

Await/Async in Unity

- [Async/Await in C# Unity](#)
- [Converting A Coroutine To An Async](#)

Async/Await in C# Unity

Preface

The original content is from the author Santosh Parihar from [Medium.com](https://medium.com). It has been modified and updated for use in this wiki.

This is an **advanced** topic! You should have an understanding of Object Oriented Programming in C# as well as an understanding behind C# Delegates.

Getting Started

Lets understand Synchronous Operations first:

In software development, synchronous operations refer to tasks that are executed in a sequential, blocking manner. This means that one task is completed before the next one begins, and each operation waits for the previous one to finish. In synchronous execution, the program's flow is linear, and tasks are processed one after the other. It's like standing in a queue: each person must wait for the one in front to finish before proceeding.

Unity's Main Thread :

Unity operates primarily on a single thread known as the "main thread" or "game loop". This main thread is responsible for several critical tasks, including:

- Input Handling, Game Logic Update, Physics Simulation, Rendering and Repeat.
- The loop continues these steps continuously, maintaining the real-time nature of the game.

Unity often requires asynchronous operations due to the following key reasons:

1. Preventing Main Thread Blocking:

- In synchronous operations, if a task takes a long time to complete (like loading a large file or making a network request), it can cause the main thread to become unresponsive. This results in the game freezing or stuttering, leading to a poor user experience.
- Asynchronous operations allow you to perform these time-consuming tasks in the background, ensuring that the main thread remains free to handle other critical tasks like

rendering, input processing, and game logic.

2. Handling Concurrent Tasks:

- Games often need to perform multiple tasks simultaneously, like loading assets, processing AI behavior, and handling network communication. Asynchronous operations enable you to manage these tasks concurrently without one task blocking the others.

3. Network Communication:

- When making network requests, especially over the internet, the response time can vary. Asynchronous operations allow the game to continue functioning while waiting for data from the network, preventing the game from feeling sluggish.

4. Error Handling:

- Asynchronous operations often offer better error handling. If something goes wrong during a synchronous operation, it can disrupt the entire process and may not be easy to recover from. With asynchronous operations, errors can be caught and handled more gracefully, allowing the game to continue running without catastrophic failures.

```
async Task HandleAsyncOperation()
{
    try
    {
        // Asynchronous operation
        await SomeAsyncTask();
    }
    catch (Exception e)
    {
        Debug.Log($"error : {e.Message}");
    }
}
```

Here's a simplified explanation and different examples of async/await for beginners :

To use async/await, you first need to declare your method as `async`. This tells the compiler that the method can return an `async Task` object. You can then use the `await` keyword to wait for an asynchronous operation to complete.

```

using System.Threading.Tasks;
using UnityEngine;

public class AsyncExample : MonoBehaviour
{
    async void Start()
    {
        Debug.Log("Start of the method");

        await SomeAsyncOperation();

        Debug.Log("After the async operation");
    }

    async Task SomeAsyncOperation()
    {
        await Task.Delay(2000); // Simulating a time-consuming operation
    }
}

```

Example 1

The following code loads an asset, creates a new object with the asset, and then moves the object to a specific location:

```

void Start()
{
    SpawnPlayer();
}

async void SpawnPlayer()
{
    GameObject player = await AssetDatabase.LoadAssetAsync("Assets/Player.asset");
    GameObject playerBody = new GameObject(playerBody);
    await playerBody.transform.position = new Vector3(1, 5, 4);
}

```

Example 2

Imagine you're a magician performing a series of magic tricks. Each trick requires some setup time, and you want to perform them one after another without making the audience wait too long between tricks.

Using `async/await`, you can perform multiple tricks concurrently and ensure a smooth flow of your performance. Let's see how it works:

```
using System;
using UnityEngine;
using System.Threading.Tasks;

public class Magician : MonoBehaviour
{
    public async Task PerformMagicShow()
    {
        Debug.Log("Preparing for the magic show...");

        Task trick1Task = PerformTrick("Card Trick");
        Task trick2Task = PerformTrick("Coin Trick");

        // While the tricks are being prepared, you can engage the audience
        Debug.Log("Engaging the audience with some jokes...");

        // Await the completion of each trick
        await trick1Task;
        Debug.Log("Card Trick performed!");

        await trick2Task;
        Debug.Log("Coin Trick performed!");

        Debug.Log("Magic show completed!");
    }

    private async Task PerformTrick(string trickName)
    {
        Debug.Log($"Preparing for {trickName}...");

        // Simulate the time it takes to set up the trick
        await Task.Delay(3000); // 3 seconds

        Debug.Log($"Performing {trickName}!");
    }
}
```

```

    }

    public async Task StartMagic( )
    {
        await PerformMagicShow();
    }

    public void Start()
    {
        StartMagic();
    }
}

```

In this example, the `Magician` class represents a magician performing a magic show. Each trick is represented by the `PerformTrick` method, which is called with the name of the trick.

Inside the `PerformMagicShow` method, two tricks (`Card Trick` and `Coin Trick`) are initiated concurrently using `PerformTrick` and stored in separate `Task` objects (`trick1Task` and `trick2Task`).

While the tricks are being prepared, you engage the audience with jokes, giving them an entertaining experience during the setup phase.

By assigning `PerformTrick("Card Trick")` to the `trick1Task` variable, you are creating a `Task` object that represents the asynchronous operation of performing card trick. However, at this point, the `PerformTrick("Card Trick")` method will start executing, and the program will proceed to the next line without waiting for the 3-second delay to complete.

The execution will continue to the line `Task trick2Task = PerformTrick("Coin Trick");`, which starts the `PerformTrick("Coin Trick")` method. Similarly, this method contains an asynchronous operation simulated by `Task.Delay(3000)`. Here again, the program will start executing this method, but it won't wait for the 3-second delay to finish.

The line `await trick1Task;` will pause the execution of the `PerformMagicShow()` method and wait until the `trick1Task` is completed before moving on to the next line of code.

When you use the `await` keyword, it essentially tells the program to pause at that point and allow other tasks to continue executing while it waits for the awaited task to complete. In this case, it will wait for the `trick1Task` to finish.

Once the `trick1Task` is completed, the program will resume execution at the next line after the `await` statement. In this code example, it will print "Card Trick performed!" to the console.

So, the `await` keyword ensures that the subsequent code is not executed until the awaited task (in this case, `trick1Task`) has finished its execution.

Then, using `await`, the program waits for each trick to complete before moving on to the next line. Once a trick is completed, a corresponding message is printed to the console.

Finally, when all the tricks are finished, the program prints a completion message, indicating that the magic show is complete.

This playful example demonstrates how `async/await` can be used to perform multiple tasks concurrently, allowing you to keep the audience engaged while seamlessly transitioning between different tricks during a magic show.

Unity is single threaded. Coroutines are useful for executing methods over a number of frames.

Async methods are useful for executing methods after a given task has finished.

Unity's main thread is single-threaded, meaning that all Unity-related operations, such as rendering and updating the game, are handled on that thread. However, `async/await` can still be beneficial in Unity for handling certain types of tasks, even if they don't run on separate threads.

The key idea is that while the `async` operations themselves may not run on separate threads, they still allow the Unity main thread to continue processing other tasks and remain responsive.

In summary, while Unity's main thread is single-threaded, `async/await` can still be utilized in Unity to handle `async` operations in a non-blocking manner, allowing the main thread to continue processing other tasks while waiting for the completion of the `async` operations.

When utilizing coroutines, a feature in Unity, they will automatically stop running as soon as you stop the Unity editor or the game ends. However, when using threads, they can continue running in the background even after stopping the Unity editor. To ensure proper handling, it becomes necessary to manually stop the thread when stopping the Unity editor to prevent any unintended background execution.

To solve the problem we will use Task Cancellation feature of `c#`, you can read more about it here :

<https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-cancellation>

```
public class TestAsync : MonoBehaviour
{

    private CancellationTokenSource token;

    private void OnEnable()
```

```

{
    token = new CancellationTokenSource();
    Debug.Log(" task , on enable");

    // Start the work in a separate task
    Task.Run(DoWork);
}

private void OnDisable()
{
    Debug.Log(" task , on disable , task cancel");

    // Cancel the task when the script is disabled
    token.Cancel();
}

private async void DoWork()
{
    Debug.Log(" task , do work");
    await Task.Run(() =>
    {
        for (int i = 0; i < 5; i++)
        {
            Debug.Log("Hello");
            if (token.IsCancellationRequested)
            {
                return;
            }
        }
    }, token.Token);

    if (token.IsCancellationRequested)
    {
        Debug.Log("task was cancelled.");
        return; // further code will not be executed
    }

    // Code to be executed after the task completes successfully
    Debug.Log("task completed successfully.");
}

```

```
}
```

Hope you understood the concept.

Converting A Coroutine To An Async

Preface

The original content is from the author Michael Quinn from [Medium.com](https://medium.com). It has been modified and updated for use in this wiki.

This is an **advanced** topic! You should have an understanding of Object Oriented Programming in C# as well as an understanding behind C# Delegates.

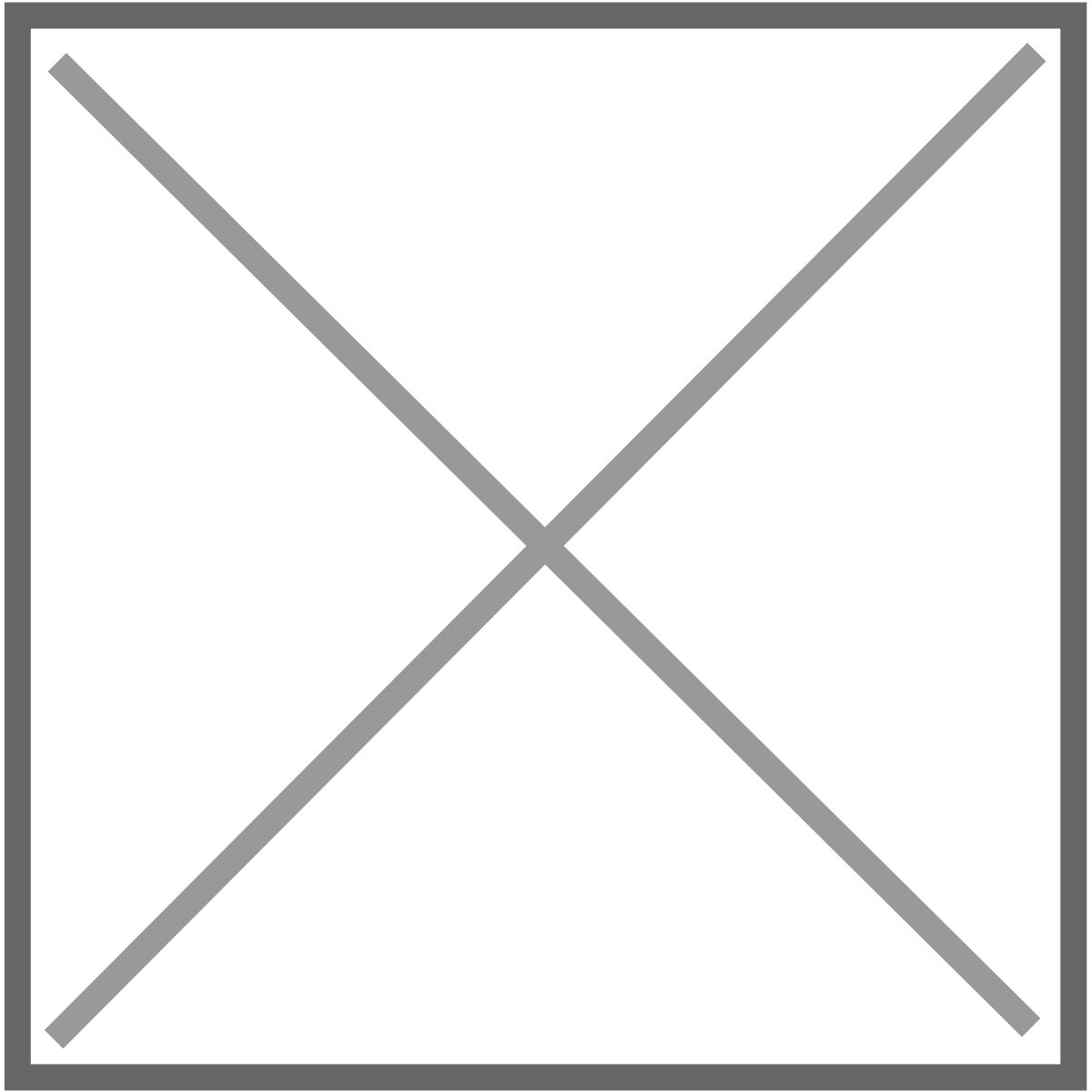
I could have translated the code below into code snippets, but I found it more important to physically type what's needed for the sake of understanding.

About

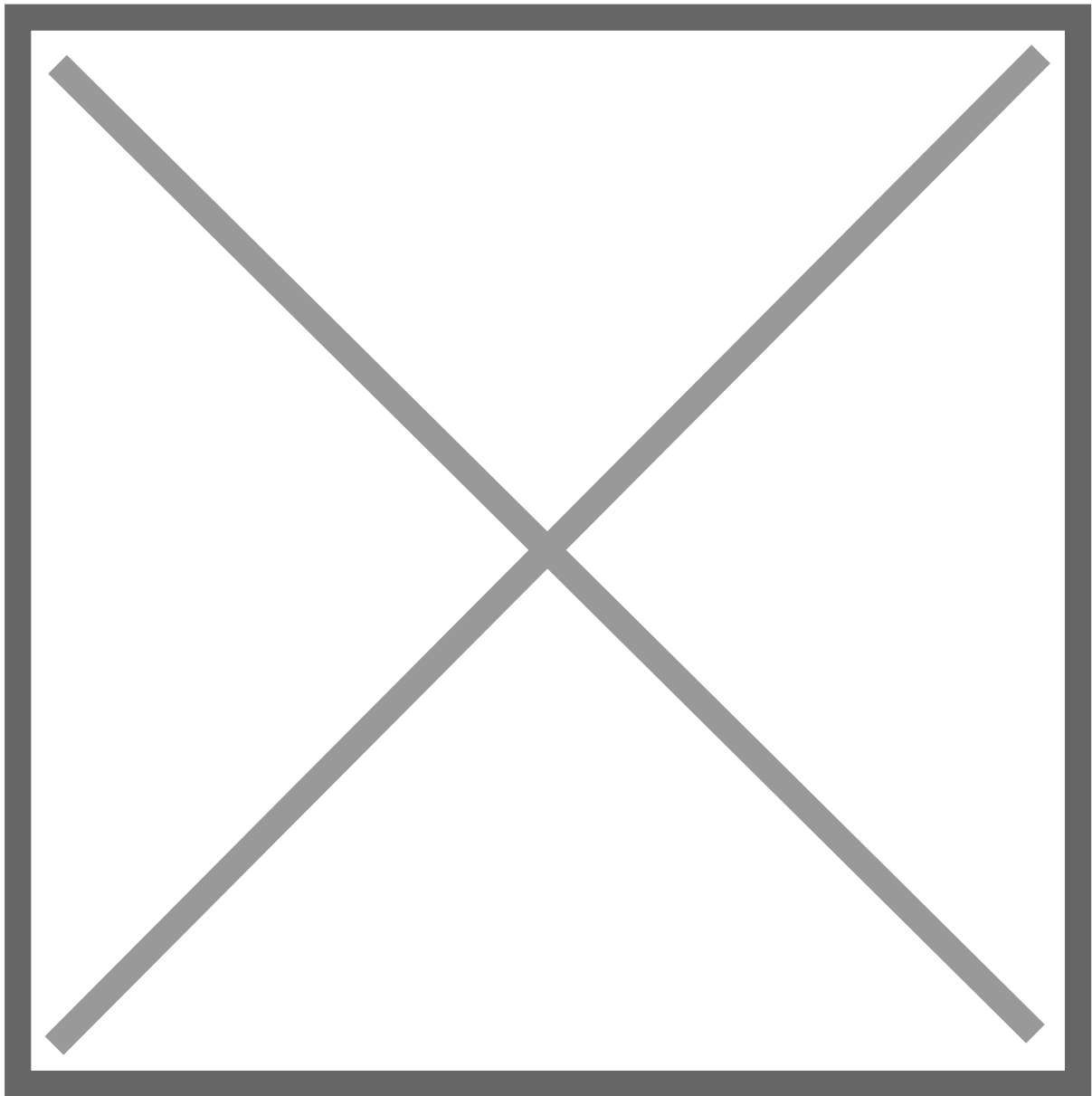
Coroutines are amazing ways to sequence logic together over many frames. Asynchronous programming, or **async**, allows the code to similarly be split over multiple frames, but allows for multithreading which has the benefit of logic being concurrently executed instead of sequentially.

The Coroutine Way

I have a very simple example that rotates cubes for a certain amount of time.

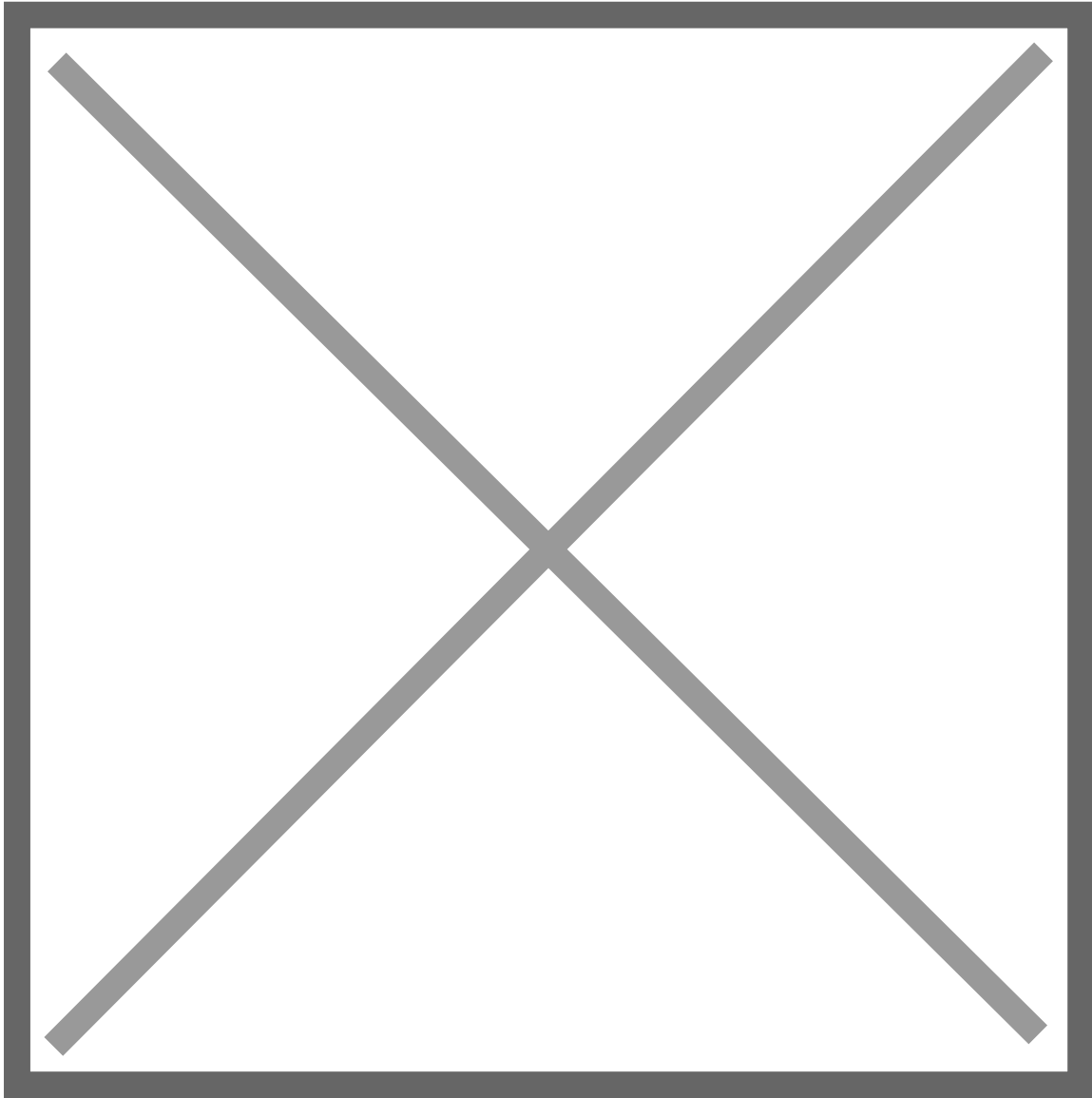


This script will do a loop over all the cubes and trigger a coroutine that will rotate them. Setting this function to run when a button is pressed and we have the following effect.

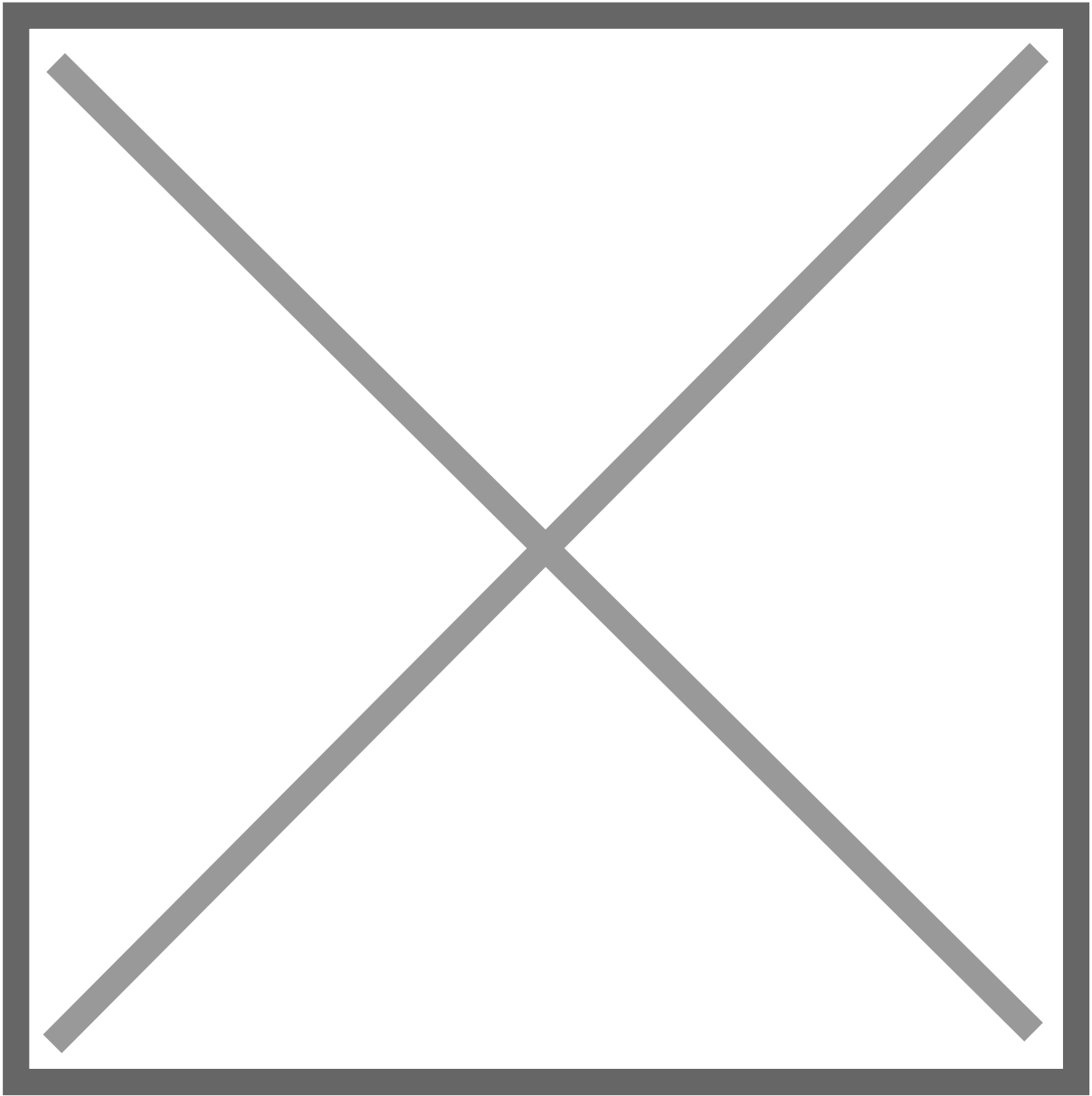


The Async Way

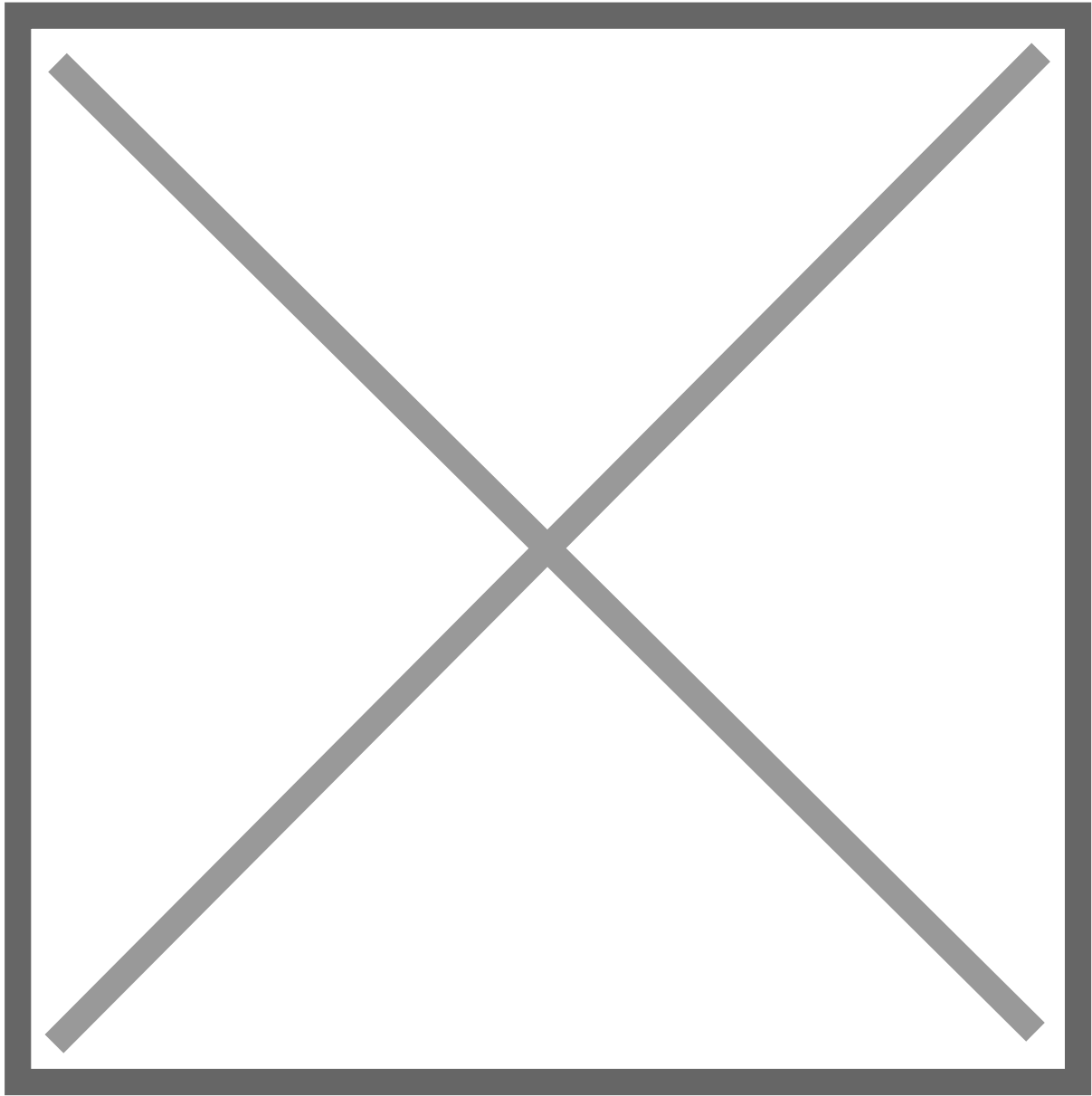
To convert the current method we are going to swap **IEnumerator** to **async** and then change the *yield* to an *await task*.



With the proper using statement, we can simplify the naming and finish with the following code.

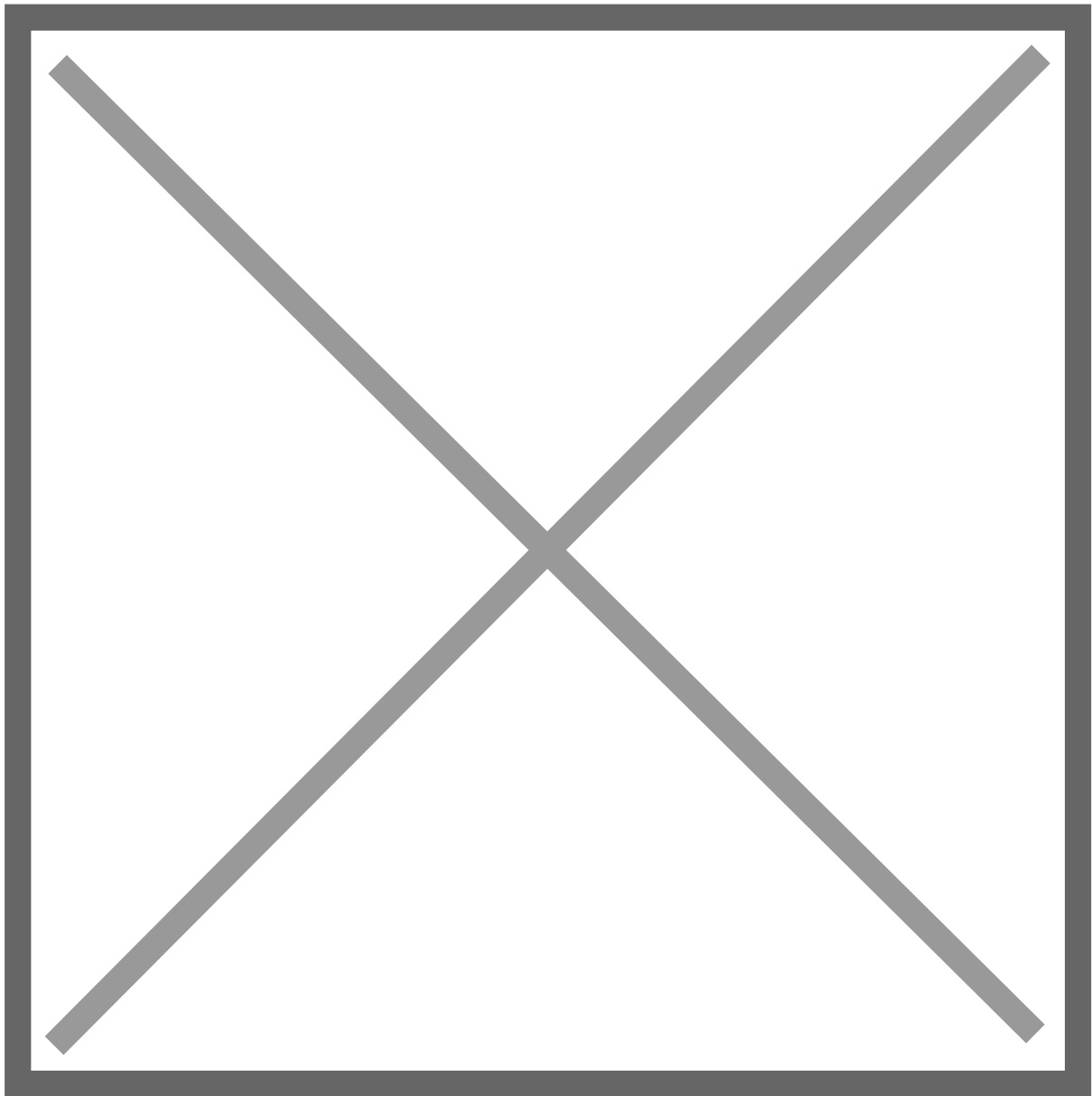


Get ready for the results! They are ... exactly the same.



Okay so where is the power in this? The benefits are immense. The very first we may be able to see was that I had to add a return type when converting the function from a coroutine to an async method. Coroutines don't return values like that.

To get around this shortcoming, I've had to do crazy things like this before.



Ugh, yuck. Who wants to read that? With async, everything is far more readable and easier when it comes to return values.

The strongest immediate benefit of async is the power of multithreading. Coroutines run on what is called the **main thread**. The longer it takes the main thread to move, the more lag your application will experience. Async, however, can run on threads besides the main thread. This benefit from async can give a major benefit to an applications performance.

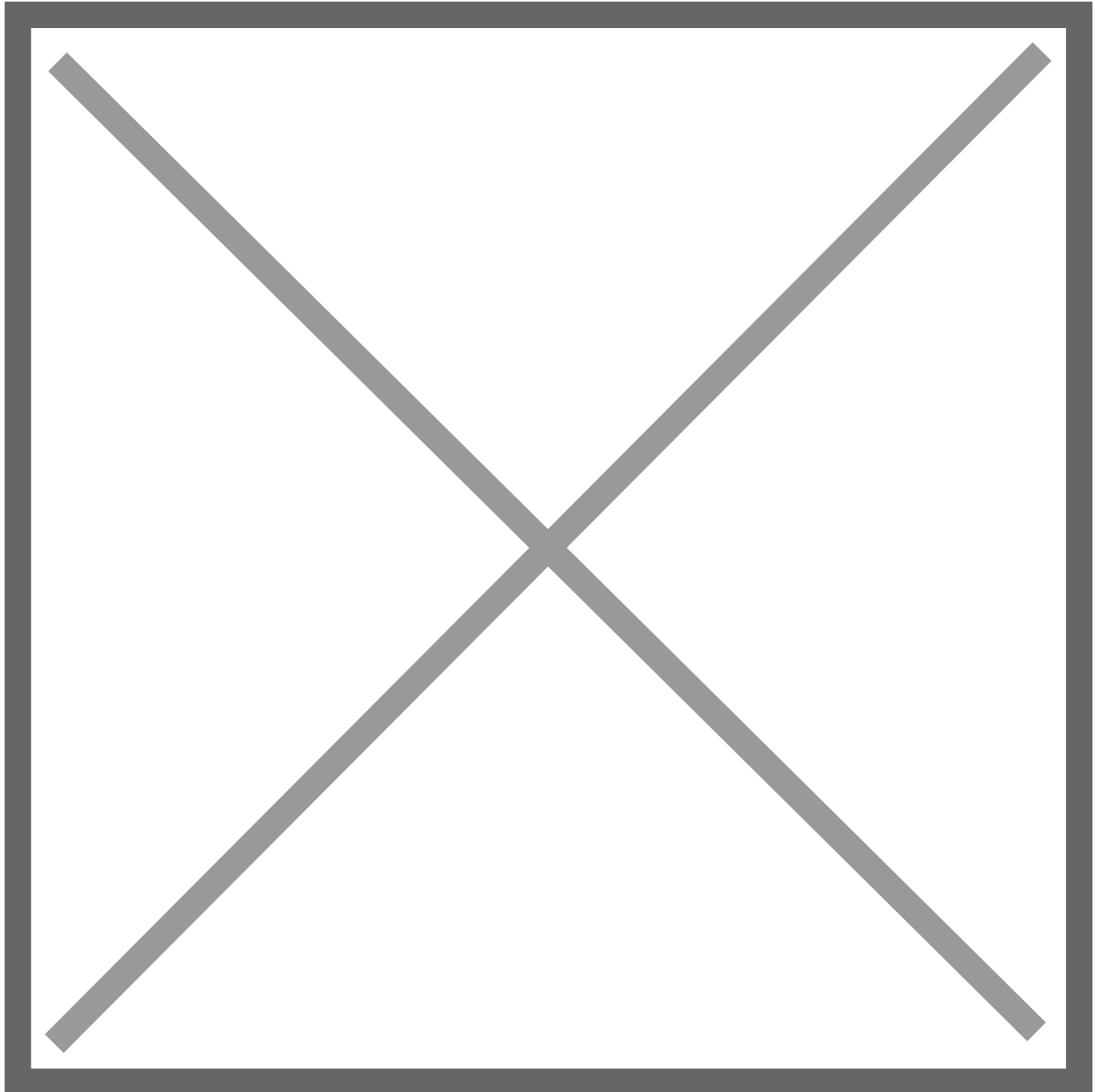
Multithreading is not totally noticeable in this example, but this next benefit is. Async has the ability to run sequentially using the keyword **await**.

But if Coroutines are made for sequential logic, then how is this a benefit? Coroutines *are* made for sequential logic, but they are not made for sequentially executing coroutines.

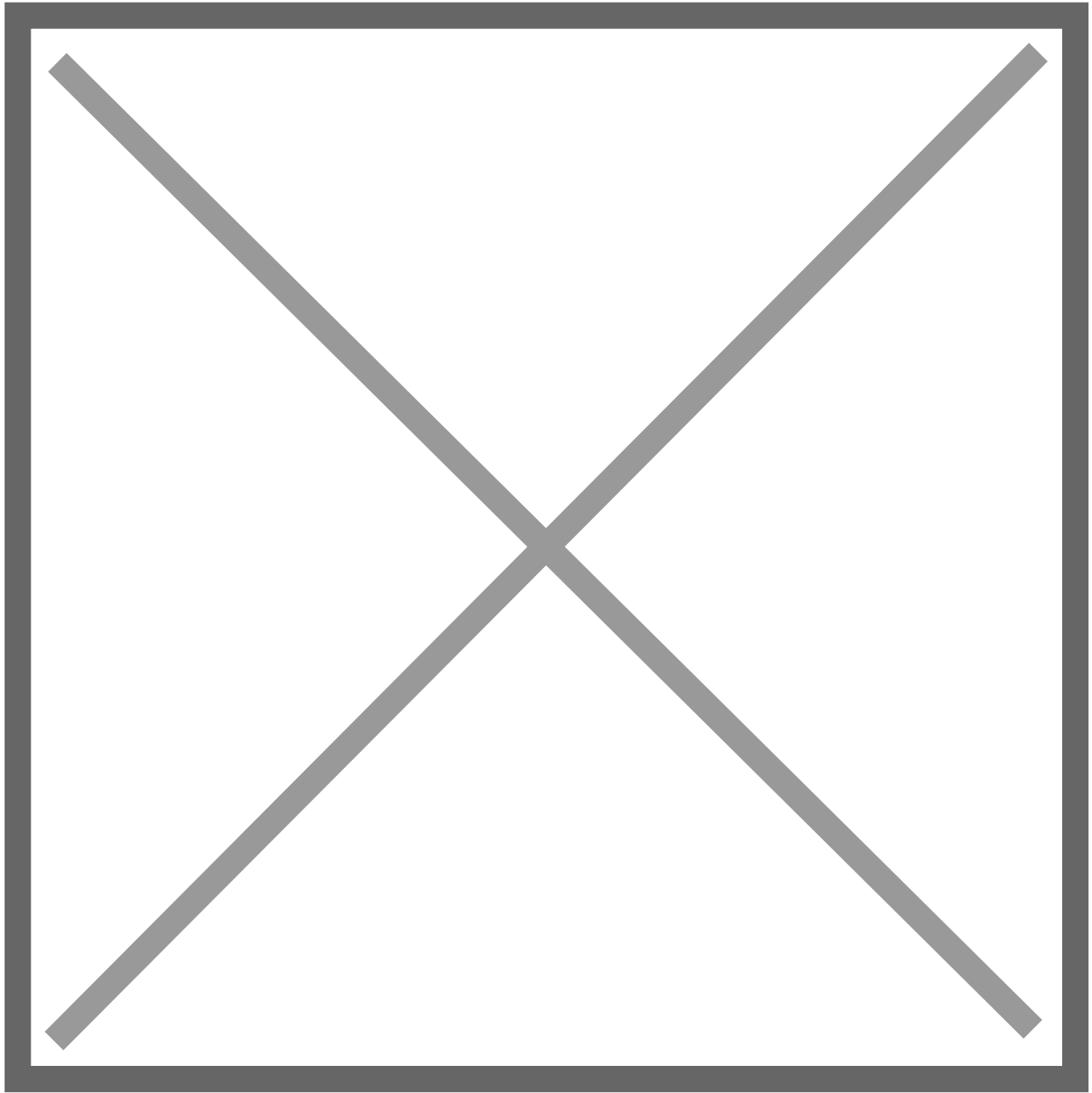
Now, there are ways to sequentially execute coroutines. The absolute simplest way would be to have a **StartCoroutine()** function at the end of a coroutine. This chaining however, will lead to

very difficult code to unpick when it comes time to refactor. A cleaner way to handle this with async methods is using **await**.

Our first step is to change the return type, from **void** to **Task**. Then we can use the await functionality by making the original function also async.

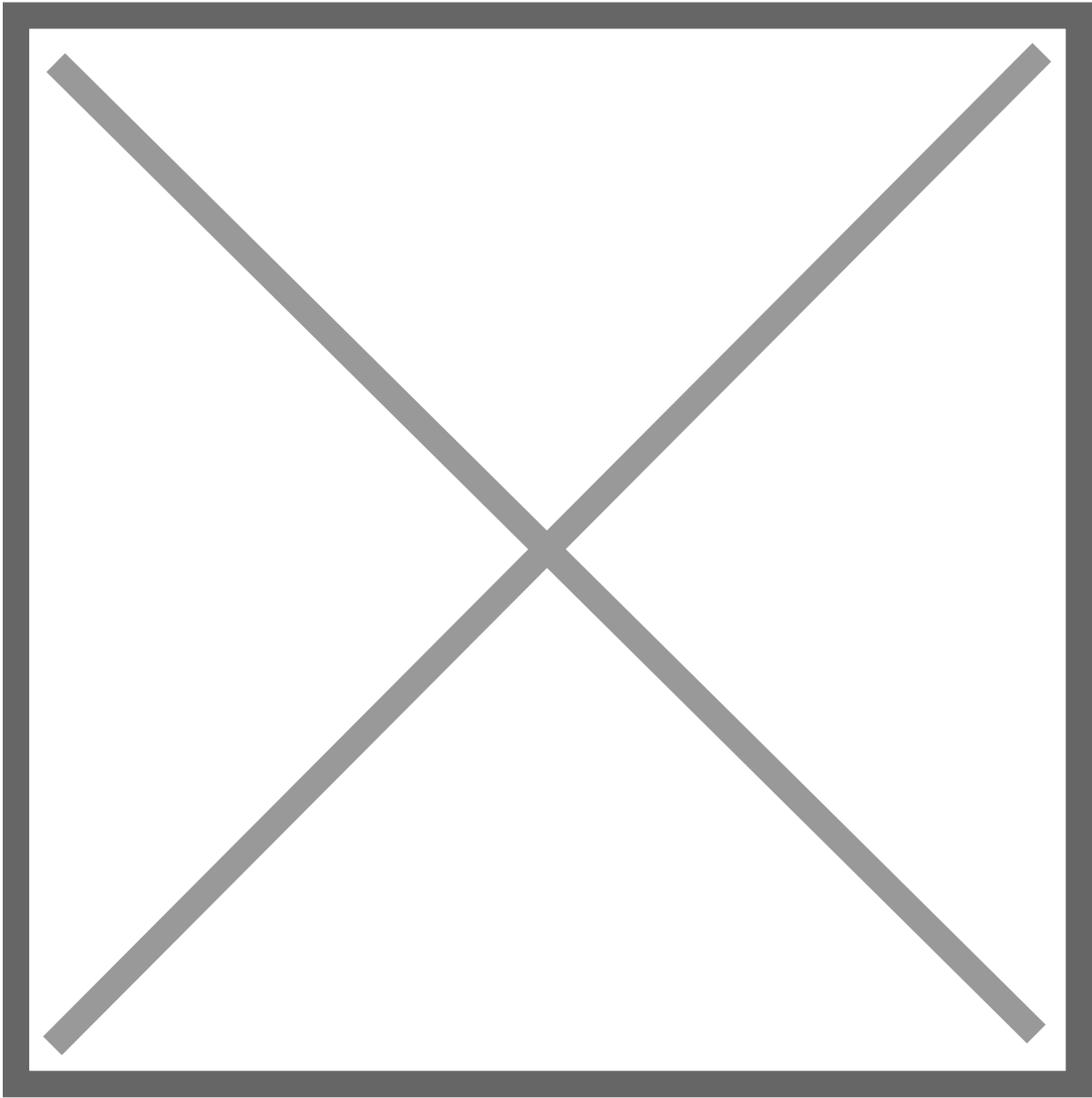


Now when we test this logic, we will see the cubes only rotate after the one before it is finished.

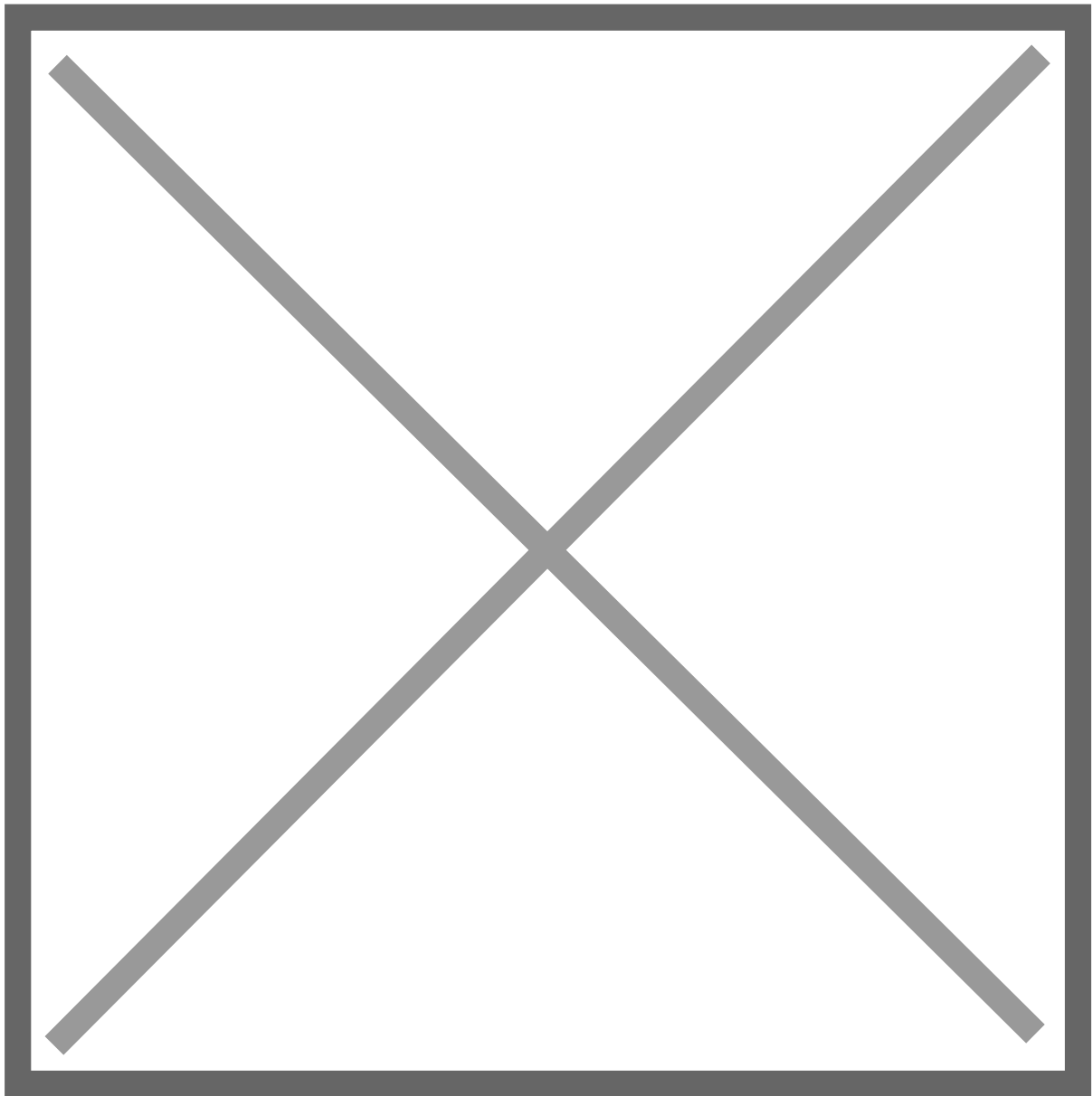


Another benefit, that might convince you to never use coroutines again, is that with `async`, you can make logic wait until a series of tasks have completed.

To demonstrate this, I'm going to create a simple array of colors, then have the button change **ONLY** after all the cubes have finished moving.

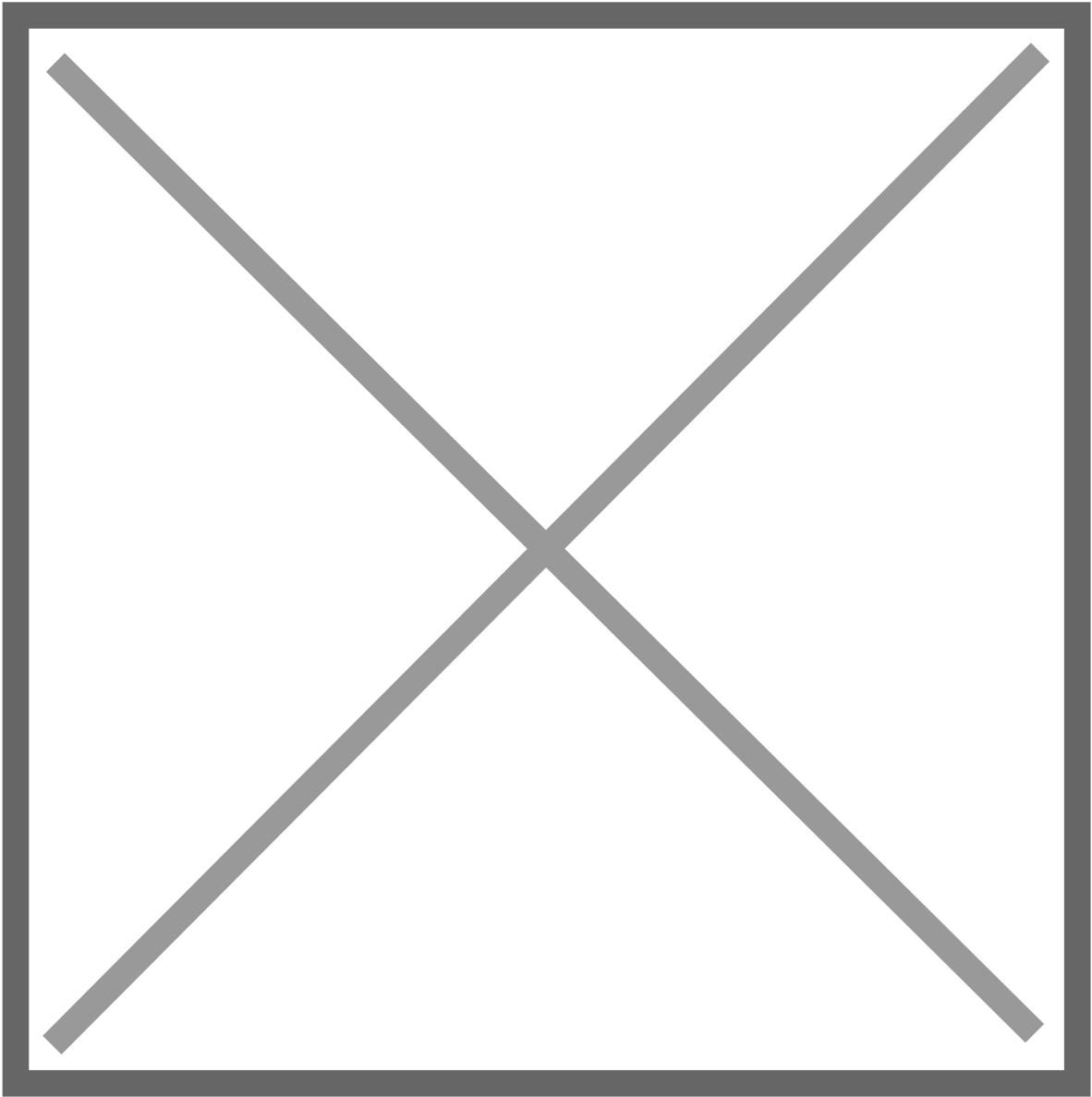


Before we change this logic, let's see what we have currently.

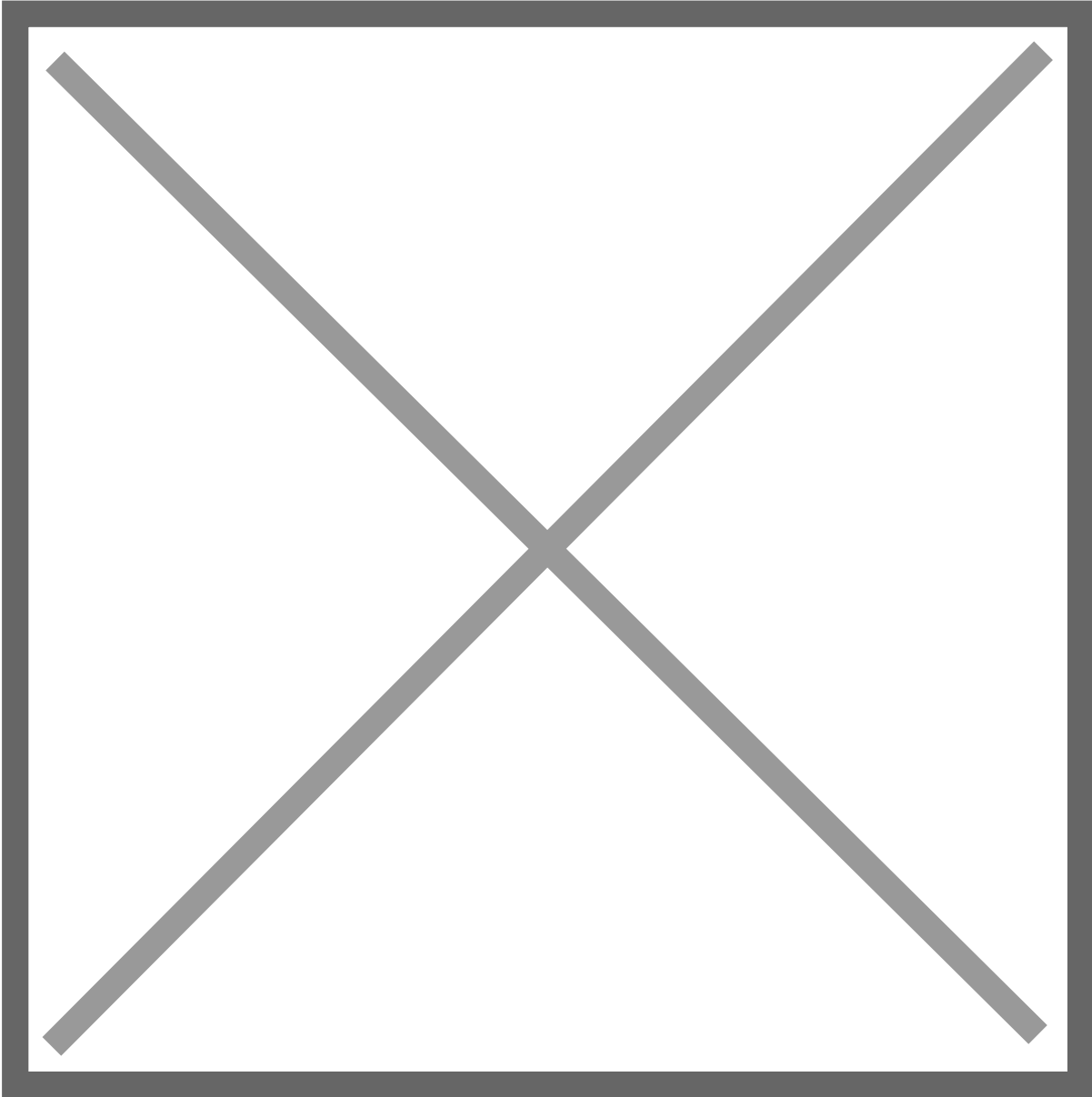


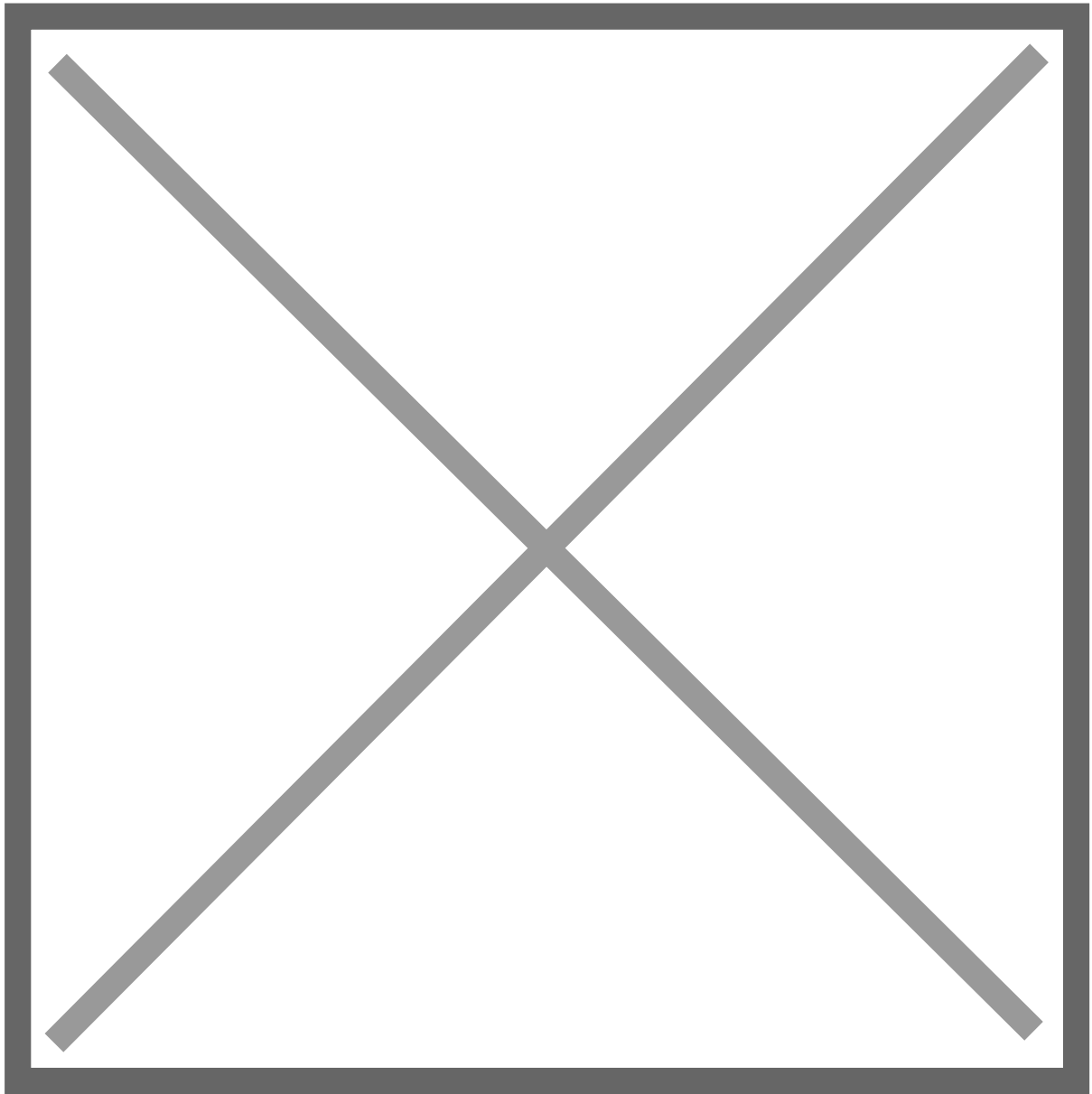
Okay, so now with `async`, we will be able to change the behavior around so that all the cubes will rotate at the same time, but the button will still only change after they are all finished.

We are going to want to create a list of tasks that we can add to, then check to see if all those tasks have finished before changing the color of the button.



The entire code will now look like the following.





That's pretty cool right there.

Takeaway

This article has only scratched the surface of the power of asynchronous programming but it has definitely convinced me to make the switch from coroutines to async.