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 MovementEffects namespace and they also rely on the System.Collections.Generic functionality rather then the System.Collections functionality that unity coroutines use. System.Collections is hardly ever used for anything except Unity's coroutines, so an easy way to switch is to do a "Find and Replace," and then just change the lines that have errors. So make sure these two using statements are at the top of every C# script that is using MEC coroutines:

```
1. using System.Collections.Generic;
2. using MovementEffects;
```

Next, replace every instance of StartCoroutine, so this

```
1. StartCoroutine(_CheckForWin());
```

..is replaced with RunCoroutine. (You have to pick the execution loop when you define the process, and it defaults to "Segment.Update".)

```
1. Timing.RunCoroutine(_CheckForWin()); // To run in the Update segment.
2. // To run in the FixedUpdate segment:
3. Timing.RunCoroutine(_CheckForWin(), Segment.FixedUpdate);
4. // To run in the LateUpdate segment:
5. Timing.RunCoroutine(_CheckForWin(), Segment.LateUpdate);
6. // To run in the SlowUpdate segment:
7. Timing.RunCoroutine(_CheckForWin(), Segment.SlowUpdate);
```

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

```
1. IEnumerator _CheckForWin()
2. {
3. ...
4. }
```

To this:

```
1. IEnumerator<float> _CheckForWin()
2. {
3. ...
4. }
```

It is a very good idea to get in the habit of always putting an underscore before all coroutine functions. The reason for this is that coroutines (both Unity and MEC ones) have an annoying tenancy to pretend to run correctly but to actually quit on the first yield statement if you try to run the function without using RunCoroutine. The "_" before the function helps you to remember to always use "Timing.RunCoroutine(_CheckForWin());" rather than trying to call it using the line "_CheckForWin();".

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

```
1.  IEnumerator _CheckForWin()
2.  {
3.      while (_cubesHit < TotalCubes)
4.      {
5.          WinText.text = "Have not won yet.";
6.
7.          yield return null;
8.      }
9.
10.     WinText.text = "You win!";
11. }
```

Would turn into this:

```
1.  IEnumerator<float> _CheckForWin()
2.  {
3.      while (_cubesHit < TotalCubes)
4.      {
5.          WinText.text = "Have not won yet.";
6.
7.          yield return 0f;
8.      }
9.
10.     WinText.text = "You win!";
11. }
```

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,

```
1.  IEnumerator _CheckForWin()
2.  {
3.      while (_cubesHit < TotalCubes)
4.      {
5.          WinText.text = "Have not won yet.";
6.
7.          yield return new WaitForSeconds(0.1f);
8.      }
9.      WinText.text = "You win!";
10. }
```

Turns into this:

```
1. IEnumerator<float> _CheckForWin()
2. {
3.     while (_cubesHit < TotalCubes)
4.     {
5.         WinText.text = "Have not won yet.";
6.
7.         yield return Timing.WaitForSeconds(0.1f);
8.     }
9.
10.     WinText.text = "You win!";
11. }
```

SlowUpdate

Unity's coroutines don't have a concept of a slow update loop, but MEC coroutines do.

The slow update loop runs (by default) at 7 times a second. It uses absolute timescale, so when you slow down Unity's timescale it will not slow down SlowUpdate. SlowUpdate works great for tasks like displaying text to the user, since if you were to update the value any faster than that then the user wouldn't really be able to see those rapid changes anyway.

```
1. Timing.RunCoroutine(_UpdateTime(), Segment.SlowUpdate);
2.
3. private IEnumerator<float> _UpdateTime()
4. {
5.     while(true)
6.     {
7.         clock = Time.time;
8.
9.         yield return 0f;
10.     }
11. }
```

SlowUpdate also works well for checking temporary debugging variables. For instance, if it takes a long time to rebuild your project you might set up a public bool in your script that will reset the values on that script. You'll need to check if the value of that bool has been set to true periodically, and the perfect period to do that check is on SlowUpdate. When the user checks the checkbox it will feel like it responds immediately, but it will use far less processing in your app to check it every 1/7th of a second than 30 – 100 times per second (depending on your framerate).

You can change the rate that SlowUpdate runs.

```
1. Timing.Instance.TimeBetweenSlowUpdateCalls = 3f;
```

The line above will make SlowUpdate only run once every 3 seconds.

Additional Functionality

There are two helper functions that are also included in the Timing object: CallDelayed and CallContinously.

- CallDelayed calls the specified action after some number of seconds.
- CallContinously calls the action every frame for some number of seconds.

Both of these could easily be created using coroutines, but this basic functionality ends up being used so often that we've included it in the base module.

```
1. // This will start the _RunFor5Seconds coroutine, 2 seconds from now.
2. Timing.CallDelayed(2f, delegate {
   Timing.RunCoroutine(_RunFor5Seconds(handle)); });
3.
4. // This does the same thing, but without creating a closure.
5. //  (A closure creates a GC alloc.)
6. //  "handle" is being passed in to CallDelayed and CallDelayed passes
7. //  it back to RunCoroutine as the variable "x".
8. Timing.CallDelayed<IEnumerator<float>>(handle, 2f, x => {
   Timing.RunCoroutine(_RunFor5Seconds(x)); });
```

```
1. private void PushOnGameObject(GameObject go, Vector3 amount)
2. {
3.     go.transform.position += amount * Time.deltaTime;
4. }
5. // Push "myObject" forward one world unit per second for 4 seconds.
6. Timing.CallContinuously(4f, PushOnGameObject(myObject,
   Vector3.forward), Segment.FixedUpdate);
7.
8. // CallContinously also has a non-closure version. It's extra important
9. //  to try not to make closures on CallContinously, since it will
10.  //  result in a GC alloc every frame.
11.  Timing.CallContinuously<GameObject>(myObject, 4f, x =>
   PushOnGameObject(x, Vector3.forward), Segment.FixedUpdate);
```

By default MEC prints any exceptions out to the console and quits the coroutine that threw the exception. However, if you want to do something different with exceptions then you can define your own custom error receiver:

```
1. Timing.Instance.OnError = OnError;
2.
3. private void OnError(Exception exception,
4.     IEnumerator<float> enumerator)
5. {
6.     Debug.LogError("The exception was seen: " + exception.Message);
7. }
```

If you decide to run with more than one Timing instance then you can define different error handlers for different instances.

FYI: You'll save yourself some trouble if you never point the OnError function to a lambda expression, since lambda expressions break their links whenever the Unity debugger recompiles.. and you're much better off if your error handling code is solid.

One MEC coroutine can yield to another MEC coroutine using the WaitUntilDone function. WaitUntilDone can also wait for the WWW object.

```
1.   yield return Timing.WaitUntilDone(coroutineOrWWWobjectToWaitFor);
```

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 default behavior is to kill all running coroutines when changing scenes.

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?

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.

NOTE: There is some confusion about what WaitForEndOfFrame actually does. When you just want to yield until the next frame then WaitForEndOfFrame is not really an ideal command, it's better to use "yield return null;" (which is "yield return 0f;" in MEC). Many people use WaitForEndOfFrame because it's the closest thing they can find to WaitForNextFrame and they either don't know to use "yield return null;" or they enjoy returning objects with explicit variable names.

This page has a graph that shows the timing for each frame. As the graph shows, WaitForEndOfFrame executes after all rendering has finished. If you were to use that call in a Unity coroutine to move a button across the screen then the button would always be drawn in the position you had set it to on the previous frame. In most cases the frame rate is high enough that you wouldn't notice the difference visually, but this practice can cause subtle visual glitches that are difficult to explain or debug.

For instance, if you had an enemy ship and an "enemy ship explodes" animation it would be common practice to call Destroy on the enemy ship object and Instantiate the explosion animation at the same time. However, if you did this in a coroutine that had been calling WaitForEndOfFrame then the user might see what appears to be the ship blinking for an instant before exploding.

Q: Does MEC have a function for StopCoroutine?

A: Yes. It's called Timing.KillCorutine(). NOTE: If you want to stop the current running coroutine you should not use KillCoroutine, use "yield break;" instead.

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: Yes. From inside the coroutine that you want to hold you call "yield return Timing.WaitUntilDone(coroutineHandle);"

NOTE: The handle must be the same variable that you originally passed into or got out of RunCoroutine. If you attempt to create a new instance of the handle then the WaitUntilDone function will fail to find a match.

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.

Q: Are MEC coroutines always more memory efficient than Unity coroutines, or is it only in select cases?

A: MEC coroutines produce less GC allocation than Unity coroutines do in all cases. In theory they should run a little faster as well, but this hasn't been verified yet.

Q: Reduced GC allocs are great, but are there any other advantages to MEC coroutines over Unity's coroutines.

A: Unity's coroutines are attached to the object that you started them on while MEC uses a central object to run all it's processes. That means that the following three things are true:

1. Unity's coroutines won't start if the current object is disabled. MEC coroutines don't care.
2. If you disable a GameObject then all of the Unity coroutines that are attached to it will quit running (they do not resume on re-enable.) MEC coroutines don't do this.
3. If the GameObject that you started a Unity coroutine on is destroyed then all the attached coroutines will also be killed. MEC coroutines also don't do this.

MEC coroutines allow you to create coroutine groups, which give you the ability to pause/resume or destroy whole groups of coroutines at the same time. Unity's coroutines don't allow you to pause and resume coroutines from the outside, or easily create coroutine groups.

Lastly, MEC coroutines allow you to run the coroutine in the LateUpdate or SlowUpdate segment if you want to.

Advanced Control of Process Lifetime

If you do nothing special the Timing object will add itself to a new object named "Movement Effects". All of the coroutine processes 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 coroutines to different objects. The functions for Timing.RunCoroutine() 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.RunCoroutineOnInstance() 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.

The OnError delegate and the TimeBetweenSlowUpdateCalls variable are also attached to the Timing instance, so you can set different error handling for different timing objects or you can set SlowUpdate to run at a different rate.

Example

```
1. using UnityEngine;
2. using System.Collections.Generic;
3. using MEC;
4.
5. public class Testing : MonoBehaviour
6. {
7.     void Start ()
8.     {
9.     IEnumerator<float> handle = Timing.RunCoroutine(_RunFor10Seconds());
10.
11.         handle = Timing.RunCoroutine(_RunFor1Second(handle));
12.
13.         Timing.RunCoroutine(_RunFor5Seconds(handle));
14.     }
15.
16.     private IEnumerator<float> _RunFor10Seconds()
17.     {
18.         Debug.Log("Starting 10 second run.");
19.
20.         yield return Timing.WaitForSeconds(10f);
21.
22.         Debug.Log("Finished 10 second run.");
23.     }
24.
25.     private IEnumerator<float> _RunFor1Second(IEnumerator<float>
   waitHandle)
26.     {
27.         Debug.Log("Yielding 1s..");
28.
29.         yield return Timing.WaitUntilDone(waitHandle);
30.
31.         Debug.Log("Starting 1 second run.");
32.
33.         yield return Timing.WaitForSeconds(1f);
34.
35.         Debug.Log("Finished 1 second run.");
36.     }
37.
38.     private IEnumerator<float> _RunFor5Seconds(IEnumerator<float>
   waitHandle)
39.     {
```

```
40.            Debug.Log("Yielding 5s..");
41.
42.            yield return Timing.WaitUntilDone(waitHandle);
43.
44.            Debug.Log("Starting 5 second run.");
45.
46.            yield return Timing.WaitForSeconds(5f);
47.
48.            Debug.Log("Finished 5 second run.");
49.      }
50.  }
```

This is the output:

```
1. Starting 10 second run.
2. Yielding 1s..
3. Yielding 5s..
4. Finished 10 second run.
5. Starting 1 second run.
6. Finished 1 second run.
7. Starting 5 second run.
8. Finished 5 second run.
```

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