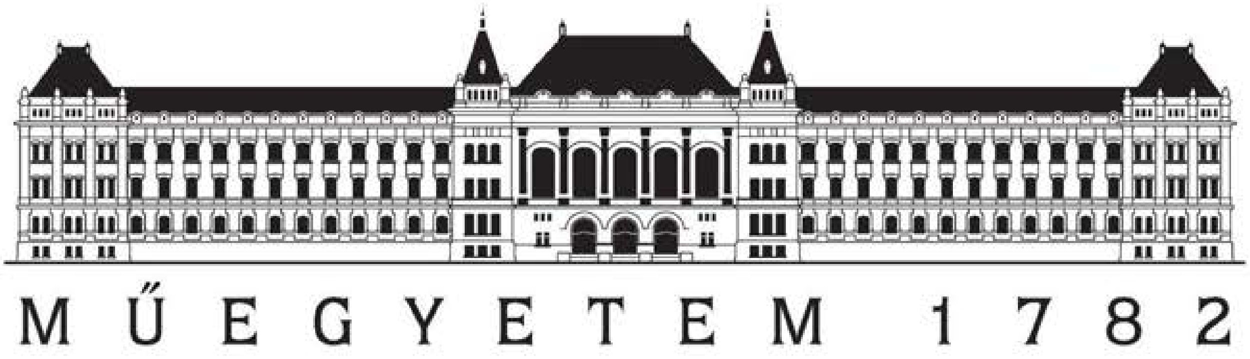
Proposing engine specific Design Patterns for Unity using Coroutines and the new Data-Oriented Technology Stack

Project Laboratory - Menyhárt Bence
Consultant - Dr. Magdics Milán



Summary

Coroutines

Coroutines

About Coroutines

What are Coroutines

In general, Coroutines are computer program components that generalize subroutines for non-preemptive multitasking[1], by allowing execution to be suspended and resumed.

In Unity, Coroutines are a type of methods which can pause execution, save state, then yield control back to Unitys game loop, so later in time (usually in the next frame) the coroutine can continue execution where it "left off".

[1] Cooperative multitasking, also known as non-preemptive multitasking, is a style of computer multitasking in which the operating system never initiates a context switch from a running process to another process. Instead, processes voluntarily yield control periodically or when idle or logically blocked.

What are Coroutines good for in Unity

- Cooperative multitasking
- Distribute logic across multiple frames
- Taking back execution more conveniently at specific game events
- Make Update methods more lightweight

Implementation of Coroutines in Unity

A good way of implementing coroutines in .Net is by using iterators.
Unity also used this concept when they implemented their own coroutines.

A coroutine yields an `IEnumerator` interface (this is the iterator), which will tell Unity's Coroutine Scheduler when the execution shall continue. Let's see an example:

```
public class CoroutineExample : Monobehaviour
{
    public bool IsReady = false;

    void Start()
    {
        // Coroutines need to be started by Monobehaviour.StartCoroutine() methoc
        // coroutines otherwise they are just plain iterator methods
        StartCoroutine(ExampleCoroutine());/*1*/
    }

    void IEnumerator ExampleCoroutine()
    {
        Debug.Log("Starting of ExampleCoroutine...");/*2*/
        // WaitUntil is one of Unitys built in yield instruction
        // https://docs.unity3d.com/ScriptReference/WaitUntil.html
        yield return new WaitUntil(() => _isReady);/*3,4*/
        Debug.Log("The coroutine is ready to continue");/*5*/
    }/*6*/
}
```

The above script does the following:

1. Starts the coroutine
2. Logs the "*Starting of ExampleCoroutine...*" string to Unitys console
3. Yields the execution of the coroutine till the supplied delegate to `WaitUntil` s constructor evaluates to true
4. After the delegate evaluates to true aka the `IsReady` variable becomes true the coroutines execution will continue
5. Logs the "*The coroutine is ready to continue*" string to Unitys console
6. The execution of the coroutine finishes.

Now let's inspect the above code snippet a bit more in depth!

1. MonoBehaviours `StartCoroutine()` method registers the coroutine into Unitys coroutine scheduler. After that the scheduler will periodically ask (basically in every frame) the yield instruction if it is done with waiting or not.

(Assuming you yielded on a YieldInstruction, not just returned a value or let the coroutine finish in that frame. Also a simple `yield return <object>;` means you yielded the execution for one frame.).

There are two types of Yield Instructions in Unity:

i, Yield Instruction that implements the interface `IEnumerator` [2]
ii, Yield Instruction that inherits from the class named `YieldInstruction` [3]

We will only speak about the first type of Yield Instructions since we don't have insight into the second type. You can learn how to write your own Yield Instruction in the Writing Yield Instructions section.

When you start your iterator method aka your coroutine with `StartCoroutine(myCoroutine())` Unity will start executing it immediately and continues execution till the first `yield` statement, unlike when you just simply call an iterator method like this

```
IEnumerator myEnumerator = myCoroutine();
```

in the above case the execution of the method won't start with the `()` operator, you need to call `myEnumerator.MoveNext()` to start the execution of the iterator method till the first `yield` statement.

So a C# equivalent of `StartCoroutine()` would be something like this

```
IEnumerator myEnumerator = myCoroutine();
myEnumerator.MoveNext();
```

Its easy to see now that what the Coroutine Scheduler does is just simply calling the `bool Movenext()` method. So a Yield Instruction's `MoveNext()` method can be translated to `Should_I_Still_Be_Suspended()` where true means yes, you **shall not** proceed to the next `yield` statement please yield control back to Unity and false means no please proceed and yield the control back to Unity at the next `yield` statement or at the end of the method.

3. Yielding back happens here, with a Yield Instruction called `WaitUntil`. It's important to note here that yielding back the execution is not a blocking operation, and also the execution of coroutines happens on the Main thread.
Let's inspect `WaitUntil` implementation[4]:

```
// Unity C# reference source
// Copyright (c) Unity Technologies. For terms of use, see
// https://unity3d.com/legal/licenses/Unity_Reference_Only_License

using System;

namespace UnityEngine
{
    public sealed class WaitUntil : CustomYieldInstruction
    {
        Func<bool> m_Predicate;

        public override bool keepWaiting { get { return !m_Predicate(); } }

        public WaitUntil(Func<bool> predicate) { m_Predicate = predicate; }
    }
}
```

Looks like there is no sign of `MoveNext()` method, but in fact `WaitUntil` s parent class `CustomYieldInstruction` implements the `IEnumerator` interface and this is how its `MoveNext()` method looks like:

```
public bool MoveNext() { return keepWaiting; }
```

It looks like it just passes the keepWaiting property. So in our case, the coroutine will be suspended till the `IsReady` Property evaluates to true.

Note, that `keepWaiting` is returning `!m_Predicate()` rather then just `!m_Predicate()` because as we talked about this earlier `IEnumerator.MoveNext()` in this context means basically `Should_I_Still_Be_Suspended()` (or according to Unity `keepWaiting`) so just returning our `IsReady` boolean default false value would let the coroutine proceed with it's execution in the next frame rather then waiting for it to become true.

▼ [2] There is an abstract class named `CustomYieldInstruction` which basically just implements the `IEnumerator` interface so you can derive from it and make a custom yield instruction. But, because of this class also introduces a sometimes useless bool property, and also prevents inheriting another class we will rather just simply stick to `IEnumerator` interface. (Click on the arrow to view the class implementation [4])

```
// Unity C# reference source
// Copyright (c) Unity Technologies. For terms of use, see
// https://unity3d.com/legal/licenses/Unity_Reference_Only_License

using System.Collections;

namespace UnityEngine
{
    public abstract class CustomYieldInstruction : IEnumerator
    {
        public abstract bool keepWaiting
        {
            get;
        }

        public object Current
        {
            get
            {
                return null;
            }
        }
        public bool MoveNext() { return keepWaiting; }
        public virtual void Reset() {}
    }
}
}
```

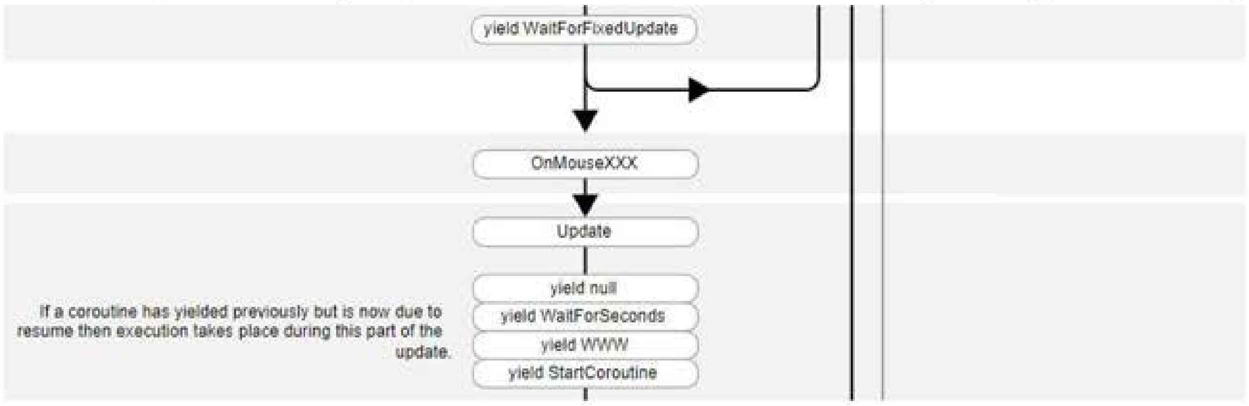
[3] As of 2020 there are only three classes that inherits from `YieldInstruction`, namely `WaitForSeconds`, `WaitForEndOfFrame` and `WaitForFixedUpdate`. All of these are pointing into Unitys Native code and we have no informations about their internal mechanism.

[4] Unitys C# code is Open Source since 2019 you can inspect it here: https://github.com/Unity-Technologies/UnityCsReference

Understanding Yield Instructions

Execution pipeline of Yield Instructions

Before diving into deeper to Yield Instructions let's see when the evaluation of the different yield instructions happens. The following picture from the Unity Manual will help us understand it[5]:


yield WaitForFixedUpdate
OnMouseXXX
Update
yield null
yield WaitForSeconds
yield WWW
yield StartCoroutine
If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.

As you can see coroutines are executed after all Fixed and normal Updates have taken place, unless you used a `WaitForFixedUpdate` Yield instruction. There is a built-in Yield Instruction that is cropped from the image namely the `WaitForEndOfFrame` which happens after Unity has renderd every Camera and GUI and just before the displaying of the current frame would happen.

As you can see it is guaranteed that our coroutines will resume execution after all Updates have been finished, but it's important to note that we have no guarantee that the order of execution of our coroutines will stay the same on the life cycle of our application. These are important to note before writing custom yield instructions or designing architectures based on coroutines.

[5] You can watch the uncropped picture here: https://docs.unity3d.com/Manual/ExecutionOrder.html

Writing Yield Instructions

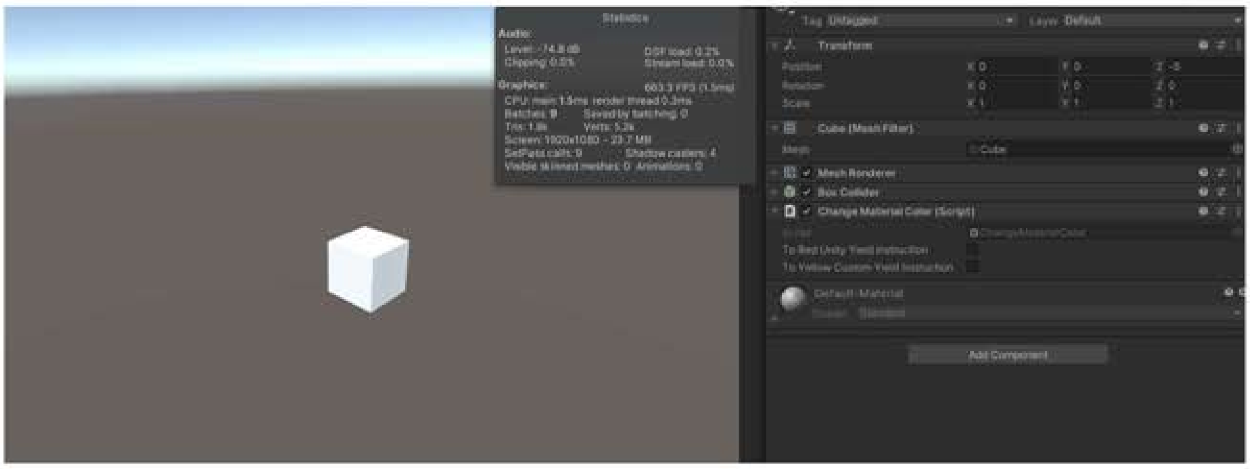
In this section we will write a custom yield instruction

As mentioned above it's easy to write a custom yield instruction, just implement the `IEnumerator` interface's `MoveNext` method and `Current` property. You can write extremely complicated yield instructions just be sure to return an appropriate bool from your `MoveNext()` method.

This is how a custom WaitUntil looks like:

```
using System;
using System.Collections;

/// <summary>
/// A custom yield instruction using IEnumerator
/// </summary>
public class CustomWaitUntil : IEnumerator
{
    /// <summary>
    /// The predicate that will be evaluated every frame
    /// </summary>
    Func<bool> _predicate;

    // This is processed after Unity's coroutine scheduler executes the MoveNext(
        public object Current { get { return null; } }

        public CustomWaitUntil(Func<bool> predicate) { _predicate = predicate; }

        // Comes from IEnumerator Interface, called by Unity in every frame after all
        public bool MoveNext()
        {
            return !_predicate();
        }

        // Comes from IEnumerator Interface, this is not processed by Unity
        public void Reset() { throw new NotImplementedException(); }

    }
```

Example:
Changes the cube color to red when using the built-in `WaitUntil` and changes the color to yellow when using our custom `WaitUntil`



Open the corresponding example found in the project to test the code for yourself you can even compare it to Unitys built-in `WaitUntil`

Writing an advanced Yield Instruction

In this section we will write an advanced yield instruction that will monitor two `Transform` by caching it in a class scoped variable. The yield instruction will signal a yielded state till the two transforms are closer than 5 meters.

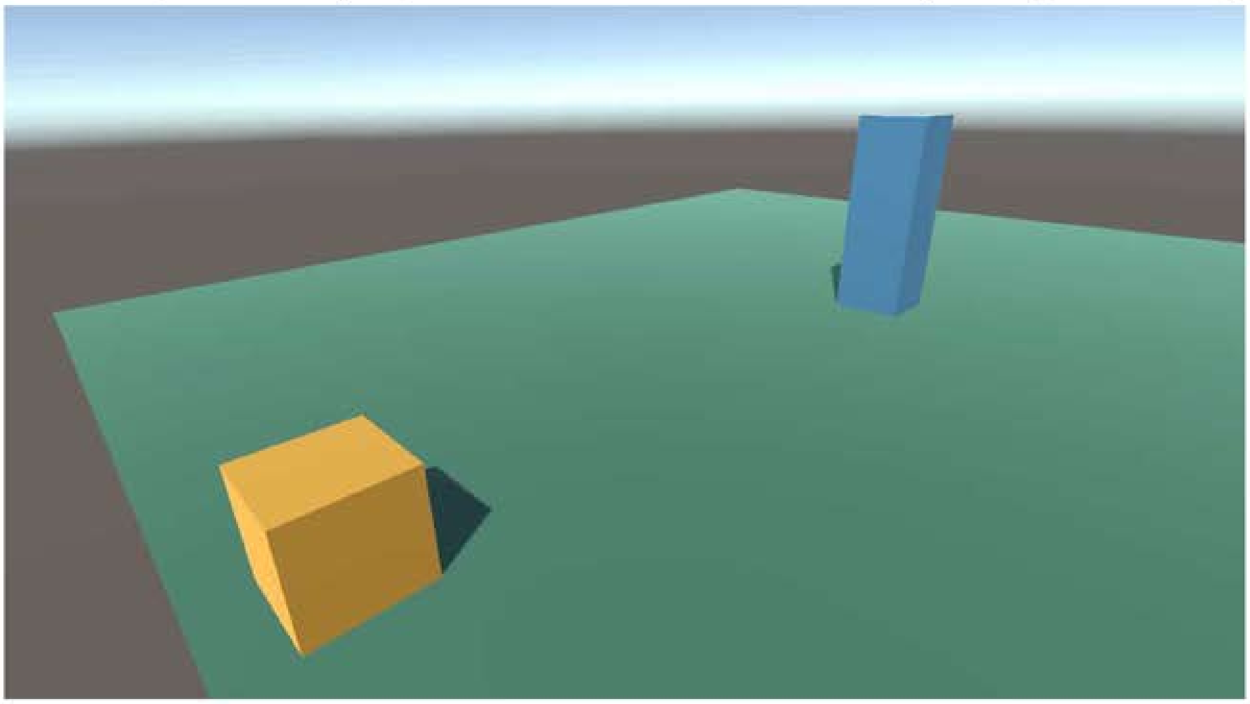
We can easily make a yield instruction like this, we just need a constructor with two `Transform` parameter and an appropriate logic in our `MoveNext()` method.
Let's see a possible implementation:

```
using System.Collections;
using UnityEngine;

/// <summary>
/// Suspends the execution of the coroutine till the supplied transforms are furt
/// </summary>
public class WaitUntilInRange : IEnumerator
{
    // Transform is a class thus it is passed by reference so we can cache it
    Transform _observer;
    Transform _observed;

    public WaitUntilInRange(Transform observer, Transform observed)
    {
        _observer = observer;
        _observed = observed;
    }

    // Should I Still Be Suspended?
    public bool MoveNext()
    {
        // Yes, the enemy is still out of range
        if (Vector3.Distance(_observer.position, _observed.position) > 5.0f)
            return true;
        // No, the enemy is in range
        else
            return false;
    }

    public void Reset() { throw new System.NotSupportedException(); }

    public object Current { get { return null; } }
}
```

Let's see this in Action!

Example:

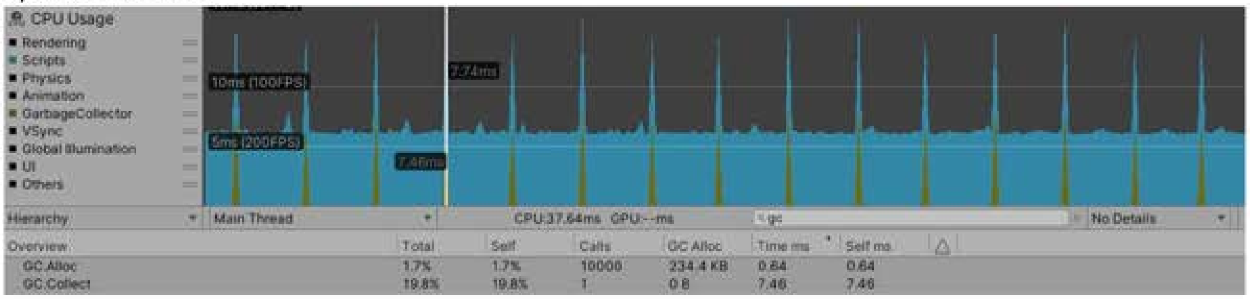
The cube Game Object will turn to red when it gets closer than 5 meters to the green tower.



Open the corresponding example found in the project to test the code for yourself!

The importance of caching Yield Instructions

In this section we will learn why we should cache yield instructions, so we can avoid GC spikes like this!



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
GC.Alloc	1.7%	1.7%	10000	234.4 KB	0.64	0.64
GC.Collect	19.8%	19.8%	1	0 B	7.46	7.46

C# is a managed language thus it is using GarbageCollector (GC) to free up unused memory. In game development where methods are executed multiple times per second efficient memory management is crucial. YieldInstructions are classes thus they are created on the heap which is managed by the GC. Every usage of the new keyword creates a new instance of the YieldInstruction which will then need to be Garbage collected when all references to the class are out of scope.

When you yield back a yield instruction be sure to use reusable instances of it whenever you can.

Good practice

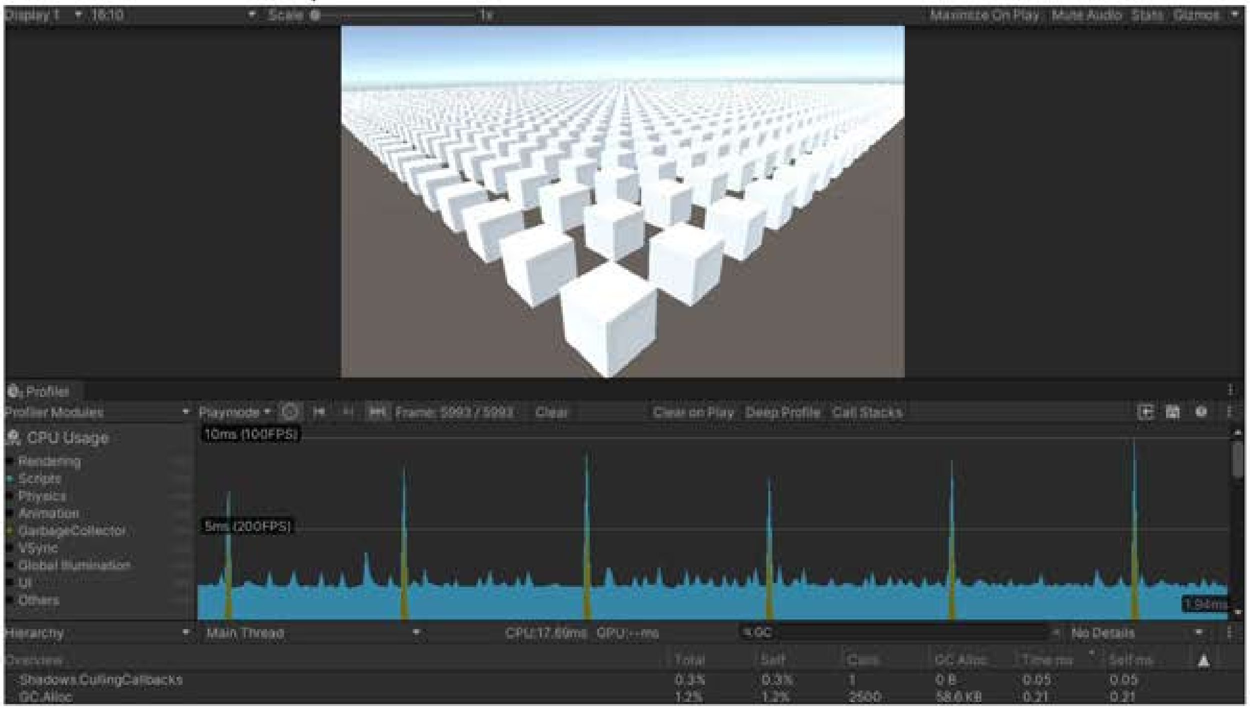
```
yield return _waitUntilPlayerDies;
```

Bad Practice

```
yield return new _waitUntilPlayerDies();
```

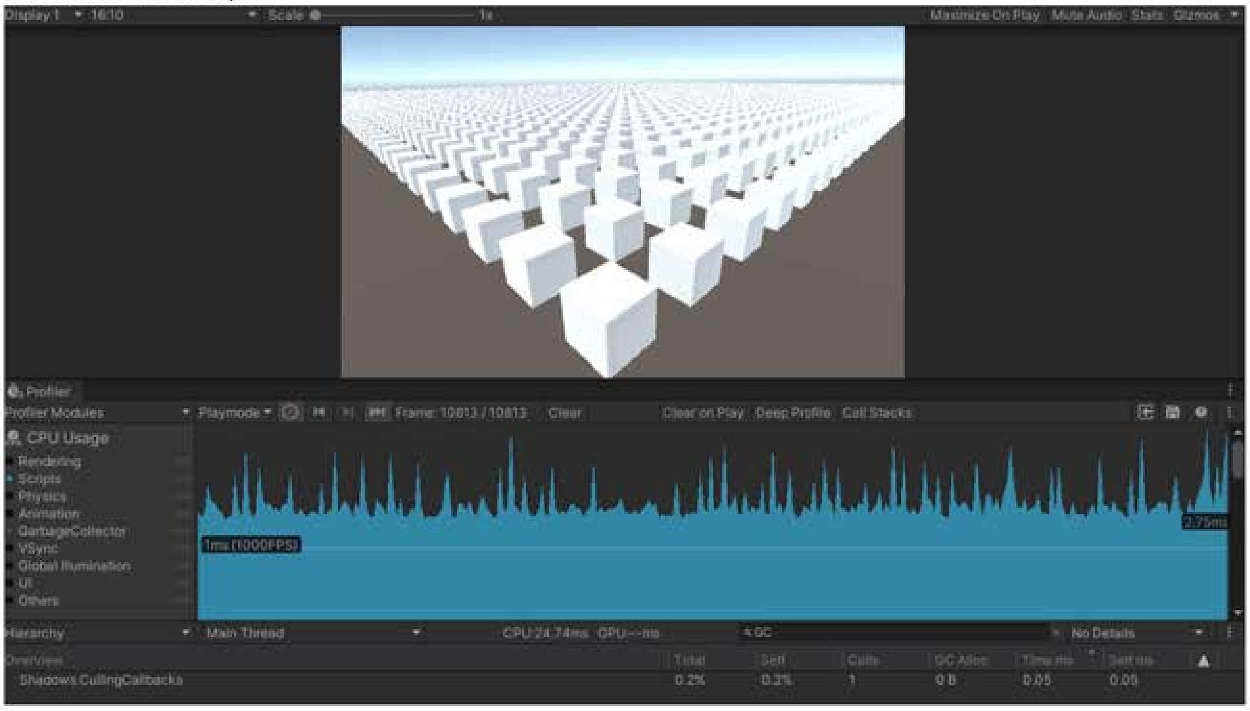
Example:
We will instantiate a lot of GameObjects each with an associated coroutine with uncached yield instruction then we will do the same with a cached solution then profile the two

Uncached - 100-200 fps



```
Overview                 Total   Self   Calls   GC Alloc   Time ms   Self ms
 Shadows.CullingCallbacks 0.3%    0.3%   1       0 B        0.05      0.05
 GC.Alloc                 1.2%    1.2%   2500    58.6 KB    0.21      0.21
```

Cached - 1000 fps



```
Overview                 Total   Self   Calls   GC Alloc   Time ms   Self ms
 Shadows.CullingCallbacks 0.2%    0.2%   1       0 B        0.05      0.05
```

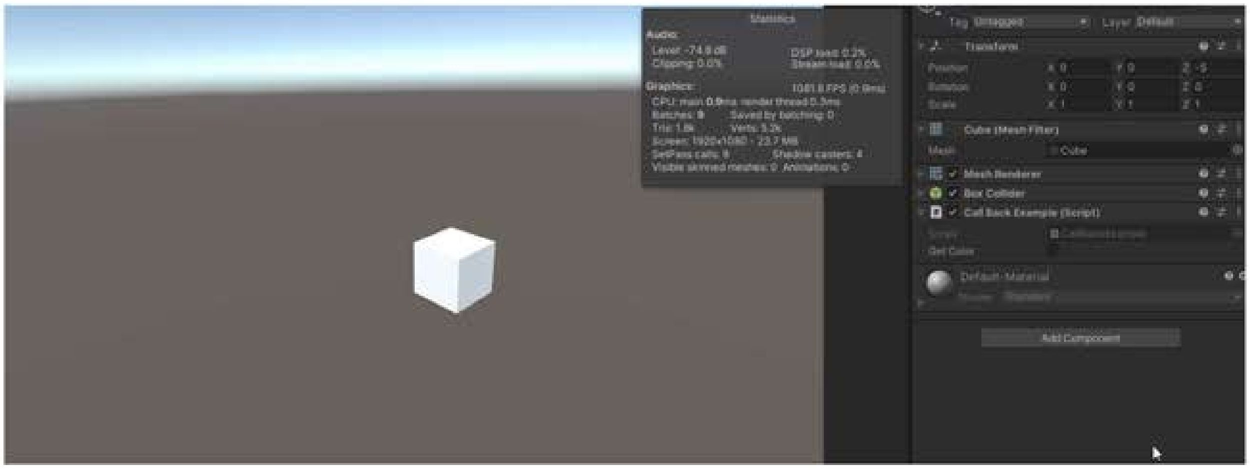
Catching the return value of a Coroutine

In this section we will learn how to catch the return value of a coroutine.

In Unity as we spoke about it earlier Coroutines are implemented by using iterator methods. Iterators methods have some restriction that make our life harder. We can't use the ref/out/inf keywords inside their parameter list, and also we just can't wait for them to return a value. One way to retrieve or work with the value produced by a coroutine is to use callbacks, in async programming it is a common practice to use callbacks.

Example:
Asks the server to what color it should paint the cube GameObject, then after the server responds the function given to the coroutine will be executed aka the callback.

```
IEnumerator GetCubeColorFromServerCoroutine(Action<Color> callBackMethod)
```



Design patterns using Coroutines

Threaded Coroutine

Description

As we talked about this earlier coroutines do not run on a separate thread, they run on Unity's main thread and get scheduled for execution by Unity's internal coroutine scheduler. Because of this coroutines naturally are not a solution for parallel computation.
However, fortunately we can still back them with threads and tasks.

For this we will need the following:

a, A robust, fast and lightweight way to make threads
b, Synchronization between the coroutine thread and the engine thread
c, A way to properly cancel the coroutine thread to avoid undesired effects specifically when developing

Solution

a, A good way to handle parallel computation in C# is using `System.Threading.Tasks` rather than directly using `System.Threading.Thread`. Tasks by default run on the threadpool thus on a worker thread rather than on a separate thread. This is good in most cases but when you have a Task that will run for an extended duration of time, running that task on a threadpool thus on a worker thread is not a good idea. The reason is because threadpool shall rotate tasks quite often not just working on a task that will run for minutes thus blocking that worker thread out from the pool.
So we will need to provide a way for ThreadedCoroutines to allow this type of behaviour, luckily we can tell the Task if it shall run on a threadpool or on a separate thread.

b, We need to handle the synchronization between the coroutine thread and the engine thread, because of the nature of game engines and their mechanic that they work in a loop called the engine or game loop, the best way to synchronize is when the coroutine scheduler schedules our coroutine. If our backing thread is ready with it's job we continue executing if not we are just simply yielding back.

c, Tasks can be easily cancelled with a lightweight token called CancellationToken, we will use this to abort the execution of our ThreadedCoroutine.

Ideas from other design patterns

The mediator design pattern:
The class `ThreadedCoroutine` will basically behave as a mediator, the engine can only interract with the task and the underlying coroutine via this class.

The factory design pattern:
The class `ThreadedCoroutineManager` will basically make the ThreadedCoroutines rather than you just create them via their constructor.

Scheduler design pattern:
The locking logic of the threaded coroutine (kinda a week comaprison tho)

Active Object design pattern:
The communication of the derived class and it's abstract base class might remind you of this design pattern.

How does it work?

You derive from the class named `TheadedCoroutine` then override the methods named `WorkOnUnityThread` and `WorkOnCoroutineThread(CancellationToken cancellationToken)`. You basically Ping-Pong between these methods, when you need a backing thread you yield to the `WorkOnCoroutineThread()` method and when you need the main engine thread you yield to the `WorkOnUnityThread()` method.

`WorkOnUnityThread`:
This is a traditional coroutine, you can execute any logic here that you would normally execute in a coroutine, plus you have access to all the Unity APIs that are not thread safe. Whenever you need the backing thread, just yield the method named `RequestThreadedCoroutineThread()` (eg.: `yield return RequestThreadedCoroutineThread();`) after this execution will resume in the `WorkOnCoroutineThread(CancellationToken cancellationToken)` method where it left off (or in the begining if it's the first call to it).

`WorkOnCoroutineThread(CancellationToken cancellationToken)`:
This is the method that contains the logic that shall be executed in parallel to the engine thread, altough you do not have access to all of the Unity APIs here! Whenever you need the main thread for syncronization or for any reason just use the method `RequestUnitysMainThread(CancellationToken cancellationToken)` this will yield control to the engine loop and execution will resume in the traditional coroutine method called `WorkOnUnityThread`.

You shall only start ThreadedCoroutines via the class named `ThreadedCoroutineManager` because the engine needs to corectly schedule these coroutines to the scheduler for execution thus this class must derive from a `Monobehaviour`.

To sum it up, this solution basically ping-pongs the control of the execution between the engine thread and the backed thread, and by using coroutines we can do this in a state machine like manner.

For the implementation details browse the code, every method and decision is well documented for education purposes.

Notes

This is just one way of implementation of threaded coroutines. You could easily transform this code into another execution logic, by using delegates. For example rather than ping-ponging the control you could subscribe and unsunscribe methods than execute them on the correct thread. I choose ping-pong though because in my view it's easier to understand, however it's true that you can easily loose the yielding logic. So feel free to transfrom my code.

For Example:

```
// You execute these function in a tight loop
public Action ExecuteOnMainThread;
public Action ExecuteOnCoroutineThread;
.
.
.
// Then from outside you just simply subscribe for these
ExecuteOnMainThread += MyMethodToExecute();
ExecuteOnMainThread -= MyMethodToNotExecuteAnymore();
```

Scheduled coroutines using inner Monobehaviours

Description

Every experienced unity programmer knows that not every script should be derived from a Monobehaviour. Monobehaviours shall only be used when you would like to make a component. A good example to this is Input Handling. When you are not satisfied with Unitys built-in handler you would want to extend it. You can freely derive from the `Input` class:

```
public sealed class TouchInputHandler : Input
```

But most of the time you will need syncronization or timing with in synch of Unitys game loop. How to achieve synchronization? How will you communicate with Unity?

Solution

A good solution is to use a private inner class that derives from a monobehaviour. This will hide the inner mechanism of our InputHandling, and also we can hide this from users in the inspector/hierarchy view, so the art/level designers won't be confused. The Inputhandler will give nothing just static values that our programmers can read.

Implementation

```
/// <summary>
/// A non monobehaviour class with access to Unitys engine loop, without any depe
/// </summary>
public class InnerMonobehaviourDesignPattern : Input
{
    static InnerMonobehaviourDesignPattern()
    {
        // Creating a GameObject that will hold our "secret" component
        var gameObject = new GameObject();
```

```
            // Properly hiding it from other colleagues that shall not modify it
            gameObject.hideFlags = HideFlags.HideInHierarchy | HideFlags.HideInInspec
            // Adding the component
            _innerMonoBehaviour = gameObject.AddComponent<InnerMonoBehaviour>();
        }

        /// <summary>
        /// A static reference to our inner monobehaviour
        /// </summary>
        static InnerMonoBehaviour _innerMonoBehaviour;
        /// <summary>
        /// The hidden inner monobehaviour
        /// </summary>
        class InnerMonoBehaviour : MonoBehaviour
        {
            void Awake()
            {
                hideFlags = HideFlags.HideInHierarchy | HideFlags.HideInInspector;
            }
        }
    }
```

How does it work?

The static constructor will register the inner monobehaviour instance into Unitys engine loop, we properly hide this class with the `HideFlags` bit flags. From now on we have access to synchronization loops via this inner class.
We have access to:

- Unity messages (Start, Awake, OnEnable, OnGui, Update etc...)
- Unity's event system
- Unity's coroutines

and many more...

Example

An extended TouhInputHandler that allows convenient access to reading out double taps on an android phone.

```
using System.Collections;
using UnityEngine;

/// <summary>
/// A non monobehaviour class with access to Unitys engine loop, without any oute
/// </summary>
public class TouchInputHandler : Input
{
    #region Double Tap

    /// <summary>
    /// Tells whether a double tap was registered.
    /// </summary>
    /// <remarks>Change the <see cref="DoubleTapTimeFrame"/> to set custom time f
    public static bool DoubleTap
    {
        get
        {
            if (CheckDoubleTap())
                return true;
            else
                return false;
        }
    }
    public static float DoubleTapTimeFrame { get; set; } = 0.35f;
    public static Vector2 DoubleTapScreenPosition { get; private set; }

    #region Helper variables & methods

    static bool _tapped;
    static Coroutine startTimerForDoubleTap;

    static bool CheckDoubleTap()
    {
        if (_tapped && touchCount == 1 && GetTouch(0).phase == TouchPhase.Ended)
        {
            _innerMonoBehaviour.StopCoroutine(startTimerForDoubleTap);
            _innerMonoBehaviour.StartCoroutine(SetDoubleClickedMouseButtonToFalse
            DoubleTapScreenPosition = GetTouch(0).position;
            return true;
        }
        else if (touchCount == 1 && GetTouch(0).phase == TouchPhase.Ended)
        {
            startTimerForDoubleTap = CoroutineManager.StartCoroutine(StartTimerFc
            return false;
        }
        else
        {
            return false;
        }
    }

    static IEnumerator StartTimerForDoubleTapCoroutine()
    {
        _tapped = true;
        yield return new WaitForSeconds(DoubleTapTimeFrame);
        _tapped = false;
    }

    static IEnumerator SetDoubleClickedMouseButtonToFalse()
    {
        yield return new WaitForEndOfFrame();
        DoubleTapScreenPosition = Vector2.zero;
        _tapped = false;
    }
```

```
        #endregion

        #endregion

        #region InnerMonobehaviour Design Pattern

        static TouchInputHandler()
        {
            // Creating a GameObject that will hold our "secret" component
            var gameObject = new GameObject();
            // Properly hiding it from other colleagues that shall not modify it
            gameObject.hideFlags = HideFlags.HideInHierarchy | HideFlags.HideInInspec
            // Adding the component
            _innerMonoBehaviour = gameObject.AddComponent<InnerMonoBehaviour>();
        }

        /// <summary>
        /// A static reference to our inner monobehaviour
        /// </summary>
        static InnerMonoBehaviour _innerMonoBehaviour;
        /// <summary>
        /// The hidden inner monobehaviour
        /// </summary>
        class InnerMonoBehaviour : MonoBehaviour
        {
            void Awake()
            {
                hideFlags = HideFlags.HideInHierarchy | HideFlags.HideInInspector;
            }
        }

        #endregion
    }
```

Sources:
https://docs.unity3d.com/
https://en.wikipedia.org/wiki/Coroutine[1]
https://en.wikipedia.org/wiki/Cooperative_multitasking[1]
https://github.com/Unity-Technologies/UnityCsReference[4]