

Coroutines

Executive Summary

Coroutines are the core mechanism in Unity3d MonoBehaviour classes to provide independent parallel processing. Activities that must occur in order use `yield return myCoroutine();` to wait on completion before continuing.

A problem occurs when we want functions to occur in a specific order but from different parts of the code. A player logs in to a remote system. They then go on to something that requires authorisation. That code does not know whether the authorisation has completed. With Askowl Coroutines a queue is created and tasks placed on it when needed. This way any coroutine with dependencies won't run until said dependencies have done their thing.

Some unity packages, specifically FireBase, use C# 4+ Tasks, a preemptive multitasking interface. Anything using Task callbacks must be treated as independent threads with semaphores and the like to protect against data corruption. The Askowl Tasks class converts Task responses into Coroutines so that they fit better and more safely into the Unity system.

Coroutines

- [Executive Summary](#)

- [Coroutines](#)

 - [Coroutines.Sequential](#)

 - [Coroutines.Queue](#)

 - [Coroutines.Completed](#)

- [Tasks](#)

 - [Tasks.WaitFor](#)

 - [Tasks.WaitFor<T>](#)

Read the code in the Examples Folder.

Coroutines

Coroutines have been around since the 50's. Microsoft used what was called cooperative multi-tasking in Windows before Windows 95. The similarities are fierce, but Windows worked on the application level while Coroutines are on a function level. With coroutines, the onus is entirely on the developer to decide when to release the CPU to other waiting coroutines. Coroutines don't work for every situation, but they are excellent for the uses made by Unity3D. They use very few resources, are more straightforward to use, and are not as prone to deadlocks.

Coroutines are much easier to deal with than preemptive multitasking. Perhaps we should call the latter uncooperative multi-tasking. Coroutines can make asynchronous code look sequential.

```

1  IEnumerator ShowScore() {
2      while (true) {
3          yield return ScoreChanged();
4          yield return UpdateScore();
5      }
6
7  IEnumerator ScoreChanged() {
8      while (score == lastScore) {
9          yield return null;
10     }
11     lastScore = score;
12 }

```

However, what to do when one coroutine is reliant on another when the calls are remote?

```

1  IEnumerator Start() {
2      yield return InitAuth();
3  }
4
5  public IEnumerator Login() {
6      yield return LoginAction();
7      yield return CheckLoginAction();
8  }
9
10 public IEnumerator SendMessage() {
11     yield return SendMessageAction();
12 }

```

Can you see the problem? If `Login()` is called before `InitAuth()` completes, it will fail. You could set a ready variable in `Start()` and wait for it whenever you call a dependent function. Do you create another variable to wait on in `SendMessage()` until `Login()` is complete? Can a server tolerate requests coming in out of order? We can recode it more cleanly with **Coroutines**.

Coroutines.Sequential

```

1  Coroutines auth = null;
2
3  IEnumerator Start() {
4      auth = Coroutines.Sequential(this, InitAuth());
5  }
6
7  public void Login() {
8      auth.Queue(LoginAction(), CheckLoginAction());
9  }
10

```

```

11     public void SendMessage() {
12         auth.Queue(SendMessageAction());
13     }
14
15     public IEnumerator Ready() {
16         yield return auth.Completed();
17     }

```

Coroutines.Queue

Here, `Coroutines.Queue(params IEnumerator[])` creates a queue of actions that are always be run one after another - in the order injected.

Coroutines.Completed

Wait for all the currently queued coroutines to complete.

Tasks

As I have discussed before, Unity3D uses Coroutines to provide a form of multitasking. While it is very efficient, it is not very granular. The refresh rate is typically 60Hz. So, 60 times a second all coroutines that are ready to continue has access to the CPU until they rerelease it.

C#/.NET also has preemptive multitasking support. Microsoft introduced an implementation of the Task-based asynchronous pattern. It is likely that Unity3D will move away from coroutines. Tasks are not limited to the 60Hz granularity. Nor are they limited to a single CPU or core. Long-running processes do not require special care. You have to understand how to make code thread-safe. With that comes the risk of the dreaded deadlock.

Google Firebase uses tasks for Unity. We need a thread-safe bridge since the rest of the current code relies on coroutines. Fortunately, most, if not all, of the asynchronous operations are not so critical that they can't fit into the 60-hertz coroutine cycle.

Tasks.WaitFor

WaitFor has a signature of:

```

1 public static IEnumerator WaitFor(Task task, Action<string> error =
   null);

```

And here is an example of converting Task time to Coroutine time:

```

1 int counter = 0;
2
3 // Start an asynchronous task that completes after a time in
  milliseconds
4 Task Delay(int ms, string msg) {

```

```

5         return Task.Run(async () => {
6             Debug.Log(msg);
7             await Task.Delay(ms);
8             counter++;
9         });
10    }
11
12    [UnityTest]
13    public IEnumerator TasksExampleWithEnumeratorPasses() {
14        Task task = Delay(500, "1. Wait for task to complete");
15        yield return Tasks.WaitFor(task);
16        Assert.AreEqual(counter, 1);
17
18        task = Delay(500, "2. Wait for task to complete with error
19processing");
20        yield return Tasks.WaitFor(task, (msg) => Debug.Log(msg));
21        Assert.AreEqual(counter, 2);
22
23        Debug.Log("3. All Done");
24    }

```

This simple example creates a task that runs on a different thread for a specified amount of time. By setting a counter, we can be sure the coroutine is not returning until the task has completed.

The error action is optional. If not supplied, the error goes to the debug log.

Tasks.WaitFor<T>

If the Task is to return a value, then use the generic version.

```

1    public static IEnumerator WaitFor<T>(Task<T> task, Action<T> action,
2    Action<string> error = null);

```

Here you must provide an action to do when the Task completes and returns a value.

The first hiccup is the DotNET version. `System.Threading.Tasks` was not implemented until DotNet 4. If you want code to compile when using DotNet 2, wrap it in `#if (!NET_2_0 && !NET_2_0_SUBSET)`.