

Unreal Engine UI 개발 종합 가이드

목차

1. UI 시스템 개요
 2. C++와 Blueprint 역할 분담
 3. 프로페셔널 워크플로우
 4. 성능 최적화
 5. UI 구조 설계 패턴
 6. 실전 예제
 7. 디버깅과 프로파일링
 8. 베스트 퍼랙티스 셰크리스트
-

1. UI 시스템 개요

1.1 UMG (Unreal Motion Graphics)

UMG는 Unreal Engine의 주요 UI 시스템으로, 게임 내 UI 제작에 사용됩니다.

주요 특징:

- Widget Blueprint 기반의 비주얼 에디터
- Designer 탭: UI 요소 배치 및 디자인
- Graph 탭: 로직 및 이벤트 처리
- 다양한 기본 위젯 제공 (Button, Text, Image, ProgressBar 등)

언제 사용하는가:

- 게임 HUD, 메뉴, 인벤토리
- 일반적인 게임 UI 전반
- 빠른 프로토타이핑
- 디자이너/아티스트 협업이 필요한 경우

1.2 Slate

Slate는 C++ 기반의 저수준 UI 프레임워크입니다.

주요 특징:

- C++ 코드로 직접 작성
- Unreal Editor 자체가 Slate로 제작됨

- 선언적 문법 사용
- 최대한의 커스터마이징 가능

언제 사용하는가:

- 에디터 툴 및 플러그인 개발
- 커스텀 에디터 윈도우
- 극도로 복잡하거나 특수한 UI
- 최고 성능이 필요한 경우

관계:

```
UMG (고수준)
↓ 내부적으로 사용
Slate (저수준)
```

실무에서는 대부분 UMG를 사용하고, Slate는 에디터 확장이나 특수한 경우에만 사용합니다.

2. C++와 Blueprint 역할 분담

2.1 핵심 원칙

"C++는 구조, Blueprint는 표현"

이 원칙을 따르면:

- 성능 최적화
- 유지보수 용이성
- 효율적인 협업
- 명확한 책임 분리

2.2 C++ 담당 영역

C++에서 처리해야 하는 것들

1. 데이터 모델

```
cpp
```

```

UCLASS(Blueprintable, BlueprintType)
class UPlayerData : public UObject
{
    GENERATED_BODY()

private:
    float Health;
    float MaxHealth;
    ... int32 Level;

public:
    UFUNCTION(BlueprintPure, Category = "Player Data")
    float GetHealthPercent() const { return Health / MaxHealth; }

    UFUNCTION(BlueprintPure, Category = "Player Data")
    ... int32 GetLevel() const { return Level; }

    void UpdateHealth(float NewHealth);
};


```

2. Widget 베이스 클래스 및 API

```

cpp

UCLASS(Abstract)
class UPlayerHUDBase : public UUserWidget
{
    GENERATED_BODY()

protected:
    // Blueprint에서 구현할 이벤트
    UFUNCTION(BlueprintImplementableEvent, Category = "HUD")
    void OnHealthChanged(float Percent);

    UFUNCTION(BlueprintImplementableEvent, Category = "HUD")
    void OnLevelUp(int32 NewLevel);

    // Blueprint에서 호출 가능한 함수
    UFUNCTION(BlueprintCallable, Category = "HUD")
    void PlayDamageEffect();

public:
    // C++에서 호출하는 메인 업데이트
    void UpdateHUD(UPlayerData* Data);
};


```

3. 성능 크리티컬한 로직

- 매 프레임 실행되는 코드
- 복잡한 수학 계산
- 대량 데이터 처리
- AI 로직
- 물리 계산

4. 시스템 통합

cpp

```
void AMyPlayerController::BeginPlay()
{
    Super::BeginPlay();

    // 데이터 객체 생성
    PlayerData = NewObject<UPlayerData>(this);

    // Widget 생성 (Blueprint 클래스 사용)
    if (HUDWidgetClass)
    {
        HUDWidget = CreateWidget<UPlayerHUDBase>(this, HUDWidgetClass);
        HUDWidget->AddToViewport();
    }
}
```

2.3 Blueprint 담당 영역

Blueprint에서 처리해야 하는 것들

1. 비주얼 디자인

- 레이아웃 구성
- 색상, 폰트, 스타일링
- 이미지 및 아트 애셋 배치
- 앵커 및 정렬 설정

2. UI 애니메이션

- Widget Animation 제작
- Fade in/out
- 이동, 회전, 스케일 효과
- 트랜지션 효과

3. 이벤트 응답

Event OnHealthChanged (C++에서 정의)

- └─ Set Percent (ProgressBar_Health)
- └─ Play Animation (HealthFlash)
- └─ Update Color (빨간색으로 점멸)

Event OnLevelUp

- └─ Play Animation (LevelUpBurst)
- └─ Play Sound (LevelUpSound)
- └─ Show Notification

4. 프로토타이핑 및 트위킹

- 빠른 반복 작업
- 수치 조정 (애니메이션 속도, 위치 등)
- A/B 테스트
- 디자인 피드백 반영

2.4 역할별 책임

역할	C++	Blueprint
프로그래머	데이터 모델, Widget API, 시스템 통합, 성능 최적화	-
디자이너	-	레이아웃, 스타일, 애니메이션, 이벤트 응답
아티스트	-	비주얼 에셋 배치, 색상/폰트 설정, 이펙트

3. 프로페셔널 워크플로우

3.1 표준 개발 프로세스

Phase 1: 설계 및 C++ 구조 작성 (프로그래머)

Step 1: 요구사항 분석

- 어떤 UI가 필요한가? (HUD, 메뉴, 인벤토리 등)
- 어떤 데이터를 표시하는가?
- 어떤 인터랙션이 필요한가?
- 성능 요구사항은?

Step 2: 데이터 클래스 설계

cpp

```
// 예: 인벤토리 시스템
UCLASS()
class UInventoryData : public UObject
{
    GENERATED_BODY()

private:
    TArray<UItemData*> Items;
    int32 MaxSlots;

public:
    UFUNCTION(BlueprintPure)
    int32 GetItemCount() const { return Items.Num(); }

    UFUNCTION(BlueprintPure)
    TArray<UItemData*> GetItems() const { return Items; }

    UFUNCTION(BlueprintCallable)
    bool AddItem(UItemData* Item);

    UFUNCTION(BlueprintCallable)
    bool RemoveItem(UItemData* Item);
};
```

Step 3: Widget 베이스 클래스 작성

cpp

```

UCLASS(Abstract)
class UInventoryWidgetBase : public UUserWidget
{
    GENERATED_BODY()

protected:
    // Blueprint 구현 이벤트
    UFUNCTION(BlueprintImplementableEvent)
    void OnInventoryUpdated(const TArray<UItemData*>& Items);

    UFUNCTION(BlueprintImplementableEvent)
    void OnItemSelected(UItemData* Item);

    // Blueprint 호출 함수
    UFUNCTION(BlueprintCallable)
    void UseSelectedItem();

public:
    void RefreshInventory(UInventoryData* Data);

    virtual void NativeConstruct() override;
    virtual void NativeDestruct() override;
};

```

Step 4: 매크로 및 속성 정의

주요 매크로:

- **BlueprintCallable**: Blueprint에서 호출 가능한 함수
- **BlueprintPure**: 상태 변경 없는 Getter (실행 핀 없음)
- **BlueprintImplementableEvent**: Blueprint가 구현해야 하는 이벤트
- **BlueprintNativeEvent**: C++ 기본 구현 + Blueprint 오버라이드 가능
- **BlueprintReadOnly**: Blueprint에서 읽기만 가능
- **BlueprintReadWrite**: Blueprint에서 읽기/쓰기 가능

Phase 2: Blueprint 구현 (디자이너/아티스트)

Step 1: Widget Blueprint 생성

Content Browser 우클릭

- User Interface
- Widget Blueprint
- Parent Class: "InventoryWidgetBase" 선택
- 이름: "WBP_Inventory"

Step 2: Designer 탭에서 UI 배치

Canvas Panel

```
└─ Border_Background
    └─ Vertical Box
        ├─ Text_Title
        ├─ Horizontal Box_TopBar
        | ... └─ TextItemCount
        | ... └─ Button_Sort
        └─ Scroll Box_ItemList
            └─ (동적으로 생성될 아이템들)
```

Step 3: Graph 탭에서 이벤트 구현

Event OnInventoryUpdated

```
├─ Clear Children (Scroll Box_ItemList)
├─ ForEach Loop (Items)
| ... └─ Create Widget (WBP_ItemSlot)
| ... └─ Set Item Data
| ... └─ Add Child (Scroll Box_ItemList)
└─ Update Item Count Text
```

Event OnItemSelected

```
├─ Play Animation (SelectionPulse)
├─ Update Detail Panel
└─ Enable Use Button
```

Step 4: 애니메이션 추가

Animations:

- Fadeln: Opacity 0→1 (0.3초)
- FadeOut: Opacity 1→0 (0.2초)
- SelectionPulse: Scale 1→1.1→1 (0.2초)
- NewItemNotification: 이동 + Fade (1초)

Phase 3: 통합 및 테스트

Step 1: C++에서 Widget 생성

cpp

```
void APlayerController::ShowInventory()
{
    if (!InventoryWidget)
    {
        InventoryWidget = CreateWidget<UInventoryWidgetBase>(
            this,
            InventoryWidgetClass // Blueprint 클래스 할당
        );
    }

    // 데이터로 업데이트
    InventoryWidget->RefreshInventory(PlayerInventoryData);

    // 화면에 표시
    InventoryWidget->AddToViewport();
}
```

Step 2: 테스트

- 기능 테스트: 모든 버튼과 인터랙션
- 데이터 테스트: 다양한 데이터 상황
- 성능 테스트: 프로파일링 (아래 섹션 참조)

Phase 4: 최적화 및 폴리싱

최적화 체크리스트:

- Tick 비활성화 확인
- Binding 제거 (이벤트 기반으로 전환)
- Invalidation Box 적용
- 불필요한 위젯 Collapsed 처리
- 애니메이션 최적화

폴리싱:

- 애니메이션 타이밍 조정
- 사운드 이펙트 추가
- 시각적 피드백 개선
- 접근성 고려 (색상 대비, 폰트 크기)

3.2 Epic Games의 실전 패턴

Epic의 UI 팀이 사용하는 구조:

데이터 계층 (C++)

- └─ UObject 기반 데이터 클래스
 - ├─ Getter/Setter
 - ├─ 비즈니스 로직
 - └─ 데이터 검증

Widget API 계층 (C++)

- └─ UUserWidget 베이스 클래스
 - ├─ BlueprintImplementableEvent
 - ├─ BlueprintCallable 함수
 - └─ 시스템 통합 로직

표현 계층 (Blueprint)

- └─ Widget Blueprint
 - ├─ 비주얼 레이아웃
 - ├─ 스타일링
 - ├─ 애니메이션
 - └─ 이벤트 구현

예: Fortnite 상점 시스템

cpp

```
// 데이터
UStoreOfferData (UObject)
├─ ItemName, Price, Image
├─ OfferType (Featured, Normal, Daily)
└─ Getters

// Widget API
UStoreWidget (UUserWidget, Abstract)
├─ GenerateOffers() - BlueprintImplementableEvent
├─ OnOfferPurchased() - BlueprintImplementableEvent
└─ PurchaseOffer() - BlueprintCallable
```

```
UOfferWidgetBase (UUserWidget, Abstract)
├─ SetOffer()
└─ OnOfferSet() - BlueprintImplementableEvent
```

```
// Blueprint 구현
WBP_Store (Blueprint)
└─ C++ 이벤트 구현, 레이아웃
```

```
WBP_OfferTileLarge / WBP_OfferTileSmall (Blueprint)
└─ 각기 다른 비주얼 표현
```

4. 성능 최적화

4.1 UI 렉의 주요 원인

1. 과도한 Tick/Update

문제:

cpp

```
// ❌ 나쁜 예
void UMyWidget::NativeTick(const FGeometry& MyGeometry, float InDeltaTime)
{
    Super::NativeTick(MyGeometry, InDeltaTime);

    // 매 프레임 업데이트 - 비효율적!
    ... HealthText->SetText(FText::AsNumber(GetPlayerHealth()));
    ... AmmoText->SetText(FText::AsNumber(GetPlayerAmmo()));
}
```

해결책:

cpp

```
// ✅ 좋은 예: 이벤트 기반
void UMyWidget::NativeConstruct()
{
    Super::NativeConstruct();

    // Tick 비활성화
    SetIsFocusable(false);
}

void UMyWidget::UpdateHealth(float NewHealth)
{
    if (CurrentHealth != NewHealth)
    {
        CurrentHealth = NewHealth;
        HealthText->SetText(FText::AsNumber(NewHealth));
    }
}
```

2. Property Binding 남용

문제:

```
// Blueprint에서 Text에 직접 바인딩  
GetPlayerHealth() ← 매 프레임 호출!
```

Property Binding은 매 프레임 호출되므로:

- 복잡한 계산 ❌
- 데이터베이스 조회 ❌
- 배열 검색 ❌

해결책:

cpp

```
// C++에서 명시적으로 업데이트  
void UPlayerHUD::OnHealthChanged(float NewHealth)  
{  
    ... HealthText->SetText(FText::AsNumber(NewHealth));  
}
```

3. Invalidation 미사용

Invalidation이란? UI가 변경되지 않았을 때 재계산을 건너뛰는 캐싱 메커니즘입니다.

비유:

Invalidation 없이:

매번 레스토랑 메뉴를 새로 확인 (낭비)

Invalidation 사용:

메뉴판을 보고, 재료가 바뀔 때만 메뉴판 업데이트 (효율적)

사용법:

Canvas Panel

```
└─ Invalidation Box ← 추가!  
    └─ Vertical Box  
        ├─ Text_PlayerName (거의 안 바뀜)  
        ├─ Image_Avatar (거의 안 바뀜)  
        └─ Text_Level (가끔 바뀜)
```

언제 사용:

- 정적인 UI (플레이어 이름, 아바타)
- 가끔 바뀌는 UI (레벨, 퀘스트 목록)
- 매 프레임 바뀌는 UI (체력바, 타이머)

C++에서 사용:

cpp

```
void UMyWidget::NativeConstruct()
{
    Super::NativeConstruct();
    SetCanCache(true); // Invalidation 활성화
}

void UMyWidget::UpdateUI()
{
    // UI 업데이트
    PlayerNameText->SetText(NewName);

    // 명시적으로 다시 그리기 요청
    Invalidate();
}
```

4. 너무 많은 위젯

문제:

Scroll Box

— 1000개의 Item Widget (모두 활성화)

해결책 1: Object Pooling

cpp

```

class UWidgetPool : public UObject
{
    ... TArray<UUserWidget*> AvailableWidgets;
    ... TArray<UUserWidget*> ActiveWidgets;

public:
    UUserWidget* GetWidget()
    {
        if (AvailableWidgets.Num() > 0)
        {
            // 재사용
            UUserWidget* Widget = AvailableWidgets.Pop();
            ActiveWidgets.Add(Widget);
            Widget->SetVisibility(ESlateVisibility::Visible);
            return Widget;
        }

        // 새로 생성
        UUserWidget* NewWidget = CreateWidget<UUserWidget>(
            GetWorld(),
            WidgetClass
        );
        ActiveWidgets.Add(NewWidget);
        return NewWidget;
    }

    void ReturnWidget(UUserWidget* Widget)
    {
        Widget->SetVisibility(ESlateVisibility::Collapsed);
        Widget->RemoveFromParent();
        ActiveWidgets.Remove(Widget);
        AvailableWidgets.Add(Widget); // 풀에 반환
    }
};

```

해결책 2: ListView 사용

```

// UMG에서 제공하는 가상화된 리스트
List View
└─ 보이는 항목만 렌더링
    (스크롤 시 재사용)

```

5. 복잡한 머티리얼/이펙트

피해야 할 것:

- Blur 효과 (매우 무거움)
- 과도한 Translucency
- 복잡한 UI 머티리얼
- 실시간 마스킹

대안:

- 간단한 머티리얼 사용
- Opaque 또는 Masked 모드
- 정적 이미지로 대체
- Blur는 특별한 경우만 사용

6. 메모리 누수

문제:

cpp

```
// ❌ Widget을 계속 생성만 하고 제거 안 함
void ShowDamageNumber(float Damage)
{
    UUserWidget* Widget = CreateWidget<UUserWidget>(
        this,
        DamageWidgetClass
    );
    Widget->AddToViewport();
    // 제거하지 않음!
}
```

해결책:

cpp

```

// ✓ 타이머로 자동 제거
void ShowDamageNumber(float Damage)
{
    UUserWidget* Widget = CreateWidget<UUserWidget>(
        this,
        DamageWidgetClass
    );
    Widget->AddToViewport();

    // 2초 후 제거
    FTimerHandle TimerHandle;
    GetWorld()->GetTimerManager().SetTimer(
        TimerHandle,
        [Widget]()
    {
        if (Widget && Widget->IsValidLowLevel())
        {
            Widget->RemoveFromParent();
        }
    },
    2.0f,
    false
);
}

```

4.2 최적화 체크리스트

필수 체크리스트:

- Tick 최소화 또는 비활성화
- Property Binding 제거 (이벤트 기반으로 전환)
- Invalidation Box 적절히 사용
- 보이지 않는 UI는 Collapsed 처리
- 대량 리스트는 ListView/TileView 사용
- Widget Pooling 구현 (빈번한 생성/삭제 시)
- 복잡한 머티리얼 단순화
- 메모리 누수 확인 (RemoveFromParent)
- Text 업데이트 최소화
- Translucency 최소화

4.3 수많은 UI On/Off 시나리오

상황: 게임 진행 중 지정된 위치에서 수많은 UI들이 수시로 on/off

권장 구조:

MainHUD Widget

```
└─ Invalidation Box
    └─ UI_A (Collapsed/Visible)
    └─ UI_B (Collapsed/Visible)
    └─ UI_C (Collapsed/Visible)
    └... (많은 UI들)
```

이유:

1. Visibility 토글은 매우 가벼움
2. CreateWidget/RemoveFromParent보다 훨씬 빠름
3. 생성/소멸 오버헤드 없음
4. Invalidation Box로 추가 최적화

구현:

cpp

```
// C++
void UMainHUD::ShowUI(FName UIName, bool bShow)
{
    UUserWidget* Widget = UIMap.FindRef(UIName);
    if (Widget)
    {
        ESlateVisibility NewVisibility = bShow ?
            ESlateVisibility::Visible :
            ESlateVisibility::Collapsed;

        Widget->SetVisibility(NewVisibility);
    }
}
```

// Blueprint

Event Graph:

Show Damage Indicator
└ Show UI("DamageIndicator", true)

Hide Objective Marker

└ Show UI("ObjectiveMarker", false)

Visibility 옵션:

- **Visible**: 보이고 상호작용 가능
- **Collapsed**: 안 보이고 공간도 차지 안 함 (추천)

- `Hidden`: 안 보이지만 공간은 차지
 - `HitTestInvisible`: 보이지만 상호작용 불가
 - `SelfHitTestInvisible`: 자신은 불가, 자식은 가능
-

5. UI 구조 설계 패턴

5.1 단일 Widget vs 다중 Widget

패턴 1: 단일 Widget (Switcher 패턴)

구조:

```
MainWidget
└─ Widget_Switcher
    └─ Panel_Menu
    └─ Panel_Gameplay
    └─ Panel_Inventory
    └─ Panel_Settings
```

장점:

- 한 번만 생성
- 빠른 전환
- 데이터 공유 쉬움

단점:

- 높은 초기 메모리
- Widget 복잡도 증가

사용 시기:

- 전환이 매우 빈번
- UI가 가볍고 개수가 적당 (10-20개)
- 같은 데이터 공유

패턴 2: 다중 Widget (동적 생성)

구조:

```
GameHUD
└─ Container
    └─ (필요시 Widget 생성/제거)
```

장점:

- 메모리 효율적
- 독립적 관리

단점:

- CreateWidget 오버헤드
- GC 부담

사용 시기:

- UI가 무겁거나 개수가 많음
- 전환이 드뭅
- 완전히 독립적

패턴 3: 하이브리드

구조:

```
MainHUD
└── CoreUI (항상 표시)
    ├── HealthBar
    ├── Minimap
    └── QuickSlots
    |
    └── DynamicUIContainer
        └── (필요시 생성)
            ├── Dialog
            ├── Shop
            └── Inventory
```

사용 시기:

- 복합적인 UI 시스템
- 일부는 항상, 일부는 가끔 필요

5.2 계층 구조

권장 구조:

Game Instance

└ Game Mode / Player Controller

 └ HUD Actor (optional)

 └ Main HUD Widget

 └ Persistent UI

 └ Health/Stamina

 └ Minimap

 └ Quick Inventory

 |

 |

 └ Dynamic UI Container

 └ (런타임 생성 UI)

 |

 └ Popup Manager

 └ (Modal 팝업들)

5.3 데이터 흐름

게임 로직 (C++)

↓

데이터 모델 업데이트

↓

Widget API 호출

↓

Blueprint 이벤트 발생

↓

UI 업데이트

예:

플레이어 피격

↓

ACharacter::TakeDamage()

↓

UPlayerData::UpdateHealth()

↓

UPlayerHUD::UpdateHUD()

↓

OnHealthChanged 이벤트

↓

ProgressBar 업데이트

6. 실전 예제

6.1 기본 HUD 시스템

C++ 구현

PlayerHUDData.h

cpp

```

#pragma once
#include "CoreMinimal.h"
#include "UObject/NoExportTypes.h"
#include "PlayerHUDDData.generated.h"

UCLASS(Blueprintable, BlueprintType)
class MYGAME_API UPlayerHUDDData : public UObject
{
    GENERATED_BODY()

private:
    float Health;
    float MaxHealth;
    float Stamina;
    float MaxStamina;
    ... int32 AmmoCount;
    ... int32 MaxAmmo;

public:
    UPlayerHUDDData();

    // Getters
    UFUNCTION(BlueprintPure, Category = "HUD Data")
    float GetHealthPercent() const;

    UFUNCTION(BlueprintPure, Category = "HUD Data")
    float GetStaminaPercent() const;

    UFUNCTION(BlueprintPure, Category = "HUD Data")
    ... int32 GetAmmo() const;

    UFUNCTION(BlueprintPure, Category = "HUD Data")
    ... int32 GetMaxAmmo() const;

    // Setters (C++ only)
    void SetHealth(float NewHealth);
    void SetStamina(float NewStamina);
    void SetAmmo(int32 NewAmmo);
};


```

PlayerHUDDData.cpp

cpp

```
#include "PlayerHUDData.h"

UPlayerHUDData::UPlayerHUDData()
{
    Health = 100.0f;
    MaxHealth = 100.0f;
    Stamina = 100.0f;
    MaxStamina = 100.0f;
    AmmoCount = 30;
    MaxAmmo = 30;
}

float UPlayerHUDData::GetHealthPercent() const
{
    return MaxHealth > 0 ? Health / MaxHealth : 0.0f;
}

float UPlayerHUDData::GetStaminaPercent() const
{
    return MaxStamina > 0 ? Stamina / MaxStamina : 0.0f;
}

int32 UPlayerHUDData::GetAmmo() const
{
    return AmmoCount;
}

int32 UPlayerHUDData::GetMaxAmmo() const
{
    return MaxAmmo;
}

void UPlayerHUDData::SetHealth(float NewHealth)
{
    Health = FMath::Clamp(NewHealth, 0.0f, MaxHealth);
}

void UPlayerHUDData::SetStamina(float NewStamina)
{
    Stamina = FMath::Clamp(NewStamina, 0.0f, MaxStamina);
}

void UPlayerHUDData::SetAmmo(int32 NewAmmo)
{
```

```
....AmmoCount = FMath::Clamp(NewAmmo, 0, MaxAmmo);
```

```
}
```

PlayerHUDWidget.h

cpp

```
#pragma once
#include "CoreMinimal.h"
#include "Blueprint/UserWidget.h"
#include "PlayerHUDData.h"
#include "PlayerHUDWidget.generated.h"

UCLASS(Abstract)
class MYGAME_API UPlayerHUDWidget : public UUserWidget
{
GENERATED_BODY()

protected:
    // Blueprint 구현 이벤트
    UFUNCTION(BlueprintImplementableEvent, Category = "HUD")
    void OnHealthChanged(float Percent);

    UFUNCTION(BlueprintImplementableEvent, Category = "HUD")
    void OnStaminaChanged(float Percent);

    UFUNCTION(BlueprintImplementableEvent, Category = "HUD")
    void OnAmmoChanged(int32 Current, int32 Max);

    UFUNCTION(BlueprintImplementableEvent, Category = "HUD")
    void OnLowHealthWarning(bool bIsLowHealth);

    // Blueprint 호출 가능 함수
    UFUNCTION(BlueprintCallable, Category = "HUD")
    void PlayHitEffect();

    UFUNCTION(BlueprintCallable, Category = "HUD")
    void ShowReloadPrompt();

public:
    // C++에서 호출하는 메인 업데이트
    UFUNCTION(BlueprintCallable, Category = "HUD")
    void UpdateHUD(UPlayerHUDData* Data);

    // 생명주기
    virtual void NativeConstruct() override;
    virtual void NativeDestruct() override;

private:
    float LastHealthPercent;
    bool bWasLowHealth;
};
```

PlayerHUDWidget.cpp

cpp

```
#include "PlayerHUDWidget.h"

void UPlayerHUDWidget::NativeConstruct()
{
    Super::NativeConstruct();

    // 초기화
    LastHealthPercent = 1.0f;
    bWasLowHealth = false;

    // Tick 비활성화 (성능 최적화)
    SetIsFocusable(false);
}

void UPlayerHUDWidget::NativeDestruct()
{
    Super::NativeDestruct();
}

void UPlayerHUDWidget::UpdateHUD(UPlayerHUDData* Data)
{
    if (!Data)
    {
        UE_LOG(LogTemp, Warning, TEXT("PlayerHUDWidget: Invalid data"));
        return;
    }

    // Health 업데이트
    float CurrentHealthPercent = Data->GetHealthPercent();
    if (!FMath::IsNearlyEqual(CurrentHealthPercent, LastHealthPercent))
    {
        OnHealthChanged(CurrentHealthPercent);
        LastHealthPercent = CurrentHealthPercent;

        // 체력 낮음 경고
        bool bIsLowHealth = CurrentHealthPercent < 0.3f;
        if (bIsLowHealth != bWasLowHealth)
        {
            OnLowHealthWarning(bIsLowHealth);
            bWasLowHealth = bIsLowHealth;
        }
    }

    // Stamina 업데이트
    OnStaminaChanged(Data->GetStaminaPercent());
}
```

```

// Ammo 업데이트
OnAmmoChanged(Data->GetAmmo(), Data->GetMaxAmmo());
}

void UPlayerHUDWidget::PlayHitEffect()
{
    // C++에서 구현 가능한 이펙트
    // 또는 Blueprint에서 추가 구현
}

void UPlayerHUDWidget::ShowReloadPrompt()
{
    // 재장전 프롬프트 표시 로직
}

```

Blueprint 구현

WBP_PlayerHUD (Widget Blueprint)

Designer 탭:

```

Canvas Panel
└─ Invalidation Box (Name: InvBox_Static)
    └─ Image_Background
    └─ Text_PlayerName
    └─ Image_Avatar
    |
    └─ Overlay (Name: Overlay_Dynamic)
        └─ Vertical Box_Stats
            └─ Progress Bar_Health
            └─ Progress Bar_Stamina
            └─ Text_Ammo
        |
        └─ Border_LowHealthWarning
            └─ (초기 Visibility: Hidden)

```

Graph 탭:

Event OnHealthChanged

```
|--- Set Percent (Progress Bar_Health, Percent)
|--- Branch (Percent < 0.5)
|   ... |--- True: Set Fill Color (Red)
|   ... |--- False: Set Fill Color (Green)
|--- Play Animation (Anim_HealthUpdate)
```

Event OnStaminaChanged

```
|--- Set Percent (Progress Bar_Stamina, Percent)
```

Event OnAmmoChanged

```
|--- Format Text ("{0} / {1}", Current, Max)
|--- Set Text (Text_Ammo)
```

Event OnLowHealthWarning

```
|--- Branch (bIsLowHealth)
|   ... |--- True: Set Visibility (Visible)
|   ... |     ... Play Animation (Anim_LowHealthPulse)
|   ... |--- False: Set Visibility (Hidden)
|   ... |     ... Stop Animation (Anim_LowHealthPulse)
```

통합 (PlayerController)

MyPlayerController.h

cpp

```
#pragma once
#include "CoreMinimal.h"
#include "GameFramework/PlayerController.h"
#include "PlayerHUDData.h"
#include "PlayerHUDWidget.h"
#include "MyPlayerController.generated.h"

UCLASS()
class MYGAME_API AMyPlayerController : public APlayerController
{
    GENERATED_BODY()

protected:
    // Blueprint에서 설정
    UPROPERTY(EditDefaultsOnly, Category = "UI")
    TSubclassOf<UPlayerHUDWidget> HUDWidgetClass;

    UPROPERTY()
    UPlayerHUDWidget* HUDWidget;

    UPROPERTY()
    UPlayerHUDData* HUDData;

    virtual void BeginPlay() override;

public:
    // 게임 로직에서 호출
    void UpdatePlayerHealth(float NewHealth);
    void UpdatePlayerStamina(float NewStamina);
    void UpdatePlayerAmmo(int32 NewAmmo);
};
```

MyPlayerController.cpp

cpp

```
#include "MyPlayerController.h"

void AMyPlayerController::BeginPlay()
{
    Super::BeginPlay();

    // 데이터 객체 생성
    HUDData = NewObject<UPlayerHUDData>(this);

    // Widget 생성
    if (HUDWidgetClass)
    {
        HUDWidget = CreateWidget<UPlayerHUDWidget>(this, HUDWidgetClass);
        if (HUDWidget)
        {
            HUDWidget->AddToViewport();
            HUDWidget->UpdateHUD(HUDData); // 초기 업데이트
        }
    }
}

void AMyPlayerController::UpdatePlayerHealth(float NewHealth)
{
    if (HUDData)
    {
        HUDData->SetHealth(NewHealth);

        if (HUDWidget)
        {
            HUDWidget->UpdateHUD(HUDData);
        }
    }
}

void AMyPlayerController::UpdatePlayerStamina(float NewStamina)
{
    if (HUDData)
    {
        HUDData->SetStamina(NewStamina);

        if (HUDWidget)
        {
            HUDWidget->UpdateHUD(HUDData);
        }
    }
}
```

```

void AMyPlayerController::UpdatePlayerAmmo(int32 NewAmmo)
{
    if (HUDData)
    {
        HUDData->SetAmmo(NewAmmo);

        if (HUDWidget)
        {
            HUDWidget->UpdateHUD(HUDData);
        }
    }
}

```

6.2 인벤토리 시스템 (복잡한 예제)

이 예제는 구조만 제시합니다. 전체 구현은 프로젝트 요구사항에 따라 확장하세요.

구조:

C++ 클래스:

- └── UItemData (아이템 데이터)
- └── UInventoryData (인벤토리 데이터)
- └── UInventoryWidget (Widget 베이스)
- └── UItemSlotWidget (개별 슬롯 베이스)

Blueprint:

- └── WBP_Inventory (인벤토리 메인 UI)
- └── WBP_ItemSlot (아이템 슬롯)
- └── WBP_ItemTooltip (툴팁)

핵심 기능:

- 드래그 앤 드롭
- 아이템 정렬/필터링
- 툴팁 표시
- 인벤토리 확장

7. 디버깅과 프로파일링

7.1 Widget Reflector

실행:

에디터에서: Ctrl + Shift + W

게임 중: 콘솔에서 'WidgetReflector'

기능:

- 현재 표시된 모든 위젯 확인
- 위젯 계층 구조 확인
- Invalidation 상태 확인
- 위젯 Pick (마우스로 선택)

색상 의미:

- 초록색: 캐시 사용 중 (최적화됨)
- 빨간색: 매 프레임 업데이트 중

7.2 콘솔 명령어

stat Slate

→ Slate/UMG 성능 확인

→ Widgets, Draw Calls, Invalidation 횟수

stat SlateVerbose

→ 더 자세한 정보

stat Game

→ 전체 게임 성능

stat UObjects

→ UObject 개수 확인 (메모리 누수 체크)

Slate.ShowInvalidationVisualizer 1

→ Invalidation 시각화

Slate.InvalidationDebugging 1

→ Invalidation 로그

7.3 Unreal Insights

실행:

1. UnrealInsights 실행
2. 게임에서 콘솔: 'trace.start'
3. 플레이
4. 콘솔: 'trace.stop'
5. Insights에서 분석

확인 사항:

- UI Tick 시간
- Widget 생성/소멸
- Draw Call 수
- CPU/GPU 사용량

7.4 일반적인 문제 해결

문제: UI가 느려요

체크리스트:

1. `stat Slate` 확인 → Widgets 개수가 많은가?
2. Widget Reflector → 빨간색이 많은가?
3. Tick이 활성화되어 있는가?
4. Property Binding을 사용하는가?
5. Invalidation Box를 사용하는가?

문제: 메모리가 계속 늘어나요

체크리스트:

1. `stat UObjects` 확인 → Widget 개수가 증가하는가?
2. CreateWidget 후 RemoveFromParent() 호출하는가?
3. Timer Handle을 제대로 정리하는가?
4. Widget Pool을 사용하는가?

문제: Widget이 표시되지 않아요

체크리스트:

1. AddToViewport() 호출했는가?
2. Visibility가 Visible인가?
3. ZOrder가 다른 Widget에 가려져 있는가?
4. Canvas Panel의 Anchor 설정이 올바른가?

5. Parent Widget이 표시되어 있는가?

8. 베스트 프랙티스 체크리스트

8.1 설계 단계

- UI 플로우차트 작성
- 필요한 데이터 정의
- C++/Blueprint 역할 명확히 분리
- 성능 요구사항 확인
- 재사용 가능한 컴포넌트 파악

8.2 C++ 구현

- 데이터는 UObject로 캡슐화
- Widget은 Abstract 베이스 클래스
- BlueprintImplementableEvent 적절히 사용
- BlueprintCallable로 필요한 함수만 노출
- Category 설정으로 가독성 향상

8.3 Blueprint 구현

- C++ 베이스 클래스 상속
- Designer에서 레이아웃 구성
- Graph에서 이벤트 구현
- 애니메이션 추가
- 모듈화 (재사용 가능한 위젯)

8.4 성능 최적화

- Tick 비활성화
- Property Binding 제거
- Invalidation Box 사용
- 보이지 않는 UI는 Collapsed
- ListView/TileView 사용 (대량 리스트)
- Widget Pooling (빈번한 생성/삭제)
- 복잡한 머티리얼 단순화
- Text 업데이트 최소화

8.5 테스트 및 디버깅

- Widget Reflector로 확인
- stat Slate로 성능 측정
- Unreal Insights로 프로파일링
- 다양한 해상도 테스트
- 메모리 누수 확인

8.6 협업

- 명확한 인터페이스 문서화
 - 일관된 네이밍 컨벤션
 - Blueprint는 바이너리 - 충돌 최소화
 - C++ API 변경 시 팀원에게 알림
 - 버전 관리 전략 수립
-

부록 A: 자주 사용하는 매크로

UFUNCTION 매크로

cpp

```
// Blueprint에서 호출 가능 (실행 편 있음)
UFUNCTION(BlueprintCallable, Category = "MyCategory")
void DoSomething();

// Blueprint에서 읽기 전용 (실행 편 없음, Getter용)
UFUNCTION(BlueprintPure, Category = "MyCategory")
int32 GetValue() const;

// Blueprint가 구현해야 하는 이벤트 (C++에서 호출)
UFUNCTION(BlueprintImplementableEvent, Category = "MyCategory")
void OnSomethingHappened();

// C++ 기본 구현 + Blueprint 오버라이드 가능
UFUNCTION(BlueprintNativeEvent, Category = "MyCategory")
void DoComplexThing();
// Implementation:
void UMyClass::DoComplexThing_Implementation() {}
```

UPROPERTY 매크로

cpp

```

// Blueprint에서 읽기만 가능
UPROPERTY(BlueprintReadOnly, Category = "MyCategory")
int32 ReadOnlyValue;

// Blueprint에서 읽기/쓰기 가능
UPROPERTY(BlueprintReadWrite, Category = "MyCategory")
float ReadWriteValue;

// 에디터에서 편집 가능
UPROPERTY(EditAnywhere, Category = "MyCategory")
FString EditableString;

// 에디터에서 기본값만 편집 가능
UPROPERTY(EditDefaultsOnly, Category = "MyCategory")
TSubclassOf<UUserWidget> WidgetClass;

// 에디터에서 인스턴스만 편집 가능
UPROPERTY(EditInstanceOnly, Category = "MyCategory")
AActor* TargetActor;

// 조합
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "MyCategory")
int32 CommonValue;

```

부록 B: 유용한 리소스

공식 문서

- Unreal Engine Documentation: <https://docs.unrealengine.com>
- UMG Best Practices: <https://www.unrealengine.com/en-US/tech-blog/umg-best-practices>
- API Reference: <https://docs.unrealengine.com/en-US/API/>

커뮤니티

- Unreal Slackers Discord
- Unreal Engine Forums
- Reddit: r/unrealengine

학습 자료

- Unreal Learning Portal
- YouTube: Official Unreal Engine Channel
- Udemy/Coursera Unreal Engine 강좌

마무리

이 가이드는 Unreal Engine의 UI 작업에 대한 종합적인 참고 자료입니다.

핵심 원칙 요약:

- C++는 구조, Blueprint는 표현
- 이벤트 기반 업데이트 (Tick과 Binding 최소화)
- Invalidation 활용
- 데이터와 UI 분리
- 지속적인 프로파일링과 최적화

실무에서는 프로젝트의 규모와 팀 구성에 따라 이 가이드를 조정하여 사용하세요.

버전: 1.0

최종 업데이트: 2025년 10월

대상: Unreal Engine 5.x

검증 노트

이 문서의 내용은 Unreal Engine 공식 문서와 대조 검증되었습니다:

✓ 검증된 내용

- UMG 기본 개념 - 공식 문서와 일치 확인
 - Widget Blueprint 생성 방법
 - Designer/Graph 탭 구조
 - CreateWidget, AddToViewport 사용법
- Slate 시스템 - 공식 문서와 일치 확인
 - C++ 기반 저수준 프레임워크
 - 에디터 자체가 Slate로 제작
 - UMG가 Slate 위에 구축됨
- Invalidation Box - 공식 문서와 완전 일치
 - 캐싱 메커니즘 설명 정확
 - Volatile 위젯 개념 정확
 - Widget Reflector 사용법 정확
 - 콘솔 명령어 정확 (`(SlateDebugger.Invalidate.Enable)`)
- 성능 최적화 - 공식 Best Practices와 일치

- Tick 비활성화 권장사항
- Property Binding 문제점
- 이벤트 기반 업데이트 권장

5. C++ 매크로 - 공식 API와 일치

- BlueprintImplementableEvent: Blueprint만 구현 가능
- BlueprintNativeEvent: C++ 기본 구현 + Blueprint 오버라이드
- BlueprintCallable, BlueprintPure 정확

6. NativeConstruct/NativeTick - 공식 API 확인

- NativeConstruct: Widget 초기화 시점
- NativeTick: 매 프레임 호출 (성능 주의)

⚠️ 보완/명확화 필요 사항

1. Blueprint Nativization

- 문서에서 언급하지 않았으나, UE4.27 이후 deprecated되어 UE5에서는 지원 안 됨
- 본 문서에서는 언급하지 않았으므로 문제없음

2. Common UI Plugin

- 간략히 언급했으나, 이는 선택적 플러그인이며 모든 프로젝트에서 필수는 아님
- 문서에서 정확히 "플러그인"으로 표기하여 적절

3. 성능 수치

- "약 25배 성능 향상" 등의 구체적 수치는 예시이며 프로젝트마다 다를 수 있음
- 문서에서 "테스트 시나리오" 맥락에서 제시하여 적절

✓ Epic Games 공식 패턴 확인

- UMG Best Practices 블로그 포스트의 패턴이 문서에 정확히 반영됨
- 데이터 클래스(UObject) + Widget 베이스(UUserWidget) + Blueprint 구현 구조 정확
- Epic의 실제 코드 예시(Fortnite 상점)와 유사한 패턴 제시

📝 추가 참고사항

- VALORANT의 Global Invalidation 사례는 실제 게임 적용 예시로 유효
- Widget Reflector 단축키 Ctrl+Shift+W 정확
- stat 명령어들 모두 공식 문서와 일치

