Async/Await & Unity Awaitable

Ali Emre Onur · Follow
8 min read · Sep 25

It has been a long time since my last article — so, I hope all of you are doing well. This article will focus on the basics of async/await, together with a brief comparison with using Coroutines and will end with a brief introduction to Unity Awaitables.

Having a non-CS background, I only had limited idea regarding Async, and you know — coroutines were just doing the job one way or another. I am pretty surprised with the ease of Async/await and I will try to explain them as lean as possible.

Being Asynchronous

Before going with the negative one, what do we mean with being synchronous? Unity runs on the main thread and runs sequentially from top to down. In order to main thread to continue executing, the started task has to be completed.

However, some in some cases, there is a need for the main thread to continue executing while also awaiting for a completion of a task — such as collecting some data from internet (imagine the smell if we were awaiting at that task). In order to make sure that the main thread is not blocked, these operations are being handled on the background, in order to allow the flow of the main thread with no blockage. You might be already using asynchronous methods if you are using Coroutines, Addressables, networking or so on.

Coroutines vs Async

So now, you might be wondering, do I really need Async if I am already using Coroutines? Yes my friend, trust me, you do.

Here's a brief comparison of Coroutines and Async methods:

Coroutine	Async
Doesn't have return values	Has return values
Can't run synchronously	Can run synchronously
Doesn't work without the engine	Works without the engine
Doesn't support try/catch	Supports try/catch
Doesn't preserve call stack	Preserves call stack
Always shows exceptions	Can hide exceptions
Doesn't always exit	Always exits
Lifetime tied to a MonoBehaviour	Lifetime handled manually
Familiar to most Unity developers	Unfamiliar to many Unity developers

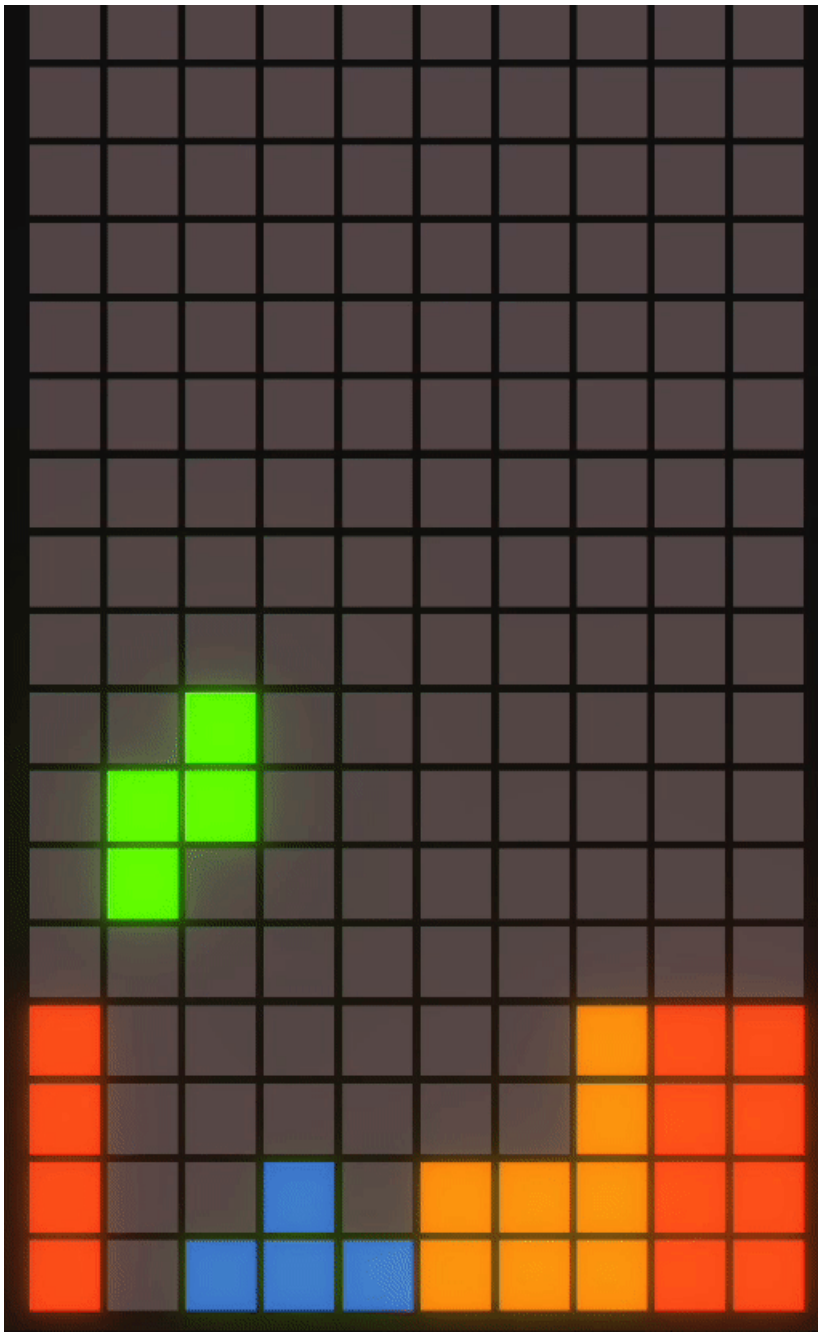
Coroutine/Async comparison — taken from the Unite Kopenhagen 2019 video

> *The chart above is taken from Unity's Unite Kopenhagen 2019 conference which can be watched here. I highly suggest you to watch the whole video in order to gather a more in-depth understanding in this subject.*

I will try to demonstrate some of the differences with an example on the rest of the article:

Tetris Game

I have created a basic Tetris game, in which the matched raws will blink twice before getting destroyed.



Sorry if you are reading this on a bright screen :)

Here's the basic code that will implement the blinking behaviour:

```
public class PieceFlasher : MonoBehaviour
{
    private Material _material;
    private Color _currentColor;
    private WaitForSeconds _blinkTime = new WaitForSeconds(0.25f);

    private void Awake()
    {
        _material = GetComponent<SpriteRenderer>().material;
        if (_material == null)
            Debug.Log("The piece could not gather its material");
        _currentColor = _material.color;
    }

    public void DestructionStarted()
    {
        StartCoroutine(DestructionRoutine());
    }

    IEnumerator DestructionRoutine()
    {
        for(int i=0; i<2; i++)
        {
            yield return _blinkTime;
            _material.color = Color.white;
            yield return _blinkTime;
            _material.color = _currentColor;
        }
        Destroy(this.gameObject);
    }
}
```

Coroutine Behaviour

In the game, I have a Monobehaviour Board class that holds the data regarding if a single cell is is filled and the piece that is currently filling that cell. Each of the cell is denoted with a Cell class that holds the data if they are filled:

```
public void CellStatusChanged(bool isFilled, Piece pieceToFill = null)
{
    _isFilled = isFilled;
    if (isFilled)
        _currentPiece = pieceToFill;
}
```

For managing the matching and related move&destroy operations, I have created a “MatchHandler” class (non-mono), which is being instanced from the Board. In order to gather a proper movement in a case that a raw is completely filled, each of the pieces that are going to move needs to know their updated target positions before moving, so that the we will not trigger wrong piece objects while executing if we are to move many objects at the same time.

Given these facts, I need to find a way to know if all of the pieces have finished their asynchronous **flashing behaviour** (or at least, the time once the last of the pieces has finished blinking). Of course there are multiple solutions to this problem, including the usage of coroutines, or longer solutions with some booleans, or maybe Action callbacks and so on. So here's the solution I would have come up with, referencing a MonoBehaviour class from a non-monobehaviour class to make usage of the Coroutines. The non-mono "MatchHandler" class could be as below:

```
AssignNewTargetToPieces(int declinedRawAmount);
ReferencedMonoObject.StartCoroutine(FlashingRoutine(1f));
```

```
FlashingRoutine(int waitTime)
{ foreach(var piece in _piecesList)
  {piece.FlashBehaviour();} //each piece will flash and then destroy
  yield return new WaitForSeconds(waitTime);
  MoveDownThePieces();
}
```

As the Coroutine relies on the Monobehavior object, the result above is tightening the coupling (the dependency) between these two classes (Board and MatchHandler in this case). I am also calling some methods of both classes from each other separately — ideally, the only connection we need between these two classes is the call from the Board class, when its time to check for matches. Additionally, I had to provide a wait time manually (I have to change the caller method's time if I decide to change the flashing intervals of the pieces), as the coroutine cannot provide a built-in structure to update me regarding the end of the execution. So you get the point, this is far away from being efficient right now.

An important point to note is that the Coroutine will only continue its operations if the relevant Monobehavior object is still active. As Unity suggest, the destruction of the Monobehaviour object while a Coroutine is running will lead to memory leakage.

Switching to Async

In order to use async methods, you need to add the "**System.Threading.Tasks**" namespace. Rather than the IEnumerator used in the coroutines, we declare an async method using the async keyword, followed by Task<T>, denoting the return type:

```
async Task AnAsyncMethod() {}
```

> *We could declare a non-value returning async method using the void attribute rather than the Task. However, Jerome Laban suggests that we might encounter some errors in some cases if we are to use void rather than Task while defining non-value returning methods. It's better get used to use Tasks as you will need them within the body or for declaring return type async methods. Lastly, in order to cancel the task once you pause or exit the game, you will also need the Task keyword.*

A task simply denotes an async operation. Usually, some cooking examples are given to explain the tasksTasks are generic types, and can have return values by using Task<T>. Thus, **async methods can have**

return values, whereas the coroutines cannot .— You will need to use some callbacks to gather some returning information from the coroutines.

Now, back to the example. In order to create the delay, rather than yield return WaitForSeconds in the coroutines, **await Task.Delay(milliseconds)** is being used. As the async methods do not rely on Monobehavior, I can also change the PieceFlasher into a non-mono class which will be instanced from the piece game object. The code is updated as below:

```
public class PieceFlasher : IFlasher
{
    private Material _material;
    private Color _currentColor;

    public PieceFlasher(Material material)
    {
        _material = material;
        _currentColor = _material.color;
    }

    public async Task StartFlashing()
    {
        for (int i = 0; i < 2; i++)
        {
            await Task.Delay(250);
            _material.color = Color.white;
            await Task.Delay(250);
            _material.color = _currentColor;
        }
    }

}
```

Now, the caller method can know when this task is completed so I can call the Destroy method from the caller method:

```
public async Task MatchedInARaw()
{
    await _pieceFlasher.StartFlashing();
    Destroy(this.gameObject);
}
```

Here, the compiler will wait for the end of the StartFlashing task, and then Destroy the object. Each of the pieces in a matched raw will be called from a foreach loop. Given the fact that the I shaped object has a 4 pieces of height, the code may execute *4* (board.Width*) number of pieces one by one. Certainly, the exact time of completion will be different on the first and the last ones —and the pieces on the upper raws shall only move after the pieces in the matched raw have been destroyed.. As I have stated before on the coroutine example, by assigning the Tasks into an array and checking once they are all completed, I can achieve what I need without any unnecessary effort. Thus, another win for the async methods is that we can know once every task has finished executing.

```
private async Task RawDown(int rawId)
{
    var tasks = new Task[_board.Width];

    for(int i=0; i<_board.Width; i++)
    {
        tasks[i] = _board.allBgTiles[i, rawId].HadAMatch();
    }

    await Task.WhenAll(tasks);
    AssignPiecesToMove();
    MoveDownPieces();
}
```

You might be wondering, how are we going to skip a frame with the async methods? Well, a point goes to the Coroutines side. Despite the fact that you can just use Task.Yield() — just like the yield return null — it is not strictly equivalent to exactly 1 frame.

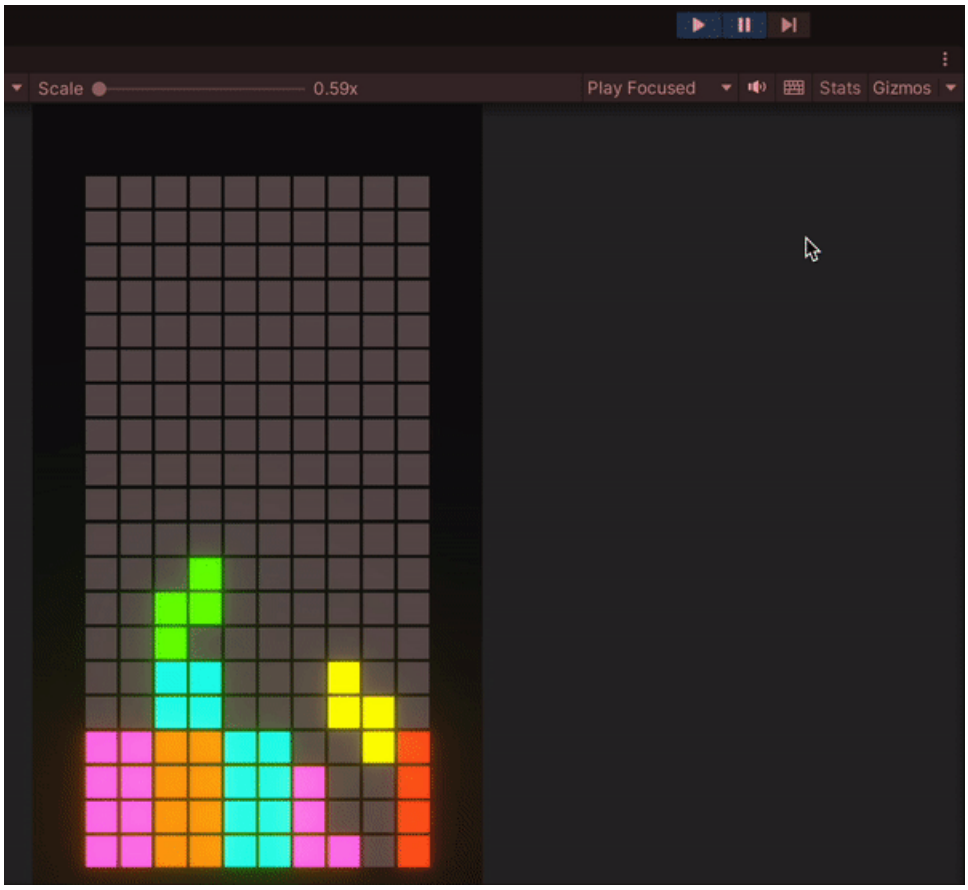
Main Problem With The Asnyc/Await

So why is the async/await system is not frame dependent? Well, there's a reason why these tasks do not rely on Monobehaviour. The Tasks are controlled by the .NET runtime, rather than Unity, making it frame independent. This may cause you some problems if you pause the game or even exit the game in the middle of a task. Even though the runtime will be paused/exited on Unity, .Net will continue operating. Being independent from the Unity Engine also results in poor Profiler readings in the case of Tasks usage.

As the flashing behaviour in the Tetris game is an async Task, I can test it easily. I have just changed the code that make easier to notice the problem.

```
//await Task.WhenAll(tasks);
await Task.Delay(1000);
```

I paused the game from the Editor once the pieces started to flash on a matched raw. Once I clicked continue, the board data was messed up. As I had the Piece Monobehaviour object to update the values of board cells, the board data on the lower two raws worked as they were empty — resulted in a total mess.



The case in which I wait for the tasks completion also cause similar problems in a lower extend. Given a case in which a task is running for a long period of time, or even in a conditional loop, the task will continue to run even after we exit the game. The solution in these cases is to cancel the tasks whenever we pause/exit by usage of the CancellationToken. For now, I will be skipping the details regarding the cancellation token.

Unity Awaitable

With Unity 2023, Unity has introduced Awaitable system, which is really easy to use while overcoming the async's frame independency. *In order to make use of the Awaitable, you need to install a 2023 version of Unity.*

You can check out the documentation following the link. Long story short, Unity successfully integrated Awaitable as a new system that covers the main negative aspects of the Task system. In my example, the flashing behaviour of the pieces were not dependent on the Unity runtime, making it vulnerable to errors. Thus, making the small change transitions the delay into Unity runtime dependent once again.

```
public async Task StartFlashing()
{
    for (int i = 0; i < 2; i++)
    {
        await Awaitable.WaitForSecondsAsync(0.25f);
        _material.color = Color.white;
        await Awaitable.WaitForSecondsAsync(0.25f);
        _material.color = _currentColor;
    }
}
```

Rather than Task.Delay in milliseconds, Awaitable allows us to delay the method using the game time.

As stated, Task.Yield() is not the equivalent of a frame skipping delay. For skipping a frame. And Awaitable is here with a solution:

```
await Awaitable.EndOfFrameAsync();
await Awaitable.FixedUpdateAsync(); //For Fixed Update
```

To cancel an Awaitable *awaitable.Cancel()* method exists. However, in order to use it, the method is also has to be defined as “Awaitable” instead of “Task”. For the purpose of following if the Awaitable is completed, *IsCompleted* has been provided as a property. Still, I decided to stick with the current await system along with the Awaitable delay time in the example above. Lastly, Unity Awaitable allows you to manually switch in between threads.

If Awaitable is still insufficient for you, you can go with the UniTask. Given these facts, I believe I will be hardly using Coroutines anymore. Please feel free to comment or contact me if you believe some of the information is misleading.

Unity Unity3d