How to Change Over

Memory Efficient Coroutines are called slightly differently than Unity's coroutines. The structure is exactly the same, so in most cases it's just a matter of find and replace inside your code.

First, you need to include the two namespaces that MEC uses. MEC coroutines are defined in the MEC namespace and they also rely on the System.Collections.Generic functionality rather then the System.Collections functionality that unity coroutines use. In many cases this means you no longer need to include System.Collections in your scripts, but it doesn't hurt much to leave it in. So make sure these two using statements are at the top of every C# script that is using MEC coroutines:

```
using System.Collections.Generic;
using MEC;
```

Next, replace every instance of StartCoroutine, so this

```
StartCoroutine(CheckForWin());
```

is replaced with one of these three lines. (You have to pick the execution loop when you define the process.)

```
Timing.StartUpdateCoroutine(CheckForWin()); // To run in the Update loop.
Timing.StartFixedUpdateCoroutine(CheckForWin()); // To run in the FixedUpdate loop.
Timing.StartLateUpdateCoroutine(CheckForWin()); // To run in the LateUpdate loop.
```

The process' header will then need to be changed as well. It turns from this:

```
IEnumerator CheckForWin()
{
...
}
```

To this:

```
IEnumerator<float> CheckForWin()
{
...
}
```

Whenever you want to wait for the next frame, just yield return 0. So this,

```
IEnumerator CheckForWin()
{
    while (_cubesHit < TotalCubes)
    {
         WinText.text = "Have not won yet.";

         yield return null;
    }

     WinText.text = "You win!";
 }
```

Would turn into this:

```
IEnumerator<float> CheckForWin()
{
    while (_cubesHit < TotalCubes)
    {
        WinText.text = "Have not won yet.";

        yield return 0f;
    }

     WinText.text = "You win!";
 }
```

If you want to pause for some number of seconds rather than just one frame then you can either pass in the time that you want to resume the process (using something like "yield return Time.time + timeToWait;") or you can use Timing.WaitForSeconds. So this,

```
IEnumerator CheckForWin()
{
    while (_cubesHit < TotalCubes)
    {
        WinText.text = "Have not won yet.";

        yield return new WaitForSeconds(0.1f);
    }

     WinText.text = "You win!";
 }
```

Turns into this:

```
IEnumerator<float> CheckForWin()
{
    while (_cubesHit < TotalCubes)
    {
        WinText.text = "Have not won yet.";

        yield return Timing.WaitForSeconds(0.1f);
    }

    WinText.text = "You win!";
}
```

FAQ

Q: I'm trying to use MEC coroutines while switching scenes and my coroutine just quits!

A: All MEC coroutines are attached to an instance of the Timing object which lives in your scene. By default when you switch scenes all the GameObjects in your old scene are destroyed to make room for the new set of GameObjects in the new scene, which deletes all running coroutines. This can be a good thing. Depending on how you use coroutines you may have set up a few that just run forever checking a variable, and you wouldn't want that type of coroutine to keep running after changing scenes, which is why the following setup is not the default setup.

The best way to allow coroutines to run between scenes is to create an instance of the Timing object on a gameObject yourself. If you create a new GameObject and add the Timing script to it then MEC will use that object for its buffers instead of making its own. You can then set the gameObject.DontDestroyOnLoad flag to true and then all the coroutines will persist between scenes.

There are other ways to keep processes running between scenes as well, see the article on advanced control of process lifetime for more on that.

Q: Does MEC have a function for WaitForEndOfFrame?

A: No. Unity only provides one way to access that particular timing in the render pipeline: Though Unity's coroutines. If you need that specific timing for a particular process then you'll have to use a Unity coroutine to do it. (some people use it to take screencaptures without the UI getting in the way, for instance.) You can still use MEC for every other process, though.

Q: Does MEC have a function for StopCoroutine?

A: Yes. Timing.KillCorutine().

Q: Does MEC have a function for StopAllCoroutines?

A: Yes. Timing.KillAllCorutines(). You can also use Timing.PauseAllCorutines() and Timing.ResumeAllCorutines() if you would rather stop everything temporarily.

Q: Does MEC have a function to yield one coroutine until another one finishes?

A: Not currently. I'm working on it, but it's tricky.

Q: Does MEC completely remove GC allocs?

A: No. MEC removes all per-frame GC allocs. (unless you allocate memory on the heap inside of your coroutine, but I have no control over that.) When a coroutine is first created the function pointer and any variables you pass into it are put on the heap and eventually have to be cleaned up by the garbage collector. This unavoidable allocation happens in both Unity's coroutines and MEC coroutines.

Advanced Control of Process Lifetime

If you do nothing special the Timing object will add itself to either your uGUI event system's gameObject or, if there isn't one in your scene, to a new object named Movement Effects. All of the process tasks will normally be handed out by that instance.

However, if you want more control over things you can attach the Timing object to one of the gameObjects in your scene yourself. You could even create more than one Timing object if you like and add different processes to different objects. The functions for Timing.StartUpdateCoroutine() are static so they can be accessed from anywhere, but if you have a handle to an instance of the Timing object then you can call yourTimingInstance.RunUpdateCoroutine() to run the coroutine on that instance. By creating multiple instances of the Timing object you can effectively create groups of processes that can all be paused or destroyed together.

For the latest documentation, FAQ, or to contact the author visit http://trinary.tech/category/mec/