Blast Course 2: Architecture & Implementation Deep-Dive

Jaime Paramo - Game Programmer

June 7, 2026

Contents

1 Core Gameplay: Rocket Architecture

1.1 Architectural Overview & Design Philosophy

In a high-speed puzzle-platformer centered around rocket jumping, projectiles must be highly performant, predictable, and modular. Relying on default Unreal Engine physics or the `UProjectileMovementComponent` introduces unnecessary overhead when spawning hundreds of projectiles.

To solve this, I bypassed standard physics in favor of a custom, `Tick`-driven mathematical movement system utilizing `SweepSingleByChannel`. This architecture relies heavily on polymorphism: a robust base class (`ABaseRocket`) handles the heavy lifting of movement and collision, allowing child classes to implement unique gameplay behaviors simply by overriding a few virtual functions.

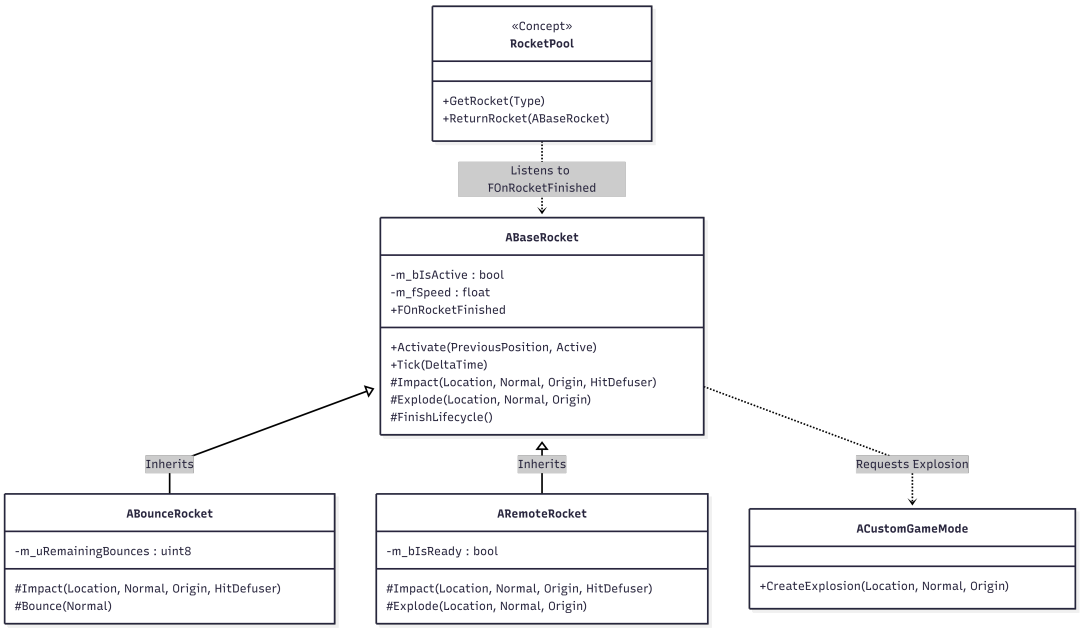

Figure 1: UML Class Diagram of the Rocket Architecture

1.2 The Base Rocket (`ABaseRocket`)

The `ABaseRocket` class acts as the workhorse of the system.

1.2.1 Collision & Defusal Logic

During the `Tick` phase, the rocket calculates its trajectory and performs a spherical sweep. Environmental interactions are handled cleanly by detecting "Defuser" surfaces via `UMaterialInterface`, keeping the system highly component-agnostic.

1.2.2 Lifecycle & Object Pooling Integration

Crucially, the rocket has zero awareness of the Object Pool that manages its memory. Instead, it utilizes an Unreal Multicast Delegate (`FOnRocketFinished`). When a rocket explodes or is defused, it broadcasts its completion and deactivates itself. The Object Pool acts as a listener, reclaiming the object asynchronously.

```
void ABaseRocket::FinishLifecycle() {
  if (!m_bIsActive) return;

  // Deactivate the rocket.
  Activate(FVector::ZeroVector, false);

  // Broadcast to the Object Pool that this actor is ready to be reclaimed.
  OnRocketFinished.Broadcast(this);
}

```

Listing 1: Lifecycle broadcasting in ABaseRocket

1.3 Extending the Architecture

The true strength of this custom base is its extensibility. Creating complex new projectile types requires minimal code duplication.

1.3.1 The Bounce Rocket (`ABounceRocket`)

By overriding the `Impact` function, the Bounce Rocket intercepts the explosion logic. It calculates the reflection vector using the impact normal, rotates the actor, and allows the base class's `Tick` to seamlessly continue moving it in the new direction.

```
void ABounceRocket::Bounce(const FVector& _rNormal) {
  const FVector vCurrentDirection = GetActorForwardVector();
  const FVector vReflectedDirection = vCurrentDirection.MirrorByVector(_rNormal).
GetSafeNormal();

  SetActorRotation(vReflectedDirection.Rotation());
  m_uRemainingBounces--;
}

```

Listing 2: Vector reflection in ABounceRocket

1.3.2 The Remote Rocket (`ARemoteRocket`)

The Remote Rocket transforms the projectile into a sticky trap. It pauses its own lifecycle upon hitting a valid surface, storing the attach location and normal, and waits for an external command to trigger the overridden `Explode` function.

2 Performance & Scaling: Object Pooling

2.1 The "Unconscious" Actor Philosophy

In a game where dozens of rockets and explosions can spawn within seconds, relying on Unreal Engine's native `SpawnActor` and `Destroy` functions would cause massive memory fragmentation and garbage collection spikes, inevitably killing the frame rate.

To solve this, I implemented a robust Object Pooling system within the `ACustomGameMode`. A core architectural pillar of this system is that **the pooled elements have absolutely no consciousness of the pool itself**. The rockets simply exist, execute their internal logic, and broadcast a delegate (`FOnRocketFinished`) when their lifecycle ends. The GameMode acts as a master observer, catching these broadcasts and managing memory silently in the background.

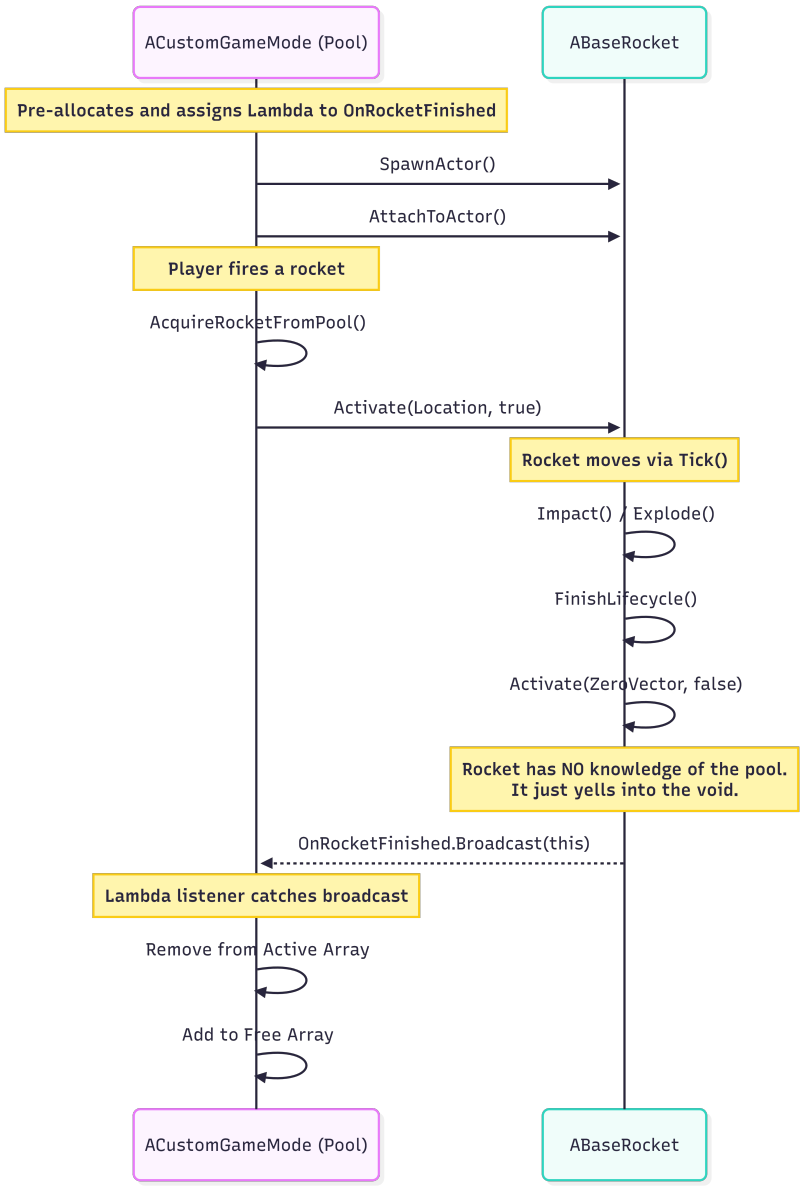

Figure 2: Sequence Diagram demonstrating the decoupled pooling lifecycle

2.2 Dual Pooling Strategies

Not all pooled objects require the same management logic. I implemented two distinct data structures based on the specific needs of the actors.

2.2.1 Explosions: The Circular Buffer

Explosions are transient, visual-heavy actors that do not require complex lifecycle tracking. For these, I utilized a highly efficient Circular Buffer. The GameMode pre-allocates an array of `APlayerExplosion` actors. When an explosion is requested, it grabs the actor at the current index, triggers it, and increments the index using a modulo operation.

```
1   void ACustomGameMode::CreateExplosion(const FVector& _vLocation, const FVector&
    _vNormal, AActor* _pOrigin) {
2     APlayerExplosion* pExplosionActor = m_lExplosionPool[m_uCurrentExplosionIndex].Get
    ();
3
4     pExplosionActor->SetActorLocationAndRotation(_vLocation, _vNormal.Rotation());
5     pExplosionActor->Explode(_pOrigin, _vNormal);
6
7     // Wrap around to ensure it stays within bounds
8     m_uCurrentExplosionIndex = (m_uCurrentExplosionIndex + 1) % m_uExplosionsPerScene;
9   }
10
```

Listing 3: O(1) Circular Buffer allocation for Explosions

2.2.2 Rockets: Active/Free Queues & Aggressive Recycling

Rockets require stricter management, as they have varied lifespans (e.g., a Bounce Rocket lives longer than a standard one). I utilized a struct containing two `TArray` containers: one for *Active* rockets and one for *Free* rockets.

To guarantee that the game never runs out of memory or crashes due to limit exhaustion, I implemented an aggressive recycling failsafe. If a player manages to exhaust the *Free* pool, the system forcefully detonates the oldest *Active* rocket to reclaim its memory.

```
1   ABaseRocket* ACustomGameMode::AcquireRocketFromPool(ERocketType _eType) {
2     FRocketPool& rPool = m_mRocketPools[_eType];
3     TObjectPtr<ABaseRocket> pRocket = nullptr;
4
5     if (!rPool.Free.IsEmpty()) pRocket = rPool.Free.Pop();
6     else if (rPool.Active.Num() > 0) {
7       // Aggressive recycling: force oldest active rocket to explode
8       pRocket = rPool.Active[0];
9       rPool.Active.RemoveAt(0);
10      pRocket->ForceExplosion();
11    }
12
13    return pRocket.Get();
14  }
15
```

Listing 4: Aggressive recycling in AcquireRocketFromPool

3 Presentation & Polish: UMG Extended Architecture

3.1 Overcoming Native UMG Limitations

Unreal Engine's default Motion Graphics UI (UMG) is powerful but can quickly become unwieldy when nesting custom widgets. Standard visibility toggles do not natively broadcast custom state changes down a deeply nested hierarchy. Furthermore, controlling exact pixel dimensions and aspect ratios from C++ while maintaining a WYSIWYG (What You See Is What You Get) experience in the editor is notoriously difficult.

To solve this, I engineered an extended UMG architecture that intercepts layout sizing and recursively manages visibility states, decoupling the UI logic from the standard blueprint constraints.

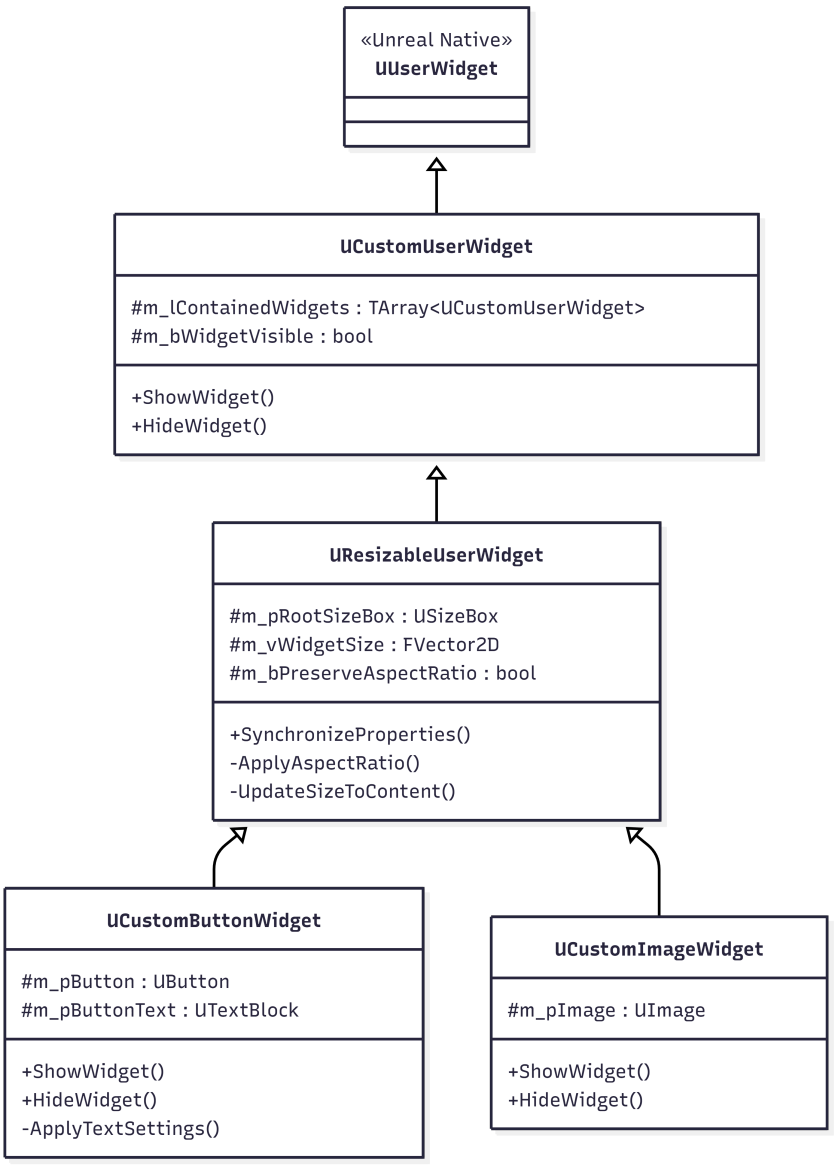

Figure 3: Class hierarchy of the custom UMG framework

3.2 Recursive State Management (`UCustomUserWidget`)

The base class, `UCustomUserWidget`, acts as a central state manager. Instead of relying on Unreal's native `SetVisibility`, I implemented custom `ShowWidget()` and `HideWidget()` functions. These functions iterate through an array of `m_lContainedWidgets`, ensuring that any complex show/hide logic propagates perfectly down the UI tree.

```
1   void UCustomUserWidget::ShowWidget() {
2     m_bWidgetVisible = true;
3
4     // Propagate visibility to all contained child widgets.
5     for (TObjectPtr<UCustomUserWidget> pCustomUserWidget : m_lContainedWidgets) {
6       if (IsValid(pCustomUserWidget)) {
7         pCustomUserWidget->ShowWidget();
8       }
9     }
10  }
11
```

Listing 5: Recursive visibility propagation in UCustomUserWidget

3.3 Editor-Synchronized Layout Control (`UResizableUserWidget`)

To fix the layout control issues, I created `UResizableUserWidget`. This class requires a root `USizeBox` bound via the `meta=(BindWidget)` tag. By overriding `NativeConstruct` (for runtime) and crucially, `SynchronizeProperties` (for the editor), the C++ class forces the UMG designer to respect the exact dimensions set in the details panel.

Additionally, I implemented an aspect-ratio preservation algorithm. If a UI designer scales the width of a widget, the code dynamically calculates the driving axis and scales the height to match, updating the viewport immediately.

```
1   void UResizableUserWidget::SynchronizeProperties() {
2     Super::SynchronizeProperties();
3
4     // Preserve aspect ratio if the flag is enabled.
5     if (m_bPreserveAspectRatio) {
6       if (m_fAspectRatio <= 0.f && m_vWidgetSize.Y != 0.f) {
7         m_fAspectRatio = m_vWidgetSize.X / m_vWidgetSize.Y;
8       }
9       ApplyAspectRatio();
10    } else {
11      m_fAspectRatio = 0.f;
12    }
13
14    SetWidgetSize(m_vWidgetSize);
15    UpdateSizeToContent();
16    m_vLastSize = m_vWidgetSize;
17  }
18
```

Listing 6: Live editor layout updates via SynchronizeProperties

3.4 Encapsulated Leaf Nodes

The system culminates in specialized leaf nodes like `UCustomButtonWidget` and `UCustomImageWidget`. Because they inherit from the resizable base, they automatically gain precise layout controls. They override the inherited `ShowWidget()` and `HideWidget()` functions to map the custom state to the native `ESlateVisibility` of their specific internal components (like `UButton` or `UTextBlock`).

3.5 Future Work: Expanding the Architecture

While the current framework successfully decouples layout control and recursive state management, the UI system is actively being expanded to support the growing needs of the project.

Current architectural goals include:

- **Data Binding (MVVM):** Implementing a formalized Model-View-ViewModel pattern to push data updates to the leaf nodes (like text and images) via delegates, further removing gameplay state from the UI logic.
- **Input-Agnostic Navigation:** Extending `UCustomUserWidget` to natively handle dynamic gamepad and keyboard navigation routing, ensuring menus are fully accessible without relying on Unreal's sometimes rigid default focus systems.