Async development with Unity

an approach to async development by Javier Peña

The Async Mindset

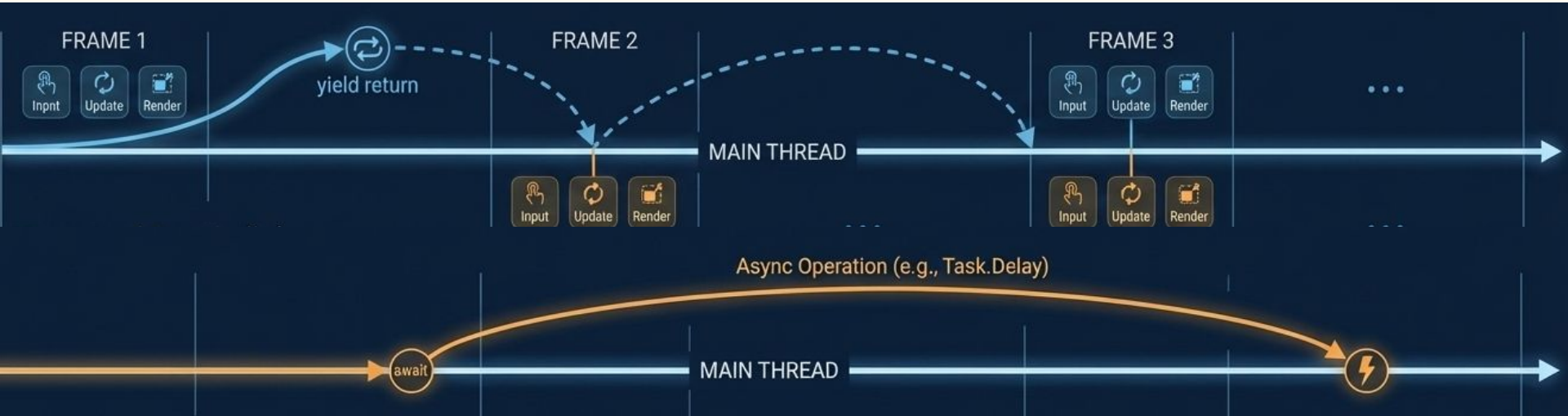
"Waiting vs. Blocking"

- **Synchronous**: The task blocks the main thread.
- **Coroutine**: Checking every chunk of time if a task is done.
- **Async/Await**: Gets notify when the task is completed.

The Async Mindset

"Frame-Based" vs. "Task-Based" Paradigm

- **Coroutines** are sequential interruptions. They say: "Stop here, let the engine render a frame, then come back to this exact line."
- **Tasks** are promises of completion. They say: "Start this operation. I don't care how many frames it takes or what thread it runs on; just notify me when the result is ready."



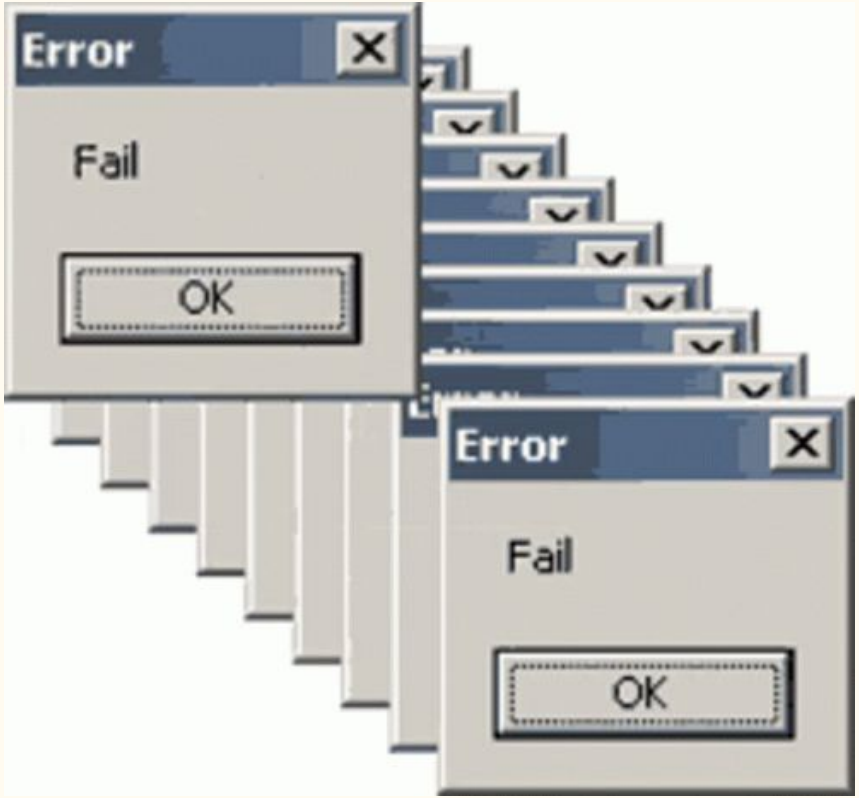
The Async Mindset

"Ghost in the Machine" (The Danger)

- **Coroutines** are tied to a MonoBehaviour

- **Tasks** are tied to the C# Runtime

The Golden Rule: "In Unity, an Async Task without a CancellationToken is a memory leak waiting to happen."



The Syntax

Concept	Coroutine (IEnumerator)	Async (async Task)
Signature	IEnumerator MyMethod()	async Task MyMethod()
Pause	yield return new WaitForSeconds(1f);	await Task.Delay(1000); *(Note: ms)*
Wait for Other	yield return StartCoroutine(Other());	await Other();
Return Value	❌ (Callbacks required)	return myValue;
Error Handling	❌ (Stops silently)	try { ... } catch (Exception e) { ... }

`async void` vs `async Task` : Which one we should use?

CancellationTokens

- **CancellationTokenSource**: The boss. It decides when to cancel.
- **CancellationToken**: The letter. It gets passed down to every worker (Task).
- **destroyCancellationToken**: Monobehaviour default token.

Never write a Task without an "Exit Strategy".

Awaitable

Feature	Standard Task	Unity Awaitable
Return Type	async Task / async Task<T>	async Awaitable / async Awaitable<T>
Namespace	System.Threading.Tasks	UnityEngine
Performance	High GC Allocation ⚠️	Zero Allocation (Pooled) 🚀
Threading	Context needs manual marshalling Task.Run(()=> {...});	Engine-aware (Auto-switching) await Awaitable.BackgroundThreadAsync(); / await Awaitable.MainThreadAsync();

Awaitable vs Task vs void

🟢 **`async Awaitable`**: Optimized, safe, native to Unity (2023+).

- Pros:
  - Zero Allocation: It uses object pooling. No GC spikes.
  - Thread Safety: It captures Unity's context automatically (switching between C++ engine threads and C# script threads).
- Cons: Doesn't have all the fancy utility methods of Task (yet).

🟡 **`async Task`**: Standard, reliable, works everywhere, but slightly heavier (GC allocation).

- Pros: Rich API (Task.WhenAll, Task.WhenAny), standard error handling.
- Cons: Garbage Collection (GC). Every Task you create allocates memory on the heap. In a tight loop (like pathfinding), this causes lag spikes.

🔴 **`async void`**: Dangerous. No return ticket. Use **only** when forced (events).

The Task Orchestration

- **Task.WhenAll**: Runs everything in parallel.
- **Task.WhenAny**: Runs everything in parallel.
- **Task.Delay**: Pauses the Method, frees the Thread.
- **Task.CompletedTask**: Returns a task that is already done.
- **Task.FromResult(T)**: Creates a task wrapped around a pre-calculated value.
- **Task.Yield()**: Forces the method to yield execution back to the caller and schedule the rest for later.

What if we want an extent API like Task but with no allocation like Awaitable?
I present UniTask

MVP

1. The Concept Diagram

- **VIEW (`MonoBehaviour`):** Handles UI references, Input, and **Lifecycle (Cancellation)**
- **PRESENTER (Pure C#):** Orchestrates the Logic. "Show Loading" → "Call Service" → "Update UI".
- **MODEL (Service):** Pure Async Data fetching.