Coroutine Manager Pro
1.0

If you have any questions, or suggestions for improvements, please email robert.wells@gandhigames.co.uk.

Overview

The coroutine manager contains three main classes/modules: CM_Job, CM_JobManager, and CM_JobQueue. Each module has a different task, as briefly outlined below.

The CM_Job class encapsulates the creation of a coroutine job. Using this class, you can easily and quickly create coroutines that can be paused (with or without a delay), resumed, killed, made repeatable etc.

The CM_JobManager class provides global named access to coroutines you create, and the ability pause, resume, and kill a coroutine from anywhere in your codebase.

The CM_JobQueue enables you to create queues of coroutines, where the next job will not be started until the previous has finished. This queue can be made repeatable; and be paused, resumed, or killed, at anytime.

For an overview of how to use these modules please see the example scripts (ExampleJobTest, ExampleJobManagerTest, and ExampleJobQueueTest) and the associated example scenes (JobTestScene, JobManagerTestScene, and JobQueueTestScene). These provide high level examples of how to call the methods for each module.

As well as the test scenes there are a number of example scenes, which contain small examples of how the coroutine manager could be used. They are located under CM/Examples/Scenes/Examples. These include:

- ExampleCharacterMovement: using a CM_JobQueue script a character is moved around the screen.
- ExampleCharacterDamage: shows how you could create an action queue for a player. Adding actions to this queue results in them being run in order. Press 1 on your keyboard to simulate fire damage, 2 to simulate poison damage, and 3 to heal.
- ExampleSpawner: using a CM_Job script an object is continuously spawned.
- ExampleJobManagerPauseResume: a number of timers are shown on screen. Their text is updated using a job added to the global job manager. Left-click to pause jobs and right-click to resume jobs.
- ExampleGUI: a simple GUI is used to show how a CM_JobQueue could be implemented to animate a GUI.

Quick Start

This section includes a number of common commands for each module. These commands (and more) are demonstrated in the test scenes (JobTestScene, JobManagerTestScene, and JobQueueTestScene).

CM_Job

```
var job = CM_Job.Make (CoroutineMethod());
```

Where "CoroutineMethod" is the job you would like to store. This returns a reference to the job so that it can later be paused/resumed/killed etc.

```
Job.Start ();
Job.Start (2f);
```

The first starts the job immediately whereas the second starts the referenced job after a 2 second delay.

```
Job.Pause ();
Job.Pause (2f);
```

The first pauses the job immediately whereas the second pauses the referenced job after a 2 second delay.

```
Job.Resume ();
Job.Resume (2f);
```

The first resumes the job immediately whereas the second resumes the referenced job after a 2 second delay.

```
job.AddChild (CoroutineMethod ());
```

Where “CoroutineMethod” is the child job you would like to store. The main job will not be completed until all child jobs have finished processing. Using this system, you can add dependant child jobs to the main job.

```
var newJob = job.Clone ();
CM_Job[] newJobs = job.Clone (5);
```

The first creates a copy of a job and the second creates five copies of the specified job.

```
job.Repeat (5);
```

This sets the job to be repeated five times.

```
job.Repeat ().StopRepeat(5f);
```

This sets the job to repeat until five seconds have passed and then it is terminated.

```
job.NotifyOnJobStarted (EventMethod).NotifyOnJobComplete (EventMethod2);
```

Where “EventMethod” is the method you want to be invoked when the job starts processing and “EventMethod2” is the method you want to be called when the job has finished processing. This subscribes to those two events.

```
CM_Job.Make (CoroutineMethod()).Start ().Pause (1f).Resume (3f);
```

Where “CoroutineMethod” is the coroutine you want to start. It is also possible to chain methods as shown above. This sets the coroutine to be paused after one second then resumed after three seconds.

CM_JobManager

```
Var testJob = CM_Job.Make (CoroutineMethod (), “coroutine_id”);
CM_JobManager.Global.AddJob (testJob);
```

Where “CoroutineMethod” is the coroutine you would like to add to the the job manager and “coroutine_id” is the id you will use to reference the coroutine from any of your scripts. The above two lines creates a new job and adds it to the job manager.

```
CM_JobManager.Global.StartCoroutine ("coroutine_id");
```

This starts the coroutine with the name “coroutine_id” that was previously added to the job manager.

```
CM_JobManager.Global.PauseCoroutine ("coroutine_id");
```

This pauses the coroutine with the name “coroutine_id” that was previously added to the job manager.

```
CM_JobManager.Global.ResumeCoroutine ("coroutine_id");
```

This resumes the coroutine with the name “coroutine_id” that was previously added to the job manager.

```
CM_JobManager.Global.StopCoroutine ("coroutine_id");
```

This stops/kills the coroutine with the name “coroutine_id” that was previously added to the job manager.

```
CM_JobManager.Global.StartAll ();
CM_JobManager.Global.StartAll (1f);
```

The first line starts all coroutines associated with the global job manager and the second line does the same but after a one second delay.

```
CM_JobManager.Global.PauseAll ();
CM_JobManager.Global.PauseAll (1f);
```

The first line pauses all coroutines associated with the global job manager and the second line does the same but after a one second delay.

```
CM_JobManager.Global.ResumeAll ()
CM_JobManager.Global.ResumeAll (1f)
```

The first line resumes all coroutines associated with the global job manager and the second line does the same but after a one second delay.

```
CM_JobManager.Global
	.NotifyOnJobAdded ((object sender, CM_JobManagerJobEditedEventArgs e) => {
	   Debug.Log ("Job added to global manager: " + e.jobEdited.id);
	}).NotifyOnJobRemoved ((object sender, CM_JobManagerJobEditedEventArgs e) => {
	   Debug.Log ("Job removed from the global manager: " + e.jobEdited.id);
	}).NotifyOnAllJobsKilled ((object sender, CM_JobManagerEventArgs e) => {
	   Debug.Log ("All jobs killed");
	}).NotifyOnAllJobsResumed ((object sender, CM_JobManagerEventArgs e) => {
	   Debug.Log ("All jobs resumed");
	}).NotifyOnAllJobsPaused ((object sender, CM_JobManagerEventArgs e) => {
	   Debug.Log ("All jobs paused");
	}).NotifyOnAllJobsCleared ((object sender, CM_JobManagerEventArgs e) => {
	   Debug.Log ("All jobs cleared");
	});
```

This subscribes to all events offered by the global job manager.

```
var localJobManager = CM_JobManager.Make ();
```

You can also make a local job manager as shown above. You can perform all of the previously outlined activities with this local copy of the job manager.

CM_JobQueue

There are a number of ways to add jobs to a queue, you can add them individually or as a list/array as shown below.

```
var jobsToQueue = new List<CM_Job> () {
			CM_Job.Make (CoroutineMethod1 ()), "job_1"),
			CM_Job.Make (CoroutineMethod2 ()), "job_2"),
			CM_Job.Make (CoroutineMethod3 ()), "job_3")};

CM_JobQueue.Global.Enqueue (jobsToQueue).Start ();
```

The first line creates a list of jobs to add to a queue (where “CoroutineMethod1”, “CoroutineMethod2”, and “CoroutineMethod3” are the names of the coroutine methods you would like to add and “job_1”, “job_2”, and “job_3” are the names of the jobs). The list is then added to the global queue on the second line and the queue is started.

```
CM_JobQueue.Global.Start ();
CM_JobQueue.Global.Start (1f);
```

The first line starts the global queue and the second line does the same but after a one second delay.

```
CM_JobQueue.Global.Pause();
CM_JobQueue.Global.Pause (1f);
```

The first line pauses the global queue and the second line does the same but after a one second delay.

```
CM_JobQueue.Global.Resume();
CM_JobQueue.Global.Resume (1f);
```

The first line resumes the global queue and the second line does the same but after a one second delay.

```
CM_JobQueue.Global.KillCurrent ();
CM_JobQueue.Global.KillCurrent (1f);
```

The first line kills the currently running job and the second line does the same but after a one second delay.

```
CM_JobQueue.Global.KillAll ();
CM_JobQueue.Global.KillAll (1f);
```

The first line kills all jobs currently queued and the second line does the same but after a one second delay.

```
CM_JobQueue.Global.Repeat ();
CM_JobQueue.Global.Repeat ().StopRepeat (5f);
CM_JobQueue.Global.Repeat (5);
```

The first line sets the queue to repeat indefinitely, the second line sets the queue to repeat for 5 seconds, and finally the last line sets the queue to repeat 5 times.

```
CM_JobQueue.Global.Start (1f).Pause (2f).Resume (3f);
```

As with CM_Job you can also chain the method calls to CM_queue. The above line starts the global queue after 1 second has passed, pauses after 2 seconds, and then resumes the queue after 3 seconds.

```
CM_JobQueue.Global
  .NotifyOnQueueStarted ((object sender, CM_QueueEventArgs e) => {
      Debug.Log ("Global queue started processing. " +
         "Number in queue: " + (e.hasJobsInQueue ? e.queuedJobs.Length.ToString () : "0"));
   })
   .NotifyOnJobProcessed ((object sender, CM_QueueEventArgs e) => {
       if (e.hasCompletedJobs) {
          Debug.Log ("Finished processing global job queue: "
              + e.completedJobs [e.completedJobs.Length - 1].id);
       }
   })
   .NotifyOnQueueComplete ((object sender, CM_QueueEventArgs e) => {
      if (e.hasCompletedJobs) {
          int numOfJobsKilled = e.completedJobs.Where (i => i.jobKilled == true).Count ();

           Debug.Log ("Global queue started processing. " +
              "Number of jobs completed successfully: "
              + (e.completedJobs.Length - numOfJobsKilled)
              + ", number of jobs killed: " + numOfJobsKilled);
      }
    });
```

The above line subscribes to all events offered by CM_Queue.

```
var localQueue = CM_JobQueue.Make ();

localQueue.Enqueue (CoroutineMethod1 ())
            .Enqueue (CoroutineMethod2 ())
            .Enqueue (CoroutineMethod3 ())
            .Start ();
```

You can also create any number of local queues, as shown above. The local queues can have their own jobs and be run separately from the global queue and each other.

```
var clonedQueue = CM_JobQueue.Global.Clone ();
CM_Queue[] clonedQueues = CM_JobQueue.Global.Clone (5);
```

Lastly, you can clone any queue. The first line above clones one copy of the global queue, whereas the second line clones 5 copies.

Events

Many of the events you can subscribe to are self-explanatory, however shown below is an outline of each event for reference.

CM_Job

For examples of how to subscribe to these events please see the ExampleJobTest class.

```
NotifyOnJobFinishedRunning
```

Raised when a job has finished running. This differs from NotifyOnJobComplete in that NotifyOnJobFinishedRunning event will only be raised when the job has finished repeating (i.e. if a job is to be repeated 5 times, NotifyOnJobComplete will be called each time the job has completed, whereas this will only be called once all 5 repeated jobs have finished or been killed).

NotifyOnJobStarted

Raised when the job starts processing.

NotifyOnJobPaused

Raised when the job is paused.

NotifyOnJobResumed

Raised when the job is resumed. This is raised even if the job was not previously paused but resume is called.

NotifyOnJobComplete

Raised when a job has been completed. This is called every time a job finishes even if it will be repeated.

NotifyOnChildJobStarted

Raised when a coroutines child job has begun processing.

NotifyOnChildJobComplete

Raised when a coroutines child job has finished processing.

CM_JobManager

For examples of how to subscribe to these events please see the ExampleJobManagerTest class.

NotifyOnJobAdded

Raised when a job has been added to the job manager.

NotifyOnJobRemoved

Raised when a job has been removed from the job manager.

NotifyOnAllJobsPaused

Raised when all jobs have been paused. This is only raised when all jobs are paused at once rather than individually.

NotifyOnAllJobsResumed

Raised when all jobs have been resumed. This is only raised when all jobs are paused at once rather than individually.

NotifyOnAllJobsKilled

Raised when all jobs have been killed. This is only raised when all jobs are paused at once rather than individually.

NotifyOnAllJobsCleared

Raised when the job managers list of owned jobs is cleared.

CM_JobQueue

For examples of how to subscribe to these events please see the ExampleJobQueueTest class.

NotifyOnQueueStarted

Raised when the job queue is started.

NotifyOnQueueComplete

Raised when the job queue has finished processing.

NotifyOnJobProcessed

Raised each time a job in the queue has complete.

Each event also includes a corresponding RemoveNotify method that removes the event listener.