Unity WiseTiming

Custom coroutines that you can control and update yourself with your own deltaTime. Can have multiple timings ticking with different speeds (deltaTimes). Compatible with Unity coroutines. Can also be used without Unity.

Asset Store | GitHub

openupm v1.0.0

What is this for?

Unity coroutines and Time.deltaTime are the main ways to simulate time in the engine. But they are tied to the Unity life-cycle and frame updates and you have no control over it:

- `Time.deltaTime` is updated once every frame by the system. You can't feed your own deltaTime values.
- Coroutines are tied to the MonoBehaviour life-cycle - if disabled, coroutines are stopped.
- Coroutines are executed after `Update()` and before `LateUpdate()`. The order of their execution is undefined, but it is tied to the MonoBehaviour `Update()` order (i.e. based on Script Execution Order). This order can often be misleading as it depends on whether the behaviour was created or enabled this frame, where in the hierarchy the object is, etc.
- Can't have multiple time simulations with different deltaTimes (different time speeds).
- Can't manually tick the coroutines when you want.
- Can't iterate or inspect coroutines - there is no API available to do this.

Example usage: imagine that your game has simple logic that doesn't rely on physics (`FixedUpdate()`). If you want to have a replay functionality, you can just record the input and deltaTime each frame - this will be enough information to reproduce the recorded playthrough. If your code uses Unity coroutines with `Time.deltaTime`, playback implementation would be impossible as you can't feed Unity your recorded deltaTime.

This is where **Wise Timing** comes in. Instead of using Unity coroutines and `Time.deltaTime`, you can use WiseTiming instance. This gives you control over the deltaTime value used by the user code and order of execution. You can also have multiple instances for parallel simulation with different speeds (i.e. each character can have different time speed, slowing down some of them).

Usage

Use **WiseTiming** the same way you use Unity coroutines, just add in the timing instance and source + use `WiseTiming.DeltaTime`. Here is a quick example usage:

```
public WiseTiming Timing { get; } = new WiseTiming();

public void OnAttackClicked(Transform target)
{
  Timing.StartCoroutine(Attack(target), this);  // Pass in source who started the coroutine.
}

private IEnumerator Attack(Transform target)
{
  yield return RotateTowardsTarget(target);
  yield return ChargeAttack();

  yield return new WaitForSeconds(1.5f);  // Suspens...

  // Move towards the target.
  while(Vector3.Distance(transform.position, target.position) > 0.001f) {
    yield return null;  // yield till next frame.

    var step = Speed * WiseTiming.DeltaTime;  // Available ONLY inside WiseTiming update.
    transform.position = Vector3.MoveTowards(transform.position, target.position, step);
  }

  // ...
}
```

When starting a coroutine, a source must be passed. If source is MonoBehaviour, you can specify if you want the coroutines to stop if the source becomes inactive. If source is destroyed the coroutine is stopped. Source is also used for tracking and debugging.

```
public enum SourceInactiveBehaviour
{
    StopCoroutine,
    SkipAndResumeWhenActive,
    KeepExecuting,
}
```

In order for the coroutines to run, you'll need to call WiseTiming update regularly. Example:

```
void Update()
{
  Timing.UpdateCoroutines(Time.deltaTime * TimeSpeedMultiplier);
}
```

IMPORTANT: The static `WiseTiming.DeltaTime` is available only during `Timing.UpdateCoroutines()` call - accessing it outside such call will result in exception. You can have multiple timing instances so it is unclear which deltaTime it should use. If you just want to take the current or last delta time, you can do this from the instance itself: `timing.LastOrCurrentDeltaTime`.

Most used Unity yield instructions are supported: `WaitForSeconds`, `WaitForEndOfFrame`, `CustomYieldInstruction`, `UnityWebRequest`, `AsyncOperation`, `Task` etc. For up-to-date list of supported yield instructions, check the code. `WaitForFixedUpdate` is not supported because it doesn't make sense.

If you want to have classic `Update()` method called by WiseTiming, subscribe to the `WiseTiming.PreUpdate` or `WiseTiming.PostUpdate` events. When called you can access the `WiseTiming.DeltaTime`.

Debugging and Exception Handling

There are several ways to debug what coroutines are running at the moment in what state:

- `WiseTiming.Coroutines` gives you access to the active coroutines.
- `WiseCoroutine` has `DebugInfo` field that gives you stats like when the coroutine was created, etc.
- Setting the `WiseTiming.DebugInfo_RecordCallstack` flag to true will allow the coroutine to collect the initial callstack while being created. It can be inspected at `WiseCoroutine.DebugInfo.StackTrace`. Note that collecting the callstack is a slow operation and should be avoided in release.
- `WiseCoroutine.GetDebugStackNames()` gives you the enumerator stack (their type names). This can help you track nested coroutines.

`WiseCoroutine.ExceptionHandling` gives you control over what should happen on exception while updating the coroutine.

Coroutines vs Async/Await Tasks

Altough coroutines and tasks are similar, they have some key differences that may affect your code workflow. Here is what you need to know before deciding what to use:

- Both run on the main thread in Unity.
- Both can wait on another call to finish before resuming (`yield` or `await`).
- Coroutines are tied to the MonoBehaviour life-time. If behaviour is disabled or destroyed, the coroutine stops. Tasks on the other hand will continue, no matter who started or awaited them - you must manually stop them on switching scenes or destroying objects. This includes stopping play mode in the editor (tasks can leak into edit mode, if assembly doesn't reload).
- Tasks can return results. WiseTiming offers DIY workaround - use the `WiseCoroutine.ResultData` property to quickly get/set any result data (example: `WiseTiming.CurrentCoroutine.ResultData = "my results"`).
- Tasks support proper exception handling - try/catch, proper callstack, proper propagation. Coroutine exceptions are just for the current enumerator - if you have nested coroutines yielding one another, the callstack information will be missing. With WiseTiming you can inspect the yield stack and still figure out what happened. You can even record the initial `StartCoroutine()` callstack. WiseTiming also allows you to listen for exceptions and decide on how to proceed.
    - When gameplay logic causes exception it usually breaks the rest of the game. It is rare for games to recover from such events, so missing proper try/catch is not a big deal.
- Tasks code is a bit harder to write (as it has more features).

Our *wise* advice: for gameplay logic use coroutines - they are tied to the scene. For system operations (like loading files, network requests, etc) use tasks.