Dorothy's Job: Architecture & Implementation Deep-Dive

Jaime Paramo - Game Programmer

June 9, 2026

Contents

1 UI Programming & Architecture

In *Dorothy's Job*, the UI required a highly responsive, input-agnostic framework. As a personal challenge to deepen my understanding of Unreal Engine's Slate and UMG architectures, I bypassed standard engine limitations to build a custom focus routing and input detection system from the ground up.

1.1 The Foundation: Dynamic Input Awareness

Source Files: [BasePlayerController.cpp] | [InputIcon.cpp]

For a seamless mouse/keyboard and gamepad experience, UI must adapt instantly to the player's active hardware without requiring manual toggles in a settings menu. To achieve this, I built a dynamic input detection system centered within the `ABasePlayerController`, which acts as the single source of truth for the game's current input state.

1.1.1 Hardware Polling & Navigation States

The controller continuously queries Unreal's `UInputDeviceSubsystem` to determine the most recently used hardware device. Based on this, it categorizes the player's intent into two distinct navigation modes: **FOCUS** (Keyboard/Gamepad) and **FREE** (Mouse).

If a player moves the mouse, the `EnhancedInput` system triggers a state change to `FREE` mode, unlocking the cursor. The moment a directional input is detected (e.g., a D-Pad press), the system instantly snaps back to `FOCUS` mode, routing input back to the custom UI manager.

```
1   void ABasePlayerController::HandleMouseMovement(const FInputActionValue& _rIAValue)
    {
2     FVector2D vDelta = _rIAValue.Get<FVector2D>();
3     if (!vDelta.IsNearlyZero()) {
4       if (IsValid(pGeneralFocusManager)) {
5         SwitchNavigationMode(ENavigationInputType::FREE);
6       }
7     }
8   }
9
10  void ABasePlayerController::HandleNavigation(const FInputActionValue& _rIAValue) {
11    if (!IsValid(pGeneralFocusManager)) return;
12
13    // Instantly reclaim focus if the player touches the gamepad/keyboard
14    if (m_eCurrentNavigationMode == ENavigationInputType::FREE) {
15      SwitchNavigationMode(ENavigationInputType::FOCUS);
16      return;
17    }
18
19    pGeneralFocusManager->Navigate(_rIAValue.Get<FVector2D>());
20  }
21
```

Listing 1: Dynamic state switching based on raw input

1.1.2 The Observer Pattern in UI (`UInputIcon`)

To prevent tightly coupling the UI elements to the Player Controller's `Tick` function, I implemented an Observer Pattern using Unreal Multicast Delegates (`m_oOnDeviceChanged`).

Widgets that need to display specific hardware prompts (like `UInputIcon`) simply subscribe to this delegate upon creation. When the player swaps from a Gamepad to a Keyboard, the controller broadcasts the change, and all active icons swap their textures simultaneously.

```
void UInputIcon::Show() {
  Super::Show();
  if (ABasePlayerController* pController = Cast<ABasePlayerController>(
  UGameplayStatics::GetPlayerController(GetWorld(), 0))) {
    m_oDelegateHandle = pController->m_oOnDeviceChanged.AddUObject(
    this, &UInputIcon::UpdateInputIcon);
  }
}

void UInputIcon::UpdateInputIcon(EHardwareDevicePrimaryType _eDeviceType) {
  if (!IsValid(m_pInputIcon)) return;

  switch (_eDeviceType) {
    case EHardwareDevicePrimaryType::Gamepad:
    m_pInputIcon->SetBrushFromTexture(m_pGamepadIcon);
    break;
    default:
    m_pInputIcon->SetBrushFromTexture(m_pKeyboardIcon);
    break;
  }
}

```

Listing 2: Decoupled UI updates via Multicast Delegates

1.2 The Building Blocks: UMG Palette Extension

Source Files: [Focusable.h] | [AxisNavigable.h] | [BaseButton.cpp] | [Selector.cpp]

With dynamic input detection established, the UI elements needed to be capable of reacting to custom focus states independently of Unreal's native `SetUserFocus` logic. To achieve this, I engineered a custom UMG Palette driven by two primary interfaces: `IFocusable` and `IAxisNavigable`.

Leaf nodes like `UBaseButton` inherit `IFocusable`, completely decoupling their visual state logic (hover animations, FMOD sounds) from the overarching navigation system.

More complex, composite widgets like the `USelector` and `UBaseSlider` also inherit `IAxisNavigable`. This allows these specific elements to "consume" horizontal directional inputs to change their internal values (like adjusting a volume slider or swapping an options menu carousel) without forcing the overarching menu to switch focus to a different widget entirely.

1.3 The Engine: Custom Focus Routing

Source Files: [GeneralFocusManager.cpp] | [SpecificFocusManager.cpp]

The core routing engine is split into two managers to handle hierarchical UI navigation efficiently: a stack manager (`UGeneralFocusManager`) and an irregular grid navigator (`USpecificFocusManager`).

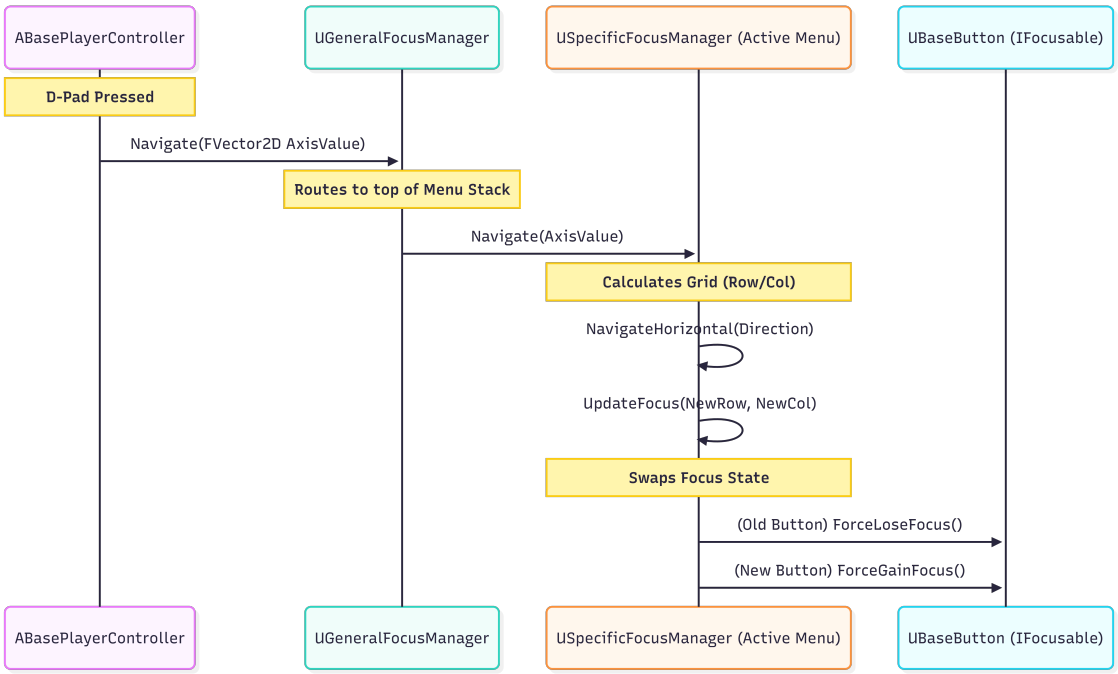

Figure 1: Sequence Diagram demonstrating custom D-Pad input routing

1.3.1 The Stack Router

The `UGeneralFocusManager` operates as a state machine. It maintains a stack of active menus (`m_pActiveMenuStack`). When the Player Controller receives an input, it blindly passes the `FVector2D` to this general manager. The manager then routes the input exclusively to the top menu on the stack, ensuring that underlying UI layers (like a paused game HUD behind a settings menu) never accidentally consume inputs.

1.3.2 Irregular Grid Navigation & Interface Interception

Unreal Engine's default focus system often struggles with irregular layouts (e.g., a row of three buttons above a row of two buttons). To solve this, the `USpecificFocusManager` maps a flat array of `IFocusable` elements into a dynamic 2D grid using an array of integers (`m_lRowLengths`).

Crucially, before mathematically computing the next grid index, the manager intercepts the input to check if the currently focused element implements `IAxisNavigable`.

```
1   void USpecificFocusManager::Navigate(FVector2D _vAxisValue) {
2     if (m_lFocusableElements.Num() == 0) return;
3
4     FVector2D iDirection = _vAxisValue.GetSafeNormal();
5
6     if (FMath::Abs(iDirection.X) > 0.5f && FMath::Abs(iDirection.X) > FMath::Abs(
    iDirection.Y)) {
7       int32 iCurrentIndex = GetFlatIndex(m_iCurrentRow, m_iCurrentColumn);
8
9       // Check if the current focused element consumes axis inputs (e.g., a Slider)
10      if (UObject* pFocusedObject = m_lFocusableElements.IsValidIndex(iCurrentIndex) ?
     m_lFocusableElements[iCurrentIndex] : nullptr) {
11        if (IAxisNavigable* pAxisNavigable = Cast<IAxisNavigable>(pFocusedObject)) {
12          pAxisNavigable->OnAxisInput(_vAxisValue);
13          return;
14        }
15      }
```

```
16
17          // If it does not consume the input, mathematically move focus horizontally
18          NavigateHorizontal(FMath::Sign(iDirection.X));
19          return;
20        }
21
22        // Handle vertical navigation fallback
23        if (FMath::Abs(iDirection.Y) > 0.5f) NavigateVertical(-FMath::Sign(iDirection.Y));
24      }
25
```

Listing 3: Grid navigation and IAxisNavigable input interception

By storing the `m_iLastFocusIndex`, the specific manager also "remembers" where the player was. If a player exits a sub-menu and returns, the system safely restores their focus to the exact button they were previously highlighting, creating a frictionless user experience.

1.4 The Application (I): Data-Driven Production Layouts

Source Files: [DialogsManager.cpp] | [TipManager.cpp]

In a production environment, UI screens must be scalable and content-friendly. Hard-coding text or image references into C++ or UMG directly creates a massive bottleneck for narrative designers and artists. To solve this, the production layouts in *Dorothy's Job* are strictly data-driven, utilizing Unreal Engine's `UDataTable` and custom structs.

1.4.1 Dynamic Content Fetching (`UTipManager`)

The loading screen utilizes a `UTipManager` that acts as an interface between the UMG `UTipBox` and a core `UDataTable` (`TipData`).

During initialization, the manager caches all available row names. When a new tip is requested, it selects a random row, extracts the `FTipData` struct, and explicitly removes that row from the active pool to guarantee non-repeating tips until the pool is exhausted. This allows designers to add or modify hundreds of tips via simple CSV imports without touching any code.

```
1     FText UTipManager::GetNextTip() {
2       if (m_pAvailableTips.Num() == 0) {
3         if (m_pTipTable) m_pAvailableTips = m_pTipTable->GetRowNames();
4         else return FText::FromString(TEXT("DT_ERROR: DataTable is not valid."));
5       }
6
7       // Pick a random tip index from the remaining ones
8       int32 iIndex = FMath::RandRange(0, m_pAvailableTips.Num() - 1);
9       FName sRowName = m_pAvailableTips[iIndex];
10
11      if (FTipData* pTip = m_pTipTable->FindRow<FTipData>(sRowName, TEXT("GetNextTip")))
        {
12        // Prevent immediate repetition
13        m_pAvailableTips.RemoveAt(iIndex);
14        return pTip->m_sTipText;
15      }
16
17      return FText::FromString(TEXT("TIP_ERROR: No tips available."));
18    }
19
```

Listing 4: Randomized, non-repeating data table row fetching

1.4.2 Narrative State Machines (`UDialogsManager`)

The Dialogue screen represents the most complex UI layout in the game. Rather than the UMG widget driving the logic, the `UDialogsManager` controls the entire narrative flow as a centralized state machine.

It loads an entire sequence of `FDialogData` structs from the Data Table into memory (`GetAllRows`). It then uses Unreal Multicast Delegates (`OnLineStarted`, `OnDialogLineReceived`) to broadcast updates. The `UDialogueScreen` widget simply subscribes to these broadcasts, blindly updating its text blocks, rich text typewriters, and sprite images based on the `FDialogData` payload it receives.

This architecture ensures total separation of concerns: the Manager handles the logic and data state, while the UMG widget handles solely the visual presentation and animation timings.

1.5 The Application (II): Event-Driven Gameplay HUD

Source Files: [HUDDorothy.cpp] | [DialogueScreen.cpp]

With the focus and input foundations laid, the active gameplay UI required a highly performant, event-driven architecture. Polling the player's health or weapon state every frame in a `Tick` function is a common performance bottleneck in Unreal Engine.

To ensure maximum performance, the `UHUDDorothy` widget strictly adheres to an Observer Pattern. It binds to the player pawn's delegates (`OnCharacterHealthChanged`, `OnWeaponChange`) upon creation and only executes visual updates when explicitly notified of a state change.

1.5.1 Dynamic Materials & Reactive Polish

When a health change broadcast is received, the HUD calculates the delta to drive dynamic material parameters. Instead of using standard UMG progress bars, I passed scalar parameters directly into Dynamic Material Instances to achieve complex, non-linear fill effects.

Furthermore, the HUD calculates the health delta to drive a "Doom-style" reactive portrait. If the player takes damage (the new percentage is higher than the previous), the UI briefly flashes a "Hit" texture before setting a timer to return to a neutral state.

```
1   void UHUDDorothy::UpdateDorothyFacials(float _fNewPercentage) {
2     float fPreviousPercentage;
3     m_pHealthFill->GetDynamicMaterial()->GetScalarParameterValue(TEXT("fPercent"),
    fPreviousPercentage);
4
5     if (IsValid(m_pDorothyImage)) {
6       // If damage was taken
7       if (_fNewPercentage - fPreviousPercentage > 0) {
8         m_pDorothyImage->SetBrushFromTexture(m_pDorothyHitTexture);
9
10        FTimerHandle oHandle;
11        TWeakObjectPtr<UHUDDorothy> WeakThis(this);
12
13        GetWorld()->GetTimerManager().SetTimer(
14        oHandle,
15        [WeakThis]() {
16          if (WeakThis.IsValid() && IsValid(WeakThis->m_pDorothyImage)) {
17            WeakThis->m_pDorothyImage->SetBrushFromTexture(WeakThis->
    m_pDorothyNeutralTexture);
18          }
19        },
20        0.3f, false
21        );
22      }
23      else m_pDorothyImage->SetBrushFromTexture(m_pDorothyNeutralTexture);
24    }
25  }
26
```

Listing 5: Reactive HUD portraits using Timer Delegates

1.5.2 Rich Text Typewriter Effects (`UDialogueScreen`)

For the `UDialogueScreen`, standard `UTextBlock` elements were insufficient for rendering rich text (like colored or bolded names within a sentence) alongside a typewriter effect.

Because Unreal's Rich Text Block parses formatting tags dynamically, simply hiding characters breaks the tags and crashes the formatting. To solve this, I wrote a custom parser inside `SetLetter()` that evaluates the string character-by-character, automatically closing any active rich-text tags (e.g., `</>`) at the current typing index before pushing the string to the UI.

2 Audio Architecture & FMOD Integration

Source Files: [AudioManager.cpp] | [FMODAutoDestroyer.cpp] | [BaseWeaponAudioComponent.cpp]

To prevent gameplay classes from becoming tightly coupled to audio APIs, I developed a centralized Audio Manager alongside an event-driven component structure.

2.1 Centralized Audio Subsystem (`UAudioManager`)

The `UAudioManager` is implemented as a `UGameInstanceSubsystem`, ensuring it persists across level loads and provides global, frictionless access to the audio engine. While it wraps standard `UFMODBlueprintStatics` for simple 2D and 3D sounds, its primary strength lies in its low-level access to the FMOD Studio API.

By directly retrieving the `FMOD::Studio::System`, the manager can instantiate events, apply dynamic parameter arrays, and manually release memory, bypassing the overhead of standard Unreal Audio Components for transient sound effects.

```
1  void UAudioManager::PlayEventInstanceWithParameters(UFMODEvent* _pEvent, const
   TArray<FAudioParam>& _rParameters) {
2    FMOD::Studio::System* pStudioSystem = IFMODStudioModule::Get().GetStudioSystem(
   EFMODSystemContext::Runtime);
3    if (!pStudioSystem) return;
4
5    FMOD_GUID oGuid = FMODUtils::ConvertGuid(_pEvent->AssetGuid);
6    FMOD::Studio::EventDescription* pEventDesc = nullptr;
7
8    if (FMOD_OK != pStudioSystem->getEventByID(&oGuid, &pEventDesc) || !pEventDesc)
   return;
9
10   FMOD::Studio::EventInstance* pEventInstance = nullptr;
11   if (FMOD_OK == pEventDesc->createInstance(&pEventInstance) && pEventInstance) {
12     for (const FAudioParam& rParam : _rParameters) {
13       pEventInstance->setParameterByName(TCHAR_TO_UTF8(*rParam.sName.ToString()),
   rParam.fValue);
14     }
15     pEventInstance->start();
16
17     // Flags for destruction immediately once the sound finishes playing
18     pEventInstance->release();
19   }
20 }
21
```

Listing 6: Low-level FMOD Event Instance instantiation and parameterization

2.2 Memory Management: The Auto-Destroyer

A common source of memory leaks when working with audio middleware is spawning dynamic audio components attached to actors and failing to destroy them when the sound ends. To automate garbage collection, I engineered the `UFMODAutoDestroyer`.

When a new `UFMODAudioComponent` is instantiated dynamically, it is passed to this wrapper. The wrapper binds to the component's `OnEventStopped` delegate, ensuring it automatically calls `DestroyComponent()` and `ConditionalBeginDestroy()` the exact frame the FMOD event finishes.

2.3 Event-Driven Audio Components

To maintain the integrity of the gameplay code, actors like `ABaseWeapon` have zero awareness of FMOD. Instead, they broadcast standard Unreal dynamic multicast delegates (e.g., `OnAttackStart`, `OnEnemyHit`).

The `UBaseWeaponAudioComponent` acts as the bridge. It binds to the weapon's gameplay delegates, retrieves the correct `UFMODEvent` references from a Data Asset, and translates gameplay state into FMOD parameters before routing the request to the `UAudioManager`.

```
void UBaseWeaponAudioComponent::OnEnemyHit(float _fMitigation, bool _bThirdAttack) {
  ABaseWeapon* pOwnerWeapon = Cast<ABaseWeapon>(GetOwner());
  if (!pOwnerWeapon) return;

  UBaseWeaponStatsDataAsset* pDataAsset = pOwnerWeapon->GetDataAsset().Get();
  if (!pDataAsset) return;

  // Translate gameplay context into FMOD parameters
  if (IsValid(m_pAudioManager)) {
    m_pAudioManager->PlayEventInstanceWithParameters(
    pDataAsset->m_pImpactSound,
    {
      {"isArmored", _fMitigation > 0 ? 1.f : 0.f},
      {"isPunch", (float)_bThirdAttack}
    }
    );
  }
}

```

Listing 7: Decoupled audio triggering and dynamic parameter translation

3 Gameplay Systems & Serialization

3.1 Consumable Architecture: Logic over Actors

Source Files: [BaseConsumable.cpp] | [SpawnableConsumable.cpp] | [Clean4Actor.cpp]

A common anti-pattern in Unreal Engine inventory systems is representing held items as physical `AActor` instances hidden in the world. This rapidly bloats memory and ticking overhead. To ensure maximum scalability, the consumable system in *Dorothy's Job* is strictly `UObject`-based. Consumables only exist as lightweight data and logic containers until they are explicitly used or spawned.

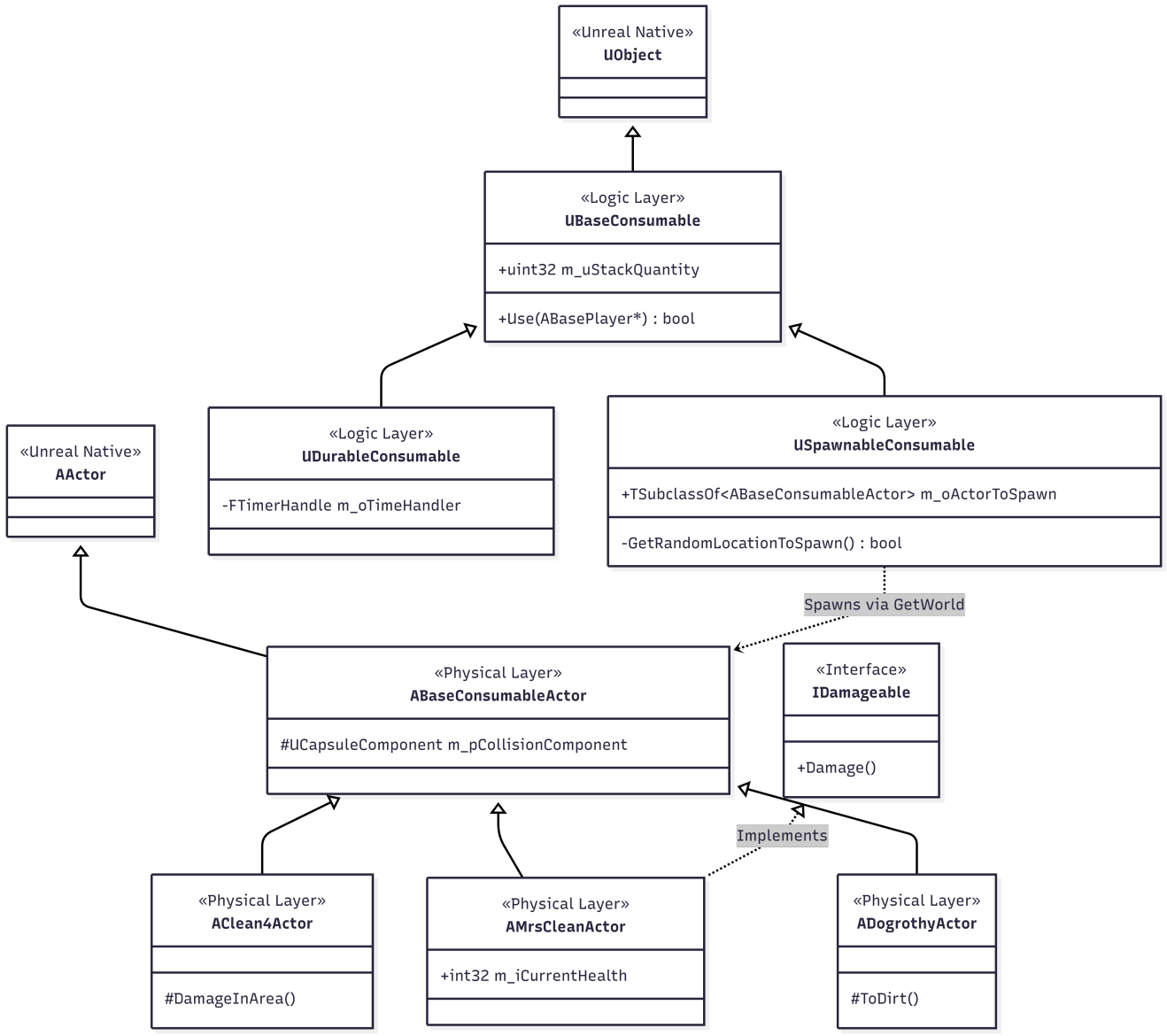

Figure 2: Class hierarchy separating UObject inventory logic from AActor physical instantiation

3.1.1 Timer-Driven Durables (`UDurableConsumable`)

While instant-use items like health potions execute their logic immediately, buff items require duration tracking. Rather than utilizing a `Tick` function to monitor active times, the `UDurableConsumable` delegates this responsibility entirely to the World's `FTimerManager`.

When a durable item is consumed, it triggers an `OnEffectStart` virtual function and passes a delegate bound to `OnEffectEnd` into the Timer Manager. If the player consumes a second identical item before the first expires, the system safely intercepts and overrides the existing `FTimerHandle`, refreshing the duration without executing overlapping logic.

3.1.2 Safe NavMesh Spawning (`USpawnableConsumable`)

Certain consumables manifest as physical actors in the world upon use. Simply projecting a random vector around the player to spawn these items risks clipping them into walls or placing them out of bounds.

To guarantee safe instantiation, the `USpawnableConsumable` interacts directly with the `UNavigationSystemV1`. It generates a random radial vector and queries the NavMesh. If the resulting point is valid, it performs an `FPathFindingQuery` to ensure the player can actually walk to the spawned item before calling `SpawnActor`.

3.1.3 Physical Instantiation & Query Filtering (`AClean4Actor`)

Once the `USpawnableConsumable` safely verifies the location, it instantiates a child of `ABaseConsumableActor`. These physical actors handle the actual world interactions.

For example, the `AClean4Actor` (a deployable bomb) utilizes Unreal's highly optimized `SphereOverlapActors` sweeps rather than attaching physical collision spheres to every target. By filtering queries through specific `EObjectTypeQuery` channels (e.g., Damage vs. Dust), the actor selectively applies distinct effects to different objects within the exact same radius. Furthermore, by casting overlapped targets to an `IDamageable` interface, the bomb remains entirely agnostic to the classes it is interacting with.

```
1   void AClean4Actor::DamageInArea(TArray<TEnumAsByte<EObjectTypeQuery>> _oChannel,
    float _fDamage) {
2     TArray<AActor*> aActorsToIgnore;
3     TArray<AActor*> aOutActors;
4
5     // Perform an optimized spatial query using specific collision channels
6     UKismetSystemLibrary::SphereOverlapActors(
7     GetWorld(),
8     GetActorLocation(),
9     m_fExplosionRadius,
10    _oChannel,
11    nullptr,
12    aActorsToIgnore,
13    aOutActors
14    );
15
16    // Apply damage purely via interfaces, avoiding hard class dependencies
17    for (AActor* pActor : aOutActors) {
18      if (IDamageable* pEnemy = Cast<IDamageable>(pActor)) {
19        pEnemy->Damage(_fDamage, EDirtType::Neutral);
20      }
21    }
22  }
23
```

Listing 8: Efficient area-of-effect logic using filtered sphere overlaps

3.2 Achievement Subsystem: Steam API & Serialization

Source Files: [AchievementSubsystem.cpp]

In *Dorothy's Job*, the achievement and statistics tracking framework was built as a persistent `UGameInstanceSubsystem`. The primary challenge was ensuring seamless synchronization between local save data and the Steamworks API, while preventing race conditions during asynchronous platform initialization.

3.2.1 Data-Driven Initialization (`RegisterFromINI`)

To avoid hardcoding achievement IDs and maximum progress values within the C++ source code, the subsystem utilizes Unreal Engine's `GConfig` parser.

During initialization, the manager dynamically reads the `DefaultEngine.ini` file, parsing key-value pairs (e.g., `Achievement_X_Id`, `Achievement_X_Max`) to construct its internal map of `FAchievementData` structs. This data-driven approach allows designers to add new progression-based or instant-unlock achievements purely through configuration files.

3.2.2 Asynchronous Synchronization & Queuing

Connecting to the Steam API (`IOnlineSubsystem`) is an asynchronous operation. If a player unlocks an achievement while the API is still initializing, a naive implementation would simply fail to register the unlock.

To solve this, the subsystem maintains an internal queue (`m_lPendingUnlocks`). If `UnlockAchievement` or `AddProgress` is called before the `OnQueryAchievementsComplete` delegate fires, the achievement is pushed to this queue.

```
1   void UAchievementSubsystem::UnlockAchievement(const FString& _rAchievementId) {
2     FAchievementData* pData = FindAchievement(_rAchievementId);
3     if (!pData || pData->bUnlocked) return;
4
5     pData->CurrentValue = pData->MaxValue;
6     pData->bUnlocked = true;
7
8     // Queue the achievement if Steam is not yet ready to receive writes
9     if (!m_bSteamAchievementsReady) {
10      m_lPendingUnlocks.AddUnique(_rAchievementId);
11    }
12    else {
13      WriteAchievementToSteam(*pData);
14    }
15
16    OnAchievementUnlocked.Broadcast(_rAchievementId);
17  }
18
```

Listing 9: Queuing achievements during asynchronous Steam initialization

Once the Steam API returns a success callback, the manager flushes the queue. Furthermore, during this callback, the system performs a bi-directional synchronization check. It compares the local save file's progress against Steam's cloud records. If the local save is ahead (e.g., the player played offline), it aggressively pushes the local data back up to the Steam cloud to force synchronization.

```
1   // Snippet from OnQueryAchievementsComplete
2   float fLocalProgressPercent = ((float)rData.CurrentValue / (float)rData.MaxValue) *
    100.0f;
3
4   // If the local save is ahead of the Steam cloud record, force a sync
5   if (fLocalProgressPercent > fSteamProgress + 0.1f) WriteAchievementToSteam(rData);
6
```

Listing 10: Bi-directional synchronization logic upon API connection

3.3 Modular Settings & Serialization

Source Files: [SettingsManager.cpp] | [VisualSettingsManager.cpp] | [AudioSettingsManager.cpp]

Modern PC titles require extensive, granular configuration options. A common pitfall is creating a monolithic settings class that handles everything from input mapping to volumetric fog. To ensure maintainability, *Dorothy's Job* utilizes a Facade pattern.

The core `USettingsManager` acts as a central coordinator. Upon initialization, it spins up dedicated, domain-specific objects (e.g., `UVisualSettingsManager`, `UAudioSettingsManager`).

3.3.1 Data Serialization & Fallbacks

Configuration data is grouped into specific structs (e.g., `FCustomVisualSettings`) and decorated with the `SaveGame` property specifier. When `LoadSettings()` is called, the manager passes these structs to a generic `USaveGameManager`.

If the player boots the game for the first time and loading fails, the system automatically intercepts the failure and falls back to a `UDefaultSettingsDataAsset`. This allows designers to tweak the default graphical and audio presets globally without touching C++ constructors. Additionally, the initialization phase queries the Operating System's default language and automatically configures the game's `FInternationalization` locale.

3.3.2 Deep Engine & Middleware Integration

The individual managers serve as translation layers between the UI data and the low-level engine APIs.

The `UVisualSettingsManager` wraps Unreal Engine's native `UGameUserSettings`. When the player changes their "Graphics Quality," the manager translates this single enum into granular engine scalability updates (Anti-Aliasing, Global Illumination, Foliage, etc.) and pushes them to the rendering thread.

Simultaneously, the `UAudioSettingsManager` ignores Unreal's native audio mixer. During initialization, it fetches the root routing buses directly from the FMOD Studio System. When a volume slider is adjusted, the manager normalizes the float and pushes it directly into the FMOD middleware.

```
1   void UAudioSettingsManager::Initialize() {
2     FMOD::Studio::System* pStudioSystem = IFMODStudioModule::Get().GetStudioSystem(
    EFMODSystemContext::Runtime);
3
4     // Cache references to the global FMOD buses
5     if (pStudioSystem) {
6       pStudioSystem->getBus("bus:/", &m_pMasterBus);
7       pStudioSystem->getBus("bus:/Music", &m_pMusicBus);
8       pStudioSystem->getBus("bus:/SoundFX", &m_pSFXBus);
9     }
10  }
11
12  void UAudioSettingsManager::SetSFXVolume(float _fNewVolume) {
13    // Normalize custom UI inputs into an FMOD-safe float range
14    float fFinalValue = NormalizeVolumeInput(_fNewVolume);
15
16    m_oAudioSettings.m_fSFXVolume = fFinalValue;
17
18    // Push directly to the middleware bus
19    if (m_pSFXBus) m_pSFXBus->setVolume(fFinalValue);
20  }
21
```

Listing 11: Direct FMOD Bus assignment and Volume Normalization