# **Unity UI Toolkit 터치 프로토타이핑 가이드 (v1.7)**

**v1.7 변경 내역 (최종):**

* 부록에 공간 분할(Spatial Partitioning) 등 고급 충돌 탐색 기법 주석 추가
* PinchZoomController에 예외 처리 방어 로직을 추가하여 안정성 강화
* 향후 다른 입력 시스템(게임패드 등)과의 연동 가능성을 명시하여 확장성 강조
* 롱터치, 두 손가락 스와이프 등 추가적인 고급 제스처 구현 방향성 제시

이 문서는 추상적인 아키텍처와 실질적인 게임플레이 사이의 간극을 메우는 것을 목표로 합니다. 그레이박싱을 통해 애플리케이션의 구조를 신속하게 구성하고, 키보드나 게임패드가 아닌 **오직 터치 인터페이스**를 통해 사용자와의 핵심적인 상호작용 채널을 구축하는 데 중점을 둡니다.

## **1. 그레이박싱 단계: 기능 중심의 프로토타이핑**

그레이박싱(또는 화이트박싱)은 간단한 기본 도형(큐브, 구, 평면 등)을 사용하여 레벨이나 UI 레이아웃의 기능적 프로토타입을 제작하는 기법입니다.⁵⁶ 최종 아트 에셋을 제작하는 데 드는 시간과 비용 없이 핵심 메커니즘, 흐름, 규모를 테스트하는 것이 목표입니다.⁵⁹

### **1.1. 구체적인 구현 방법**

* Unity의 내장 3D 오브젝트(GameObject > 3D Object)를 사용하여 벽, 플랫폼, 캐릭터, 적, UI 패널과 같은 핵심 요소를 표현합니다.⁵⁷
* 미학보다는 기능에 집중합니다. 간단한 색상의 머티리얼을 사용하여 오브젝트 유형을 구분할 수 있습니다(예: 플레이어는 파란색, 적은 빨간색, 정적 환경은 회색).⁵⁷
* 이 단계는 핵심 게임 루프와 UI 내비게이션을 구현하고 테스트하는 시기입니다. 예를 들어, 플레이어가 메인 메뉴에서 설정 화면으로 이동했다가 다시 돌아올 수 있는지, 플레이어의 이동 능력에 비해 게임 공간의 크기가 적절한지 등을 검증합니다.

### **1.2. 주의점**

세부 사항에 얽매이지 않도록 주의해야 합니다. 그레이박싱의 목적은 속도와 반복적인 개선입니다. 이 단계에서는 성능이 중요한 고려 사항이 아니며, 플레이스홀더 지오메트리는 나중에 최종 에셋으로 교체될 것입니다.⁵⁹

## **2. 터치 전용 입력 시스템 구현**

모바일 플랫폼을 주 타겟으로 하므로, 별도의 액션 기반 입력 시스템 설정 없이 UI Toolkit에 내장된 이벤트 시스템을 활용하여 직관적인 터치 상호작용을 구현합니다. 이는 "점프"나 "발사" 같은 추상적인 액션 대신, UI 요소에 대한 사용자의 직접적인 터치(탭, 드래그 등)에 반응하는 방식입니다.

### **2.1. 핵심 원리: 직접적인 UI 상호작용과 제스처 인식**

터치 기반 UI 게임에서 주된 "입력 시스템"은 사실상 UI Toolkit의 내장 이벤트 시스템입니다. 사용자의 모든 상호작용은 PointerDownEvent, PointerUpEvent, ClickEvent와 같은 이벤트의 형태로 UI 요소에 직접 전달됩니다. 따라서 별도의 입력 액션 에셋을 설정할 필요 없이, 이러한 이벤트를 수신하고 처리하는 데 집중합니다.

### **2.2. 기본 터치 입력 처리 (탭/클릭)**

가장 기본적인 터치 상호작용인 '탭'은 UI Toolkit의 ClickEvent로 간단하게 처리할 수 있습니다. 이 이벤트는 마우스 클릭과 화면 탭 모두에 동일하게 반응합니다.

**구체적인 구현 방법 (C#):**

using UnityEngine;  
using UnityEngine.UIElements;  
  
public class MainMenuController : MonoBehaviour  
{  
 // 인스펙터에서 할당할 ScriptableObject 이벤트 채널  
 public GameEventChannelSO UIEvents;  
  
 private Button startButton;  
  
 void OnEnable()  
 {  
 var root = GetComponent<UIDocument>().rootVisualElement;  
 startButton = root.Q<Button>("start-button");  
  
 if (startButton != null)  
 {  
 // 버튼의 'clicked' 이벤트에 콜백 메서드를 등록합니다.  
 startButton.clicked += OnStartButtonPressed;  
 }  
 }  
  
 void OnDisable()  
 {  
 // 오브젝트가 비활성화될 때 메모리 누수를 방지하기 위해 항상 콜백을 해제해야 합니다.  
 if (startButton != null)  
 {  
 startButton.clicked -= OnStartButtonPressed;  
 }  
 }  
  
 private void OnStartButtonPressed()  
 {  
 Debug.Log("시작 버튼이 터치(클릭)되었습니다!");  
   
 // 직접 게임 로직을 실행하는 대신, 전역 이벤트 채널을 통해 이벤트를 발생시킵니다.  
 // 이는 UI와 게임 로직의 결합도를 낮추는 좋은 아키텍처 패턴입니다.  
 if (UIEvents != null)  
 {  
 UIEvents.RaiseStartGameRequested();  
 }  
 }  
}

### **2.3. 복합 제스처 처리 (드래그 앤 드롭)**

몬스터 카드 배치와 같이 드래그 앤 드롭(Drag and Drop) 기능이 필요한 경우, 여러 포인터 이벤트를 조합하여 구현해야 합니다.

**드래그 앤 드롭 예시 코드:**

// 이 스크립트는 드래그 가능한 VisualElement에 적용되거나, 해당 요소를 제어합니다.  
private VisualElement draggableElement;  
private VisualElement dropSlot; // 드롭될 영역  
private Vector2 startDragPosition;  
private bool isDragging = false;  
  
void SetupDraggable(VisualElement element, VisualElement slot)  
{  
 draggableElement = element;  
 dropSlot = slot;  
  
 // PointerDown: 드래그 시작  
 draggableElement.RegisterCallback<PointerDownEvent>(e =>   
 {  
 startDragPosition = e.localPosition;  
 isDragging = true;  
 draggableElement.CapturePointer(e.pointerId);  
 });  
  
 // PointerMove: 드래그 중 위치 업데이트  
 draggableElement.RegisterCallback<PointerMoveEvent>(e =>  
 {  
 if (!isDragging) return;  
 Vector2 newPosition = draggableElement.transform.position + (e.localPosition - startDragPosition);  
 draggableElement.transform.position = newPosition;  
 });  
  
 // PointerUp: 드래그 종료  
 draggableElement.RegisterCallback<PointerUpEvent>(e =>  
 {  
 if (!isDragging) return;  
 isDragging = false;  
 draggableElement.ReleasePointer(e.pointerId);  
   
 // 드롭 가능 영역 감지: 드래그 중인 요소의 경계가 슬롯의 경계와 겹치는지 확인  
 if (dropSlot != null && draggableElement.worldBound.Overlaps(dropSlot.worldBound))  
 {  
 Debug.Log("드롭 성공!");  
 // 여기에 드롭 성공 로직을 구현합니다 (예: 슬롯에 카드 배치).  
 }  
 });  
}

#### **주의점**

* **성능:** 드래그나 스와이프 중 발생하는 PointerMoveEvent는 매우 빈번하게(매 프레임) 호출될 수 있습니다. 콜백 함수 내에서 UI 레이아웃을 재계산하는 복잡한 연산을 수행하면 성능 저하의 원인이 될 수 있습니다. **빈번한 드래그 위치 업데이트 시에는, 레이아웃 재계산을 유발하는 transform.position 대신 VisualElement.style.left와 VisualElement.style.top 속성을 직접 변경하는 것이 성능에 더 유리할 수 있습니다.**
* **성능 모니터링:** 빈번한 드래그 및 멀티터치 제스처 구현 시(부록 예시 참조), **Unity Profiler**를 통해 이벤트 호출량과 UI 레이아웃(Layout) 연산 비용을 주기적으로 점검하여 성능 병목 현상을 사전에 방지하는 것을 강력히 권장합니다.

### **2.4. 터치 입력을 게임플레이 로직에 연결하기**

가장 중요한 원칙은 UI 이벤트 콜백 함수(예: OnStartButtonPressed) 안에서 직접 무거운 게임 로직을 처리하지 않는 것입니다. UI 스크립트의 역할은 사용자 입력을 감지하고, 이를 시스템의 다른 부분에 알리는 것까지입니다.

**권장 흐름 다이어그램:**

+---------------------+ +------------------------+ +-------------------+  
| UI Element |----->| UI Controller |----->| Event Channel |  
| (e.g., StartButton) | | (e.g., MainMenu.cs) | | (ScriptableObject)|  
+---------------------+ +------------------------+ +-------------------+  
 | | |  
 (터치/클릭) (이벤트 감지) (이벤트 발행)  
 |  
 V  
+---------------------+ +------------------------+ +-------------------+  
| Game Logic |<-----| Game Manager |<-----| (이벤트 구독) |  
| (e.g., Scene Load) | | (e.g., GameManager.cs) | | |  
+---------------------+ +------------------------+ +-------------------+

이러한 **디커플링(decoupling)** 방식은 UI와 게임 로직을 완전히 분리하여, 향후 UI 디자인이 변경되더라도 핵심 게임 로직 코드를 수정할 필요가 없게 만들어 유지보수성과 확장성을 크게 향상시킵니다.

**[!] 중요:** UI 이벤트 콜백 함수 내에서 직접적인 게임 로직(예: 씬 로드, 데이터 계산)을 처리하는 것은 반드시 피해야 합니다. 이는 UI와 게임 로직 간의 강한 결합을 유발하여 코드의 재사용성과 테스트 용이성을 심각하게 저해합니다.

## **3. 철학적 연결: 기능과 상호작용의 추상화**

그레이박싱과 터치 기반 입력 시스템은 철학적으로 깊은 연관성을 가집니다.

* **그레이박싱**은 게임의 *기능*에 집중하기 위해 *시각적 요소*를 추상화합니다.
* **직접적인 터치 인터페이스**는 *플레이어의 의도*("이 버튼을 누른다", "이 카드를 저기로 옮긴다")에 집중하기 위해 마우스, 키보드, 게임패드와 같은 *물리적 하드웨어*를 추상화합니다.

이 두 가지를 함께 사용하면, 최종 아트 에셋이나 특정 장치에 얽매이지 않고, 타겟 플랫폼(모바일)의 핵심적인 사용자 경험과 게임플레이 루프를 매우 유연하고 추상적인 상태에서 신속하게 구축하고 테스트할 수 있는 강력한 프로토타이핑 워크플로우가 만들어집니다.

## **4. 부록: 고급 제스처 및 최적화 예시**

### **4.1. 드래그 앤 드롭 충돌 탐색 최적화 예시**

worldBound.Overlaps()는 간단하지만, 드롭 가능한 슬롯이 많아지면 모든 슬롯을 순회하며 검사해야 하므로 성능에 부담이 될 수 있습니다. Rect.Contains()를 사용하는 것이 더 직관적이고 효율적일 수 있습니다.

// PointerUp 이벤트 콜백 내에서  
private List<VisualElement> allDropSlots;  
  
// ... (PointerUp 로직)  
Vector2 dropPosition = e.position;  
VisualElement targetSlot = FindTargetSlot(dropPosition);  
  
if (targetSlot != null)  
{  
 Debug.Log($"드롭 성공: {targetSlot.name}");  
}  
  
// ...  
  
private VisualElement FindTargetSlot(Vector2 position)  
{  
 foreach (var slot in allDropSlots)  
 {  
 // worldBound는 스크린 좌표 기준이므로, position을 로컬 좌표로 변환할 필요 없이 바로 사용 가능  
 if (slot.worldBound.Contains(position))  
 {  
 return slot;  
 }  
 }  
 return null;  
 // <remarks>  
 // 슬롯이 수백 개 이상으로 매우 많아질 경우, 모든 슬롯을 순회하는 것은 비효율적입니다.  
 // 이 경우, 슬롯들을 그리드나 계층 구조로 묶어 탐색 범위를 줄이거나,  
 // 공간 분할(Spatial Partitioning) 기법(예: 쿼드트리)을 적용하여 충돌 탐색 성능을 극대화할 수 있습니다.  
 // </remarks>  
}

### **4.2. 핀치 투 줌 (Pinch-to-Zoom) 구현 예시**

멀티터치 제스처는 여러 개의 pointerId를 추적하여 구현합니다. 다음은 두 손가락을 이용한 핀치 투 줌의 기본적인 구현 예시입니다.

using UnityEngine;  
using UnityEngine.UIElements;  
using System.Collections.Generic;  
using System.Linq;  
  
public class PinchZoomController : MonoBehaviour  
{  
 private VisualElement zoomTarget;  
 private Dictionary<int, Vector2> activePointers = new Dictionary<int, Vector2>();  
 private float previousDistance = 0f;  
   
 // 스케일 제한 값  
 private Vector3 minScale = new Vector3(0.5f, 0.5f, 1f);  
 private Vector3 maxScale = new Vector3(3f, 3f, 1f);  
  
 void SetupZoomable(VisualElement element)  
 {  
 zoomTarget = element;  
  
 zoomTarget.RegisterCallback<PointerDownEvent>(e =>   
 {  
 // 멀티터치 이벤트는 포인터 수에 비례해 연산량이 증가하므로,  
 // 필요한 최대 포인터 수(여기서는 2개)를 초과하는 입력은 무시하여 불필요한 연산을 방지합니다.  
 if (activePointers.Count >= 2) return;  
   
 if (!activePointers.ContainsKey(e.pointerId))  
 {  
 activePointers.Add(e.pointerId, e.position);  
 }  
 });  
  
 zoomTarget.RegisterCallback<PointerMoveEvent>(e =>  
 {  
 if (activePointers.ContainsKey(e.pointerId))  
 {  
 activePointers[e.pointerId] = e.position;  
   
 // 두 개의 터치가 감지될 때만 줌 로직을 실행합니다.  
 if (activePointers.Count == 2)  
 {  
 PerformZoom();  
 }  
 }  
 });  
  
 zoomTarget.RegisterCallback<PointerUpEvent>(e =>  
 {  
 if (activePointers.ContainsKey(e.pointerId))  
 {  
 activePointers.Remove(e.pointerId);  
   
 // <remarks>  
 // 한 손가락이 떨어지면 제스처가 중단된 것으로 간주하고,  
 // 다음 제스처를 위해 이전 거리 값을 초기화합니다.  
 // </remarks>  
 previousDistance = 0f;  
 }  
 });  
 }  
  
 /// <summary>  
 /// 두 포인터 사이의 거리 변화를 기반으로 스케일링 로직을 수행합니다.  
 /// </summary>  
 private void PerformZoom()  
 {  
 // 예외 상황 방지: 두 개의 포인터가 아닐 경우 로직을 실행하지 않습니다.  
 if (activePointers.Count != 2) return;  
  
 // 두 포인터의 위치를 가져옵니다.  
 Vector2 pos1 = activePointers.Values.ElementAt(0);  
 Vector2 pos2 = activePointers.Values.ElementAt(1);  
 float currentDistance = Vector2.Distance(pos1, pos2);  
  
 // <remarks>  
 // 첫 프레임에서 의도치 않은 확대/축소를 방지하기 위해 이전 거리가 기록되어 있을 때만 실행합니다.  
 // </remarks>  
 if (previousDistance > 0)  
 {  
 float scaleFactor = currentDistance / previousDistance;  
 Vector3 newScale = zoomTarget.transform.scale \* scaleFactor;  
  
 // 의도치 않은 크기 변화를 막기 위해 스케일의 최소/최대값을 제한합니다.  
 newScale.x = Mathf.Clamp(newScale.x, minScale.x, maxScale.x);  
 newScale.y = Mathf.Clamp(newScale.y, minScale.y, maxScale.y);  
 newScale.z = 1f; // 2D UI에서는 z 스케일을 1로 고정  
  
 zoomTarget.transform.scale = newScale;  
 }  
  
 // 현재 거리를 다음 프레임을 위해 저장합니다.  
 previousDistance = currentDistance;  
 }  
}

## **5. 확장성 및 다음 단계**

이 가이드는 UI Toolkit의 포인터 이벤트를 활용한 터치 기반 프로토타이핑에 집중했지만, 이는 시작에 불과합니다.

* **고급 제스처 확장:** 본문에 설명된 포인터 이벤트 처리 패턴을 확장하면, 두 손가락 스와이프(패닝), 회전(Twist), 롱터치(Long-press) 등 더 복잡하고 풍부한 상호작용을 구현할 수 있습니다.
* **멀티플랫폼 대응:** 향후 프로젝트를 PC나 콘솔로 확장하여 게임패드, 마우스, 키보드 입력을 지원해야 할 경우, Unity의 **Input System 패키지**와 연동하여 액션 기반의 입력 처리를 추가하는 것을 고려할 수 있습니다. 이는 기존 터치 로직과 독립적으로 새로운 입력 방식을 유연하게 추가할 수 있게 해줍니다.