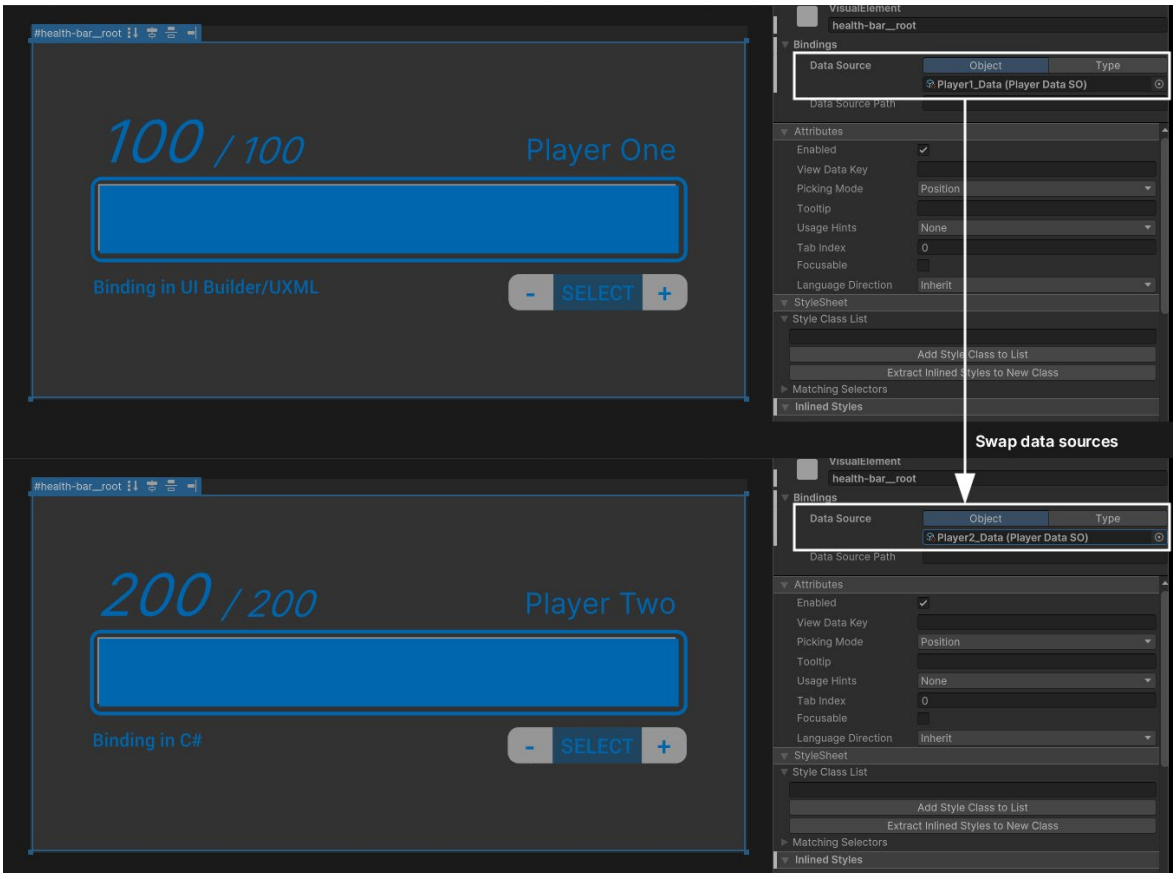
基本的な体力ゲージ

データソースをルートに設定すると、それが子要素のデフォルトデータソースとして表示されます。正しいデータソースパスを入力します。次の表は、UI 要素のプロパティと ScriptableObject を結合するデータバインディングを示しています。

UI 要素	UI 要素プロパティ	Bound プロパティ	注記
`health-bar__player-name`	`text`	`PlayerName`	プレイヤー名を表示
`health-bar__current-health`	`text`	`CurrentHealth`	現在の体力値を表示
`health-bar__max-health`	`text`	`MaximumHealth`	体力の最大値を表示
`health-bar__progress`	`style.width`	`Progress`	ゲージ幅を動的に調整

データバインディングが完了すると、体力ゲージがリアルタイムで更新され、ラベルと、プレイヤーの体力を示すプログレスバーが表示されます。

データソースの切り替えは簡単です。新しい ScriptableObject アセットを割り当てるだけで、同じバインディングを維持しながら UI に新しい値が自動的に反映されます。



データソースを切り替えると、データバインディングが更新されます。

UI Builder でデータバインディングを設定すると、UXML ファイルに直接追加され、バインドされた要素ごとに <Bindings> ブロックが作成されます。

次に、health-bar__player-name 要素の text プロパティを PlayerName プロパティにバインディングした際の UXML を示します (読みやすくするために一部の属性を省略しています)。

```
<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>
```

経験豊富なユーザーなら、UXML でこれらのバインディングを直接作成することもできます。これをコードで行えば、多数のバインディングを扱う場合に、正確な制御と迅速な編集が可能です。また、手動で記述した UXML はバージョン管理の差分をより明確にし、マージ競合の解決や変更の追跡を容易にします。

C# でのデータバインディング設定

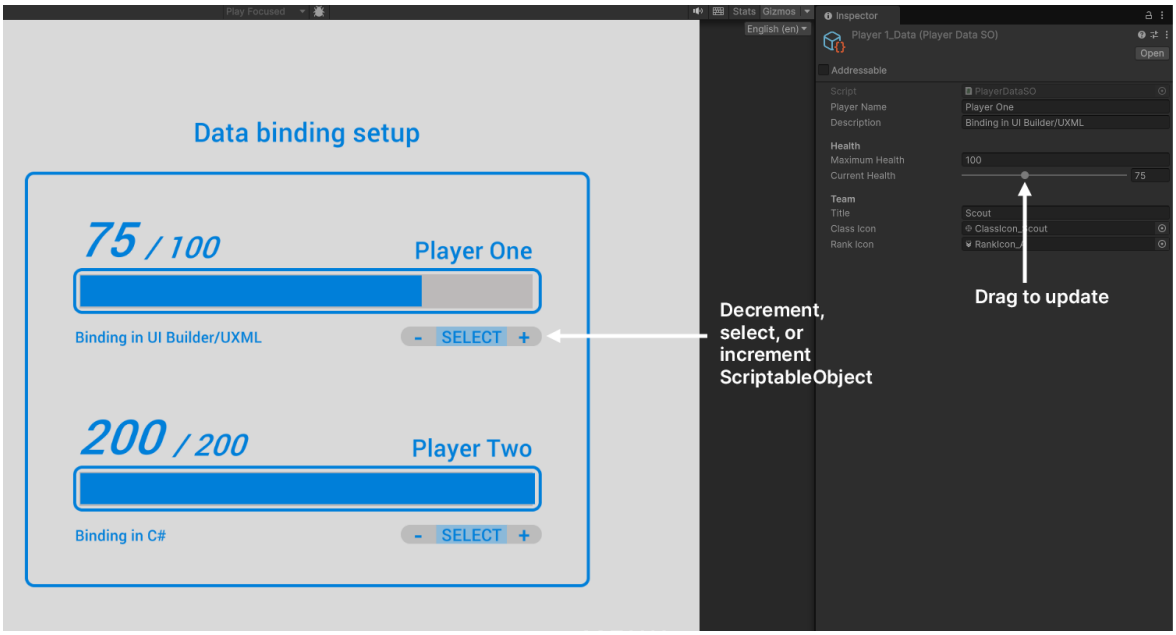
UI Builder は、静的データ (事前定義済みの ScriptableObject アセットなど) を使用したプロトタイピングに最適ですが、ランタイムデータでは C# での動的な処理が必要になることがよくあります。このコード例は、デモシーンで Player Two の体力ゲージがどのように機能するかを示しています。

```
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` スクリプトは、`OnEnable` のメインコントローラースクリプトから呼び出される `Initialize` メソッドでこれを処理します。

- まず、`health-bar__player-name` 要素のクエリを実行します。次に、ScriptableObject データをソースとして割り当てます。
- そして、SetBinding メソッドで、`text` プロパティを新しい DataBinding インスタンスにバインドし、`dataSourcePath` および `bindingMode` パラメーターを設定します。

前述の表の 4 つのバインディングはすべて、同様に設定されます。ScriptableObject スライダーまたはカスタムエディタープロパティドロワーを使用して、`CurrentHealth` を調整します。デモには、ScriptableObject を増分、減分、選択するためのプレイテストコントロール (+、-、Select) が含まれています。体力ゲージは、変化があると動的に更新されます。

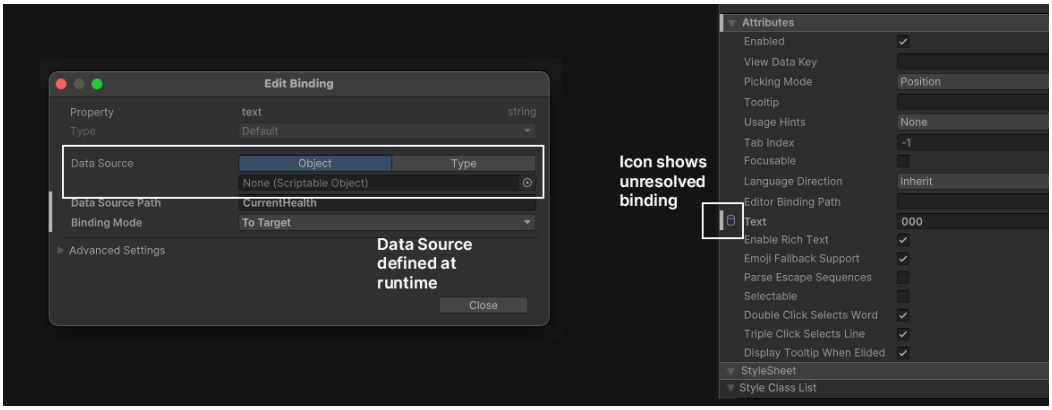


HealthBar は CurrentHealth 値と同期します。

未解決データバインディングのワークフロー

Unity 6 は、UI Builder のビジュアルの設定と柔軟なスクリプティングを融合したハイブリッドデータバインディングワークフローもサポートしています。

UXML でデータソースをハードコーディングする代わりに、Data Source Type を指定して、実際のデータソースを未解決のままにすることができます。UI Builder では、これらの不完全なバインディングは線画アイコンで示されます。これは、パスと型が設定されているものの、データソースがまだ割り当てられていないことを意味します。



未解決のデータバインディングは線画アイコンで示されます。

ランタイムでは、1 行のコード だけでデータソースを割り当てることができます。例:

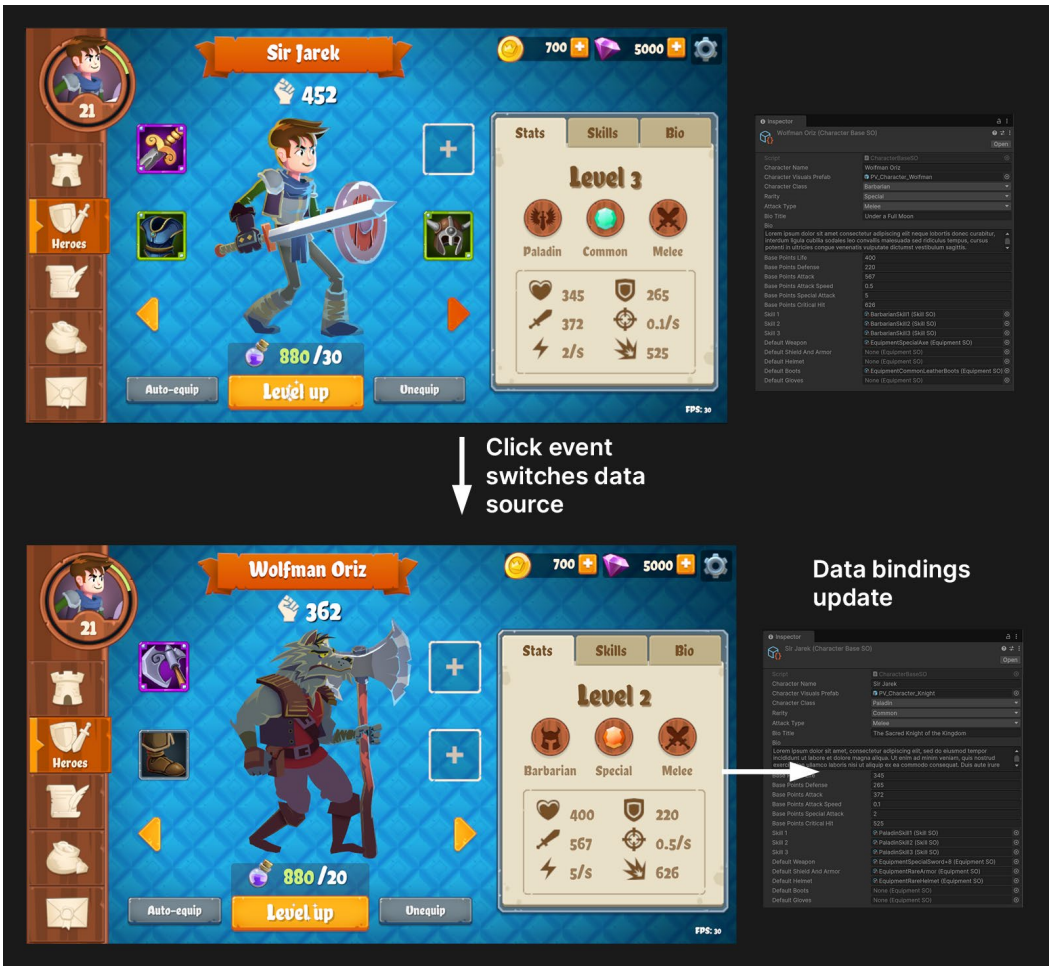
```
myElement.dataSource = myNewDataSource;
```

ここで、`myNewDataSource` を `myElement` に割り当てると、UXML で定義されたプレースホルダーのバインディングが解決され、UI が自動的に更新されます。これにより、`SetBinding` 呼び出しの繰り返しがなくなり、UXML の柔軟性が維持されます。

例えば、『*Dragon Crashers*』サンプルでは、UXML でデータパスを事前定義し、ランタイムに実際のデータソースを設定しています。

UI の ”次へ” や ”最後” のボタンをクリックすると、現在選択されているキャラクターがデータソースとして設定されます。データソースを変更しても、UXML に変更は必要ありません。

新しいデータソースが設定されると、未解決のバインディングに正しいキャラクターの統計が表示されます。



『Dragon Crashers』サンプルのデータソースの更新

注: UXML ファイルが特定のデータソース (例えば、`data-source="PlayerDataSO.asset"`) を設定すると、バインディングは固定になり、ランタイム時に変更できなくなります。ランタイム時の変更を有効にするには、`data-source` 属性を空のままにするか、`data-source-type` を使用します。

このハイブリッドデータバインディングワークフローの例については、ListView へのリストのバインディングを参照してください。

型変換

Unity 6 の型変換機能を使用すると、生データをユーザーに理解しやすい形式に変換して UI に表示できます。データソースと UI の間の仲介者として機能し、ユーザーにとってより直感的な形式にデータを変換します。

例えば、型換算機能を使用して、ラジアンを度に変換したり、体力値を体力ゲージの色に変換したりできます。これにより、UI が明確でわかりやすい形式で情報を表示できます。型変換は、多くの手動の変換ロジックを必要とせずに実行します。

Unity 6 では、次の 2 種類の変換機能がサポートされています。

- **グローバルコンバーター:**特定の型変換が必要なバインディングに適用します。例えば、グローバルコンバーターは Float の体力率を色値に変換したり、`Color` オブジェクトを`StyleColor` 型に変換したりして、UI 全体で一貫した動作を確保します。
- **バインディングごとのコンバーター:**特定のデータバインディングに適用することで、より詳細な制御が可能になります。

例: 値を色に変換

プレイヤーの体力に応じて色が変わる体力ゲージは、型変換によるバインディングの使用例です。プレイヤーの現在の体力をカラーグラデーション (緑色が体力フル、黄色が体力低、赤色がクリティカル) にマッピングすることで、プレイヤーはゲームプレイ中にステータスを簡単に把握できます。

QuizU プロジェクト内の **DataBindingDemo** シーンでこの動作を確認できます。

HealthDataConverter の設定

DataBindingDemo シーンでは、`HealthBarWithConverter` クラスは静的な `HealthDataConverter` の機能を使用して、いくつかの DataConverter を登録しています。

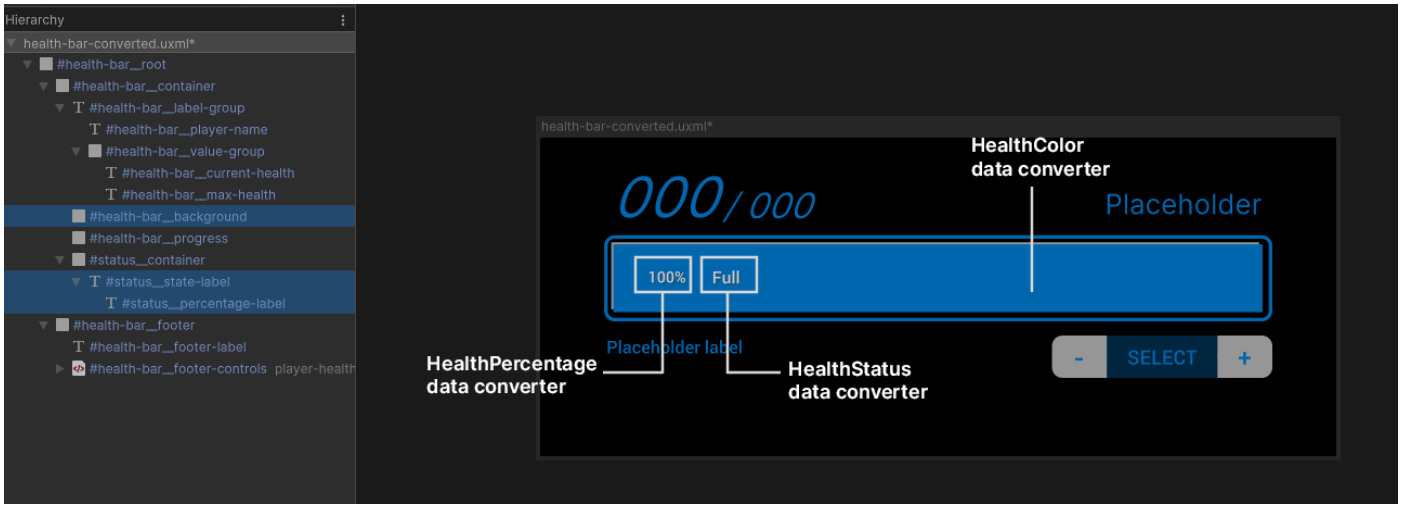
- 体力率は、体力ゲージのカラーグラデーションに作用し、緑色 (体力フル) から赤 (体力危険) に変わります。
- ラベルは、パーセント文字列として数値を表示できます (例: "75%")。
- 別のラベルで同じ体力率を "Full"、"Mid"、"Critical" などのステータスラベルにマップできます。

これは HealthDataConverter クラスの一部です。

```
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);
   }

   // …

}
```

上記のロジックで HealthColor ConverterGroup を作成し、Float の体力率 (0 から 1) を、赤 (体力低) と緑 (体力フル) の間の対応する StyleColor 値に変換します。

HealthDataConverter クラスには、2 つのラベルのコンバーターも含まれます。これらは、PlayerDataSO の HealthPercentage プロパティを、フォーマットされた文字列値として表すことができます。複数のコンバーターを 1 つの ConverterGroup にバンドルすることもできますが、このデモでは読みやすくするために個別の ConverterGroup に分けています。



UI Builder で型変換機能を使用します。

HealthBarWithConverter の使用

HealthDataConverter クラスには実際の機能が含まれていることに注意してください。HealthBarWithConverter は単純です。

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

次の点に注意してください。

— `UnityEditor.InitializeOnLoadMethod` を使用すると、ConverterGroup が登録されて UI Builder で使用できるようになり、それをエディターで確認して適用できます。

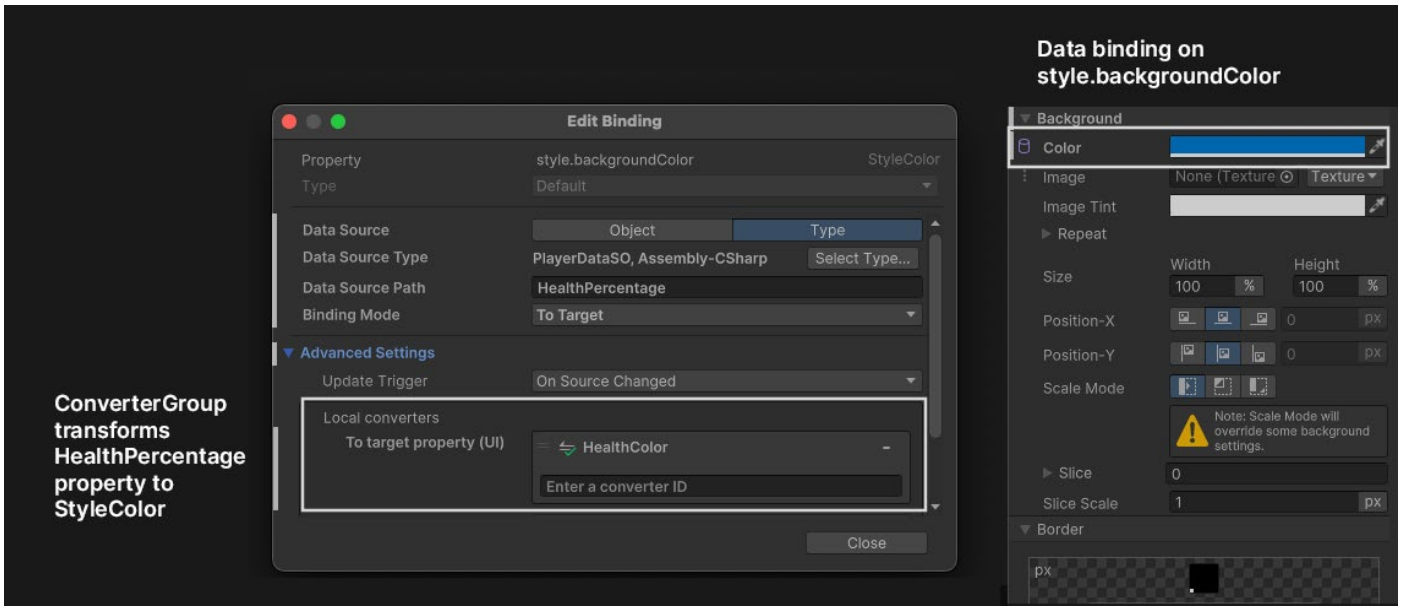
— `RuntimeInitializeOnLoadMethod` で、ゲーム実行時のランタイムにおいて ConverterGroup を利用できるようにします。

`#if UNITY_EDITOR` プリプロセッサーディレクティブで、コードがエディターで実行されるかゲームプレイ中に実行されるかに応じて、適切なメソッドが適用されるようにします。

UI Builder での DataConverters の適用

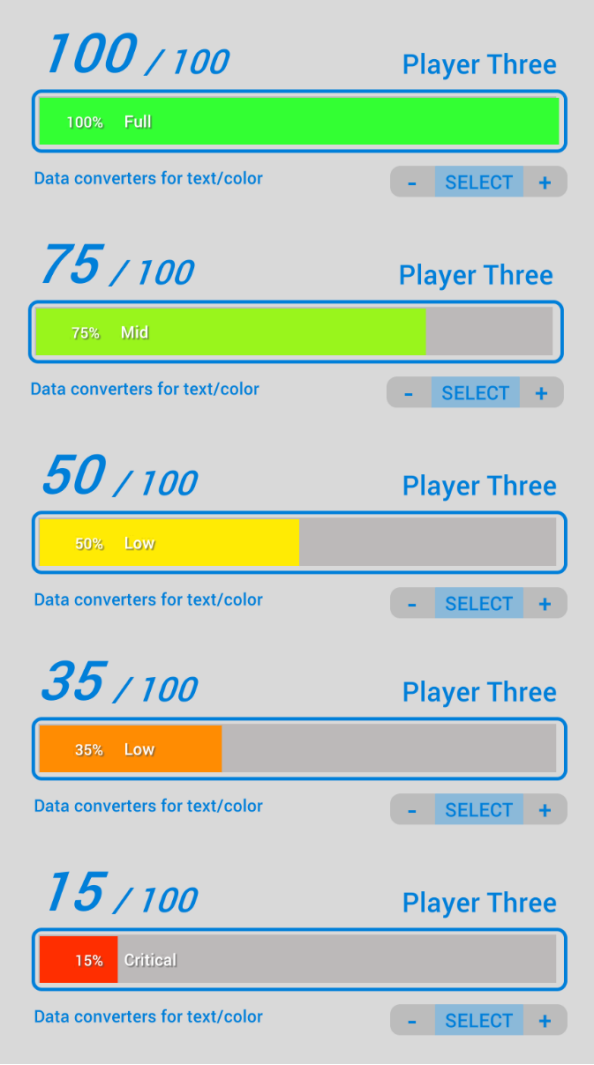
DataConverter を登録すると、その変換を必要とするあらゆるバインディングに適用できます。UI Builder で直接使用するには、以下の手順を実行します。

1. UXML ファイルを開き、プログレスバー要素を選択します。QuizU プロジェクトでは、**RuntimeDataBinding.uxml** ファイルを開いてその設定を確認できます。
2. Data Source を `PlayerDataSO` ScriptableObject に設定します。
3. プログレスバーの `backgroundColor` スタイルプロパティを `HealthPercentage` データパスにバインドします。
4. `HealthColor` ConverterGroup を使用して、体力率の値をプログレスバーの背景色に変換します。



体力ゲージのデータバインディングを設定します。

5. PlayerDataSO ScriptableObject の `CurrentHealth` 値をドラッグすると、体力ゲージの色が更新されるようになりました。グラデーションは、緑色 (フル) から黄色 (中)、オレンジ色 (低)、赤 (危険) へと滑らかに変化していきます。



HealthColor コンバーターでプログレスバーの色を変えます。

このグローバルな DataConverter は、Float 値をこのカラーグラデーションに変換する必要があるアプリケーション内のあらゆる場面で利用できるようになりました。

推奨ガイド

型変換機能を使用する際は、以下のヒントを参考にしてください。

- **割り当ての最小化:**変換のデリゲートを (特に頻繁な操作の場合に) 軽量に保ち、不要なパフォーマンスオーバーヘッドを回避します。
- **簡素化:**素早く変換できるよう、シンプルで焦点を絞ったコンバーターを記述します。複雑なロジックやリソースを消費するロジックを埋め込むことは避けます。
- **変換をデータソースに統合:**データソース自体で基本的な変換を扱います (例: ScriptableObject プロパティで体力率を事前にフォーマット)。UI バインディング固有の変換用に DataConverter を予約します。

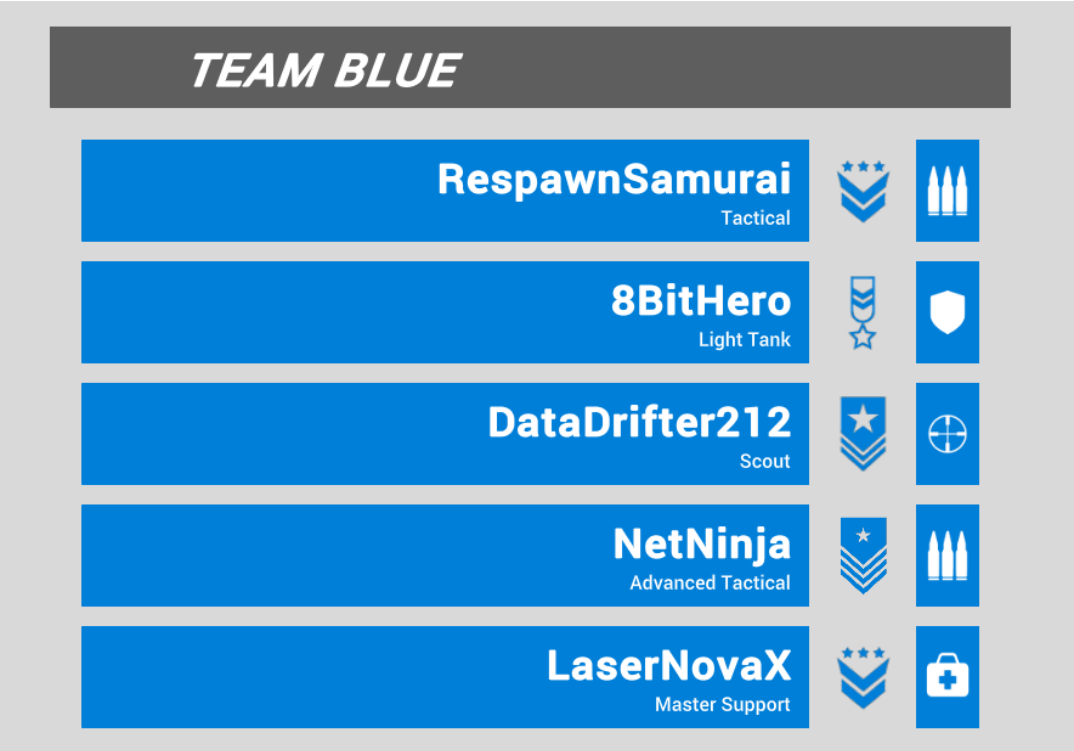
例: ListView へのリストのバインディング

アプリケーションでは、ゲーム UI に応じて、収集アイテムのインベントリ、目標を記録するクエストログ、プレイヤーのランキングを表示するリーダーボードなど、さまざまなデータコレクションを画面に表示する必要があります。

ListView は、スクロール可能ですっきりとしたインターフェースを提供し、情報の管理と表示を容易にします。Unity 6 では、ランタイムデータバインディングでこのプロセスが合理化され、データが変化したときに UI を手動やカスタムスクリプトで更新する必要がありません。

以前のバージョンの Unity では、ListView を設定するには、リストを生成し、データ変化時の更新を処理するカスタムコードを記述する必要がありました。Unity 6 では、ListView をデータソースに直接バインドし、UI の変更を自動的に追跡して反映できます。

デモシーンには、PlayerDataSO ScriptableObject のリストにバインドする簡単な ListView が含まれています。これにより、マルチプレイヤーゲームのロビーやハイスコアリーダーボードで見られるようなインターフェースを作成できます。



TeamList は、リストを ListView にバインドします。

ランタイムデータバインディングを使用すると、ListView を ScriptableObject などのデータソースに直接リンクできます。ListView はデータの変化を自動的に追跡し、設定とメンテナンスを効率化します。

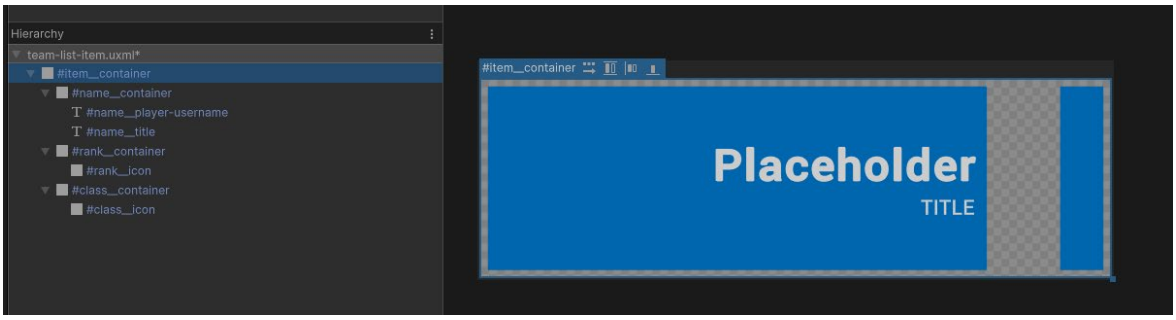
ListView をリストにデータバインディングするには、未解決のバインディングをいくつか設定してから、ランタイム時にデータバインディングを完了する必要があります。

リストとテンプレートの設定

以下の手順に従って、ListView でデータバインディングの準備を行います。

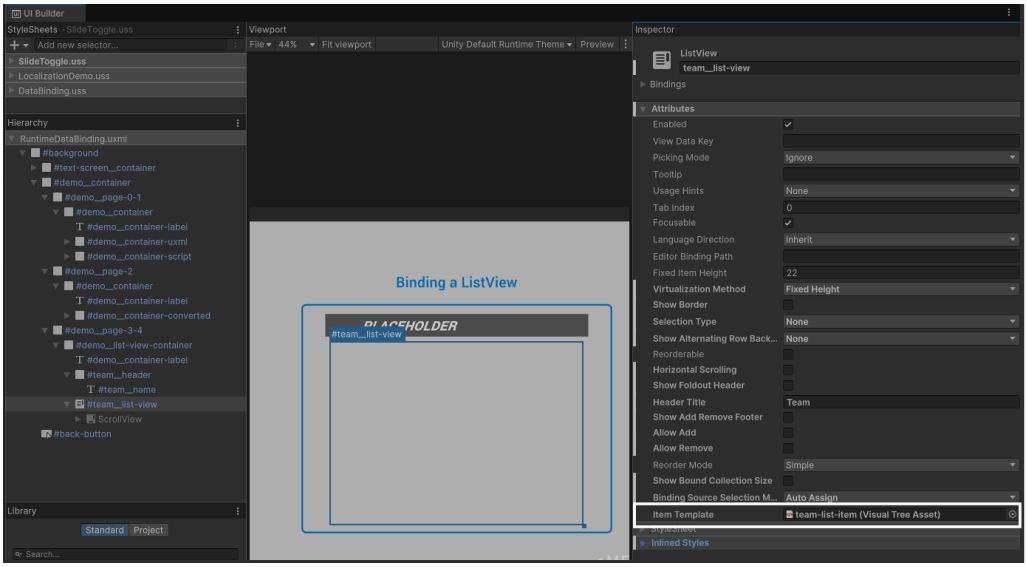
1. **データソースの定義**:ListView にはデータのリストが必要です。このデモでは、TeamSO ScriptableObject が PlayerDataSO オブジェクトのリストを保持しています。そのリスト内の各項目は、ListView の行に対応しています。

2. **UXML アイテムテンプレートの作成**:UI Builder で、1 つのリスト項目の外観を定義する UXML テンプレート (VisualTreeAsset) をデザインします。例えば、デモの `team-list-item` テンプレートには、プレイヤー名といくつかの Texture2D プロパティが含まれています。データソースを直接参照するのではなく、Data Source Type と Data Source Path を UI Builder で設定します。これでバインディングが未解決になり、あとでランタイム時に完了できるようになります。



UI Builder でビジュアルツリーアセットをデザインします。

3. **ListView をメインユーザーインターフェースに追加**:別の UXML ファイルで、プレイヤーのリスト全体を表示する ListView 要素を追加します。アイテムテンプレートを ListView の **Item Template** として割り当てます。この時点では、ListView は各行がどのように見えるかを認識していますが、どのデータソースを使用すべきかはまだわかりません。



Add VisualTreeAsset as Item Template

テンプレートを ListView に追加します。

デモシーンの ListView では、いくつかの基本設定 (上記参照) のみを使用しています。さらに高度な機能については、ListView の公式ドキュメントを参照してください。

ランタイムのバインディングの完了

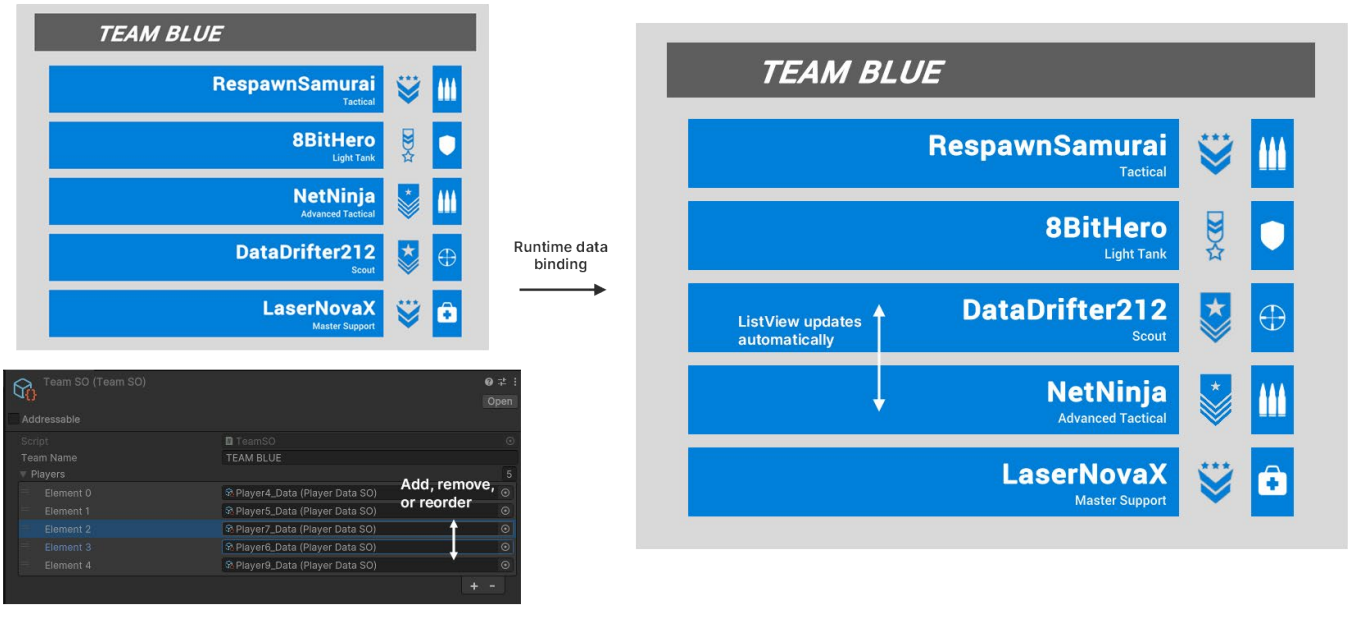
ランタイム時に、実際のデータソースを提供することで、単純な `TeamList` スクリプトによりバインディングが完了します。以下の行で、未解決だったバインディングが完了します。

```
// データソースを設定する
m_ListView.dataSource = m_TeamData;

// "itemsSource" を Players リストにバインドする
m_ListView.SetBinding("itemsSource", new DataBinding
{
    dataSourcePath = new PropertyPath("Players")
});
```

ここでは、`m_TeamData` (TeamSO のインスタンス) が ListView に割り当てられています。`SetBinding` を呼び出すと、`Players` プロパティが `itemsSource` に関連付けられます。これにより、ListView で UI の行を生成できます。

これらのバインディングは UXML ではランタイムまで未解決であるため、各リスト要素を個別に接続する必要はありません。UI Toolkit でこれらのバインディングを独自に解決し、すべてのリスト項目のデータを入力します。

データソースのリストに加えられた変更 (プレイヤーの追加、削除、再配置など) は、スクリプトを追加することなく、すぐに UI に表示されます。



Player リストの変更が UI に反映されます。

このハイブリッドなデータバインディングアプローチにより、定型コードの大量の繰り返しを減らすことができることを覚えておいてください。UXML でデータパスのプレースホルダーを設定することにより、実際のデータソースの割り当てをランタイム時まで延期できます。

データモデルを変更した場合、バインディングロジック全体を書き換える必要はありません。起動時の 1 回の更新で、UI を新しいソースに再接続できます。

ListView のリストへのバインディングについては、こちらの ドキュメントページ を参照してください。

データバインディングの最適化

効率的なバインディングは、高いパフォーマンスを発揮する UI を維持するのに役立ちます。冗長なバインディングや過剰なバインディングは、システムに過負荷を与え、不要な更新やパフォーマンスの低下につながるおそれがあります。これは、インターフェースが複雑であったり、リソースを消費したりする場合、特に重要です。

デフォルトでは、ランタイムバインディングシステムは UI 要素をフレームごとに更新します。これは小規模なアプリケーションでは応答性がありますが、バインディングが多くなるとパフォーマンスのボトルネックになるおそれがあります。

このセクションでは、大規模プロジェクトのデータバインディングの効率を改善する方法について説明します。

値型の管理

データソースで値型 (`int`、`float`、`struct` など) を使用する場合は、ボックス化のコストに注意してください。`dataSource` プロパティはオブジェクトとして動作するため、値型からの頻繁な変換はオーバーヘッドが増えるおそれがあります。

これを減らすには、値型のプロパティを使用するときに、不要なバインディングや冗長な更新を最小化します。

オーバーヘッドの最小化

同じ要素を複数回更新したり、ほとんど変更されないデータを追跡したりするバインディングを特定することから始めます。これらのバインディングを統合または削除して、不要な動作を減らします。可能な場合は、複雑な階層構造ではなく、フラットでシンプルなデータ構造を使用します。これにより、頻繁なデータ検索に起因するパフォーマンスのボトルネックを回避できます。

計算負荷の高い値については、事前に計算しておりたり、キャッシュしたりすることを検討します。これらの事前計算された値にバインディングすると、バインディングシステムの計算負荷が減り、再計算の繰り返しを避けることができます。

頻繁な更新を必要とする要素にバインディングするようにしてください。定常的に同期する必要のない要素については、不要なバインディングを削除し、値を直接割り当てるか、イベントによってトリガーされた場合にのみ更新してください。

更新トリガーの使用

バインディングは、UI がデータソースと同期する頻度を決定する 更新トリガー に基づいて更新されます。これにより、パフォーマンスと応答性のバランスを取ることができます。以下に示すオプションによって、バインディングの更新頻度が決まります。

- **フレームごと:**継続的に更新されます。体力ゲージのように定常的な更新を必要とする要素に使用します。
- **変更の検出時:**データソースが変更されると更新されます。検出できない場合は、フレームごとに更新されます。例えば、観測可能なデータに依存する統計パネルやインベントリリストに使用します。
- **ダーティとマークされている場合:**更新頻度が低いシナリオでは、`MarkDirty` でバインディングをダーティと明示的にマークすることで、不要な更新サイクルを回避できます。この更新トリガーは、特定のコンテキストでのみ変化する設定メニューなどの要素に対して機能します。

更新トリガーを各 UI 要素のニーズに合わせることで、応答性と効率のバランスを取ることができます。

バージョン管理と変更追跡

不要な更新を減らすために、バージョン管理と変更追跡をデータソースに統合できます。

次の 2 つのインターフェースがデータバインディングの効率化に役立ちます。

- IDataSourceViewHashProvider:バージョンハッシュを使用して全体的な変更を追跡し、データソースが変更されたときにのみ更新をトリガーします。これは、更新頻度が低い静的または半静的データの場合に便利です。
- INotifyBindablePropertyChanged:これはプロパティレベルで変更を追跡し、影響を受けるバインディングが確実に更新されるようにします。これにより、細かい制御が可能になります。

これらのインターフェースをデータソースに追加します。個別に使用したり、組み合わせて使用したりすることで、更新を細かく制御できます。使用法とベストプラクティスについては、こちらの ドキュメントページ を参照してください。

ヒント: その他の UI Toolkit 最適化のヒント

UI Toolkit の最適化に関するこの Unite 2024 での講演 では、連鎖的なドローコールの実装とバッファサイズの影響、動的アトラス化のベストプラクティス、カスタムシェーダーや 3D UI などの制限への対応といったトピックについて学ぶことができます。

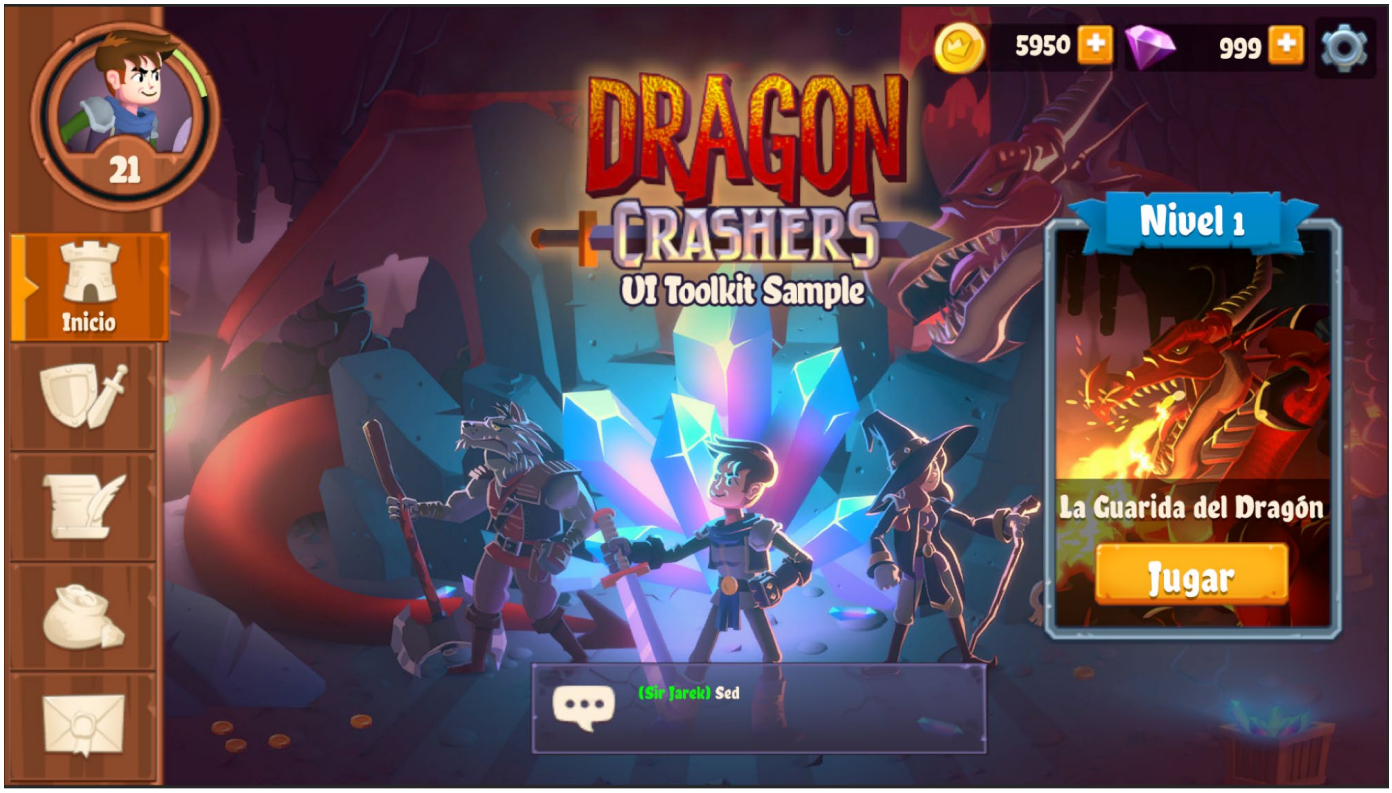
ローカライゼーション

UI をローカライズすることで、アプリケーションをあらゆる言語で直感的かつ親しみやすいものにして、ゲームをグローバルなオーディエンスに届けられるようになります。

Unity 6 は、Localization パッケージを UI Toolkit に統合しており、このプロセスが簡略化されています。この統合により、プレイヤーがどこにいても、地域固有のコンテンツを楽しめます。

Unity のローカライゼーションの鍵は Locale クラスです。このクラスで特定の言語を表し、通貨や数値の書式設定など、地域固有の細部を管理します。

UI Toolkit でローカライズを設定する方法の簡単な例を見てみましょう。この設定により、アプリケーションは選択したロケールに基づいてコンテンツを動的に調整できます。



UI Toolkit サンプル – 『Dragon Crashers』におけるスペイン語ローカライゼーションの例

仕組み

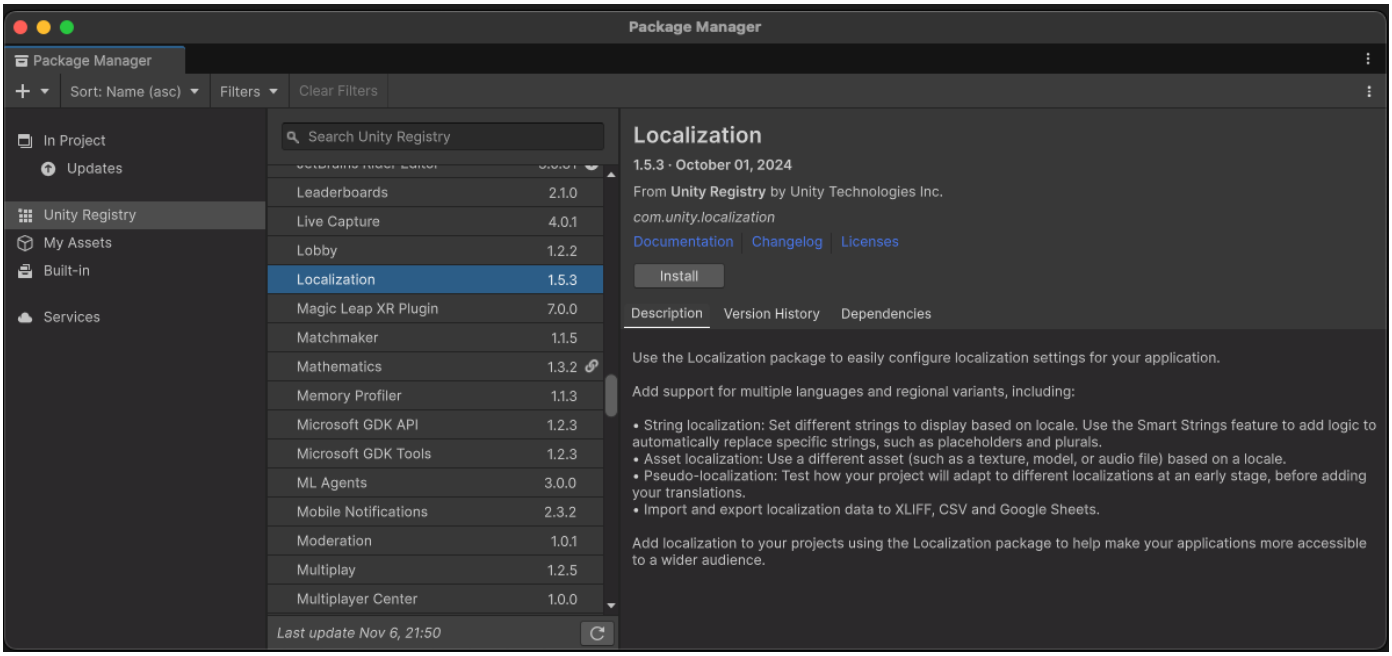
Localization パッケージの主要機能をいくつか紹介します。

- **文字列ローカライゼーション:**LocalizedString クラスを使用すると、ランタイム時にロケールを切り替えた際に自動更新される文字列を管理できます。Smart Strings を使用すると、プレースホルダーの追加、複数形の扱い、言語固有のその他のニュアンスの調整を行うことができます。
- **アセットローカライゼーション:**ロケールに基づいてテクスチャやその他のアセットを切り替えられるため、単純なテキスト以外の地域固有のコンテンツも作成できます。
- **データバインディング:**Localization パッケージは UI Toolkit のランタイムデータバインディングと統合され、UI 要素を String Table と Asset Table にリンクします。データ、ロケール、ロード状態の変更により、自動更新が行われます。
- **String Table と Asset Table の管理:**String Table と Asset Table には、テキストやその他のアセットをロケール固有の形式に変換するためのキーと値のペアが格納されます。一元的な UI インターフェースにより、プロジェクト内のローカライズされたすべてのテキストとアセットの概要を確認できます。
- **ロケールの切り替え:**アプリケーションを再起動することなく、言語をリアルタイムで切り替えることができます。ランタイム時に新しいロケールを選択すると、UI がただちに更新され、変更が反映されます。

テキストの長さや形式の変更に対応するときは、UI Toolkit の FlexBox コンテナと自動サイズ調整要素を活用することを忘れないでください。これにより、さまざまな言語をサポートする際の UI の応答性が向上します。

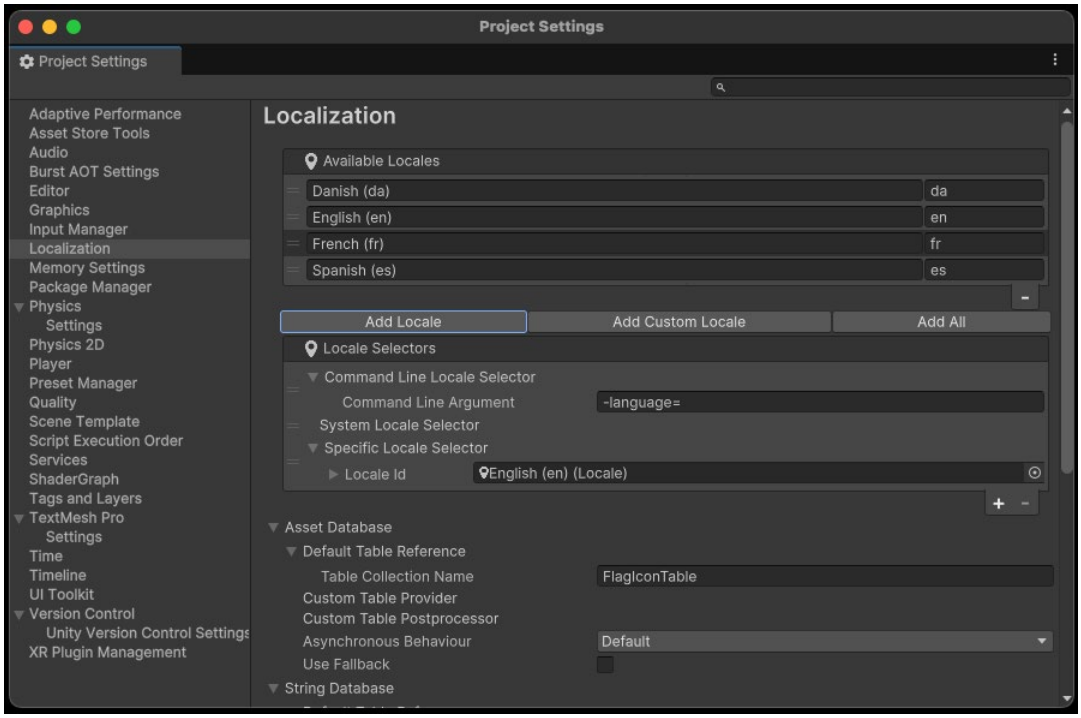
ローカライゼーション設定

UI Toolkit で Unity の Localization パッケージの使用を開始するには、以下の基本手順に従ってローカライズされたコンテンツを設定し、UI Builder で UI 要素にバインドします。



Package Manager から Localization パッケージをインストールします。

1. **Localization Settings の設定:**Package Manager から Localization パッケージをインストールし、**Project Settings > Localization** に移動して、ローカライズされたすべてのアセットを管理する Localization Settings アセットを作成して設定します。

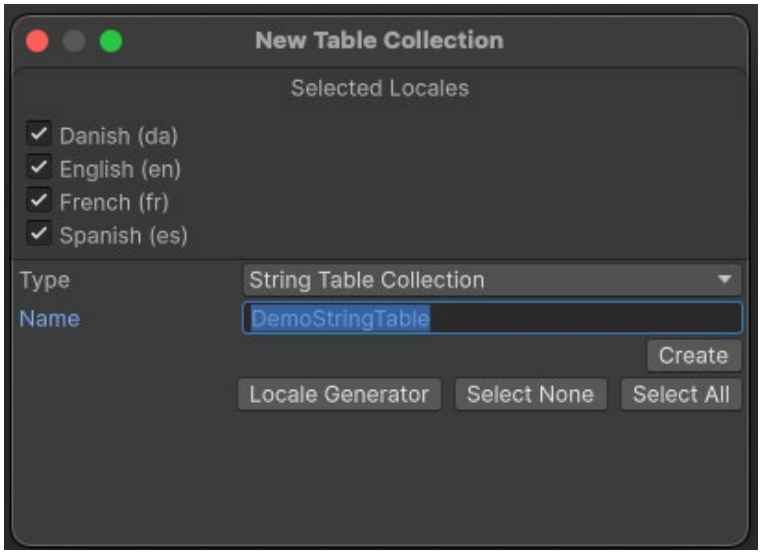


Project Settings で Localization Settings を作成して設定します。

2. **ロケールの作成:** Locale Generator を使用して、プロジェクトがサポートする言語と地域を定義します。これにより、固有の 2 文字のコード (英語の場合は "en"、フランス語は "fr"、スペイン語は "es"、など) で識別されるアセットがロケールごとに作成されます。アプリケーション起動時に使用するデフォルトロケールを設定します。

ロケールを追加します。

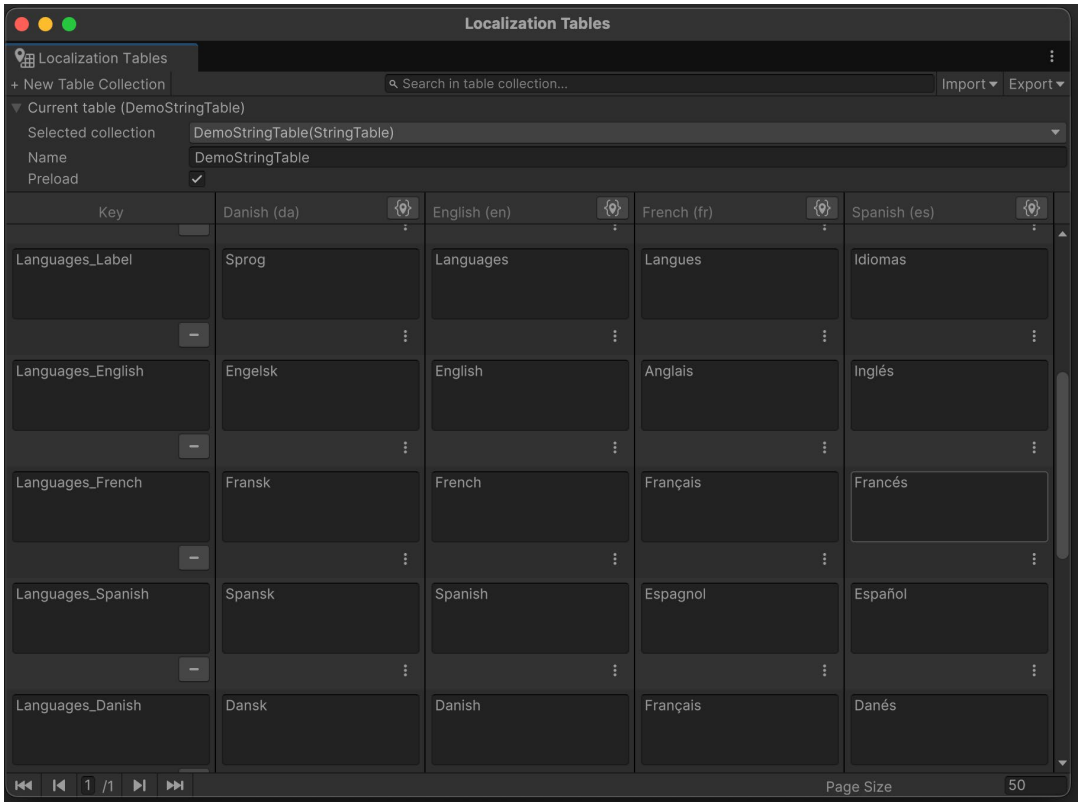
3. **String Table と Asset Table の作成:** String Table (文字列テーブル) を使用して、各ロケールのテキストエントリーを格納します。ラベル、ボタン、ドロップダウンオプションなどの UI テキスト要素のエントリーを追加します。



String Table (文字列テーブル) と Asset Table (アセットテーブル) を作成します。

4. Localization Tables ウィンドウ (**Window > Asset Management > Localization Tables**) で、UI 内の各テキスト要素のキーと値のペアを追加します。各キーは特定のテキスト項目 (ラベルやボタンなど) を表し、各値は各ロケールでその項目に対して翻訳されたテキストです。

5. 画像やゲームオブジェクトなどの地域固有のアセットを Asset Table に格納します。例えば、各ロケールを表すアイコンにスプライトやテクスチャを追加できます。キーごとに、各ロケールに固有のアセットをリンクして、地域や文化の好みを反映させます。

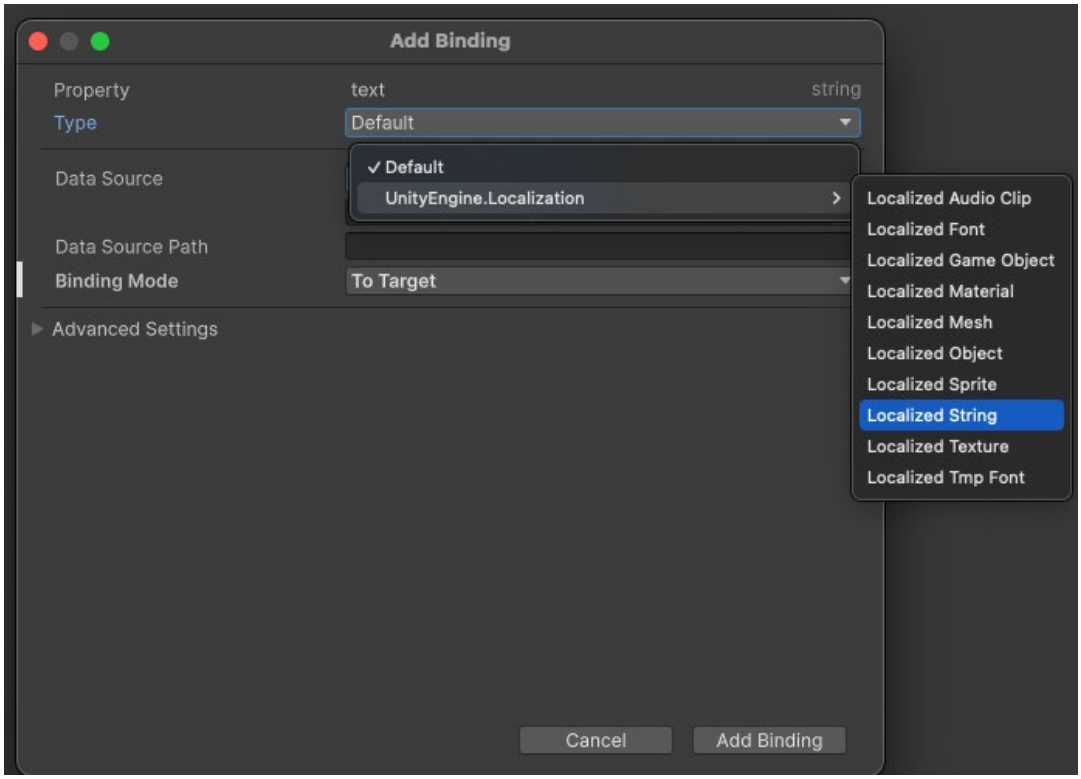


キーと値のペアは、ローカライズする各要素を表します。

6. **UXML インターフェースの定義:**UI Builder を使用して、ボタン、ドロップダウンフィールド、ラベルなどの要素を含む UXML ファイルを作成します。デモシーンでは、この UXML でローカライズ可能な要素をいくつか示しています。テキストフィールドの場合は String Table のエントリーを使用します。テクスチャなど、テキスト以外のフィールドについては Asset Table を使用します。

デモシーン
QuizU プロジェクトに含まれる **LocalizationDemo** シーンで、ローカライゼーションのサンプル実装を確認できます。

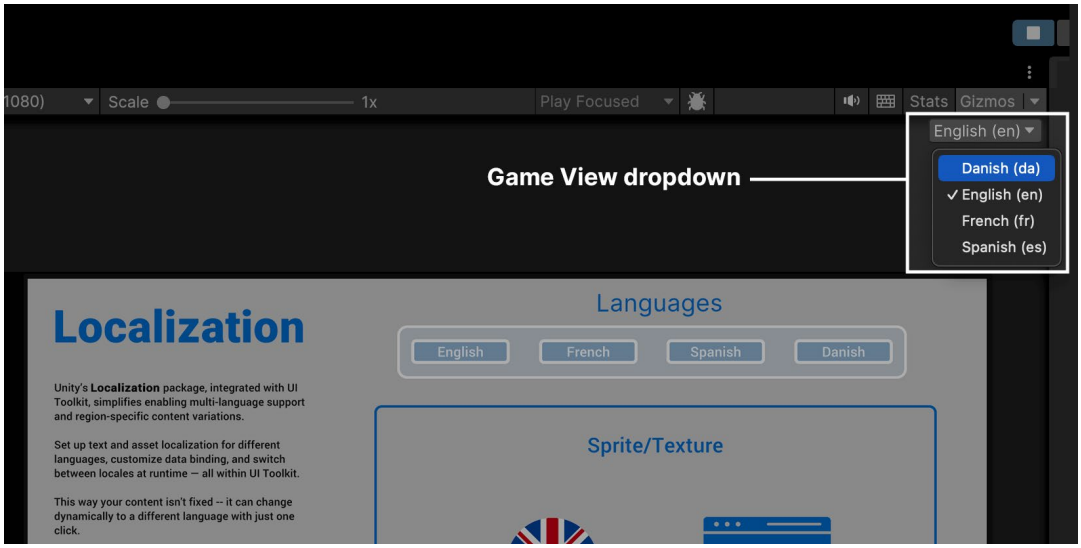
これにアクセスするには、ランタイムでメインメニューに移動し、**Demos > Localization** を選択するか、ブートローダーを無効にした (**Quiz > Don't Load Bootstrap Scene on Play**) 直後に **LocalizationDemo** シーンをロードします。



ローカライズされた文字列やアセットに UI Builder でデータバインディングを追加します。

7. **UI Builder で UI 要素にデータをバインド:**UI Toolkit のランタイムデータバインディングシステムの能力が発揮されます。UI Builder で、ローカライズする要素を選択します。Inspector パネルを開き、コンテンツフィールドの **Add Binding** を選択します (ラベルの場合は `text`、画像の場合は `backgroundImage` など)。

   バインディングの種類として **LocalizedString** またはローカライズされたその他のアセットを選択し、String Table または Asset Table 内の対応するエントリーにリンクします。他の要素をローカライズする必要がある場合は、テーブルにその他のエントリーを追加します。



ローカライゼーションをプレビューするには、ゲームビューのロケールドロップダウンを使用します。

8. テストするには、ゲームビューのロケールドロップダウンを使用してさまざまな言語で UI をプレビューし、各ロケールで要素が正しく表示されるか確認します。

以上が基本設定です。この例では、ボタンとラベルのテキストプロパティを、設定されている他の言語に切り替えることができるようになりました。UI 全体をローカライズするには、すべてのテキストで String Table に独自のエントリーがあることを確認します。

Localization パッケージはコンテンツの整理方法に柔軟性があります。複数の String Table を使用して、大規模なプロジェクトをより管理しやすいセクションに分割したり、さまざまなエントリーを分類したりできます。次に、Asset Table を使用して、テクスチャやテキスト以外のその他のアセットをローカライズします。

UI Builder でローカライゼーションバインディングを追加した後、UXML ファイルにより UI 要素にローカライゼーションが直接組み込まれます。その結果、次のようなコードブロックが生成されます。ローカライズされた各プロパティは、String Table または Asset Table の特定のエントリーに関連付けられています。

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

この UXML は、UI 要素の text プロパティを文字列テーブルまたはアセットテーブルのエントリーにリンクするデータバインディングを設定します。

Localization Tables を更新するたびに、リンクされた UI 要素に、ローカライズされた最新のコンテンツが自動的に反映されます。

注:UI Builder でデータバインディングの作成を簡素化できますが、ローカライズされたコンテンツを細かく制御するために、経験豊富なユーザーであれば UXML を直接編集することもできます。

Localizatizon API の使用

エディターのゲームビューのロケールドロップダウンは、さまざまな言語をテストするのに役立ちますが、アプリケーションのビルドでは使用できません。最終的なアプリケーションでユーザーが言語を変更できるようにするには、ロケールを切り替えるための独自の UI を作成する必要があります。

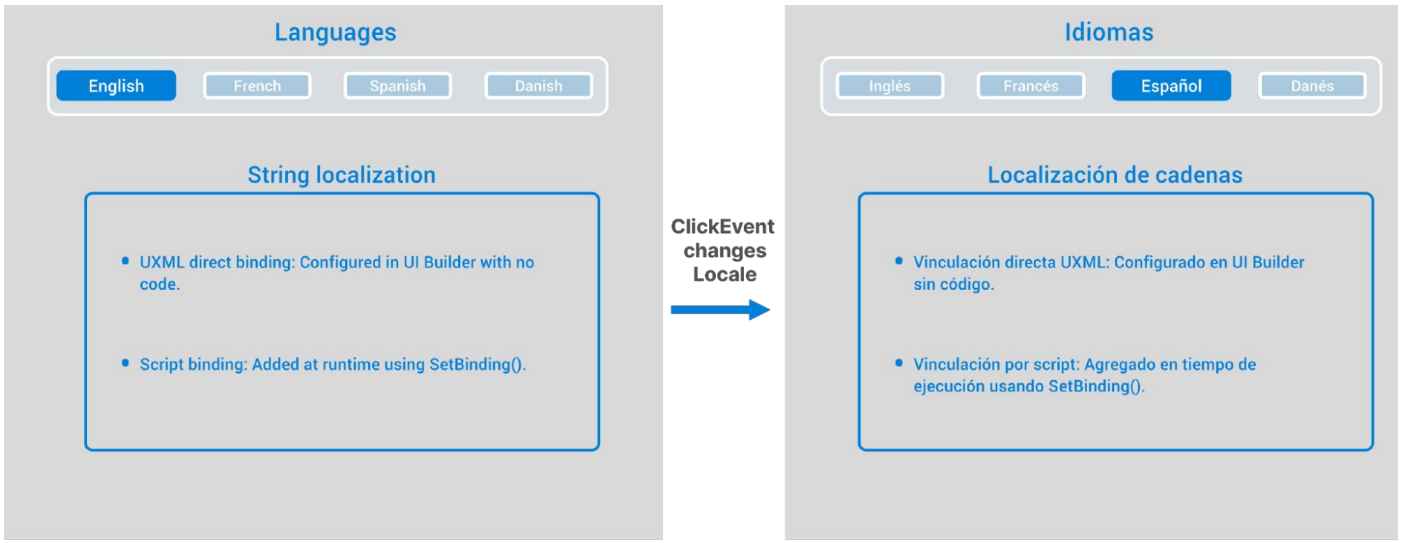
ロケールの選択

ロケールの 2 文字の識別子がある場合は、`LocalizationSettings` でアクティブなロケールを設定できます。次に、各ボタンのクリック マニピュレーターまたは `RegisterCallback<ClickEvent>` メソッドを使用して、このアクションをボタンに接続します。

サンプルプロジェクトの LocalizationDemo スクリプトは、1 つの実装例です。

```
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");
}
```

各ボタンで、ロケールをそのボタンが示すロケールに変更できます。英語、フランス語、スペイン語、デンマーク語のボタンを押すと、UI 内のテキストがランタイム時に変更されます。



ボタンでロケールを変更できます。

SetBinding の使用

UI Builder を使用したデータバインディングは対話的で簡単に設定できます。ただし場合によっては、ランタイム時にスクリプトを使用してバインディングを設定する必要があります。例えば、UI 要素を動的に作成する場合や、ゲームプレイ中にのみ利用できるデータに依存するバインディングを作成する場合です。

C# でデータバインディングを設定するには、ビジュアル要素で SetBinding メソッドを使用します。ラベルの `text` プロパティを StringTable の LocalizedString エントリーにバインドする方法を示します。

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


public class LocalizationDemo : MonoBehaviour
{
    // Inspector で設定
    [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);
    }
}
```

この設定では、Inspector で `m_LocalizedText` が DemoStringTable のエントリーに割り当てられます。このコードで、`m_LocalizedLabel` の `text` プロパティを指定した LocalizedString にリンクし、ロケールが変更されたときに自動的に更新できるようにします。

ロケール変更のリッスン

場合によっては、ロケールが変更されたときに、ローカライズされた文字列の更新以外の追加のアクションが必要になることがあります。ロケールが更新されるたびに何らかのロジックを実行する場合は、LocalizationSettings API の `SelectedLocaleChanged` イベントをリッスンします。

例を挙げてみましょう。

```
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)
   {
        // ロケール変更時に UI 要素の更新などのアクションを実行する
        Debug.Log($"Locale changed to: {newLocale.Identifier.Code}");
    }
}
```

この場合、ロケールが変更されるたびに OnLocaleChanged が呼び出され、ほかの要素を更新したり、カスタムロジックを実行したりできます。特に、翻訳されたテキストが現在のレイアウトにうまく収まらない場合は、このイベントハンドラーを使用して UI のプロパティやスタイルを調整します。

String Table の操作

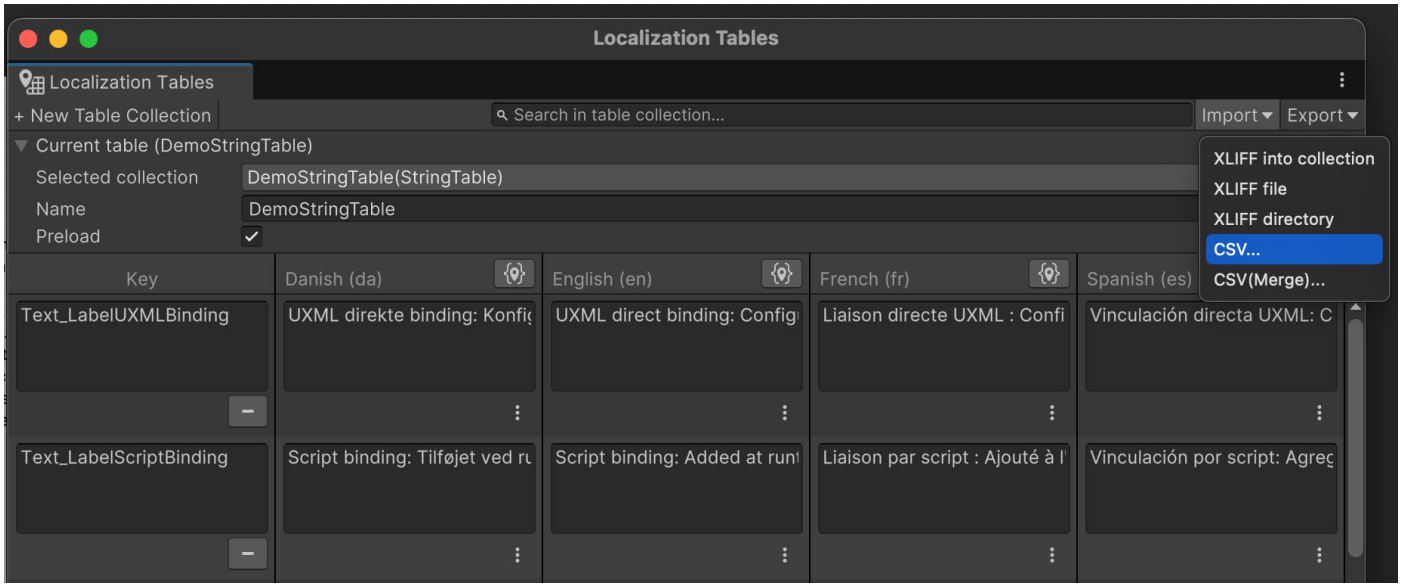
ローカライゼーション作業のほとんどは、UI とラベルのすべてのテキストベースの翻訳を扱う String Table が関与します。

Localization Tables ウィンドウを開き (**Window > Asset Management > Localization Tables**)、String Table Collection を作成または選択します。ここからロケールごとに、新しいエントリーの追加、一意のキーの定義、翻訳の入力を行うことができます。

文字列データのインポートとエクスポート

CSV ファイル

CSV (カンマ区切り形式) ファイルからデータをインポートして String Table に入力し、デザイナーが外部でテキストを設定するようにできます。プレーンテキスト形式のエントリーを編集するには、既存の String Table を CSV としてエクスポートします。



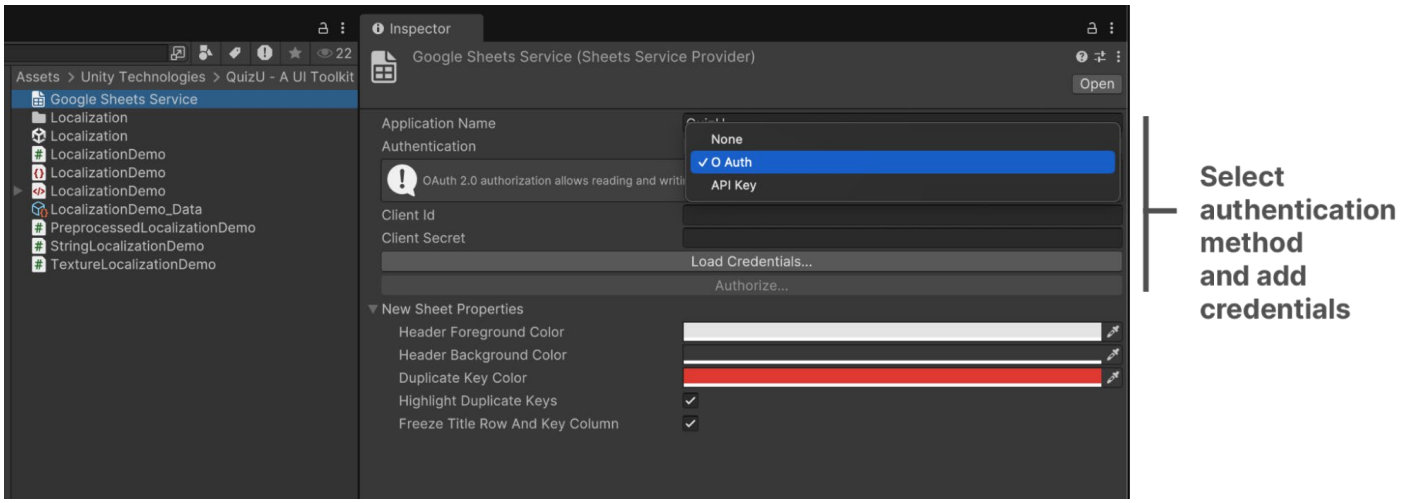
CSV ファイルをインポートまたはエクスポートします。

ファイルの更新後、Unity に再インポートすると、キーに基づいてエントリーが自動的に更新されます。

Google スプレッドシートの同期

プロジェクトを Google スプレッドシートに接続するには、Sheets Service Provider アセットを使用します。このアセットで認証を管理し、エディター内で直接新しいスプレッドシートを作成できます。

これを作成するには、**Assets > Create > Localization > Google Sheets Service** に移動します。

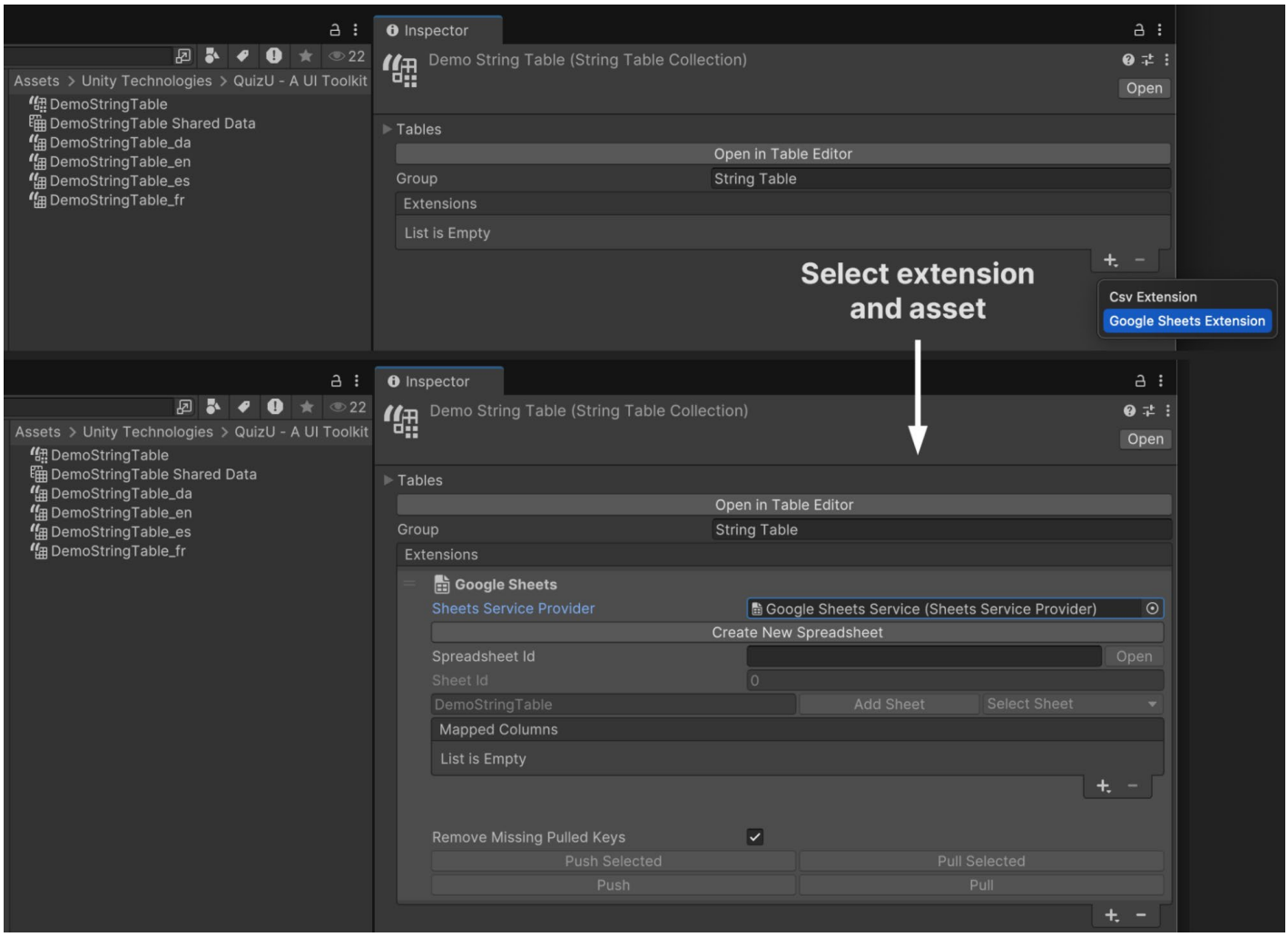


Google Sheets Service を作成して認証します。

Google Sheets Service には、2 つの認証オプションがあります。**OAuth** と **API キー** です。読み取りや書き込みのためにプライベートシートにアクセスする必要がある場合は、OAuth を使用します。パブリックシートからのみ読み取る必要がある場合は API キーを使用します。完全な読み取り/書き込みアクセスを行うには、Google に認証をリクエストする必要があります。詳細については Google スプレッドシートのドキュメント:認証のリクエストを参照してください。

String Table Collection を Google スプレッドシートにリンクするには、コレクションの **Extensions** リストに **Google Sheet Extension** を追加します。String Table アセットを選択し、Inspector で **Extensions** の横の **Add (+)** ボタンをクリックします。1 つの String Table Collection に複数の拡張を追加し、必要に応じて各ロケールに異なるシートを割り当てることができます。

String Table を Google スプレッドシートに同期させるには、テーブルを Sheets Service Provider アセットに接続します。Sheets Service Provider の作成と設定については、こちら を参照してください。



Google Sheets Service を文字列テーブルの拡張に追加します。

一度設定すれば、この同期機能により、デザイナーや、開発者以外のユーザーも、Google スプレッドシートでローカライゼーションのエントリーを直接編集できます。

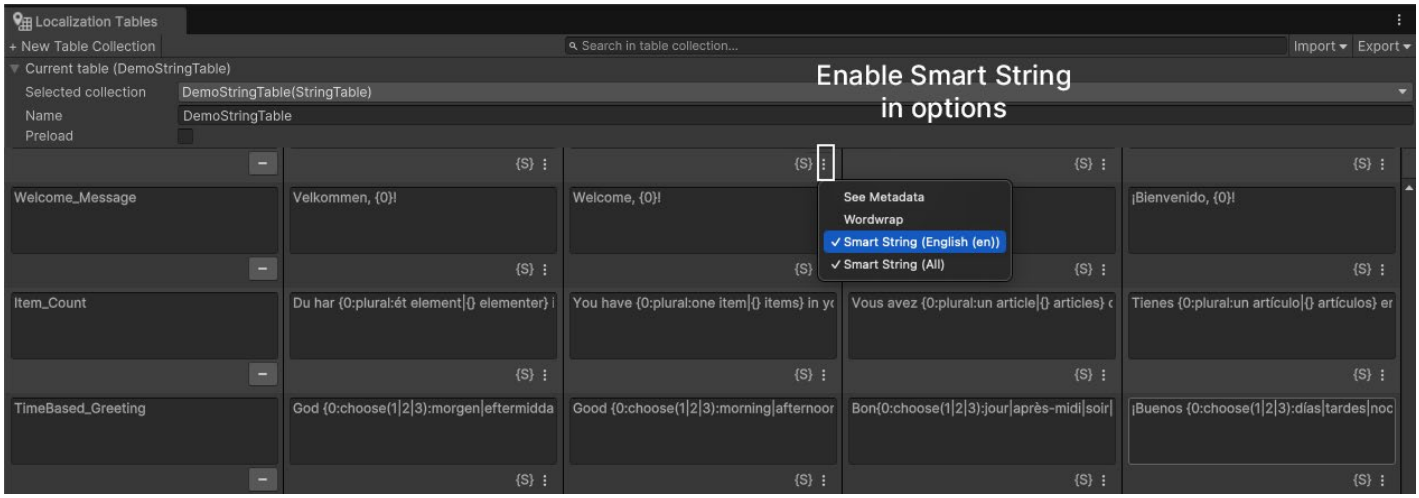
Smart Strings の使用

Smart Strings は、動的な文字列の生成時に String.Format を使用する強力な代替案です。複数形化、条件付き書式、リスト、言語固有のその他のルールなどの機能をサポートするデータ駆動型のテンプレートを使用できます。これらの機能により、ローカライゼーションの設定を簡略化できます。

Smart Strings を使用するには、Localization Tables ウィンドウで文字列をスマートとしてマークします。

Localization Tables ウィンドウを開きます。次に、メニューオプション (⋮) から **Smart Format** を選択します。エントリーの横に **{S}** アイコンが表示されていることを確認します。

または、Inspector の**Localized String Editor**の **Smart** フィールドで Smart Strings を有効にします。



StringTable ウィンドウまたは Inspector で Smart Strings を有効にします。

スクリプトでの Smart String の設定

スクリプトから SmartString を管理する方法は次の通りです。

— **プレースホルダーを使用してローカライズされた文字列を設定:**Smart String は {} カッコ内にプレースホルダーが配置されたリテラルテキストで構成されます。これは String.Format と似ていますが、柔軟性が強化されています。String Table に、`"Welcome, {0}!"` のようにプレースホルダーを使用してエントリーを作成します。ここで、`{0}` はランタイムデータのプレースホルダーです。

— **LocalizedString と Arguments の使用:**スクリプトで、このエントリーの LocalizedString を作成し、`Arguments` プロパティを使用してランタイムデータを指定します。例えば、以下は SmartStringDemo の抜粋は、1 つのプレースホルダーを置き換える方法を示しています。

```
// プレースホルダーをプレイヤー名に置き換える (例:
//       "Welcome, {0}!" => "Welcome, Player One!")

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

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

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

これにより、LocalizedString がラベルの text プロパティにバインドされ、ランタイム時にプレイヤー名が挿入されます。`"Welcome, {0}!"` の元のエントリーは、画面上に "Welcome, Player One!" のように表示されます。

プレースホルダーの理解

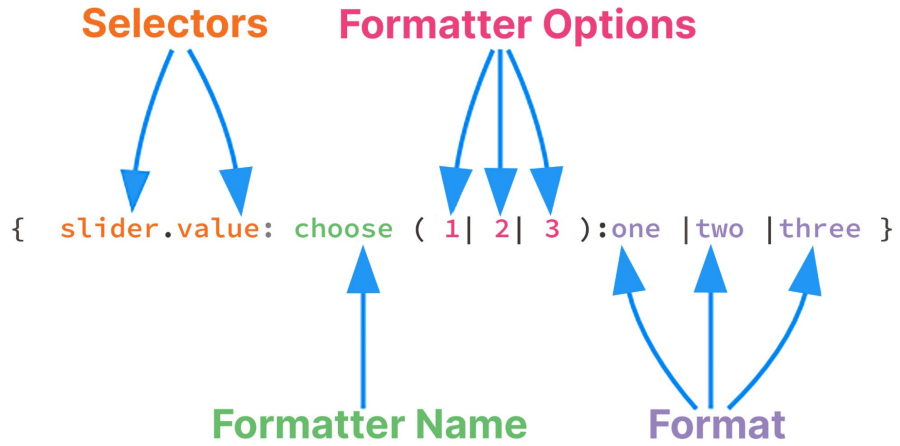
Smart Strings のプレースホルダーは、単純な `{0}` 置換だけではありません。より複雑な場合があり、高度なシナリオに対応するよう設計されています。実際、プレースホルダーは次のような複数のパーツで構成されます。

— **Selector (セレクター):**使用するデータを決めます (例えば、`{player.name}` はプレイヤーオブジェクトの name プロパティを選択します)。

— **Formatter Name (フォーマッター名):**適用するフォーマッターを定義します (複数形化のための `plural` など)。

— **Formatter Options (フォーマッターオプション):**フォーマッターの動作 (単数形と複数形の指定など) をカスタマイズします。

— **Format (フォーマット):**出力の表示方法を決定します (数字を複数形の単語にする、日付や時刻をフォーマットする、入力に基づいてフレーズを選択するなど)。



プレースホルダーは、複数の部分で構成されます。

セレクター は柔軟性が高く、ランタイム時にデータを動的に取得できます。ランタイム時にオブジェクトのプロパティやフィールドを照会できます。例えば、セレクター `{gameObject.name}` を使用するとゲームオブジェクトの name プロパティを、`{slider.value}` のセレクターを使用するとスライダーの value プロパティを取得できます。

フォーマッター は、取得したデータを最終的な文字列形式に変換します。フォーマッターを使用すると、日付、時間、リスト、複数形をフォーマットしたり、条件付きロジックを適用したりできます。

データ取得後、**フォーマッター** でデータを最終出力形式に変換します。各フォーマッターは、独自のオプションとフォーマットルールを定義します。サンプルプロジェクトには、いくつかのフォーマッターが含まれています。

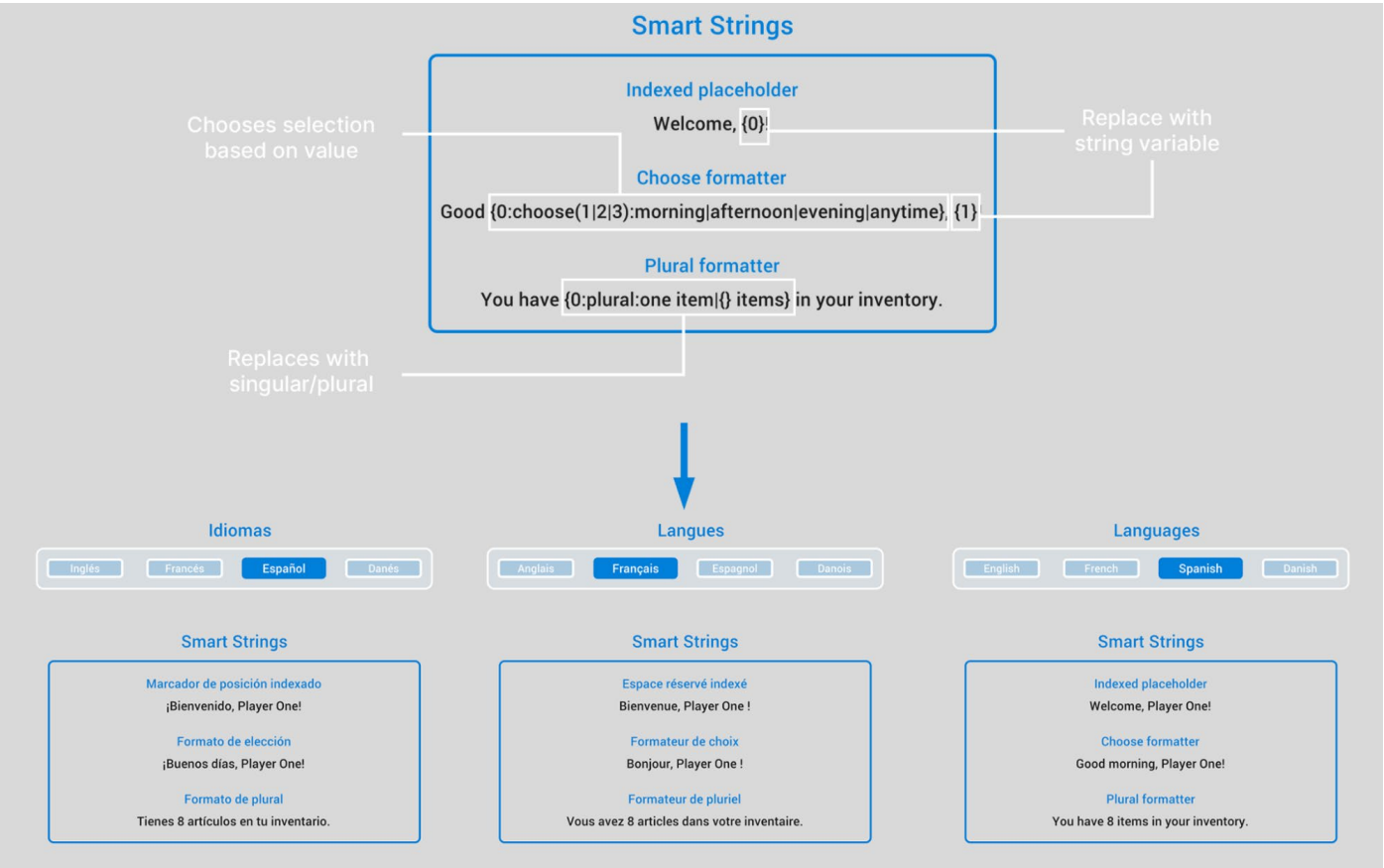
— **Choose フォーマッター**:

`{0:choose(1|2|3): morning|afternoon|evening|anytime}` は、入力に基づいて "morning"、"afternoon"、"evening" を選択します。

— **Plural フォーマッター**:

`{0:plural:one item|{} items}` は、テキストを単数形か複数形に調整します。

Inspector で値を変更し、再生モードを開始して結果のテキストを確認してください。



Smart Strings は `String.Format` の代替となります。

Smart Strings には、ローカライゼーションを強化する数多くのビルトインフォーマッターが備わっており、ゲームの状態やコンテキストに基づいてテキストを適応させることができます。

- **Choose フォーマッター**:数値入力に基づいて条件ロジックを適用できます
- **Plural フォーマッター**:数量に基づいて複数形化ルールを自動的に適用します
- **Time フォーマッター**:日付と時刻を表示します
- **Conditional フォーマッター**:if/else のようなロジック用です
- **List フォーマッター**:配列またはリストをフォーマットします
- **Is Match フォーマッター**:正規表現パターンに基づいて条件付きテキストを表示します

API を使用して カスタムフォーマッター を作成することもできます。フォーマッターの詳細については、Smart String のドキュメント を参照してください。

文字列の前処理

UI 要素を LocalizedString に直接バインディングすることが便利ではない場合もあります。例えば、ローカライズされたテキストを表示する前に、一部の要素で追加のフォーマットや変更が必要になる場合です。その場合は、UI に表示される前に LocalizedString を前処理できます。

GetLocalizedString

ここで役立つのが、ランタイム時に LocalizedString を標準文字列に変換する `GetLocalizedString` メソッドです。これにより、処理された文字列を UI に公開する前に、プレフィックスの追加や文字列の結合などのカスタムフォーマットを適用できます。例を挙げてみましょう。

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

// ローカライズされた文字列を取得し、プレースホルダー {0} をプレイヤーのレベルに置き換えるプロパティ
[CreateProperty] public string LevelMessage =>
m_LevelMessage.GetLocalizedString(m_PlayerLevel);
```

この例では、`LevelMessage` プロパティがローカライズされた文字列内の `{0}` プレースホルダーをプレイヤーの現在のレベルに置き換えます。`[CreateProperty]` 属性を使用すると、このプロパティをランタイムデータバインディングで使用できるので、UI 要素に直接簡単にバインドできます。

単純なユースケースでは、前述の `LevelMessage` のようなプロパティを定義してフォーマットロジックを扱えるため、追加のイベントハンドラーは不要です。

StringChanged イベントの使用

LocalizedString の StringChanged イベントは、このような前処理に役立ちます。LocalizedString が更新されるたびに (つまりロケールが変更されるたびに) トリガーされ、レンダリングする前にテキストを変更できます。

使用するには、StringChanged イベントにハンドラーをアタッチします。以下に、My_Entry エントリーを使用して My_Table StringTable から新しい LocalizedString を作成するコードの抜粋を示します。

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

localizedString.StringChanged += OnLocalizedStringChanged;
```

OnLocalizedStringChanged イベントハンドラーが標準文字列への変換をどのように自動的に処理するかに注目してください。その後、カスタムロジックを適用して、表示する前にテキストを変更できます。

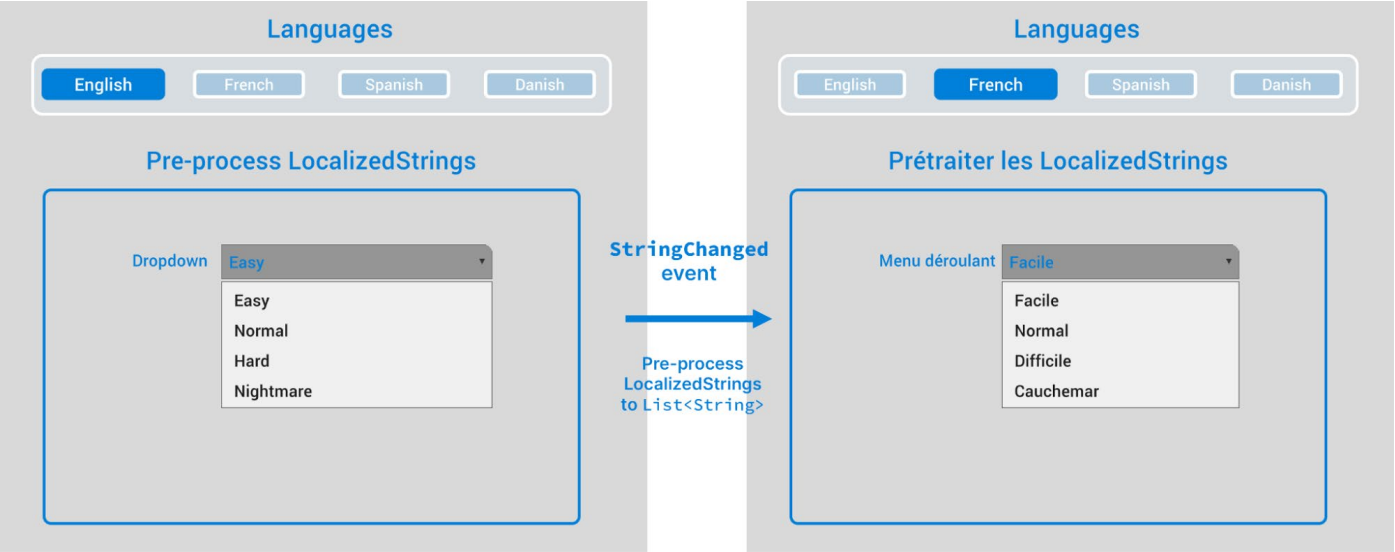
```
void OnLocalizedStringChanged(string value)
{
   // 例:特定の条件に基づいてプレフィックスを追加する
   string processedString = $"[Prefix] {value}";

   // 処理された文字列で UI 要素を更新する
   m_TextLabel.text = processedString;
}
```

動的 UI コントロール

もちろん、LocalizedString の前処理は基本的なテキストフィールドに限定されません。これは、Smart String だけでは必要なロジックやフォーマットを処理できない複雑なプロパティや UI 構造を扱うときに特に便利です。

例えば、DropdownField には文字列のリストで構成される choices プロパティがあります。前処理は、このオプションのリストを動的にローカライズし、アクティブなロケールを反映するのに役立ちます。

ここで、PreprocessDemo スクリプトは DropdownField の選択肢をローカライズし、プレイヤーが新しい言語を選択するたびに更新します。ここでも、StringChanged イベントに応答してリストをリビルドするロジックが実行されます。



StringChanged イベントを使用して LocalizedString を前処理します。

その仕組みを示すPreprocessDemo スクリプトからの抜粋を示します。

```
[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;

   // … 他の選択肢を登録する

   // 初期生成
   UpdateDropdownChoices(null);
}
```

StringChanged イベントがトリガーされると、ドロップダウンのオプションがリビルドされ、現在の選択内容が保持されます。

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

    // 現在の選択を保存する
    int selection = m_DropdownField.index;

    // 以前の選択を削除
    m_DropdownField.choices.Clear();

    // ローカライズされた現在の値を追加する

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

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

    // …他の選択肢を追加する

    // 選択したインデックスと値を復元する
    m_DropdownField.index = selection;

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

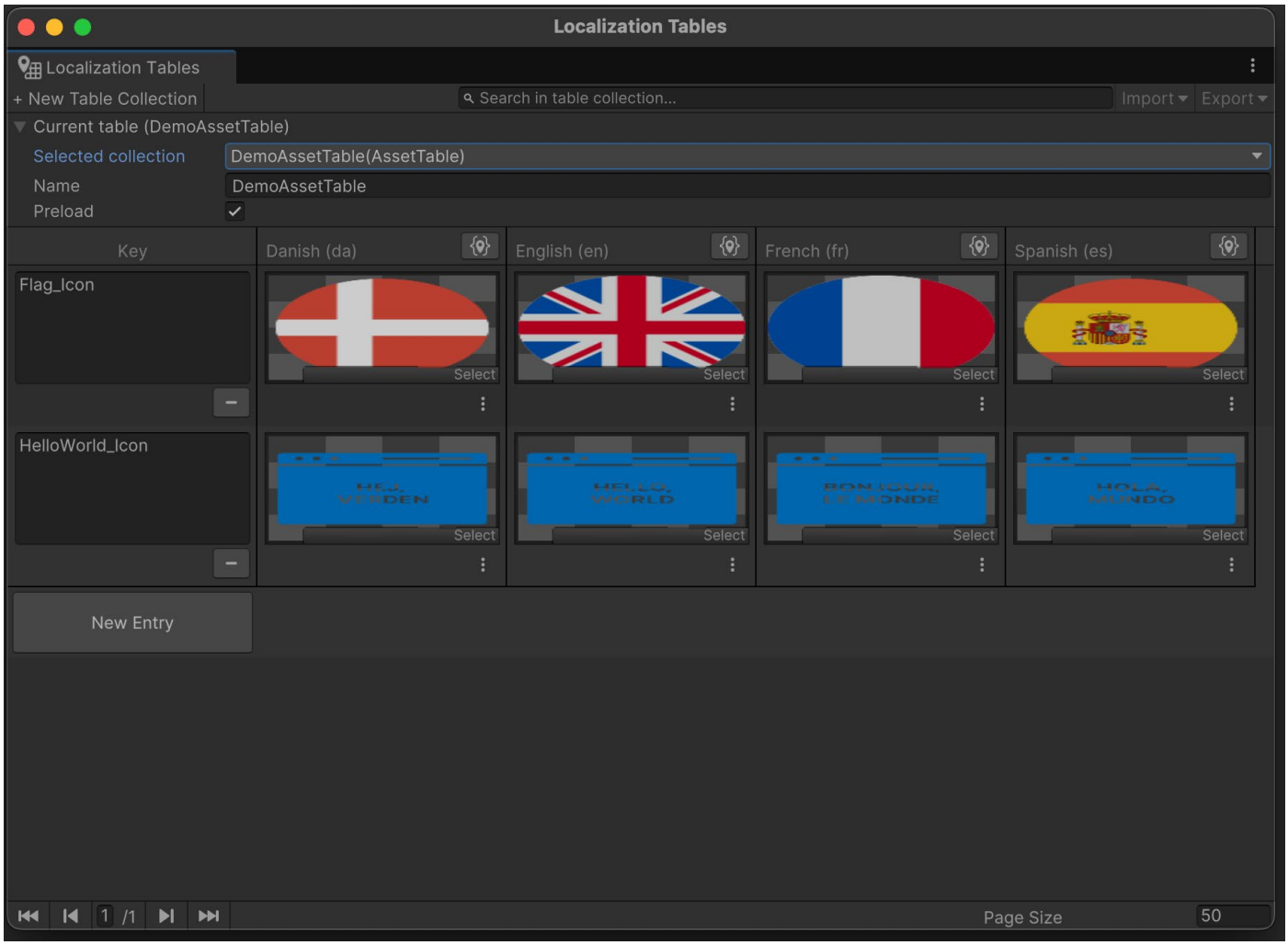
`SetValueWithoutNotify` を使用すると、ChangeEvent をトリガーせずにドロップダウンの表示が更新されます。これにより、再帰的な更新が防止され、ドロップダウンオプションが変更されたときにユーザーの選択が保持されます。

サンプルプロジェクトでは、DropdownField が LocalizedString 値に基づいて選択を動的に更新します。新しいロケールを選択するたびに、更新された言語がドロップダウンオプションに反映されます。

前処理は、ローカライズされコンテキストに配慮した UI を作成するのに役立つテクニックです。Smart Strings はプレースホルダーや複数形化など、多くのローカライゼーションタスクを処理できますが、追加の前処理により、Smart Strings だけでは処理できない柔軟性と書式設定を実現できます。

アセットのローカライズ

ローカライゼーションでは、文字列が大きな割合を占めますが、テキストに加えてアセットのローカライズが必要になる場合もあります。例えば、サンプルプロジェクトには、設定が異なるロケールの代わりになるアイコンが含まれています。



旗と "Hello, world" のアイコンは各ロケールを表します。

アセットローカライゼーションの設定

アセットローカライゼーションは文字列ローカライゼーションと同様に機能します。ローカライズされたテキストに String Table を使用するのと同様に、ローカライズされたアセットに Asset Table を使用します。どちらのテーブルも、エントリーの追加や、スクリプトや UXML ファイルでのエントリーの参照など、類似のワークフローを共有しています。

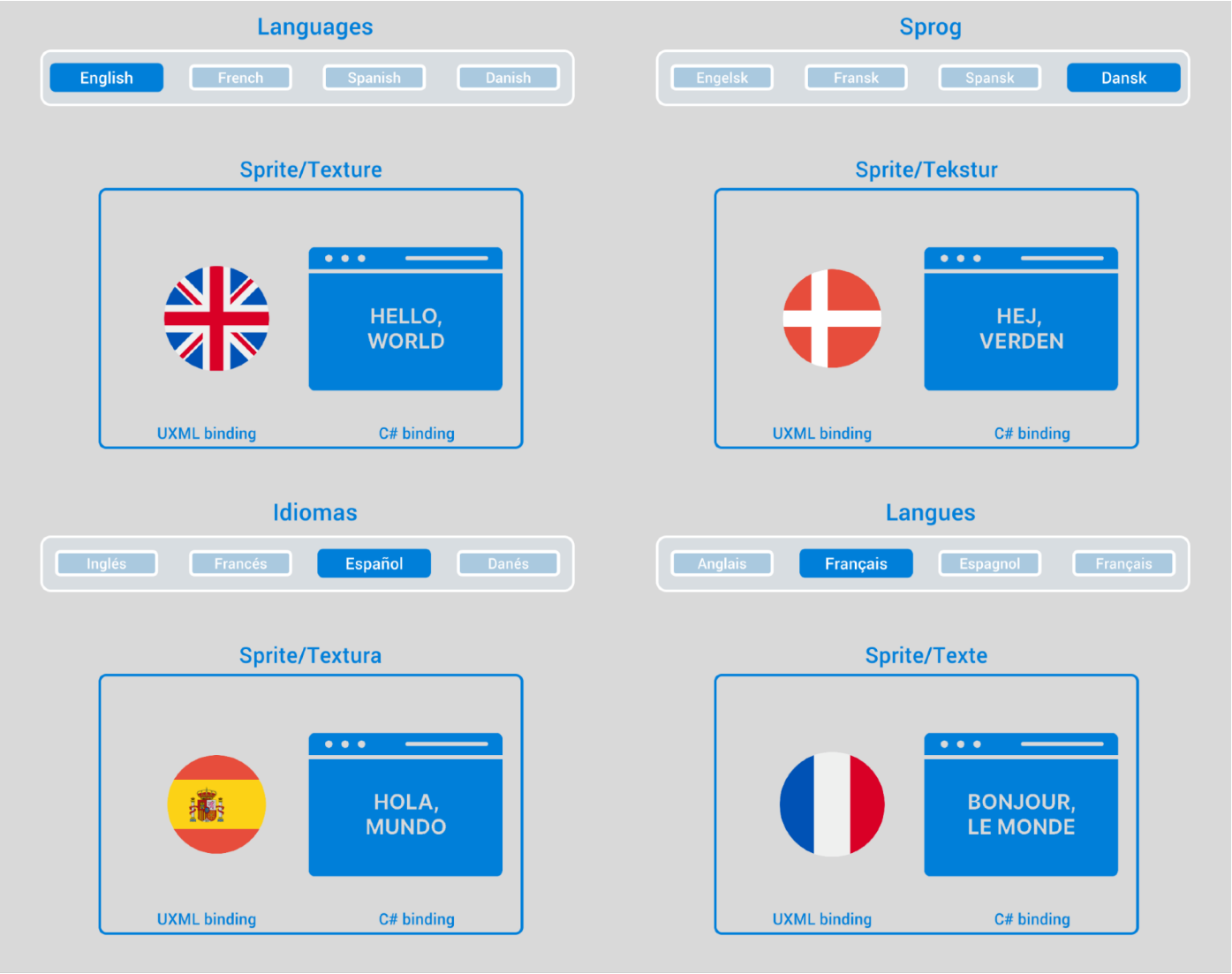
ローカライズされたアセットは、UI Builder または C# スクリプティングを通じて UI 要素にバインドできます。例えば、ビジュアル要素の `style.backgroundImage` プロパティをローカライズされたスプライトやテクスチャにバインドできます。

サンプルプロジェクトでは:

— ある要素が、UI Builder を介して UXML でデータバインディングが定義されています。

— 別の要素のバインディングが C# スクリプトで設定されています。

ランタイム時にロケールを選択すると、テキストラベルとともにアイコンが更新され、アクティブなロケールが一目でわかるようになりました。



LocalizedTextures はロケールごとに更新されます。

Asset Table と String Table

Asset Table の操作プロセスは、String Table と似ています。どちらもロケールでエントリーを定義し、ランタイム時にエントリーを取得できます。以下の違いに注意してください。

- **イベント処理**:Asset Table は、文字列に対する StringChanged イベントの代わりに、AssetChanged イベントを使用してローカライズされたアセットの変更を通知します。
- **バインディング方法**:文字列とアセットのバインディングは両方とも SetBinding で機能しますが、バインドされるプロパティ (文字列の場合は text、テクスチャの場合は style.backgroundImage) はアセットの種類によって異なります。

デモの例では、LocalizedTexture を Asset Table から名前別に取得し、`style.backgroundImage` プロパティにバインドしています。

```
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 Toolkit のローカライズされた共通アセット

ローカライズされたアセットにはさまざまな形式があります。UI Toolkit の使用時に発生する可能性があるものをいくつか紹介します。

- **ローカライズされたテクスチャ:**アイコン、背景、その他の装飾的な見た目に最適で、`style.backgroundImage` などのビジュアル要素のプロパティに直接バインドできます。
- **ローカライズされたスプライト:**あまり一般的ではありませんが、カスタムコンポーネントやスプライトベースのビジュアルには便利です。
- **ローカライズされたフォント:**さまざまな言語に必要な特定のスクリプトやタイポグラフィックスタイルをサポートするようにフォントを切り替えることができます。
- **ローカライズされたオブジェクト:**ロケールによって変える必要があるプレハブやデータ駆動型のアセットなどの複雑なリソースを参照する場合に便利です。

Asset Table を使用してこれらのアセットを活用することで、UI のテキストだけでなくビジュアルも、アクティブなロケールに合わせて動的に対応できます。

『Dragon Crashers』サンプルでのローカライズ

*UI Toolkit サンプル –『Dragon Crashers』*デモでは、いくつかのローカライゼーションテクニックが実際に使用されています。Setting ビューでは、ドロップダウンメニューを使用して、サポートされている言語の 1 つを選択できます。新しい言語が選択されると、LocalizationSettings システムは変更を検出し、リアルタイムで UI を更新します。

言語ドロップダウンメニューからロケールを選択します。

プロジェクトの詳細を自分で探る際に確認すべき点をいくつか紹介します。

— **SettingsScreen ロケールの選択:**Settings 画面では、ドロップダウンメニューでロケールを選択できます。この UI は、`LocalizationSettings` の変更をリッスンして新しいロケールの選択を検出し、ドロップダウンの変更に応じてリアルタイムで更新します。

— **データバインディング手法:**UI にはローカライゼーション手法が組み合わせて使用されています。静的プロパティは UI Builder に直接バインドされ、UXML に保存されます。

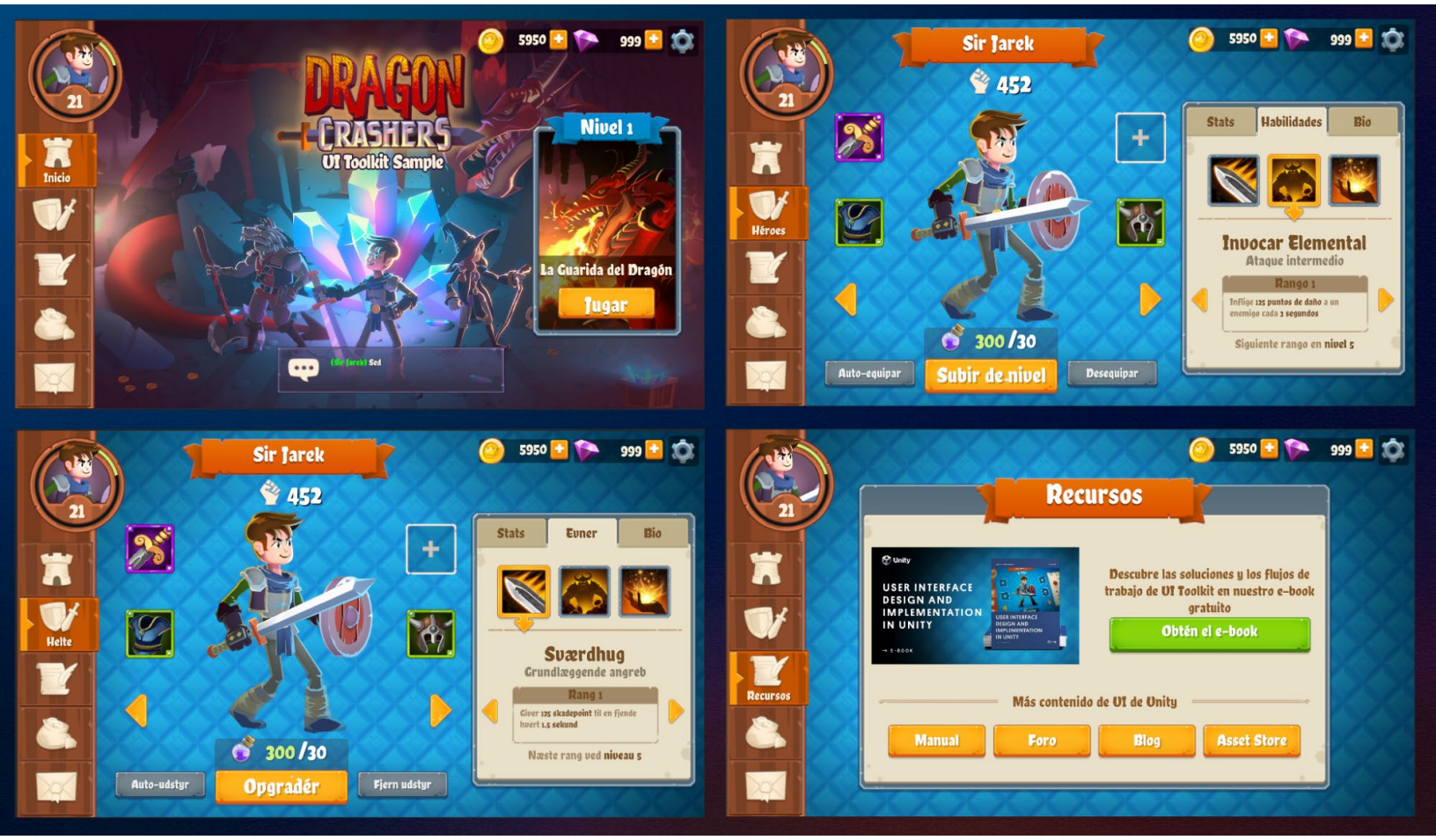
一方、動的に入力されるフィールドは、データバインディングにランタイムスクリプトを使用します。`SetBinding` メソッドは、テキストのプロパティを `LocalizedString` オブジェクトに接続し、選択したロケールを UI に反映します。

— **ScriptableObject の事前フォーマット済み LocalizedString:**一部の ScriptableObject アセットには、事前フォーマット済みの `LocalizedString` プロパティが含まれています。例えば、Settings 画面の Theme と Language ドロップダウンフィールドは、ローカライズされた値からリストを動的にリビルドし、使用可能な選択肢を翻訳します。

  RadioButtonGroup やカスタム SlideToggle などのその他の要素も、`StringChanged` イベントを扱うことで LocalizedStrings を前処理します。

UXML によるデータバインディングでも C# スクリプティングでも、ローカライゼーションの手法に関係なく、UI はロケールの変更にリアルタイムで応答します。

このサンプルプロジェクトの手法を参考に、自身の Unity プロジェクトで、ローカライズされたインターフェースをバインディングしてください。データバインディングと UI Toolkit を組み合わせることで、多言語対応の柔軟な UI を作成し、世界中のプレイヤーを迎えることができます。



UI Toolkit サンプル –『Dragon Crashers』でローカライズについてさらに調べます。

カスタムコントロール

UI Toolkit にはインターフェースを構築するための標準的な要素セットが用意されていますが、アプリケーションのニーズに合わせたカスタムコントロールを作成することもできます。

例えば、カスタムの体力ゲージは体力値に基づいて色が変わり、体力が減ると緑色から黄色、赤色にアニメーション化します。追加設定なしでキャラクター間で再利用することも、マナやパワーなどの他の統計を表すのに使用することもできます。このカプセル化されたコントロールにより、UI Toolkit 標準ライブラリのスライダーが視覚的に大幅に改善されます。

カスタムコントロールを使用すると、機能をスタンドアロン要素にカプセル化し、インターフェースのさまざまな部分で再利用できます。適切に設計されたコントロールは、抽象的、自己完結型で、コード再利用をサポートしており、プロジェクトメンテナンスの簡略化に役立ちます。カスタムコントロールを実装するときは、単独では機能しない特定のコンポーネントに結び付けられた要素（ゲームメニューなど）と一緒に使用しないようにします。

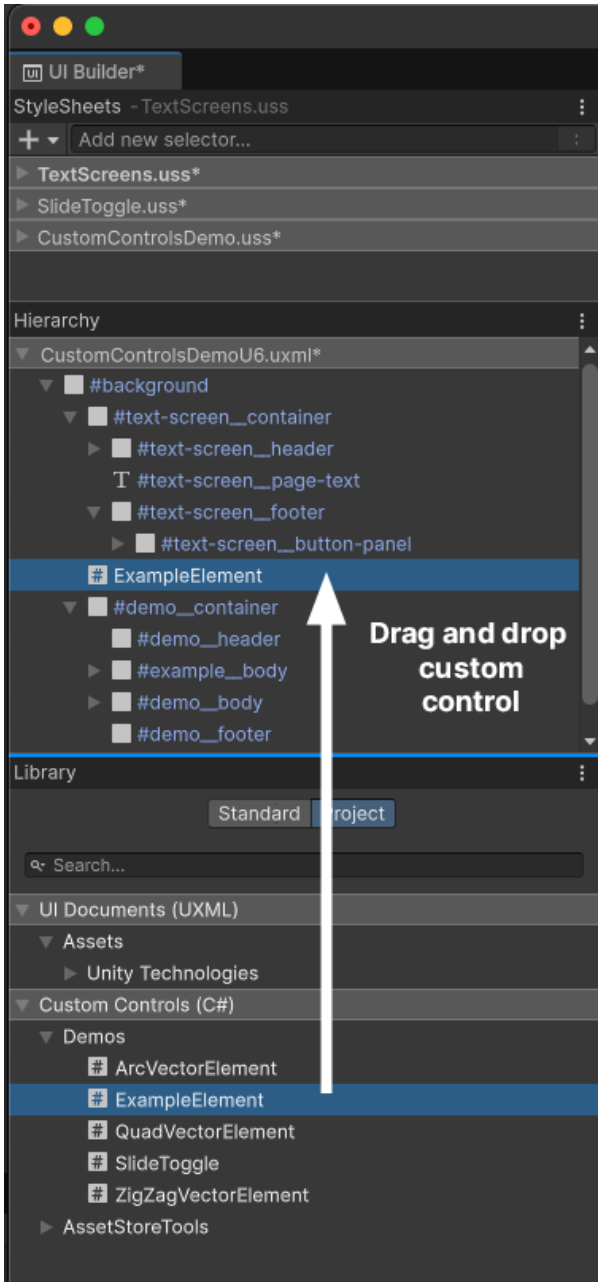
UxmlElement 属性

カスタムコントロールを作成するには、VisualElement クラスから継承する新しい C# スクリプトを定義するか、作成するクラスにぴったり合うサブクラスを定義します。ボタンのようなコントロールが必要ですか? それならば、Button クラスから継承しましょう。

カスタムコントロールを UXML と UI Builder で使用できるようにするには、クラスに `UxmlElement` 属性を追加します。カスタム要素が `public partial` クラスとして定義されるようにします。

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


}
```

作成したカスタムコントロールは、UI Builder の Library セクションの **Custom Controls (C#)** カテゴリに表示されます。次に、UI Builder の Hierarchy ウィンドウにドラッグします。



カスタムコントロールは UI Builder ライブラリに表示されます。

ビジュアル要素はゲームオブジェクトではないため、Awake、OnEnable、OnDisable、OnDestroy などの通常のライフサイクルイベントはありません。代わりに、そのコンストラクターを使用してカスタムコントロールを初期化します。

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

    // コンストラクター
    public ExampleElement()
    {
        // 初期化
    }
}
```

カスタムコントロールが UI に追加されるまで初期化を遅延することもできます。そのためには、**AttachToPanelEvent** のコールバックを登録します。

カスタムコントロールが UI から削除されたことを検出するには、**DetachFromPanelEvent** のコールバックを使用します。

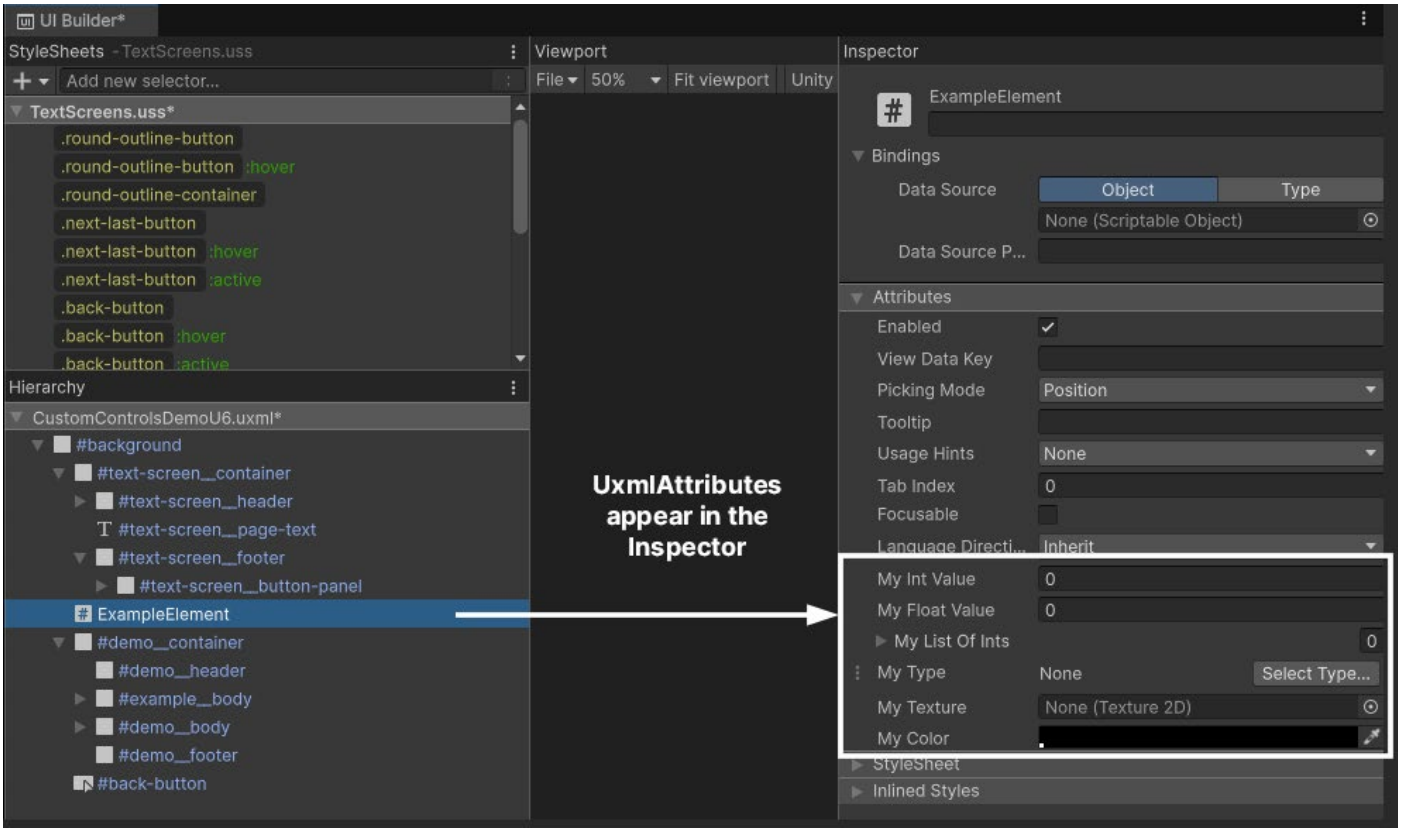
UxmlAttribute 属性

UxmlAttribute 属性をプロパティに追加すると、そのプロパティが UI Builder の Inspector ウィンドウに表示されます。これにより、初期値をインタラクティブに設定できます。Inspector での変更はコードの修正が不要なため、UxmlAttributes はデザイナーとの共同作業の際に便利です。

公開する各プロパティに UxmlAttribute 属性を適用します。また、**name** 引数を使用して、属性名をカスタマイズすることもできます。

Hierarchy でコントロールを選択すると、Inspector ウィンドウにカスタム属性が表示され、直接設定できます。

デコレーター属性で、MonoBehaviour の操作と同じようにカスタム属性フィールドを変更できます。便利なデコレーター属性には、TextArea、Tooltip、Range、Header、Min、Multiline、Space、Delayed があります。例えば、Range 属性を使用すると、範囲内の値を選択するためのスライダーが追加されます。



カスタム属性は UI Builder の Inspector に表示されます。

カスタムコントロールに UxmlElement 属性を追加する基本的な例を示します。これには、UxmlAttribute 属性を使用する 2 つの公開プロパティが含まれています。

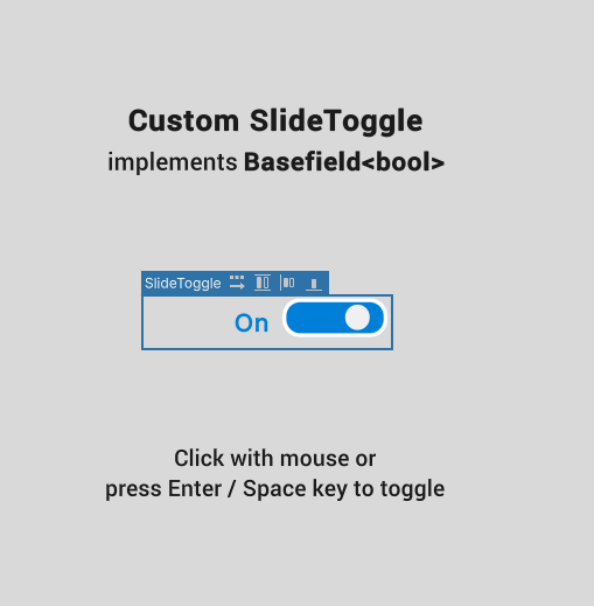
```
[UxmlElement]
public partial class ExampleElement: VisualElement
{
   [UxmlAttribute(name:"my-text")]
   public string myStringValue { get; set; }

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

}
```

この例では、name パラメーターを使用して `MyStringValue` が Inspector に **"My Text"** として表示されます。`MyStringValue` と `MyIntValue` は両方とも、Hierarchy で ExampleElement のインスタンスが選択されているときは、Inspector で編集できます。

Unity 6 より前は、カスタムコントロールを作成するには、カスタム要素の属性登録とオブジェクトのインスタンス化を処理する **UxmlTraits** クラス と **UxmlFactory** クラスを実装する必要がありました。

Unity 6 では、UxmlElement 属性と UxmlAttribute 属性を導入することで、カスタム要素の作成が簡略化されています。これらは UXML と UI Builder でカスタムコントロールとプロパティを直接公開します。この新しいワークフローにより、定型コードの量が減り、UI 要素を素早くカスタマイズできます。

例: カスタムスライドトグルコントロール



カスタムスライドトグルコントロールはブーリアン値を表わします。

単純なカスタムコントロールの例として、スライドトグルを考えます。これは、ブーリアン値を表すスイッチのような要素です。

これは、標準のトグルよりも魅力的なユーザー体験を提供することがあります。アニメーション化されたスイッチ、色の変更、動的なテキストなどの視覚的なフィードバックを追加することで、より直感的な UI にすることができます。

カスタムコントロールの定義

QuizU プロジェクトの CustomControlsDemo シーンで、このカスタムコントロールの簡単な実装を確認できます。SlideToggle.cs スクリプトを開き、その仕組みを確認します (下の抜粋参照)。

スライドトグルのカスタムコントロールは、最も適切な基本クラス (この場合は BaseField<bool>) から継承します。UxmlElement 属性は、UXML と UI Builder でコントロールを公開し、再利用できるようにします。

```
[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;
```

ビジュアル構造は背景 (m_Input) とノブ (m_Knob) で構成され、USS クラスが外観を定義します。

```
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";
```

クリック、キー入力、ナビゲーションイベントに応答するイベント処理が実装されています。これにより、複数の方法で状態を変更できます。

```
    // …
    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();
    }
}
```

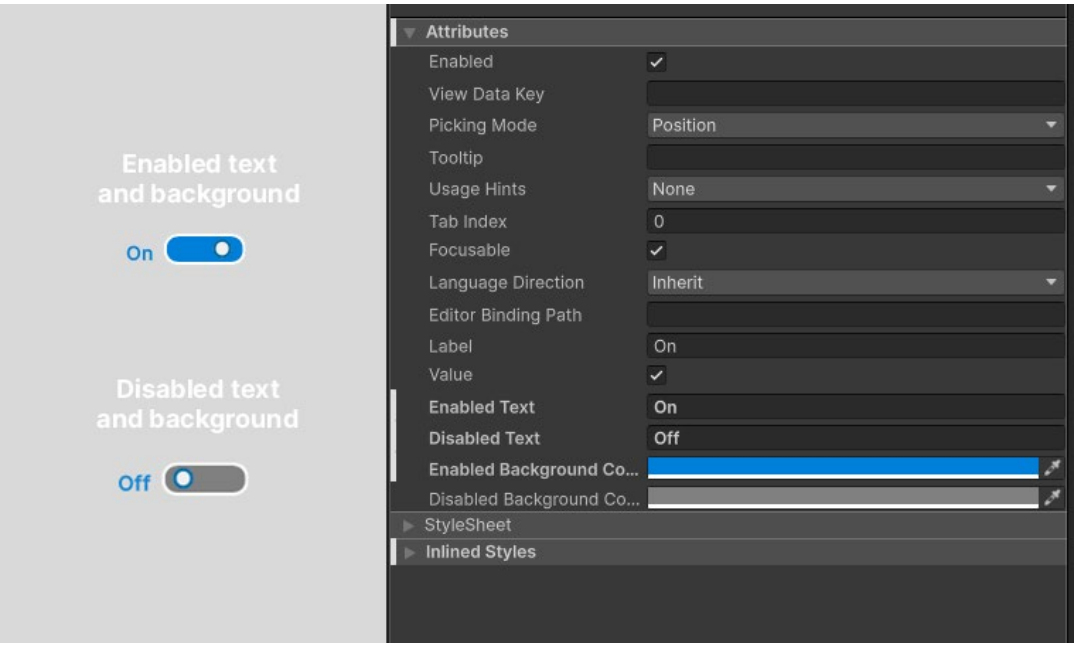
ユーザーがスイッチを切り替えるとラベルと背景色が自動的に更新され、視覚的にフィードバックされます。

ここでは、SetValueWithoutNotify を使用して、ChangeEvent をトリガーせずにトグルの見た目の状態を更新します。このメソッドは値が変化すると内部で呼び出されるため、更新の無限ループが発生することなく UI が正しく更新されます。

```
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 シーンのサンプル実装を見てみます。要素をマウスでクリックするか、Enter キーまたは Space キーを押してアクティブ状態を切り替えます。このサンプルでは、ユーザーがスライドコントロールを切り替えると、視覚的なフィードバックとなる簡単なアニメーションと共に、ラベルと背景色が動的に更新されます。

Inspector を使用して、有効状態と無効状態に対応する文字列ラベルと背景色を設定します。



スライドトグルのテキストと色をカスタマイズします。

スライドトグルの使用

コンパイルが完了したら、スライドトグルを UI の任意の部分に組み込むことができます。カスタム SlideToggle クラスを他のビジュアル要素と同じように使用します。次に、SlideToggle クラスを使用してサウンドをミュートまたはミュート解除する実装例を示します。

```
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);
        }
    }
}
```

このケースでは、SlideToggle は既存の UXML ドキュメントの一部です。MonoBehaviour でビジュアルツリー内を名前で検索し、RegisterValueChangedCallback メソッドを使用してトグル状態をオーディオ設定にリンクします。

スライドトグルはスタンドアロンのカスタム要素であるため、UI のあらゆる種類のトグルスイッチに使用できます。例えば、『Dragon Crashers』UI Toolkit サンプルでは、同様の SlideToggle によって FPS カウンターを有効および無効にできます。

UI Toolkit サンプル – 『Dragon Crashers』の様式化されたトグル

SlideToggle は、アプリケーションの要件に合わせてカスタマイズできます。ビジュアル、サウンド、ゲームプレイのオプションなどの設定に最適です。一度ビルドすれば、カスタムスイッチによってユーザー体験を強化できる場合はいつでも再利用できます。

実装全体については、QuizU プロジェクトの **SlideToggle.cs** スクリプトを参照してください。

その他のカスタムコントロールの作成

コントロールが標準 UI Toolkit ライブラリに含まれていない場合は、独自に作成できます。ここで紹介するのは、自作のゲームにカスタムコントロールを展開する方法を考えるための例です。

- **体力ゲージ/プログレスバー:**体力、マナ、パワーなどのゲーム属性はゲームプレイによって大きく変わるため、カスタムコントロールの有力な候補です。最大値、現在値、状態の色などのUxmlAttributes を公開して、カラーグラデーションのオプションを追加します。
- **星評価:**このコントロールは、整数値を表すセグメント化されたプログレスバーのように機能します(ステージクリア時の星評価など)。入力済みの状態と未入力の状態を切り替えることができる複数の子要素を持つビジュアル要素から始めます。Inspector で最大値を持つ `int` を公開し、ユーザーがUxmlAttributes を使用してスプライト画像をカスタマイズできるようにします。
- **タブビューコントロール:**タブ付きインターフェースは、同じウィンドウ内でさまざまなビューやセクションを切り替えるための一般的な UI です。これを実装するには、タブの行とコンテンツ領域を持つカスタム要素を作成します。各タブはボタンのようなビジュアル要素にすることができ、動的にタブを追加または削除するオプションがあります。

ほとんどの場合、USS 遷移をトリガーしてアニメーションで外観を華やかにすることもできます。カスタムコントロールにより、ユーザーはそのゲーム独自の UI で、つまんだり、クリックしたり、スクロールしたり、トグルしたりできます。

皆さんの作品を楽しみにしています。

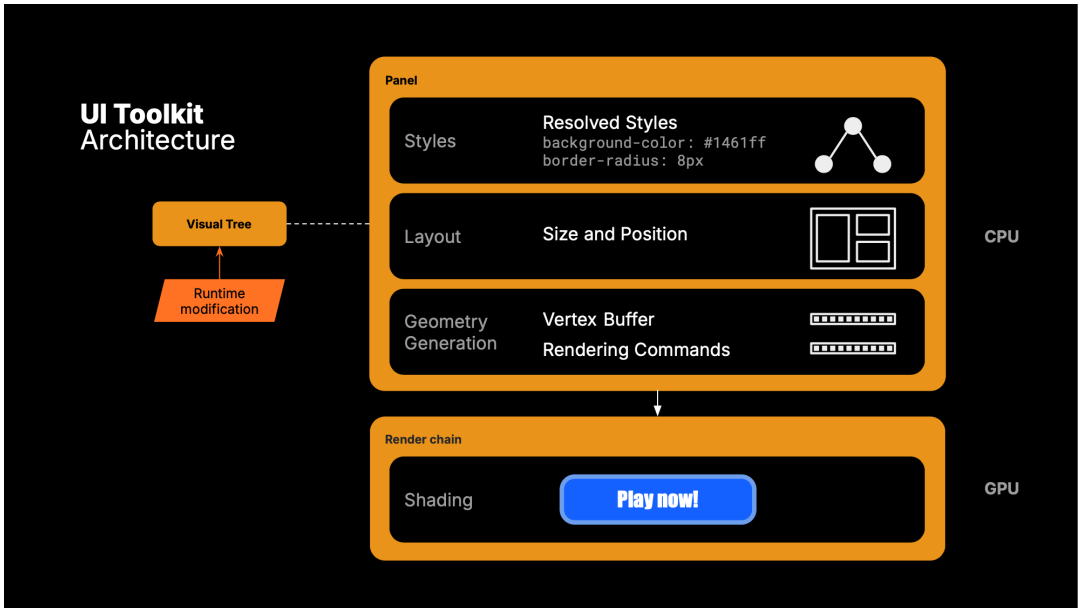
パフォーマンスの最適化

洗練されたゲーム UI を構築するには、多くの場合、画面上の要素の大規模な階層を管理する必要があります。何百もの要素が存在するため、技術的な課題が生じる可能性があります。わずかな非効率性でも、ランタイム時のスタッターやフレーム落ちにつながり、プレイヤー体験に悪影響を及ぼすおそれがあります。

幸いなことに、これらの課題のほとんどはいくつかの最適化手法によって解決できます。Unity 6 では UI Toolkit キットが大幅に改善され、そのままでもパフォーマンスは向上していますが、真に効率的なユーザーインターフェースにはやはり開発者の努力が必要です。

多くの場合、この作業の大部分は、不要なオーバーヘッドの排除とドローコールの削減に帰結します。Unity 6 の UI Toolkit を最大限活用するために役立つヒントを探っていきましょう。

更新メカニズム



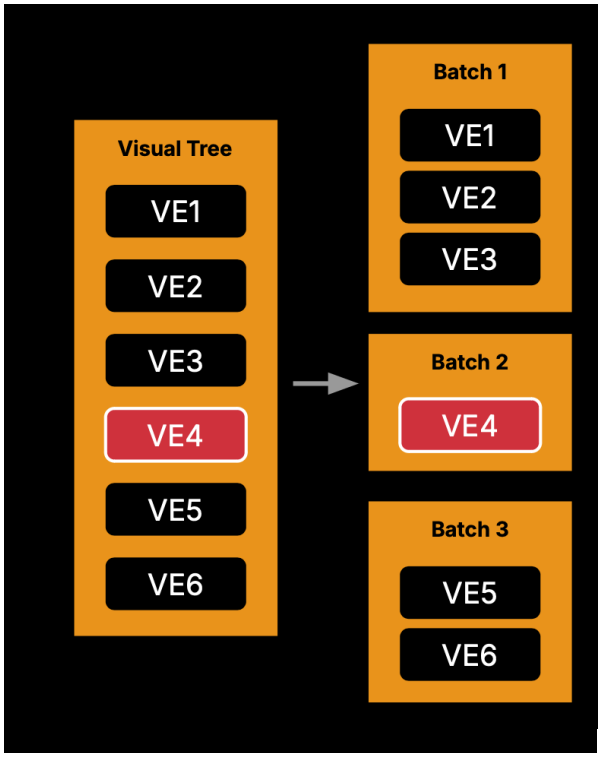
ビジュアルツリーには、複数の更新メカニズムが含まれています。

ビジュアルツリーには、ランタイム時のスタイル、レイアウト、コンテンツの変更に反応する複数の更新メカニズムが含まれています。これらの更新メカニズムのいずれかがパフォーマンスに影響を及ぼす可能性があります。下表に、それらの発生タイミングと、パフォーマンスへの影響をまとめました。

更新メカニズム	説明	発生タイミング	パフォーマンスへの影響
スタイル解像度	USS セレクターとスタイルを適用して、要素の最終的な外観を決定します	スタイルクラスの追加や色の変更など、クラスやスタイルが変更されたときにトリガーされます	大規模な階層や、深くネストされた階層では、このプロセスはコストがかかります。頻繁な変更を最小化します。
レイアウト再計算	UI 階層に正しく収まるように要素のサイズと位置を調整します	要素のサイズ、位置、整列の変更 (パネルのサイズ変更、要素の移動など) によってトリガーされます	レイアウトを頻繁に更新すると、コストが高くなることがあります。アニメーションには、位置を直接変更する代わりに Transform を使用します。
頂点バッファ更新	長方形や角丸など、UI 要素のレンダリングに使用される幾何学的形状を更新します	角丸の付加や境界線の変更など、要素のジオメトリが変更されたときにトリガーされます	頂点バッファの更新には大量のリソースを要します。頻繁なジオメトリの変更は避けます。
レンダリング状態の変更	テクスチャなどのレンダリング状態や、要素の描画に必要なブレンドモードを変更します	マスキングや独自のテクスチャなど機能によってトリガーされ、バッチ処理が中断されます	過剰な状態変更は CPU オーバーヘッドを増加させます。バッチ処理を活用し、固有のテクスチャやマスクの仕様を制限することで最適化します。

もちろん、これらの操作のコストは、UI 要素変更の頻度と範囲によって異なります。

バッチ処理要素



バッチを分割すると、パフォーマンスが低下します。

ユーザーインターフェースをレンダリングするときは、すべてのビジュアル要素で GPU に命令を送信する必要があります。UI Toolkit はバッチ処理を通じてこれらのドローコールを最適化します。同一の GPU 要件を持つビジュアル要素をグループ化し、まとめて処理できるようにします。バッチ処理は、ゲームオブジェクトでの ドローコールバッチ処理 と同様に、GPU との通信オーバーヘッドを大幅に削減します。

要素を効率的にバッチ処理するには、同じ GPU 状態 (同じシェーダー、テクスチャ、メッシュデータ、GPU 固有のその他のパラメーター) を共有する必要があります。例えば、同じフォントとスタイルを使用する一連のテキスト要素はまとめてバッチ処理できます。ただし、その間に画像を挿入するには、異なる GPU 設定が必要です。これにより、新しいバッチが強制的に作成されます。

このようにバッチを "分割" するたびに、少し非効率になります。高いパフォーマンスを維持するには、分割を最小限にするよう UI を構成することが重要です。

すべてのバッチが GPU に 1 つ以上のドローコールを発行する可能性があるため、一般的にバッチ数が少ないほどオーバーヘッドが減り、パフォーマンスが向上します。

次のセクションでは、UI のバッチ数を最適化し、一貫したパフォーマンスを実現するための手法を探ってみましょう。

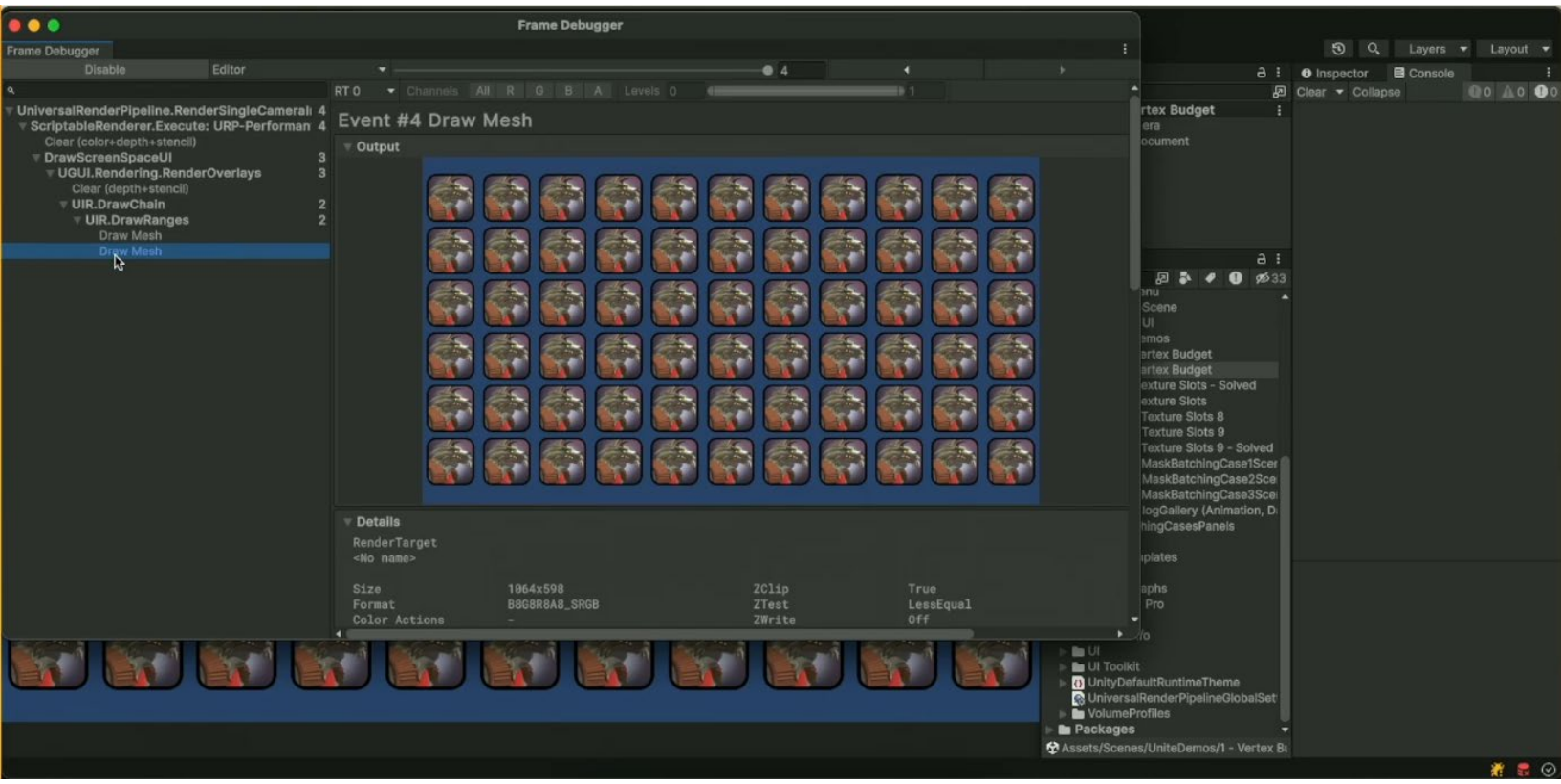
頂点バッファ

UI Toolkit では、UI のレンダリングに必要なジオメトリ（頂点）は頂点バッファに格納されます。UIDocument がランタイム時に Panel を作成するときに、ビジュアル要素を処理する単一の頂点バッファを事前に割り当てます。このバッファは、ビジュアル要素用の "ヒープアロケーター" だと考えてください。要素が UI に追加されるときに動的にメモリが割り当てられます。

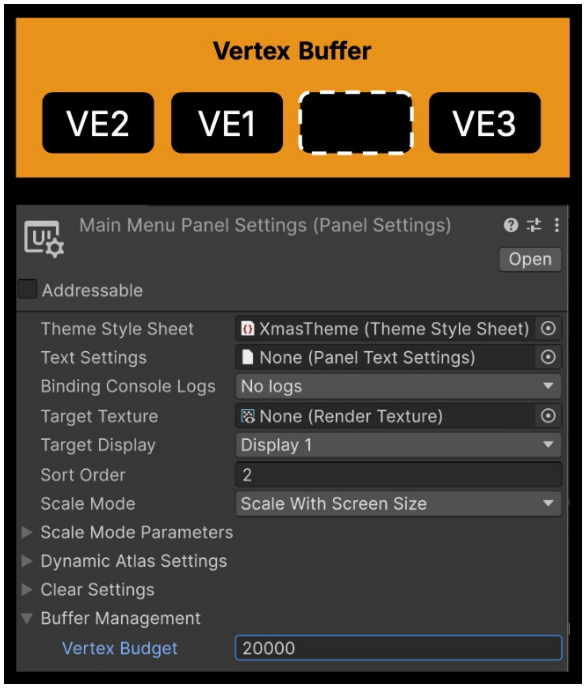
UI が頂点バッファの容量を超えている場合は、追加のバッファが作成されます。これによりバッチ処理がフラグメント化され、ドローコールの回数が増加し、最終的にパフォーマンスが低下するおそれがあります。

これに対処するには、Panel Settings で **Vertex Budget** を調整して、頂点バッファの初期サイズを設定します。デフォルト値は 0 で、Unity が自動的にサイズを決定します。ただし、複雑な UI の場合にこの値を手動で増やせば、ドローコールの回数が減るため、パフォーマンスが向上する可能性があります。

以下に例を挙げます。この UI には、1 つの頂点バッファに収まらない多数の要素があります。Frame Debugger (フレームデバッガー) は、ドローコールが 1 回ではなく 2 回になることを示します。

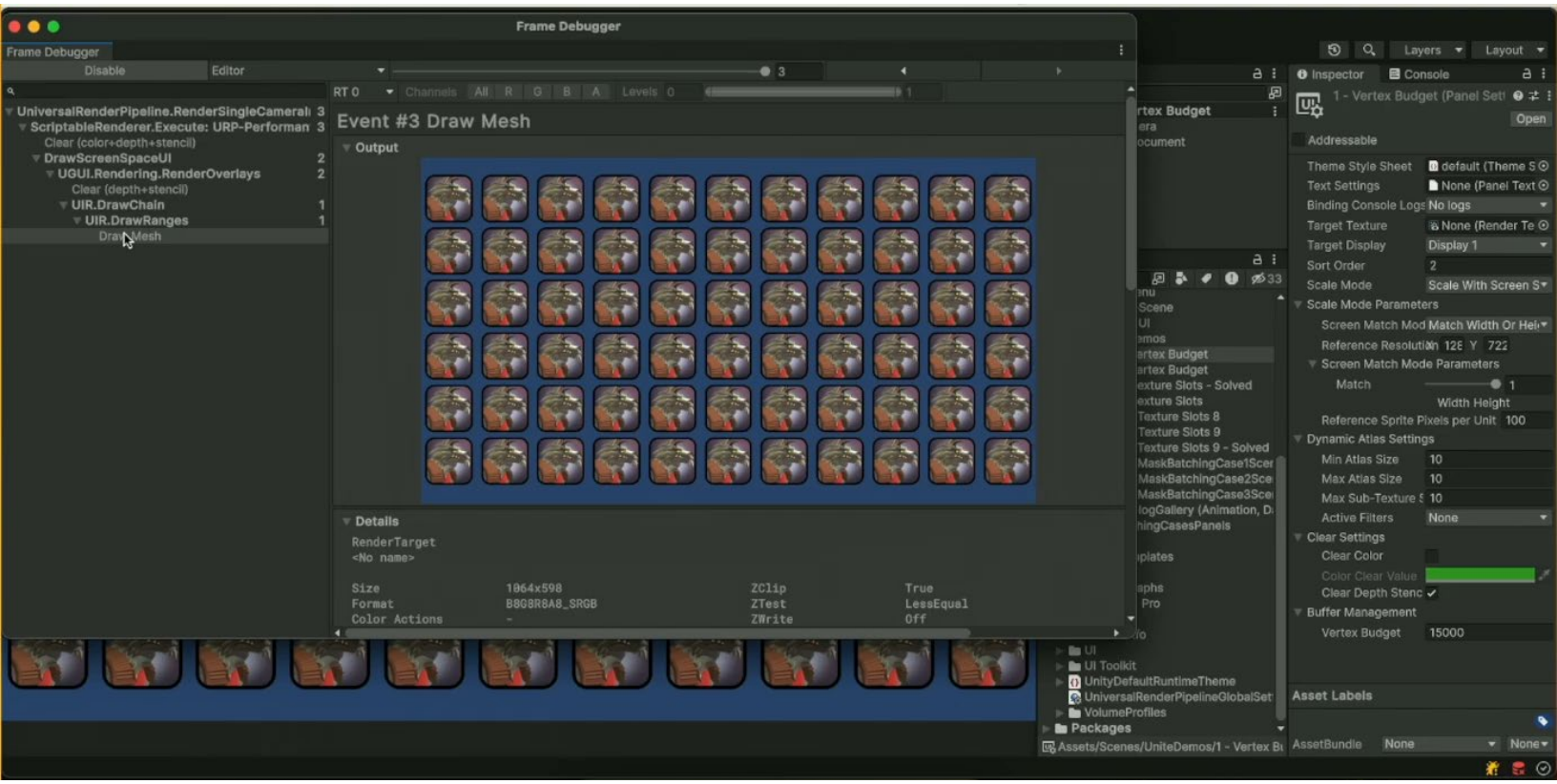


この UI には複数のドローコールが必要です。



Vertex Budget の値を、例えば 20,000 頂点に増やすと、1 回のドローコールでフレームバッファが UI 要素を収めることができるようになります。設定を 1 つ変更することでサンプル UI の効率が向上します。

Vertex Budget を増やすと、ドローコールが減る場合があります。



Vertex Budget を調整すると、1 回のドローコールが復元されます。

複雑な UI の場合、この値を手動で増やしてドローコールが減ることでパフォーマンスが向上する可能性がありますが、メモリの過剰割り当てには注意してください。フレームデバッガーと Unity プロファイラー を使用して、メモリ使用量とドローコール回数の最適なバランスを見つけます。

ウーバーシェーダーと 8 テクスチャの制限

UI Toolkit は、すべての UI レンダリング機能を汎用性の高い 1 つの "ウーバーシェーダー" に統合します。このシェーダーは、複数のシェーダーバリアントに依存するのではなく、動的分岐 を使用してランタイム時に適切なレンダリングパスを選択します。これにより、シェーダーの切り替えを最小限に抑えることで CPU オーバーヘッドは削減されますが、分岐ロジックにより GPU コストはいくらか増加します。

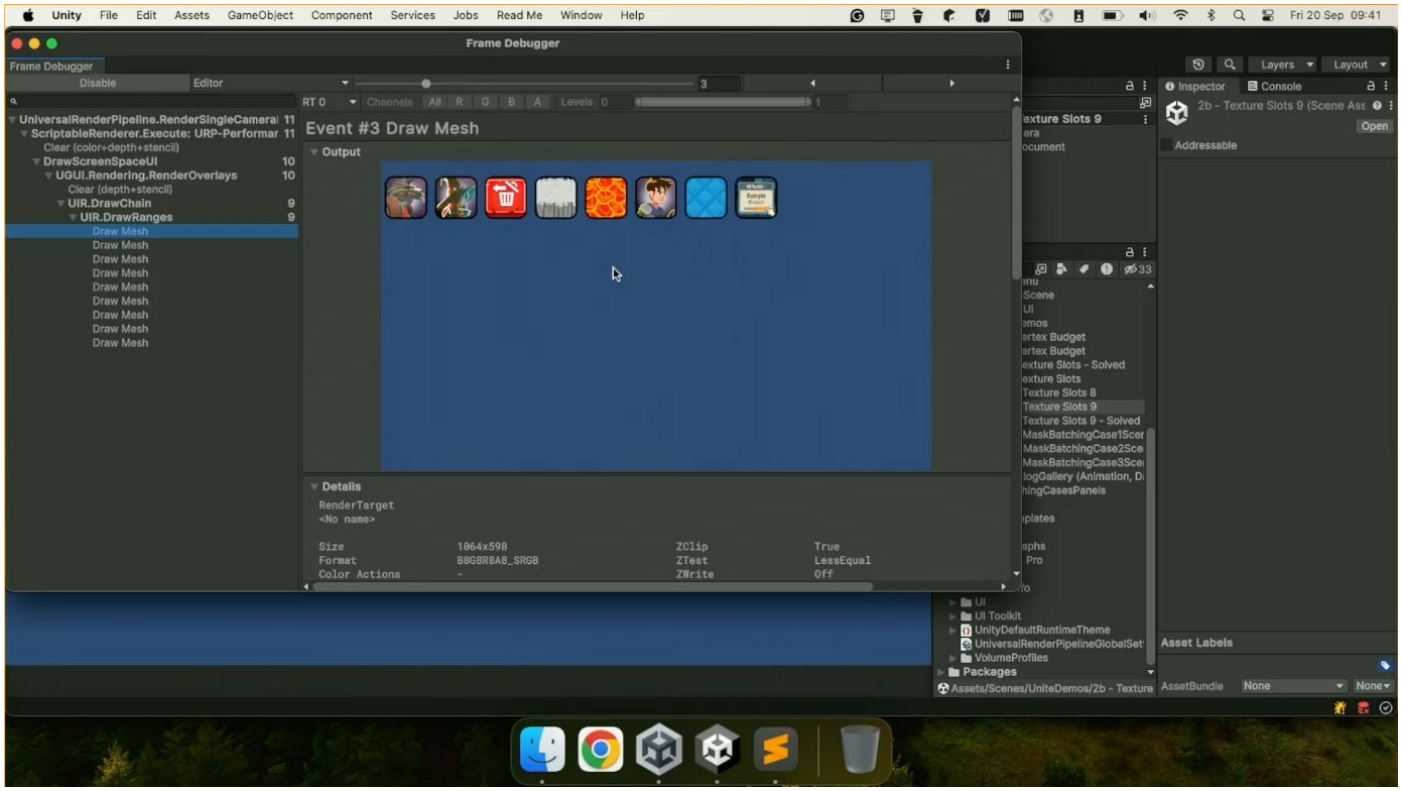
このシェーダーを強力なものにしている特徴の 1 つは、同じバッチ内での最大 8 つのテクスチャのサポートです。これにより、さまざまなテクスチャを持つ要素を同じドローコールでレンダリングできます。次ページの画像では、UI が 8 つの異なるテクスチャで構成されていることがわかります。

このサンプル UI には、8 つのテクスチャが含まれています。

"ウーバーシェーダー" は、1 回のドローコールとしてレンダリングされます。

Frame Debugger に示されているように、Unity は最大 8 つのテクスチャに対して 1 回のドローコールでサンプル UI をレンダリングします。ただし、8 テクスチャという制限を超えると、別々のバッチに分割され、オーバーヘッドが増加します。

8 テクスチャの制限を超えた場合は、次のようになります。1 回のドローコールが次のように増えます。



テクスチャが多すぎるとバッチが分割されます。

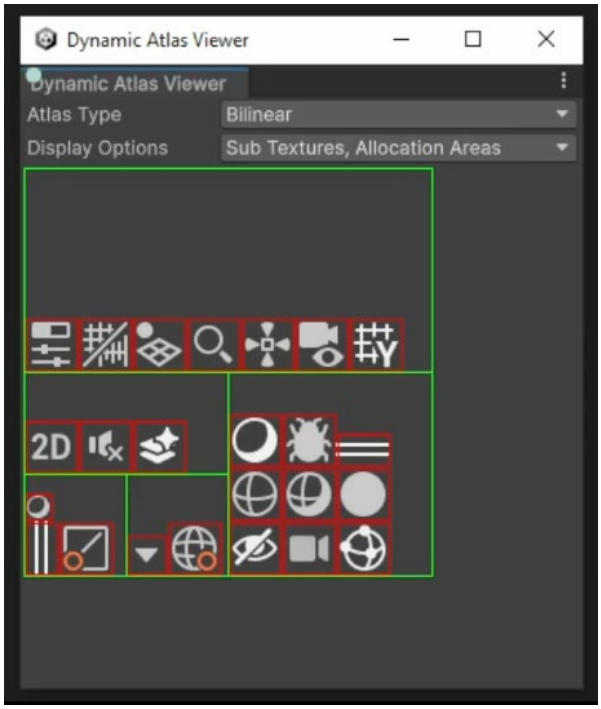
この制限を緩和するために、UI Toolkit にはテクスチャの使用を最適化するツールが用意されています。例えば、テクスチャをアトラスに統合することで、テクスチャの数をサポート制限内に収め、バッチの効率性を維持し、ドローコールを減らすことができます。

動的なテクスチャアトラス

複数のテクスチャを切り替えると、UI Toolkit がバッチを分割し、ドローコールが増え、パフォーマンスが低下することがあります。この問題の一般的な解決策はテクスチャの **アトラス化** です。これは、複数の小さなテクスチャを 1 つの大きなテクスチャに結合するものです。

2D Sprite アトラス をご存知であれば、パフォーマンスを改善する効果的な方法もご存じでしょう。複数のスプライトを 1 つのスプライトアトラスにパッキングすることで、Unity はそれらを 1 つのテクスチャとして扱い、バッチの分割やドローコールを減らします。2D Sprite アトラスは UI Toolkit とシームレスに連携するため、静的コンテンツや事前定義済みコンテンツに最適です。
ただし、2D Sprite アトラスには、ランタイム生成のテクスチャを処理できないなどの制限があります。また、設定とスプライトレイアウトを事前に行う必要があり、時間がかかる場合があります。

『Dragon Crashers』サンプルは、2D Sprite アトラスを使用しています。



Frame Debugger の Dynamic Atlas Viewer を使用します。

UI Toolkit の動的テクスチャアトラスは、複数の画像を効果的に 1 つのテクスチャにマージし、テクスチャの状態変化を減らします。アトラスの設定は Panel Settings で行い、アトラスのレイアウトは Dynamic Atlas Viewer (UI Toolkit の Debugger ウィンドウで利用可能) で視覚化できます。

UI に大幅な変更 (時間経過に伴う大量のテクスチャの追加や削除など) を行うと、アトラスがフラグメント化される可能性があります。このような場合、ResetDynamicAtlas API でアトラスを初期状態に復元できます。

UI Toolkit では、2D Sprite アトラスと動的テクスチャアトラスを同時に使用できることを覚えておいてください。スプライトアトラスは静的な事前定義済みコンテンツに最適で、動的アトラスは UI コンテンツがランタイム駆動型の場合に優れています。

マスキング

UI Toolkit はステンシルバッファを使用してマスク (UI 要素の一部を表示または非表示にする領域) を作成します。ステンシルバッファは GPU システムの一部であるため、マスク設定を変更すると、UI Toolkit のバッチ処理が分割される場合があります。

マスクされた要素を階層的にレイヤー化すると、ネストされた深度ごとに追加の状態を追跡するためのステンシルバッファが必要になるため、複雑さが増すことにご注意ください。そのため、GPU の負荷が増加します。

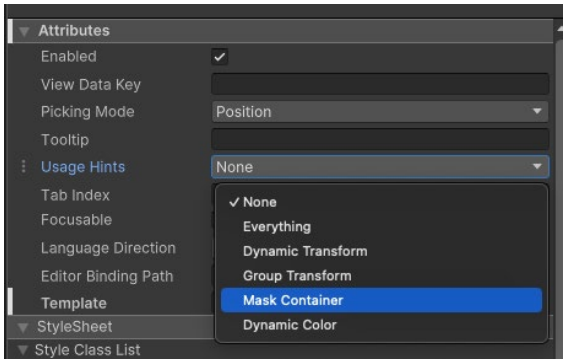
UI Toolkit では、2 種類のマスキングをサポートしています。

- **矩形ベースマスキング:**矩形マスクはシェーダーベースの操作を使用し、GPU の状態を変更することなくバッチの整合性を維持します。この手法はステンシルバッファを使用しないため、深度制限なしで矩形のマスクをネストできます。
- **角丸と複雑なマスク (ステンシルバッファ):**角丸やその他の複雑な形状にはステンシルバッファ操作が必要であり、各マスキングレベルでバッチ処理が分割される可能性があります。この手法では、最大 7 レベルにネストされたマスキングがサポートされます。

矩形と角丸のマスク

マスクされた要素のパフォーマンスを最適化するには:

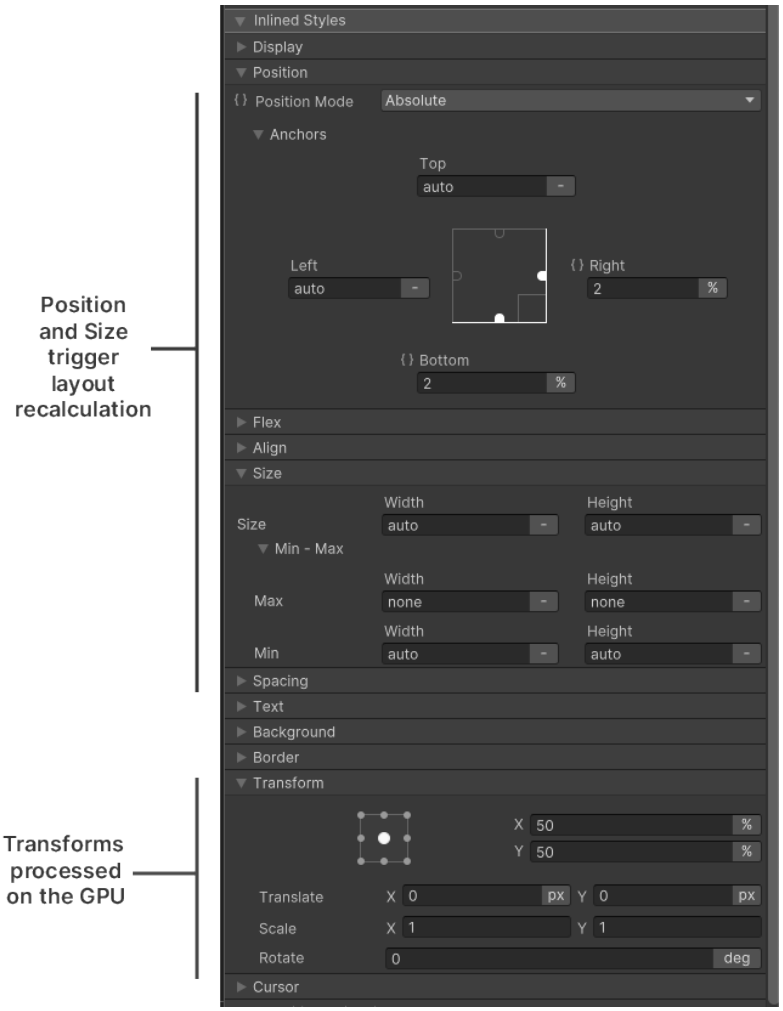
- ステンシル操作を避けるには、可能な限り矩形のマスクを使用します。
- マスクのネスティング深度を最小化します。マスクを階層内でフラットに保つことで、ステンシル再計算の回数を減らすことができます。
- 可能な場合は、子要素に複数のマスクを使用するのではなく、親要素に単一のマスクを使用します。
- 複数のマスキンググレイヤーが避けられない場合は、マスクコンテナの使用に関するヒントを適用して、ステンシル状態の設定を最適化します。ただし、バッチが分割されないように、控えめに使用してください。



マスクコンテナの使用に関するヒントを使用します。

最後に、フレームデバッガーを使用してこれらの最適化の影響を検証し、レンダリングとバッチ処理を効率化します。

アニメーションと遷移



レイアウトプロパティの代わりに Transform をアニメーション化します。

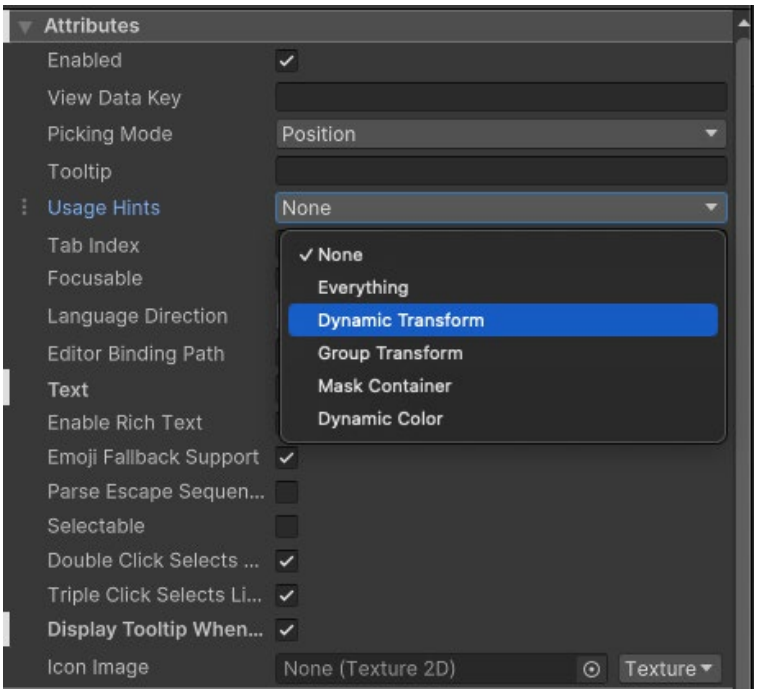
UI Toolkit の USS 遷移はシンプルなプロパティアニメーションを提供しますが、サイズや位置などのレイアウトプロパティを変更すると、高コストなレイアウト再計算をトリガーする可能性があります。アニメーションを最適化してパフォーマンスオーバーヘッドを減らすために、いくつかの戦略を試すことができます。

まず、レイアウトプロパティの変更よりも Transform ベースのアニメーションを優先します。`Width`、`Height`、`Top`、`Left` などのプロパティをアニメーション化するのではなく、Transform の`Translate`、`Scale`、`Rotate`を使用します。これらの操作は GPU 上で直接処理されるため、レイアウトを再計算する必要がありません。その結果、アニメーションが滑らかになります。

アニメーション化する必要のある任意のビジュアル要素で **ヒント** を有効にすることもできます。

DynamicTransform のヒントにより、UI Toolkit で GPU の位置と Transform の更新を処理し、高コストな頂点データの再計算を回避します。

アニメーション化された複数の子を持つ親コンテナの場合、**GroupTransform** のヒントを使用するとオーバーヘッドを大幅に削減できます。単一の Transform を親に適用し、GPU がそれをすべての子要素に効率的に伝播することで、大規模なグループのアニメーションを最適化します。



各ビジュアル要素に使い方のヒントが用意されています。

また、原則として、大規模な階層のスタイル変更のためにアニメーション中にクラスを切り替えることは避けるようにしてください。クラスの変更は、特に複雑な UI 構造において、広範なスタイル再計算を引き起こします。代わりに、インラインプロパティの変更を使用してスタイルを直接更新し、計算コストを最小化します。

最後に、Unity のフレームデバッガーを使用してアニメーションパフォーマンスを監視します。このツールを使用すると、これらの最適化が意図したとおりに機能していることを確認できます。

ランタイムデータバインディング

Unity のランタイムデータバインディングは、基礎となるデータの変更を自動的に反映することで UI 要素の更新を簡素化します。これにより、手動で更新する必要がなくなり、UI 開発がより効率的でメンテナンスが容易になります。

このプロセスを最適化できるテクニックもあります。

プロパティバッグとソース生成

プロパティバッグ は、特定の型のデータの効率的な走査と操作を可能にする補助オブジェクトです。デフォルトでは、Unity は型の初回アクセス時に リフレクション を使用してプロパティバッグを生成します。このリフレクションアプローチは便利ですが、プロパティバッグがまだ登録されていないときにのみ遅延して実行されるため、ランタイムのオーバーヘッドが少し高くなります。

パフォーマンスを改善するには、プロパティバッグの コード生成 を有効にします。型に `[Unity.Properties.GeneratePropertyBag]` というタグを付け、アセンブリにもコード生成用のタグを付けます。Unity はコンパイル時にプロパティバッグを生成して登録するため、ランタイム中のリフレクションは不要です。詳細については、プロパティバッグ のドキュメントを参照してください。

GeneratePropertyBag 属性は型全体を最適化しますが、個々のプロパティに CreateProperty 属性を追加すると、コンパイル時にバインディングコードを生成できます。これにより、プロパティを検出して接続するためのランタイムリフレクションが不要になり、より高速で効率的なデータバインディングが可能になります。

多くの場合、ランタイムデータバインディングの最適化には `[CreateProperty]` だけで十分です。ただし、効率的なシリアル化やすべてのプロパティの頻繁な走査など、型に追加の最適化が必要な場合は、`[CreateProperty]` を `[GeneratePropertyBag]` と組み合わせることで、全体的なパフォーマンスが最高になります。

変更追跡

ランタイムデータバインディングには、データバインディングの更新頻度を最適化する 2 つのインターフェースが含まれています。

- IDataSourceViewHashProvider:このインターフェースはハッシュベースの等価性チェックを提供し、データに重要な変更があったときにのみバインディングが更新されるようにします。
- INotifyBindablePropertyChanged:このインターフェースは、特定のプロパティ値が変更されたときにのみ更新をトリガーします。

これらのインターフェースは複雑な UI に特に役立ち、データに重要な変更がないときの不要な更新を防止します。

この ScriptableObject は、これらの最適化を実装するサンプルを示しています。

```
[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;
}
```

このクラス例では、[GeneratePropertyBag] を使用してコンパイル時にプロパティバッグを生成し、[CreateProperty] を使用して Name プロパティのランタイムデータバインディングを最適化します。

変更追跡のために、INotifyBindablePropertyChanged を実装します。Notify メソッドは、Name が更新されたときに propertyChanged イベントをトリガーします。これは変更を UI に通知し、すべてのリスナーに伝達します。

このクラスで IDataSourceViewHashProvider も実装します。GetViewHashCode メソッドは、Name が変更されるたびにバージョンが増加するハッシュを返すため、更新を簡単に検出できます。

要素の表示と非表示

UI 要素を非表示にするときは、不透明度を変更したり、画面外に移動したりするだけでは、パフォーマンスが最善とは限りません。これらの要素は、非表示にされていてもレイアウト計算、スタイルの更新、データバインディング操作に関与し、パフォーマンスに影響を与える可能性があります。

UI Toolkit には要素を非表示にする方法がいくつかあり、それぞれに代償があります。こちらの表で概要を確認してください。

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

異なる方法で要素を切り替えます。

UI 要素を非表示にする場合に、Opacity (不透明度) を 0 に設定したり、画面外に移動したりしても、GPU とレイアウトシステムで可視のままであり、Medium (中) から High (高) のレンダリングコストがかかります。これらのメソッドは遷移には便利ですが、メモリやレイアウトのオーバーヘッドは軽減されません。

要素の `Visible` プロパティを false に設定すると、レンダリングは行われなくなりますが、レイアウトの一部として保持されます。これは、ステンシルメモリの使用中に一時的に要素を隠すという妥協です。

より効率的なパフォーマンスのために、`style.display` 属性を `DisplayStyle.None` に設定すると、レンダリングとレイアウト更新が完全に停止します。ただし、これには要素を切り替えて元に戻す際のレイアウト再計算のコストが伴います。

ダイアログボックスや設定パネルなど、表示頻度の低い要素については、`RemoveFromHierarchy` を使用して階層から削除するだけで、進行中のオーバーヘッドを削減できます。レイアウトを完全に再構築する必要があるため、要素を再追加するとパフォーマンスのスパイクが高くなることに注意してください。

要素を切り替える頻度に基づいてメソッドを選択します。その後、短期的なレンダリングのニーズと長期的なパフォーマンスのバランスを取ります。

オーバードロー

UI Toolkit では、要素に透明度を持たせてレンダリングします。各ピクセルが複数回処理される可能性があるため、要素が重なると大幅なオーバードローとなる可能性があります。これは、UI Toolkit のウーバーシェーダーでは特に高コストになり、重複する要素の各レイヤーが複雑になります。透明または半透明の要素を何層も重ねると、パフォーマンスにさらに影響するおそれがあります。

オーバードローによるパフォーマンスへの影響を緩和するには、以下の戦略が役立ちます。

- 要素を完全に非表示にするために、`style.opacity = 0` の代わりに `style.display = DisplayStyle.None` を使用します。この場合も、要素は透明としてレンダリングされます。
- 複数の要素を重ねるのではなく、完全に隠れている要素を削除または非表示にします。
- スクロール可能なコンテンツを扱う場合は、ListViews を使用して仮想化を実装します。ListView は、画面上の可視要素だけを効率的にレンダリングできます。
- また、`style.overflow = Overflow.Hidden` を設定してコンテンツを特定の領域にクリップし、可視範囲外での不要なレンダリングを減らすこともできます。

メモリ管理

USS および UXML ファイルはフォント、テクスチャ、その他のアセットを直接参照します。これらのファイルを読み込むと、参照されているすべてのアセットがメモリに残ることになり、メモリ使用量が増加するおそれがあります。USS の例から、参照されたアセットを次に示します。

USS がアセットを参照します。

これらのアセットはインポートされると、使用していないときでもすぐにメモリを消費します。アセットを適切に管理しないと、メモリの非効率的な使用につながります。

アセット使用を最適化するには、次の戦略を検討してください。

- **Asset Bundle または Addressables の使用:**可能な場合は、特定のシーンやコンテキストに必要な UI ドキュメントとスタイルシートのみをロードします。これは、メモリ消費を抑えるのに役立ちます。
- **不要なアセットのアンロード:**UI 要素またはドキュメントが使用されなくなった場合は、`RemoveFromHierarchy` を使用して階層から削除します。次に、`Addressables.Release` または `AssetBundle.Unload(true)` を使用してアンロードし、ほかの操作のためにメモリを解放します。
- **複雑な UI 向けの選択的なロード:**サイズの大きな UXML ファイルや USS ファイルを、モジュール化された小さなテンプレート（VisualTreeAssets）に分割し、必要に応じて動的にロードします。表示される要素のリソースのみをロードすることで、メモリ使用量を低く抑えることができます。

プロファイリングツール

Unity には、アプリケーション内の UI パフォーマンスの問題を特定して解決するための複数のツールが用意されています。

Unity プロファイラー、UI Toolkit デバッガー、および フレームデバッガー は、パフォーマンス問題を診断するために不可欠です。これらのツールは、ドローコール、バッチのほか、レイアウトの再計算、スタイルの更新、頂点バッファの変更などの負荷の高い操作を分析するのに役立ちます。

UI 変更のビューをさらに細かくするには、Panel Settings から SetPanelChangeReceiver メソッドを使用します。これにより、UI に対する変更を監視し、そのソースを追跡できます。エディターと開発ビルドに限定されますが、速度低下の原因となる可能性がある特定の UI 動作を分離するのに便利です。

UI に対するすべての変更をログに記録するスクリプト例を次に示します。

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

これをゲームオブジェクトにアタッチし、Inspector で PanelSettings を設定するだけです。OnVisualElementChange メソッドは、ビジュアル要素 (レイアウト、スタイル、Transform など) が変更されるたびにトリガーされ、コンソールメッセージを記録します。これは、UI のどの部分が現在変更されているかを理解するのに役立ちます。

Unity 6 のパフォーマンス強化

Unity 6では、エディター環境とランタイム環境の両方で滑らかで応答性の高い体験を実現するために、さまざまなパフォーマンスの改善が行われています。

- **イベントディスパッチ:**イベントディスパッチルールが簡略化され、よりわかりやすく、2 倍高速になりました。
- **メッシュ生成の強化**:主要な改善点として、従来の要素ジオメトリ用のジョブ化されたジオメトリ生成と、ベクター API のネイティブ実装への遷移が挙げられます。テキストの生成も並列化されるようになりました。
- **カスタムジオメトリ API**:新しい公開 API により、開発者は同レベルのパフォーマンスでカスタムジオメトリを生成できるため、高度に最適化された UI コンポーネントを作成できます。
- **詳細階層レイアウトパフォーマンス**:レイアウト計算のキャッシングの改善により、深い階層でのパフォーマンスが大幅に向上し、より滑らかなユーザー体験を実現します。
- **大規模データセット用に最適化されたツリービュー**:ツリービューの制御は、以前は大規模なデータセットでは非効率的でしたが、Entities 専用の高パフォーマンスの新しいバックエンドによって強化されました。

その他のパフォーマンス最適化リソース

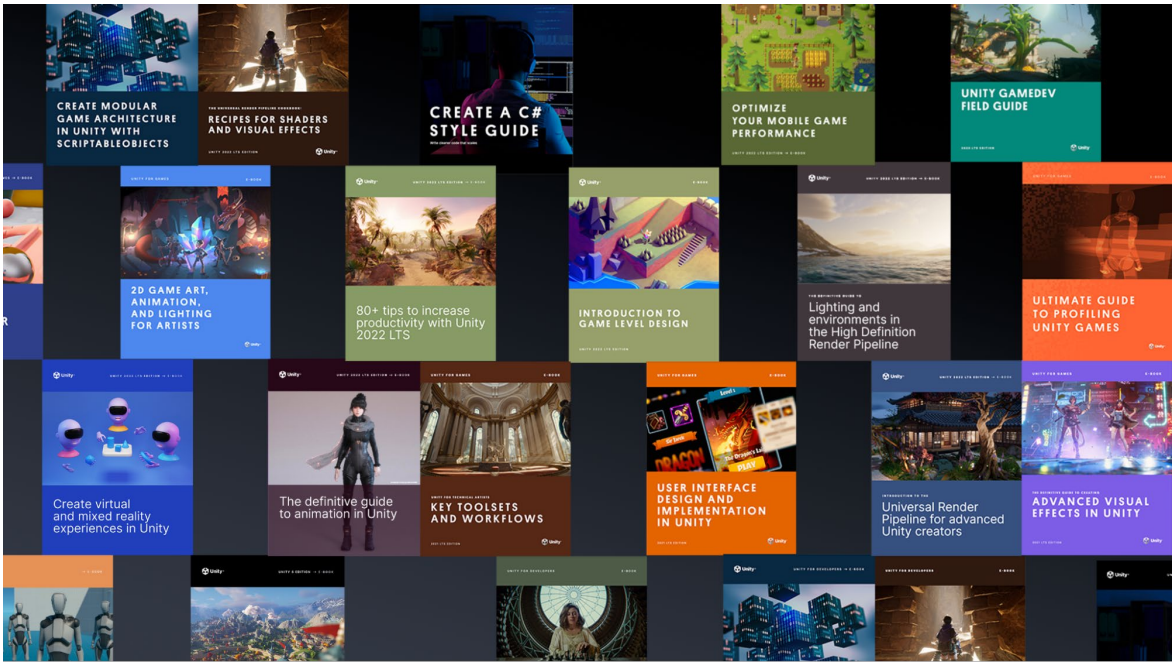
Unite 2024:UI Toolkit で最高のパフォーマンスを実現

eBook:Unity ゲームのプロファイリングに関する決定版ガイド

eBook:Unity におけるモバイル、XR、ウェブのパフォーマンス最適化

eBook:Unity によるコンソールと PC 向けゲームのパフォーマンス最適化

上級開発者およびアーティスト向けのリソース



Unity のベストプラクティスハブでは、上級 Unity 開発者およびクリエイター向けさまざまな eBook をダウンロードできます。業界の専門家や Unity のエンジニア、テクニカルアーティストによって作成された 30 以上のガイドの中からお選びください。Unity のツールセットやシステムを使った効率的な開発のベストプラクティスを紹介しています。

また、Unity Blog、UnityDiscussions、Unity Learn、**#unitytips** でヒント、ベストプラクティス、ニュースを確認できます。


Unity®