# **세련된 경험을 위한 고급 UI Toolkit 기법 v1.1**

기본적인 기능 위에, 이 섹션에서는 기능적인 UI를 전문적이고 역동적이며 세련된 사용자 경험으로 격상시키는 고급 UI Toolkit 기능들을 탐구합니다. 이러한 기술들은 살아 움직이는 듯한 느낌을 주고 유지보수가 효율적인 인터페이스를 만드는 데 매우 중요합니다.

### **5.1. 데이터 바인딩을 이용한 동적 UI**

데이터 바인딩은 데이터 소스(예: C# 객체의 속성)와 UI 요소의 속성(예: Label의 텍스트) 사이에 직접적인 연결을 생성합니다. 데이터 소스가 변경되면 UI가 자동으로 업데이트되어 수동 업데이트 코드가 필요 없어집니다. 이는 MVVM 패턴의 핵심입니다.

#### **구현 전략: ViewModel과 INotifyPropertyChanged**

가장 견고한 데이터 바인딩을 위해, ViewModel 클래스가 INotifyPropertyChanged 인터페이스를 구현하도록 합니다. 이 인터페이스는 속성이 변경될 때마다 PropertyChanged 이벤트를 발생시켜 UI가 즉시 업데이트되도록 보장합니다.

#### **구체적인 구현 방법**

1. **ViewModel 클래스 작성:**  
   using System.ComponentModel;  
   using System.Runtime.CompilerServices;  
     
   public class PlayerStatsViewModel : INotifyPropertyChanged  
   {  
    public event PropertyChangedEventHandler PropertyChanged;  
     
    private string \_playerName;  
    public string PlayerName  
    {  
    get => \_playerName;  
    set  
    {  
    if (\_playerName != value)  
    {  
    \_playerName = value;  
    OnPropertyChanged(); // 속성 변경 알림  
    }  
    }  
    }  
     
    private int \_playerLevel;  
    public int PlayerLevel   
    {  
    get => \_playerLevel;  
    set  
    {  
    if (\_playerLevel != value)  
    {  
    \_playerLevel = value;  
    OnPropertyChanged();  
    OnPropertyChanged(nameof(PlayerLevelText)); // 파생 속성도 변경 알림  
    }  
    }  
    }  
     
    // UI 표시를 위한 파생 속성  
    public string PlayerLevelText => $"Lv. {PlayerLevel}";  
     
    // PropertyChanged 이벤트를 발생시키는 헬퍼 메서드  
    protected void OnPropertyChanged([CallerMemberName] string propertyName = null)  
    {  
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));  
    }  
   }
2. **UI 컨트롤러에서 ViewModel 연결:**  
   using UnityEngine;  
   using UnityEngine.UIElements;  
     
   public class PlayerHUDController : MonoBehaviour  
   {  
    private PlayerStatsViewModel \_viewModel;  
     
    void Start()  
    {  
    var root = GetComponent<UIDocument>().rootVisualElement;  
    \_viewModel = new PlayerStatsViewModel { PlayerName = "DungeonMaster", PlayerLevel = 1 };  
     
    // 루트 요소에 ViewModel 인스턴스를 바인딩 소스로 설정합니다.  
    root.dataSource = \_viewModel;  
    }  
     
    // 예시: 5초마다 레벨업  
    void Update()  
    {  
    if (Time.frameCount % 300 == 0)  
    {  
    \_viewModel.PlayerLevel++;  
    }  
    }  
   }
3. **UI Builder에서 바인딩 설정:**
   * 플레이어 이름을 표시할 Label을 선택합니다.
   * 인스펙터의 Binding 섹션에서 Binding Path에 PlayerName을 입력합니다.
   * 레벨을 표시할 Label을 선택하고, Binding Path에 PlayerLevelText를 입력합니다.

이제 코드에서 \_viewModel.PlayerLevel++가 실행될 때마다, PlayerLevel과 PlayerLevelText 속성에 바인딩된 UI Label들이 자동으로 업데이트됩니다.

#### **주의점**

* **바인딩 경로 오류:** Binding Path는 ViewModel의 public 속성 이름과 대소문자까지 정확히 일치해야 합니다. 오타는 가장 흔한 오류 원인이지만, 콘솔에 명확한 에러가 출력되지 않아 찾기 어려울 수 있습니다.
* **성능:** INotifyPropertyChanged는 리플렉션을 사용하지 않아 매우 빠릅니다. 하지만 매우 많은 수의 요소를 매우 빈번하게 업데이트해야 하는 극단적인 경우, 성능을 고려해야 할 수 있습니다.

### **5.2. 재사용 가능한 커스텀 컨트롤 제작**

캐릭터 초상화와 체력 바, 이름이 결합된 UI처럼 복잡하거나 자주 사용되는 UI 컴포넌트의 경우, 커스텀 VisualElement 서브클래스를 만드는 것이 효율적입니다. 이는 구조(UXML), 스타일(USS), 로직(C#)을 하나의 재사용 가능한 컴포넌트로 캡슐화합니다.

#### **구체적인 구현 방법: UnitStatusBadge 컨트롤**

1. **컨트롤의 내부 구조 정의 (UnitStatusBadge.uxml):**  
   <ui:VisualElement xmlns:ui="UnityEngine.UIElements" class="badge-root">  
    <ui:VisualElement name="HealthBarBackground" class="health-bar-background">  
    <ui:VisualElement name="HealthBarForeground" class="health-bar-foreground" />  
    </ui:VisualElement>  
    <ui:Label name="UnitNameLabel" text="Unit Name" class="unit-name-label" />  
   </ui:VisualElement>
2. **컨트롤의 스타일 정의 (UnitStatusBadge.uss):**  
   .badge-root { flex-direction: column; align-items: center; }  
   .health-bar-background { width: 100px; height: 10px; background-color: grey; border-radius: 5px; }  
   .health-bar-foreground { width: 100%; height: 100%; background-color: red; border-radius: 5px; }  
   .unit-name-label { font-size: 12px; -unity-font-style: bold; }
3. **컨트롤의 로직 및 UXML 노출 정의 (UnitStatusBadge.cs):**  
   using UnityEngine;  
   using UnityEngine.UIElements;  
     
   public class UnitStatusBadge : VisualElement  
   {  
    // UI Builder에 이 컨트롤을 노출시키기 위한 팩토리 클래스  
    public new class UxmlFactory : UxmlFactory<UnitStatusBadge, UxmlTraits> { }  
     
    // UI Builder 인스펙터에서 설정할 수 있는 커스텀 속성 정의  
    public new class UxmlTraits : VisualElement.UxmlTraits  
    {  
    UxmlStringAttributeDescription m\_UnitName =  
    new UxmlStringAttributeDescription { name = "unit-name", defaultValue = "Default Name" };  
    UxmlFloatAttributeDescription m\_HealthPercent =  
    new UxmlFloatAttributeDescription { name = "health-percent", defaultValue = 1f };  
     
    public override void Init(VisualElement ve, IUxmlAttributes bag, CreationContext cc)  
    {  
    base.Init(ve, bag, cc);  
    var ate = ve as UnitStatusBadge;  
     
    ate.UnitName = m\_UnitName.GetValueFromBag(bag, cc);  
    ate.HealthPercent = m\_HealthPercent.GetValueFromBag(bag, cc);  
    }  
    }  
     
    private Label \_nameLabel;  
    private VisualElement \_healthBar;  
     
    public string UnitName  
    {  
    get => \_nameLabel.text;  
    set => \_nameLabel.text = value;  
    }  
     
    public float HealthPercent  
    {  
    get => \_healthBar.style.width.value.value / 100f;  
    set => \_healthBar.style.width = new Length(Mathf.Clamp01(value) \* 100, LengthUnit.Percent);  
    }  
     
    public UnitStatusBadge()  
    {  
    // UXML 템플릿 로드 (템플릿 파일은 Resources 폴더에 있어야 함)  
    var asset = Resources.Load<VisualTreeAsset>("UnitStatusBadge");  
    asset.CloneTree(this);  
     
    // 내부 요소 쿼리 및 캐싱  
    \_nameLabel = this.Q<Label>("UnitNameLabel");  
    \_healthBar = this.Q<VisualElement>("HealthBarForeground");  
     
    // 스타일시트 적용  
    var styleSheet = Resources.Load<StyleSheet>("UnitStatusBadge");  
    this.styleSheets.Add(styleSheet);  
    }  
   }

이제 UI Builder의 Library 탭에 UnitStatusBadge가 나타나며, 다른 컨트롤처럼 드래그 앤 드롭으로 사용하고 인스펙터에서 unit-name과 health-percent를 직접 설정할 수 있습니다.

[심화 노트] 상속보다 조합 (Composition over Inheritance)

위 예제는 VisualElement를 상속하지만, 실제로는 내부에 UXML로 정의된 여러 요소(Label, VisualElement 등)를 \*\*조합(Composition)\*\*하여 하나의 컨트롤을 만드는 방식입니다. 이는 Unity가 권장하는 패턴으로, 복잡한 C# 클래스를 상속받는 것보다 유연하고 재사용성이 높습니다.

### **5.3. USS와 C#을 이용한 UI 애니메이션**

UI Toolkit은 CSS와 유사한 트랜지션과 C# 조작을 통해 UI에 생동감을 불어넣을 수 있습니다.

#### **방법 1: USS 트랜지션을 이용한 상태 기반 애니메이션**

USS 트랜지션은 :hover (마우스 오버), :active (클릭 중) 같은 의사 클래스(pseudo-class)나 C#으로 추가/제거하는 일반 클래스에 반응하여 부드러운 상태 변화를 만듭니다.

* **클래스 기반 상태 변화 애니메이션 예시:**  
  /\* 기본 상태 \*/  
  .notification-panel {  
   opacity: 0;  
   translate: 0 -50px; /\* 화면 위쪽에 숨겨진 상태 \*/  
   transition: opacity 0.5s ease-out, translate 0.5s ease-out;  
  }  
    
  /\* 'visible' 클래스가 추가되었을 때의 상태 \*/  
  .notification-panel.visible {  
   opacity: 1;  
   translate: 0 0; /\* 제자리로 슬라이드되며 나타남 \*/  
  }  
  ```csharp  
  // C#에서 상태 변경  
  private VisualElement \_notificationPanel;  
    
  void ShowNotification() {  
   \_notificationPanel.AddToClassList("visible");  
  }  
    
  void HideNotification() {  
   \_notificationPanel.RemoveFromClassList("visible");  
  }

#### **방법 2: C#과 schedule을 이용한 애니메이션**

더 복잡한 애니메이션이나 시간 기반 애니메이션은 C#의 스케줄러를 사용하여 구현할 수 있습니다.

* **'흔들림(Wobble)' 효과 예시:**  
  private VisualElement \_iconToWobble;  
    
  public void WobbleIcon()  
  {  
   float duration = 0.5f;  
   float startTime = Time.time;  
    
   \_iconToWobble.schedule.Execute(() =>  
   {  
   float elapsed = Time.time - startTime;  
   if (elapsed >= duration)  
   {  
   \_iconToWobble.style.rotate = new Rotate(Angle.None());  
   return; // 애니메이션 종료  
   }  
    
   float progress = elapsed / duration;  
   float angle = Mathf.Sin(progress \* 10 \* Mathf.PI) \* (1 - progress) \* 15f; // 시간이 지남에 따라 진폭 감소  
   \_iconToWobble.style.rotate = new Rotate(new Angle(angle, AngleUnit.Degree));  
    
   }).Every(16).Until(() => Time.time - startTime >= duration);  
  }

#### **주의점**

성능을 위해 레이아웃 재계산을 유발하지 않는 속성에 애니메이션을 적용하는 것이 가장 좋습니다. **transform 속성(translate, rotate, scale)과 opacity는 애니메이션에 매우 최적화**되어 있습니다. width, height, margin과 같은 속성에 애니메이션을 적용하면 성능이 저하될 수 있습니다.

### **5.4. 여러 해상도에 걸친 UI 반응성 보장**

반응형 UI는 다양한 화면 크기와 종횡비에 자연스럽게 적응합니다. 이는 Flexbox, 백분율, 미디어 쿼리, 그리고 PanelSettings의 조합을 통해 달성됩니다.

#### **구현 전략 1: 모바일 우선 접근법과 미디어 쿼리**

1. **기본 스타일 (모바일 세로):** 가장 작은 화면(모바일 세로)을 기준으로 기본 레이아웃을 column 기반으로 설계합니다.
2. **Flexbox와 백분율:** flex-grow를 사용하여 요소들이 공간을 유연하게 채우도록 하고, width와 height에 px 대신 %를 적극적으로 사용합니다.
3. **미디어 쿼리:** 더 넓은 화면을 위해 미디어 쿼리를 사용하여 스타일을 덮어씁니다.

* **USS 스타일 예시 (모바일 우선):**  
  /\* 기본 (모바일) 스타일: 모든 것이 세로로 쌓임 \*/  
  .container {  
   flex-direction: column;  
   width: 100%;  
   height: 100%;  
  }  
  .main-content {  
   flex-grow: 1; /\* 남은 공간 모두 차지 \*/  
  }  
  .side-bar {  
   height: 200px;  
   width: 100%;  
  }  
    
  /\* 태블릿 및 데스크톱 스타일: 화면 너비가 768px 이상일 때 적용 \*/  
  @media (min-width: 768px) {  
   .container {  
   flex-direction: row; /\* 가로로 배치 \*/  
   }  
   .side-bar {  
   width: 30%;  
   height: 100%;  
   }  
  }

#### **구현 전략 2: PanelSettings를 이용한 전체 스케일링**

PanelSettings 에셋은 UGUI의 CanvasScaler와 유사하게, 전체 UI의 크기를 기준 해상도에 맞춰 조절하는 강력한 기능을 제공합니다.

1. **PanelSettings 에셋 생성:** Assets > Create > UI Toolkit > Panel Settings Asset을 통해 에셋을 생성합니다.
2. **UI Document에 연결:** 씬의 UI Document 컴포넌트에 생성한 PanelSettings 에셋을 연결합니다.
3. **주요 속성 설정:**
   * **Scale Mode:** Scale With Screen Size로 설정합니다.
   * **Reference Resolution:** UI를 디자인한 기준 해상도를 입력합니다 (예: 1920 x 1080).
   * **Screen Match Mode:** Match Width Or Height로 설정하고, 슬라이더를 조절하여 너비와 높이 중 어느 쪽에 더 비중을 두고 스케일링할지 결정합니다. (0 = 너비 기준, 1 = 높이 기준)

이 두 가지 전략을 함께 사용하면, PanelSettings로 전체적인 UI 크기를 맞추고, 미디어 쿼리와 Flexbox로 특정 해상도에서의 레이아웃을 세밀하게 조정하는 견고한 반응형 UI를 구축할 수 있습니다.

### **부록: 심화 학습 및 추천 자료**

#### **A.1. 리액티브 프로그래밍과 데이터 바인딩 자동화**

INotifyPropertyChanged를 수동으로 구현하는 것이 번거롭다면, **UniRx (Reactive Extensions for Unity)** 와 같은 리액티브 프로그래밍 라이브러리를 도입하는 것을 고려해볼 수 있습니다. UniRx는 데이터의 '흐름(Stream)'을 관찰하고 반응하는 방식으로, 복잡한 비동기 처리나 데이터 연동 로직을 훨씬 간결하고 우아하게 작성할 수 있도록 도와줍니다. 대규모 프로젝트에서 데이터 바인딩 코드를 크게 줄여줄 수 있는 강력한 대안입니다.

#### **A.2. 외부 애니메이션 라이브러리 연동**

USS 트랜지션은 간단한 상태 변화에 적합하지만, 여러 요소가 순차적으로 움직이는 복잡한 연출에는 한계가 있습니다. 이 경우, **DOTween**과 같은 전문 트위닝 라이브러리를 사용하는 것이 훨씬 효율적입니다. C# 스크립트에서 DOTween을 사용하여 VisualElement의 transform이나 style 속성을 제어하면, 정교하고 아름다운 UI 애니메이션을 쉽게 구현할 수 있습니다.

#### **A.3. 실무 UI/UX 참고 자료 및 추천 에셋**

훌륭한 UI는 다른 좋은 예시들을 참고하는 것에서 시작됩니다. 아래 자료들은 영감을 얻고 실무적인 UI 키트를 탐색하는 데 도움이 될 수 있습니다.

* **영감 및 학습:**
  + **Game UI Database:** 실제 출시된 다양한 게임들의 UI 스크린샷을 모아놓은 사이트로, 레이아웃, 아이콘, 색상 사용 등에 대한 영감을 얻기 좋습니다.
* **Unity 에셋 스토어 추천 UI 키트:**
  + **GUI PRO Kit - Fantasy RPG:** 판타지 RPG 장르에 어울리는 고품질의 UI 에셋과 프리팹을 제공합니다.
  + **Modern UI Pack:** 깔끔하고 현대적인 스타일의 UI 요소들을 포함하고 있어, 다양한 장르에 범용적으로 사용하기 좋습니다.