Easy Invoke API Documentation

Table of Contents

1. Introduction

What is Easy Invoke?

Easy Invoke is a powerful and flexible tool designed to enhance scripting in Unity environments by simplifying the management and execution of coroutines and asynchronous tasks. Aimed at developers looking to streamline complex timing and task sequences in their applications, Easy Invoke provides a robust API that integrates seamlessly with Unity's native coroutine system and extends its capabilities to non-MonoBehaviour scripts.

Key Features of EasyInvoke:

- **Delay Code Execution:** Easily delay execution of parameterless and parameterized methods.
- **Coroutine Handling**: Allows the initiation, management, and stopping of coroutines from scripts that do not inherit from MonoBehaviour, facilitating more modular and reusable code architectures.
- **Task Conversion**: Can run coroutines as tasks for use in multithreaded applications, enhancing the ability to perform asynchronous operations alongside Unity's main thread.
- **Frame Skipping**: Offers methods to delay the execution of actions by skipping frames, providing more control over the timing and sequencing of game logic.
- **Chainable Methods**: Supports the creation of chains of methods and delays, which can be executed in sequence to coordinate complex sets of operations with precise timing.
- **Repeatable Jobs**: Enables the scheduling of repeatable tasks that can execute actions at fixed intervals, which is ideal for tasks such as polling or continuous checks within the game environment.

Easy Invoke is designed to be intuitive and easy to integrate, requiring minimal setup to get started. Whether you are developing complex game mechanics, interactive animations, or any system that requires precise timing control, Easy Invoke aims to provide a straightforward and powerful solution to manage these needs effectively.

1. Introduction (Cont.)

1.1 - How to use Easy Invoke

To set up Easy Invoke all you need to do is add the following using statement to your source file:

```
using Doofah.EasyInvoke;
```

All APIs are offered through the **EasyInvoker** class.

1.2 – Initialization

Initialization of Easy Invoke is completely automatic and requires no action on your part.

2. Delaying Code Execution

This section covers the various methods provided by **EasyInvoker** to delay code execution. Methods in this section return **a RunJob.**

2.1 Delay a Parameterless Method

Syntax:

RunRepeat(float delay, Action method)
RunRepeat(Action method, float delay,)

- *delay*: the time, in seconds, before executing the method
- *method*: actual code to repeat:

```
EasyInvoker.Run(2f, () => ExplodeBomb())
```

Shortened form:

```
EasyInvoker.Run(2f, ExplodeBomb);
```

Specify the delay after the method:

```
EasyInvoker.Run(ExplodeBomb, 2f);
```

2.2 Delay a Method with Parameters

To delay execution of a method that takes parameters use the same syntax as when delaying a parameterless method: *Note: The shortened form can't be used*.

```
EasyInvoker.Run(3f, () => DetonateBombAt(location));
```

Specify the delay after the method:

```
EasyInvoker.Run(() => DetonateBombAt(location), 3f);
```

2.3 Delay In-line Code

To delay a block of in-line code using lambdas or delegates you can use the following:

Using lambdas:

```
EasyInvoker.Run(10f, () => {
     Debug.Log("Launch Rocket");
     LaunchRocket();
});
```

Using delegates:

```
EasyInvoker.Run(10f, delegate {
    Debug.Log("Launch Rocket ");
    LaunchRocket();
});
```

2.4 Execute Code On Job Completion

To execute a callback once the action is complete, utilize the **OnComplete** method on the returned **RunJob.**

```
var job = EasyInvoker.Run(10f, delegate {
    Debug.Log("Secure Launch Perimeter");
    SecurePerimeter();
});
job.OnComplete(() => NotifyCommandCenter("Perimeter
Secured"));
```

3. Repeating Code Execution

This section covers the various methods provided by **EasyInvoker** to schedule and control repeating tasks. Methods in this section return **a RepeatJob.**

3.1 Run Repeat

Executes a method or code block repeatedly at fixed intervals, with or without an initial delay.

3.1.1 Without Initial Delay

Syntax: *RunRepeat(float repeatRate, int repeats, Action method)*

- *repeatRate*: the time, in seconds, between each repeat
- *repeats*: the number of repeats (either a fixed number or infinite times)
- *method*: actual code to repeat

This executes EmitStars () every 0.5 seconds, repeating 10 times.

```
EasyInvoker.RunRepeat(0.5f, 10, () => EmitStars());
```

3.1.2 With Initial Delay

Syntax: *RunRepeat(float delay, float repeatRate, int repeats, Action method)*

- *delay*: the time, in seconds, starting the first repeat
- *repeatRate*: the time, in seconds, between each repeat
- *repeats*: the number of repeats (either a fixed number or infinite times)
- *method*: actual code to repeat:

This waits for 1.0 second, then starts executing EmitStars () every 0.5 seconds, repeating 10 times

```
EasyInvoker.RunRepeat(1.0f, 0.5f, 10, () => EmitStars ());
```

3. Repeating Code Execution (Cont.)

3.2 Run While

Executes a method repeatedly as long as a condition remains true, with options for specifying repeat rates and initial delays.

3.2.1 Without Initial Delay

Syntax: *RunWhile(Func<bool> condition, float repeatRate, Action method)*

- *condition*: the condition to evaluate
- *repeatRate*: the number of repeats (either a fixed number or infinite times)
- *method*: actual code to repeat

Continuously monitors the player's health every second as long as the player is active.

```
EasyInvoker.RunWhile(() => player.isActive, 1.0f, delegate {
    MonitorHealth();
});
```

3.2.2 With Initial Delay

Syntax: *RunWhile(Func<bool> condition, float delay, float repeatRate, Action method)*

- *condition*: the condition to evaluate
- *delay*: the time, in seconds, before starting the first repeat
- *repeatRate*: the number of repeats (either a fixed number or infinite times)
- *method*: actual code to repeat

Waits for 2 seconds before beginning to monitor the player's health every second, continuing as long as the player is active.

```
EasyInvoker.RunWhile(() => player.isActive, 2.0f, 1.0f,
delegate {
    MonitorHealth();
});
```

3. Repeating Code Execution (Cont.)

3.3 Run Until

Executes a method at regular intervals until a specified condition is met, with options for specifying repeat rates and initial delays.

3.3.1 Without Initial Delay

Syntax: *RunUntil(Func<bool> condition, float repeatRate, Action method)*

- *condition*: the condition to evaluate
- *repeatRate*: the number of repeats (either a fixed number or infinite times)
- *method*: actual code to repeat

Gets the player's position every 0.25 seconds and sets the mob destination to it, until the player is caught.

```
EasyInvoker.RunUntil(() => player.isCaught, 0.25f, delegate {
    var playerPos = GetPosition(player);
    mob.SetDestination(playerPos);
});
```

3.3.2 With Initial Delay

Syntax: *RunUntil(Func<bool> condition, float delay, Single repeatRate, Action method)*

- *condition*: the condition to evaluate
- *delay*: the time, in seconds, before starting the first repeat
- *repeatRate*: the number of repeats (either a fixed number or infinite times)
- *method*: actual code to repeat

After an initial 2 second delay, starts getting the player's position every 0.25 seconds and sets the mob destination to it, until the player is caught.

```
EasyInvoker.RunUntil(() => player.isCaught, 2f, 0.25f,
delegate {
    var playerPos = GetPosition(player);
    mob.SetDestination(playerPos);
});
```

3. Repeating Code Execution (Cont.)

3.4 Run For

Executes a method at regular intervals for a specified duration.

3.4.1 Without Initial Delay

Syntax: *RunFor(float duration, float repeatRate, Action method)*

- *duration*: the duration this will run for
- *repeatRate*: the number of repeats (either a fixed number or infinite times)
- *method*: actual code to repeat

Updates the game state every second for 30 seconds.

```
EasyInvoker.RunFor(30.0f, 1.0f, () => UpdateGameState());
```

3.4.2 With Initial Delay

Syntax: *RunFor(float duration, float delay, float repeatRate, Action method)*

- *duration*: the duration this will run for
- *repeatRate*: the number of repeats (either a fixed number or infinite times)
- *method*: actual code to repeat

Waits for 2 seconds, then updates the game state every second for 30 seconds.

```
EasyInvoker.RunFor(30.0f, 2.0f, 1.0f, delegate {
    UpdateGameState();
});
```

Repeats:
The value for repeats can be any positive number to repeat for a specified number of times.

If repeats is set to **EasyInvoke.Infinity** or -1, the job will repeat until cancelled.

4. RunJob and RepeatJob

The **RunJob** and **RepeatJob** classes in EasyInvoke are designed to manage the lifecycle of asynchronous tasks, providing control over task completion, cancellation, and monitoring. These classes enable precise control over code execution that is scheduled using various **EasyInvoker** methods.

4.1 Managing Job Lifecycle

RunJob and RepeatJob share several methods that allow developers to interact with running tasks more effectively.

SetCancellationTokenSource:
Associates a cancellation token with the job, allowing for controlled cancellation.

Syntax: *SetCancellationTokenSource(EasyInvokeCancellationTokenSource cancellationTokenSource)*
- *cancellationTokenSource*: the *EasyInvokeCancellationTokenSource*

```
var job = EasyInvoker.Run(5f, () => LaunchMissiles());
var cts = new EasyInvokeCancellationTokenSource();
job.SetCancellationTokenSource(cts);
```

Note: The CancellationTokenSource is automatically handled by default, and the option to manually assign it is primarily to allow multiple jobs to share the same CancellationToken, enabling them to all be cancelled at the same time.

OnComplete:
Registers a callback to be executed when the job completes.

Syntax: *OnComplete(Action callback)*
- *callback*: the callback code to execute

```
var job = EasyInvoker.Run(2f, () => DisplayVictoryMessage());
job.OnComplete(() => CelebrateEndOfGame());
```

4. RunJob and RepeatJob (Cont.)

4.2 Job Cancellation and Termination

Jobs can be cancelled or terminated based on conditions or timeouts, ensuring that resources are managed efficiently, and unwanted executions are prevented.

CancelAfter:
Schedules the cancellation of the job after a specified delay. This can optionally invoke a cancellation callback if the job has not yet completed.

Syntax:

CancelAfter(float secondsDelay)
CancelAfter(float secondsDelay, bool invokeOnCancellation)

- *secondsDelay*: the time before cancellation occurs
- *invokeOnCancellation*: if true, the completion callback is invoked when cancellation occurs

Example 1: Cancel and invoke callback

```
var job = EasyInvoker.Run(10f, () => StartCountdown());
job.CancelAfter(5f, true); // Cancels the countdown halfway
through and invokes the completed callback
```

Example 2: Cancel without invoking callback

```
var job = EasyInvoker.Run(10f, () => StartCountdown());
job.CancelAfter(5f, false); // Cancels the countdown halfway
through without invoking the callback
```

Kill:
Immediately terminates the job. This can also optionally invoke a cancellation callback.

Example 1: Kill Job and invoke callback

```
var job = EasyInvoker.RunRepeat(1f, 10, () =>
UpdateSensorData());
job.Kill(true); // Stops the sensor data updates immediately
and triggers and triggers the completed callback
```

Example 2: Kill Job without invoking callback

```
var job = EasyInvoker.RunRepeat(1f, 10, () =>
UpdateSensorData());
job.Kill(false); // Stops the sensor data updates immediately.
```

4. RunJob and RepeatJob (Cont.)

4.3 Monitoring Job State

The state of a job can be queried to make decisions in your application based on whether the job is still running, has been completed, or was cancelled.

JobState: Each job maintains a state that can be one of several values: initialised, running, complete, killed. This state can be used to help in decision-making processes throughout the job's lifecycle.

```
var job = EasyInvoker.Run(3f, () => ActivateAlarms());
if (job.state == JobState.complete)
{
Debug.Log("Alarms activated successfully.");
}
```

4.4 Usage in Advanced Scenarios

Both **RunJob** and **RepeatJob** are versatile tools for complex scenarios where tasks need to be managed dynamically based on game state or external conditions.

Dynamic Task Management:

Jobs can be adjusted or cancelled based on gameplay events, user actions, or performance considerations.

Example:

```
var monitoringJob = EasyInvoker.RunWhile(() =>
player.isInDangerZone,
    0.5f,
    delegate {
        MonitorHealth(); // Monitor health while in
                         // danger zone.
    }
);

if (!player.isInDangerZone())
{
    monitoringJob.Kill(); //Stop health monitoring
                          // when the player is safe.
}
```

5. Chains

This section covers the methods provided by **EasyInvoker** to create, manipulate, and execute chains of actions and delays. The **EasyInvokeChain** provides a powerful way to coordinate complex sequences of methods with precise timing.

5.1 Create Chain

Starts the creation of a new method chain.

Syntax: *CreateChain()*

Example:

```
var chain = EasyInvoker.CreateChain();
```

Initializes a new chain for adding methods and delays.

5.2 Add Method to Chain

Adds a method to the current method chain.

Syntax: *AddMethod(Action method)*

Example:

```
chain.AddMethod(() => OpenGate());
```

Adds the OpenGate method to the chain.

5.3 Add Delay to Chain

Inserts a delay into the chain before the next method is executed.

Syntax: *AddDelay(float delayInSeconds)*

Example:

```
chain.AddDelay(1.5f);
```

Adds a 1.5 second delay to the chain before the next method.

5. Chains (Cont.)

5.4 Run Chain Immediately

Syntax: *Run()*

Example: Runs the chain immediately without any initial delay.

```
chain = EasyInvoker.CreateChain();
chain.AddMethod(() => MoveRocket(5f));
chain.AddDelay(5f);
chain.AddMethod(() => Rotate(45));
chain.AddMethod(() => MoveRocket(5f));
chain.AddDelay(5f);
chain.Run();
```

5.5 Run Chain with Initial Delay

Syntax: *Run(float delay)*

- *delay*: the time, in seconds, before executing the chain

Example: Waits for 2 seconds before starting the execution of the chain.

```
chain = EasyInvoker.CreateChain();
chain.AddMethod(() => MoveRocket(5f));
chain.AddDelay(5f);
chain.AddMethod(() => Rotate(45));
chain.AddMethod(() => MoveRocket(5f));
chain.AddDelay(5f);
chain.Run(2f);
```

5.6 Completion Callback, CancelAfter and Kill

EasyInvokeChain extends **EasyInvokeJob** and therefore also supports **OnComplete, CancelAfter** and **Kill** as described in section 4. RunJob and RepeatJob

5. Chains (Cont.)

5.7 Repeat Chain Execution

It is also possible to repeat the entire chain of methods and delays a specified number of times or infinitely, with an optional initial delay.

5.7.1 Repeat Chain without Initial Delay

Syntax: *RunRepeat(float repeatDelay, int repeats)*

- *repeatDelay*: the time, in seconds, between each repeat of the chain
- *repeats*: the number of times to repeat the chain

Example: Repeats the entire chain every 0.5 seconds, doing so 3 times.

```
chain = EasyInvoker.CreateChain();
chain.AddMethod(() => MoveRocket(5f));
chain.AddDelay(5f);
chain.AddMethod(() => Rotate(45));
chain.AddMethod(() => MoveRocket(5f));
chain.AddDelay(5f);
chain.RunRepeat(0.5f, 3f);
```

5.7.2 Repeat Chain with Initial Delay

Syntax: *RunRepeat(float initialDelay, float repeatDelay, int repeats)*

- *initialDelay*: the time, in seconds, before starting execution
- *repeatDelay*: the time, in seconds, between each repeat of the chain
- *repeats*: the number of times to repeat the chain

Example: Waits for 1 second before starting the first repetition, then repeats the chain every 0.5 seconds, doing so 3 times.

```
chain = EasyInvoker.CreateChain();
chain.AddMethod(() => MoveRocket(5f));
chain.AddDelay(5f);
chain.AddMethod(() => Rotate(45));
chain.AddMethod(() => MoveRocket(5f));
chain.AddDelay(5f);
chain.RunRepeat(1.0, 0.5f, 3f);
```

6. Additional methods for delaying code execution

6.1 Get Wait Time

Retrieve a cached WaitForSeconds instance, useful for incorporating delays within coroutines.

Syntax: *GetWait(float duration)*

-*duration*: The time in seconds to wait.

Example: To add a 1.5-second wait in a coroutine:

```
yield return EasyInvoker.GetWait(1.5f);
```

6.2 Wait Until Condition

Execute a callback when a specified condition becomes true.

Syntax: *WaitUntil(Func<bool> condition, Action callback)*

-*condition*: A function that evaluates to true or false.

-*callback*: The method to call when the condition evaluates to true.

Example: Perform an action when the system is ready:

```
EasyInvoker.WaitUntil(() => isReady, OnReady);
```

6.3 Wait While Condition

Wait while a condition is true and perform a callback once the condition is false.

Syntax: *WaitWhile(Func<bool> condition, Action callback)*

-*condition*: A function that evaluates to true or false.

-*callback*: The method to call once the condition is false.

Example: Continue operations once loading is complete:

```
EasyInvoker.WaitWhile(() => isLoading, OnLoadingComplete);
```

6. Additional methods for delaying code execution (Cont.)

6.4 Skip a Single Frame

Skips a single frame and then executes a specified method.

Syntax: *SkipFrame(Action method)*

-*method*: The method to execute after skipping a frame.

Example: To refresh the user interface after a frame:

```
EasyInvoker.SkipFrame(() => RefreshUI());
```

6.5 Skip Multiple Frames

Allows skipping a specified number of frames before executing a method, useful for delaying actions without halting other processes.

Syntax: *SkipFrames(int count, Action method)*

-*count*: The number of frames to skip.

-*method*: The method to execute after skipping the specified number of frames.

Example: To update positions after skipping 5 frames:

```
EasyInvoker.SkipFrames(5, () => UpdatePositions());
```

7. Unity Editor Delayed Execution

This section focuses on functionalities exclusive to the Unity Editor .

7.1 EditorRunSingle

Executes a delayed method in the editor. If an existing job is provided, it will be terminated before starting a new one with the specified settings.

Syntax: *EditorRunSingle(ref EditorJob editorJob, int delay, UnityEngine.Object owner, Action method)*

-*editorJob*: Reference to an existing EditorJob. If non-null, the existing job will be terminated.
-*delay*: The time, in seconds, to wait before executing the job.
-*owner*: The Unity object that owns the job. This is optional and can be null.
-*method*: The action to be performed after the delay.

Example:

```
EditorJob compileJob;
EasyInvoker.EditorRunSingle(ref compileJob, 5f, null,
CompileAssets);
```

7.2 EditorRun

Syntax: *EditorRun(int delay, Action method)*
-*delay*: The delay, in seconds, before the job starts.
-*method*: The action to perform.

Example: Schedule an asset cleanup operation with a 3-second delay:

```
EditorJob cleanupJob = EasyInvoker.EditorRun(3,
CleanUpAssets);
```

Syntax: *EditorRun(int delay, UnityEngine.Object owner, Action method)*
-*delay*: The delay, in seconds, before the job starts.
-*owner*: The Unity object that owns the job. If the owner is destroyed while the job is running, the job is terminated.
-*method*: The action to perform.

Example: Initiate a re-import of a specific asset after a 2-second delay:

```
EditorJob reimportJob = EasyInvoker.EditorRun(2, myAsset,
ReimportAsset);
```

8. Coroutines

This section details methods provided by EasyInvoker for managing coroutines. These methods allow for starting, pausing, resuming, stopping, and resetting coroutines.

8.1 Starting Coroutines EasyInvoker provides multiple methods to start coroutines, accommodating various scenarios:

8.1.1 StartCR

Syntax: *StartCR(Func<IEnumerator> routineFunc)*

- *routineFunc*: A delegate that returns an IEnumerator. This delegate encapsulates the method that will be executed as a coroutine.

StartCR(string methodName)
-methodName: The name of the method to run as a coroutine.

StartCR(string methodName, object value)
-methodName: The name of the method to run as a coroutine.
-value: The object to pass to the coroutine method as an argument.

Examples:
Start a coroutine that fades an audio source:

```
StartCR(() => FadeAudio(audioSource, targetVolume, duration));
```

Start a coroutine named "FadeOut":

```
StartCR("FadeOut");
```

Start a coroutine that loads a level with a specified name:

```
StartCR("LoadLevel", levelName);
```

8.1.2 StartCRWithReset

Starts a new coroutine and stops the referenced coroutine if it is already running.

Syntax: *StartCRWithReset(ref Coroutine crRef, IEnumerator routine)*

- *crRef:* A reference to an existing Coroutine instance.
- *routine*: An IEnumerator that defines the coroutine to be executed.

Example: Restart a health regeneration coroutine:

```
StartCRWithReset(ref hpRegenCoroutine, RegenerateHealth());
```

8.2 Managing Coroutines Manage the execution state of coroutines with pause, resume, restart and stop functionalities:

8.2.1 PauseCR

Temporarily halts the execution of a specified coroutine.

Syntax: *PauseCR(Coroutine coroutine)*

- coroutine: The Coroutine instance that is currently active and needs to be paused.

Example: Pause a coroutine managing enemy AI:

```
PauseCR(enemyAICoroutine);
```

8.2.2 ResumeCR Resumes the execution of a paused coroutine.

Syntax: *ResumeCR(Coroutine coroutine)*

- coroutine: The Coroutine instance that is currently active and needs to be paused.

Example: To resume the enemy AI after a player interaction:

```
ResumeCR(enemyAICoroutine);
```

8.2.3 StopCR Stop a running coroutine

Syntax: *StopCR(Coroutine coroutine)*

-coroutine: The Coroutine instance that is currently active and needs to be stopped.

Example: Stop a coroutine that controls environmental effects:

```
StopCR(weatherEffectCoroutine);
```

8. Coroutines (Cont.)

8.2.4 RestartCR Restarts a running or paused coroutine.

Syntax: *RestartCR(Coroutine coroutine)*

-coroutine: The Coroutine instance that is currently active and needs to be restarted.

Example: Restart a coroutine that manages day/night cycles:

```
RestartCR(dayNightCycleCoroutine);
```

8.3 IsCoroutinePaused

Check the execution state of coroutines to make informed decisions within your game logic:

Syntax: IsCoroutinePaused(Coroutine coroutine)

-coroutine: The Coroutine instance to check.

Example: To check if the coroutine managing dialogue sequences is paused:

```
IsCoroutinePaused(dialogueCoroutine);
```

8. Coroutines (Cont.)

8.4 GetWait

Retrieve cached wait objects to use within multiple coroutines, to reduce memory allocations.

Syntax: *GetWait(float duration)*
-duration: Duration of time to wait

Example:

```
yield return GetWait(1.5f);
```

8.4 WaitForFixedUpdate

Gets a static instance of WaitForFixedUpdate, to reduce memory allocations.

Syntax: *WaitForFixedUpdate()*

Example: To perform physics calculations in a fixed update cycle:

```
yield return WaitForFixedUpdate;
```