Getting Started

This guide takes you from importing the asset to running the demo and integrating SolidCore Coroutines into your own project.

---

1. Install

After importing the asset, you'll have:

- `Assets/Solid24 Systems/Runtime/Plugins/` — the precompiled `SolidCore.Coroutines`, `SolidCore.Burst`, and `SolidCore.Collections` assemblies. These are auto-referenced, so any script in your project can `using SolidCore.Coroutines;` with no `.asmdef` setup required.

- `Assets/Solid24 Systems/Samples/` — the demo scenes and their scripts.

No further setup is needed. If you're on **IL2CPP**, no special configuration is required — SolidCore uses ordinary managed delegates and is fully IL2CPP-compatible.

> ***Note:*** *The demo scenes use* ***TextMeshPro*** *for the on-screen HUD. If prompted, import the TMP Essentials (Window → TextMeshPro → Import TMP Essential Resources).*

> ***Input (demo scenes only):*** *The samples fire with the* ***legacy Input Manager*** *(`Input.GetKeyDown`), kept deliberately dependency-free. If your project's* ***Active Input Handling*** *(Edit → Project Settings → Player) is set to* ***Input System Package (New)****, switch it to* ***Both*** *(or* ***Input Manager (Old)****) or the Space-to-fire hotkey will throw an* `InvalidOperationException`*. This affects only the demo — the SolidCore runtime has no input dependency.*

> ***Render pipeline (demo scenes only):*** *The sample scenes and materials are authored for the* ***Universal Render Pipeline (URP)****. In a Built-in Render Pipeline project the demo materials will appear* ***magenta*** *— either view the demo in a URP project, or upgrade the sample materials (Edit → Rendering → Materials → Convert…). This affects only the demo — the SolidCore runtime is render-pipeline agnostic.*

---

2. Run the demo

First, run demo setup (one time). *The launcher loads scenes by name, which requires them*

in Build Settings — and Build Settings don't travel inside a `.unitypackage`*. After importing,*

run `Tools ▸ Solid24 Systems ▸ Set Up Demo Scenes` *once. This registers the launcher*

and both demo scenes in Build Settings. (Safe to re-run; idempotent.)

The asset ships three scenes under `Assets/Solid24 Systems/Samples/`:

Scene	What it shows
`Launcher/Launcher.unity`	Start menu — pick which demo to run. **Open this one first.**
`Unity_Coroutines_Demo/Unity_Coroutines_Demo.unity`	The baseline: Unity's `StartCoroutine`, one call per projectile.
`SolidCoreDemo/SolidCore_Coroutines_Demo.unity`	The same volley driven by a SolidCore `CoroutineRunner`.

From the launcher, click a demo to load it. Inside a demo, the ≡ **Menu** button (top-right of the

HUD) or the **Esc** key returns to the launcher.

To compare them:

1. Open a scene and press **Play**.
2. Press **Space** to fire a single volley, or tick **Auto Fire** on the `PlayerActions` component to hold a sustained stream.

3. Watch the HUD: **GC allocated/frame**, **worst frame**, **FPS**, and (SolidCore only) **tick time**.
4. Raise **Projectiles Per Shot** on the `PlayerActions` component to push both systems harder.

You'll see Unity's GC-allocated line and worst-frame time climb with the coroutine count, while SolidCore holds flat at ~0 B/frame.

The knobs on the SolidCore `PlayerActions_SolidCore` component:

- **Projectiles Per Shot** — how many coroutines one shot spawns (in a single batch call).
- **Auto Fire** — fire every frame for a sustained stream.
- **Expected Peak** — the peak concurrent-coroutine count the runner pre-allocates for. Keep this ≥ the most you expect alive at once so volleys never trigger a native resize. This pre-sizing is how SolidCore holds 0 B GC under load.

---

3. Core concepts

SolidCore has four moving parts:

Concept	Type	Role
Runner	`CoroutineRunner`	The front door. Owns the scheduler and native memory. You create one, tick it each frame, and dispose it.

Template	`InstructionSequence`	An immutable, reusable "recipe" for one coroutine's life. Built **once**; spawned any number of times.
Builder	`SequenceBuilder`	The fluent API (`.Do().WaitForSeconds()...`) used to author a template.
Handle	`CoroutineHandle`	A lightweight value-type id for one running instance — use it to `Stop` a specific coroutine.

The golden rule: **build templates at authoring time, spawn instances at runtime.**

All allocation happens when you build; spawning and ticking are allocation-free.

---

4. Your first coroutine

```
using SolidCore.Coroutines;
using UnityEngine;

// Create a runner pre-sized for the peak number of concurrent
coroutines.
using var runner = CoroutineRunner.Create(expectedPeak: 1000);

// Author a template once. This is the ONLY place allocation
happens.
InstructionSequence flash = runner.Sequence()
    .Do(() => Debug.Log("on"))
    .WaitForSeconds(0.25f)
    .Do(() => Debug.Log("off"))
    .Build();

// Start one instance...
CoroutineHandle handle = runner.Start(flash);
```



```
// ...or a whole batch of 500 from the same template — zero per-
instance allocation.
runner.Start(flash, count: 500);

// Drive it. In a game loop this is one Tick per frame.
while (runner.ActiveCount > 0)
    runner.Tick(Time.deltaTime);
```

For tests or batch simulation, `RunToCompletion` or `StartAndRun` drain a wave as fast as the CPU allows (simulated time — a `WaitForSeconds(2)` completes in microseconds):

```
int ticks = runner.StartAndRun(flash, count: 500, deltaTime: 1f /
60f);
```

---

5. Using it in a MonoBehaviour

This is the pattern the demo uses (`PlayerActions_SolidCore.cs`). Create the runner in `Awake`, build templates once, `Tick` in `Update`, and **always** `Dispose` **in** `OnDestroy`.

```
using UnityEngine;
using SolidCore.Coroutines;

public sealed class ProjectileSpawner : MonoBehaviour
{
    [SerializeField, Min(1)] int projectilesPerShot = 500;
    [SerializeField, Min(1)] int expectedPeak = 20000;

    CoroutineRunner     _runner;
    InstructionSequence _projectile;
```

```
    void Awake()
    {
        // Pre-size internal buffers to the peak you expect, so
waves never resize.
        _runner = CoroutineRunner.Create(expectedPeak);

        // Build the template once. Closures capture `this` — no
allocations at runtime.
        _projectile = _runner.Sequence()
            .Do(() => Spawn())                          // muzzle
            .WaitForSeconds(1.5f)                       // travel time
            .Do(() => Despawn())                        // expire /
recycle
            .Build();
    }

    void Update()
    {
        // SolidCore requires an explicit tick — this advances every
coroutine.
        _runner.Tick(Time.deltaTime);

        if (Input.GetKeyDown(KeyCode.Space))
            _runner.Start(_projectile, projectilesPerShot); // whole
volley, one batch call
    }

    void OnDestroy() => _runner.Dispose(); // release native buffers
— required

    void Spawn()   { /* ... */ }
    void Despawn() { /* ... */ }
}
```

Contrast with Unity

The Unity equivalent must loop `StartCoroutine`, and each call allocates a fresh iterator plus a `WaitForSeconds` object — cost that grows linearly with the volley size and feeds the GC:

```
// Unity: one allocation-heavy StartCoroutine per projectile.
for (int i = 0; i < projectilesPerShot; i++)
    StartCoroutine(Projectile());
```

SolidCore replaces the whole loop with `_runner.Start(_projectile, projectilesPerShot)` and adds zero GC pressure regardless of count.

---

6. The instruction set

Author templates with the fluent `SequenceBuilder` returned by `runner.Sequence()`:

Step	Meaning
`.Do(Action)`	Run a managed action immediately, then advance to the next step.
`.WaitForSeconds(float)`	Wait a duration in **simulated** seconds (driven by your `Tick` delta).
`.WaitForFrames(int)`	Wait a number of scheduler ticks.
`.WaitUntil(Func<bool>)`	Block until the predicate returns `true`.
`.Build()`	Bake the steps into an allocator-owned template. Call once; the builder isn't reusable afterward.

`Do` and `WaitUntil` accept **ordinary C# delegates** — including lambdas that capture state. This is what makes SolidCore IL2CPP/Mono-friendly (it never needs unmanaged function pointers).

Spawning from inside a coroutine

To spawn more coroutines from within a running `Do` action, use `CoroutineRuntime.SpawnLocal` — it spawns into the runner/shard currently ticking on this thread, lock-free:

```
.Do(() => CoroutineRuntime.SpawnLocal(childTemplate))
```

---

7. Going multi-threaded (`CoroutineJobSystem`)

For very large populations you can shard coroutines across worker threads. The authoring API is the same (build templates, spawn from them); the difference is that ticks run in parallel.

```
using var jobs = CoroutineJobSystem.Create(expectedPeak: 200_000,
workerCount: 0); // 0 = ProcessorCount - 1

InstructionSequence seq = jobs.Sequence()
    .Do(() => DoWork())
    .WaitForSeconds(1f)
    .Build();

jobs.Schedule(seq, count: 200_000);   // distributed evenly across
shards
```

```
void Update() => jobs.Tick(Time.deltaTime); // advances every shard
in parallel, then joins
```

Notes:

- Call scheduling/tick methods from **one owning thread**. Workers are idle between ticks, so read `ActiveCount` / `GetDiagnostics()` between frames.
- A coroutine lives on one shard for its whole life, so ticks touch no shared memory.
- `FlushSpawns()` front-loads a wave's creation cost in parallel without ticking.
- Use the single-threaded `CoroutineRunner` unless you actually have a population large enough to benefit from threading – it's simpler and has no fork/join overhead.

---

8. Diagnostics

Single-threaded (`CoroutineRunner`):

```
int   alive   = runner.ActiveCount;                  // O(1)
float tickMs  = (float)runner.Scheduler.LastTickTimeMs; // wall-
clock cost of the last Tick
long  started = runner.TotalStarted;
Debug.Log(runner.GetPerformanceSummary());
```


`LastTickTimeMs` is the real per-frame cost of the coroutine system — a number Unity's engine-driven coroutines don't expose. The demo HUD reads exactly these fields via the `ICoroutinePerfSource` interface in `Samples/Shared/Scripts/`.

Multi-threaded (`CoroutineJobSystem`):

```
CoroutineDiagnostics d = jobs.GetDiagnostics(); // call between
frames
Debug.Log(d.Summary());   // active count, throughput/sec, parallel
speedup, peak tick, ...
```

---

9. Common pitfalls

- **Forgetting to `Tick`.** Unlike Unity coroutines, SolidCore does nothing until you call `runner.Tick(dt)` (or `jobs.Tick(dt)`). Nothing advances on its own.
- **Forgetting to `Dispose`.** The runner owns native memory. Call `Dispose()` (or use `using`) in `OnDestroy` to free it.
- **Rebuilding templates at runtime.** Build each `InstructionSequence` once at authoring time and reuse it. Rebuilding per shot defeats the zero-allocation design.
- **Under-sizing `expectedPeak`.** If more coroutines go live than you reserved, the runner resizes its native buffers once — a one-time allocation. Set the peak ≥ your worst-case concurrent count to stay at 0 B GC.

- **Reading job-system stats mid-tick.** `ActiveCount` / `GetDiagnostics()` on `CoroutineJobSystem` are meant to be read between frames, while workers are idle.

Next: the full **API Reference**.