Editor Update

Unity's coroutines don't run in the editor when the app is not playing, but MEC coroutines can.

Editor coroutines are generally used to give the user a preview of an effect before the game runs, or to perform update or networking sync tasks that need to be done before the actual app is compiled. Editor coroutines will all be destroyed when the user hits the play button, and will not run while in play mode. Time.time and Time.deltaTime will not work while in editor mode, instead you can either use EditorApplication.timeSinceStartup (in the UnityEditor namespace) or you can just use Timing.LocalTime and Timing.DeltaTime.

Be conservative when using editor coroutines: An infinite loop or a very processor intensive function here can destroy your entire project. Always make sure to have a backup of your project available whenever you mess with these. Also keep in mind that any changes you make to an object in your scene in the editor will change the initial state of that object when you run or compile the app. For example, you could make an editor coroutine that made your character wink at the developer before play mode, but this would have the side effect of making the character randomly start mid-wink occasionally when they started or compiled the app.

In order to run in the editor, set the Segment to EditorUpdate or EditorSlowUpdate. Make sure your class has the [ExecuteInEditMode] tag attached, and then use them normally. Here is an example:

```
1. using UnityEngine;
2. using System.Collections.Generic;
3. using MovementEffects;
4.
5. [ExecuteInEditMode]
6. public class EditorTesting : MonoBehaviour
7. {
8.
9.     void OnEnable()
10.      {
11.          Timing.RunCoroutine(_RunOverAndOver(), Segment.EditorUpdate);
12.      }
13.
14.      IEnumerator<float> _RunOverAndOver()
15.      {
16.          while(true)
17.          {
18.              if(!enabled)
19.                  yield break;
20.
21.              Timing.RunCoroutine(_MoveThisObject(),
   Segment.EditorUpdate);
22.
23.              yield return Timing.WaitForSeconds(1f);
24.          }
25.      }
26.
```

```
27.
28.      IEnumerator<float> _MoveThisObject()
29.      {
30.          double startTime = Timing.LocalTime;
31.          Vector3 direction = Random.onUnitSphere;
32.
33.          while(Timing.LocalTime - startTime < 7d)
34.          {
35.              Vector3 tmp = transform.position;
36.              tmp += direction * Timing.DeltaTime;
37.              transform.position = tmp;
38.
39.              yield return 0f;
40.          }
41.      }
42.  }
```

Realtime Update

Another timing segment you can use in MEC Pro is RealtimeUpdate. This segment is just like update, but Timing.LocalTime and Timing.DeltaTime ignore Unity's timescale. This can be useful while controlling menus that happen while your game is paused.

Pause and Resume

In More Effective Coroutines Pro you can pause all coroutines that have a particular tag or layer and resume them later.

This could be useful, for instance, if you had a two layer menu system. If you tagged every movement coroutine with the string "layer1" and opened a menu on layer 2 you could easily stop all movement on layer1 with the command "Timing.PauseCoroutines("layer1″);". Later, when your layer 2 menu closed you could resume all movement on layer 1 right where it left off by calling "Timing.ResumeCoroutines("layer1″);"

Remember that with tags the string has to match exactly every time, so "layer1" is not the same as "Layer1", "layer 1", or "laier1". Always make sure you can spell all tags consistently and watch out for capitalization and spaces (both are ok, but you have to use them the same way every time).

Layers and Tags

The free version of MEC only has tags, but MEC Pro adds layers. Collectively tags and layers are called graffiti because they are both just ways to identify a particular instance of a coroutine. The only real difference between a layer and a tag is that a layer is an integer and a tag is a

string, but in MEC Pro you have the option of supplying one, the other, or both. Once you supply graffiti for an instance, you can pause/resume, kill, or use that graffiti for RunCoroutineSingleton.

In Unity every GameObject is assigned a unique number, which can be accessed by calling gameObject.GetInstanceID(). The instance id can change every time you run the application, but it is guaranteed that no other gameObject instance will be assigned the same id during a single run. So even if you have a swarm of enemies that all have the same scripts attached to them you can still run them on a layer which is the instance id of the gameObject they were created on and then kill all coroutines attached to one particular enemy using

```
Timing.KillCoroutines(enemy.gameObject.GetInstanceID());
```

If you had, say, an amnesia gun you could graffiti all your enemy's AI logic with both the instance id and a tag:

```
 Timing.RunCoroutine(_EnemyAI(), gameObject.GetInstanceID(), "AI");
 // Somewhere else in code:
void HitEnemyWithAmnesiaGun(EnemyController enemy)
{
    Timing.KillCoroutines(enemy.gameObject.GetInstanceID(), "AI");
}
```

KillCoroutines only kills the instance that match all the graffiti that you pass in, so if you coded it right the above code would make the enemy stand there and do nothing, but would also leave any other coroutines that might be running on that particular character untouched.
Unity's coroutines will always cancel all of the coroutines that are associated with a GameObject whenever you disable that object. Sometimes you would rather just pause a coroutine while it's disabled and then resume it as soon as the object is re-enabled. That sort of pattern is really easy to set up in MEC like this:

```
void Start ()
{
    Timing.RunCoroutine(_MoveUpAndDown(), gameObject.GetInstanceID());
}
void OnEnable()
{
    Timing.ResumeCoroutines(gameObject.GetInstanceID());
}
void OnDisable()
{
    Timing.PauseCoroutines(gameObject.GetInstanceID());
}
```

Extension Methods

Extension methods can be extremely powerful tools. They allow you to run the same coroutine, but change the way it runs while running it. That probably sounds confusing, so here's an example:

AddDelay
AddDelay runs the coroutine after a delay.

```
1. using UnityEngine;
2. using UnityEngine.UI;
3. using System.Collections.Generic;
4. using MovementEffects;
5.
6. public class ButtonExample : MonoBehaviour
7. {
8.     public Button button1;
9.     public Button button2;
10.      public Button button3;
11.
12.
13.      void Start ()
14.      {
15.          Timing.RunCoroutine(_MoveBackAndForth(button1).AddDelay(1f));
16.          Timing.RunCoroutine(_MoveBackAndForth(button2).AddDelay(2f));
17.          Timing.RunCoroutine(_MoveBackAndForth(button3).AddDelay(3f));
18.      }
19.
20.      private IEnumerator<float> _MoveBackAndForth(Button myButton)
21.      {
22.          Vector3 myPos = myButton.transform.localPosition;
23.          float time = 0f;
24.
25.          while(time < 2f)
26.          {
27.              myPos.x += Timing.DeltaTime;
28.              time += Timing.DeltaTime;
29.              transform.localPosition = myPos;
30.              yield return 0f;
31.          }
32.
33.          while (time < 4f)
34.          {
35.              myPos.x -= Timing.DeltaTime;
36.              time += Timing.DeltaTime;
37.              transform.localPosition = myPos;
38.              yield return 0f;
39.          }
40.      }
41.  }
```

The code above will simply move three buttons, first 2 units to the right over 2 seconds, and then 2 units to the left. The interesting part is that during the Timing.RunCoroutine calls we used the

AddDelay function to add a different delay to each call, so button1 will start moving after one second, button2 will start moving after 2 seconds, and button3 after 3 seconds. This is far cleaner than passing in the delay and putting it at the top of the function.

CancelWith
This can take either a GameObject or a function that returns a bool. If you pass in a GameObject then the coroutine will automatically be terminated if the GameObject is destroyed or disabled. You might want to use this if your coroutine is moving some UI element and that UI element might occasionally end up being destroyed before the movement completes. CancelWith will ensure that the coroutine quits cleanly without throwing any exceptions. If you want to safely modify two or three game objects there are overloads to pass in two or three at a time. If you want to be safe from more than three game objects going out of scope then you can chain calls to CancelWith.

If you pass in a function then the coroutine will be terminated as soon as that function returns false.

```
1. int framesLeft = 100;
2. GameObject obj1;
3. GameObject obj2;
4. GameObject obj3;
5. GameObject obj4;
6.
7. void Start()
8. {
9.     Timing.RunCoroutine(_Coroutine().CancelWith(obj1, obj2,
   obj3).CancelWith(obj4));
10.      Timing.RunCoroutine(_Coroutine().CancelWith(CancelFunction));
11.  }
12.
13.  bool CancelFunction()
14.  {
15.      if(framesLeft <= 0)
16.          return false;
17.
18.      framesLeft--;
19.      return true;
20.  }
```

Append and Prepend
These two functions can be used to chain two (or more) coroutines together. So this would turn right and then turn left:

```
1. Timing.RunCoroutine(_TurnRight().Append(_TurnLeft()));
```

They can also be used to append a delegate to the end of a coroutine. So this might be used to run a coroutine and then destroy the object once it was done:

```
1. Timing.RunCoroutine(_MoveToFinalPosition(obj1).Append(delegate {
   Destroy(obj1); }));
```

Of course in many cases you could have also just added the line to Destroy(obj1) to the end of the movement coroutine, but if you are using the same move function in several places and you don't normally want to destroy the object at the end then this can be a clean way to add that functionality without fracturing your code base.

There are also many cases where you might want to call some event once a coroutine has finished. If that event is related more to how you are calling the coroutine than what the coroutine is doing then it is better to encapsulate the code for the event in the place that is calling it, and this pattern does that.

Superimpose
This quirky function superimposes two coroutines into a single handle. The combined coroutine won't finish until both of its contributing coroutines are done. This can be combined with the WaitUntilDone function to make a coroutine wait until both functions finish before continuing. Here's an example of that:

```
1. void Start()
2. {
3.     var handle =
   Timing.RunCoroutine(_NetworkStream1().Superimpose(_NetworkStream2()));
4.     Timing.RunCoroutine(_RunWhenDone(handle));
5. }
6.
7. IEnumerator<float> _NetworkStream1()
8. {
9.     // Networking stuff happens here.
10.  }
11.
12.  IEnumerator<float> _NetworkStream2()
13.  {
14.      // Other networking stuff happens here.
15.  }
16.
17.  IEnumerator<float> _RunWhenDone(IEnumerator<float> handle)
18.  {
19.      yield return Timing.WaitUntilDone(handle);
20.      // This part gets run as soon as both networking streams are done.
21.  }
```

Hijack
This fun little function alters the return value of a coroutine. This can be useful for cutscenes or replays where you want the same code to be executed but in slow motion.

```
1. float slowdown = 0.1f;
2.
3. Timing.RunCoroutine(_MoveButton().Hijack(input =>
4. {
5.     if(input <= Timing.LocalTime)
6.         input = (float)Timing.LocalTime;
7.     return input + slowdown;
8. }));
```

All extension functions are very lightweight additions, so feel free to use them liberally or chain them together if you like.

If you would like to know more about the performance of MEC vs Unity's coroutines, take a look at this video: https://www.youtube.com/watch?v=sUYN8XtuUFA

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