



Thread-Safe Logger

Documentation

Contents

1	Introduction	2
2	Documentation	3
2.1	Setup	3
2.2	Usage	4
2.2.1	Instantiation	4
2.2.2	Logging	5
2.2.3	Advanced Options	6
2.2.4	Examples	8
2.3	Tests	9
3	Advanced	10
3.1	Thread-Safe IO	10
3.2	Log Buffer	10
3.3	Flushing Buffered Messages On Exit	10
3.4	Logger State	11
4	Support	13

Section 1

Introduction

The Thread-Safe Logger package is a lightweight thread-safe logging utility for writing simple log messages to log files. This logging utility is thread-safe, meaning that many loggers are able to safely write to a single log file.

The following document serves as an offline documentation, briefly describing the necessary setup of the package, as well as how to use it in your Unity projects.

Section 2

Documentation

The following section describes how to import, set up, and use the Thread-Safe Logger package in your Unity project.

For detailed information about the API, consult the API document included in the package, visit the [Project G.A.M.E.D.E.V. documentation website](#), or read the comments in the source files.

2.1 Setup

Firstly, import the package in your project like any other package on the Unity Asset Store – go to **Window\Package Manager**, select **My Assets** as the package source, and import the Thread-Safe Logger package.

Then, add the main component located at **Project GAMEDEV\Tools\Logger\Prefabs** to your starting scene. Once added to an active scene, the object will stay alive across all scenes for the duration of the application. If there are multiple starting scenes, add the main component to all of them. The important thing is that the component is included in the scene that is started first. Without the main component, the Logger will function just fine; however, buffered log messages may not be always written to a log file on application exit (see section 2.2.3 for more information), resulting in lost log messages on application exit.

That's it! You are now ready to use the full logging functionality as shown in the next section.

NOTE

In case you are using assembly definition files for your scripts, add the Thread-Safe Logger assembly located at **Project GAMEDEV\Tools\Logger\Scripts** as an **assembly definition reference** to your assembly files. Do this for all assemblies containing scripts that need to use the `ProjectGamedev.Logging` namespace.

2.2 Usage

2.2.1 Instantiation

There are 2 basic ways to instantiate a `Logger` object:

```
1  Logger logger1 = new Logger("MY_LOGS_DIR", "logfile1");
```

This creates a logger with a log file **logfile1.txt** located in `Application.persistentDataPath\MY_LOGS_DIR`.

i NOTE

`Application.persistentDataPath[1]` is a static property provided by Unity that points to an OS-independent writable directory. On Windows, this points to **C:\Users\<user>\AppData\LocalLow\<company name>**. On other operating systems, the path will vary. Please consult with the Unity documentation cited above for further information.

Notice how only the name of the log file was specified during construction - the `Logger` class automatically appends the appropriate file extension to the log files.

For specifying a custom log directory, use the second constructor provided by the `Logger` class:

```
1  Logger logger2 = new Logger(new DirectoryInfo("path\to\logdir"), "logfile2");
```

The path specified can be both relative and absolute. The resulting log file will be located at **path\to\logdir\logfile2.txt**.

i NOTE

Please keep in mind that by manually creating a log directory, you are responsible for making sure the directory is both available on other operating systems and is writable.

2.2.2 Logging

You can instantiate as little or as many `Logger` objects as you'd like from various scripts. You can have a single logger for everything, or multiple loggers for individual aspects of your game (e.g., one logger object for player-related events, one logger object per enemy, etc.).

Additionally, you may specify multiple loggers to use the same log file – this logger utility is thread-safe and optimized for working with many files and logger objects.

Once you have instantiated a logger object, there are 2 ways to log a message:

```
1     Logger logger1 = new Logger("MY_LOGS_DIR", "logfile1");
2     logger1.Log("PLAYER COLLECTED 30 COINS!");
```

This will timestamp the message using `System.DateTime.Now[2]` and append it to the log file, creating it if it does not exist.

WARNING

By default, log messages are buffered in-memory before being written to log files for better performance. The example above **will not create a log file yet**. See 2.2.3 for more information.

NOTE

By default, existing log files left from previous application runs are not deleted, and new log messages are appended to them. See 2.2.3 for information on how to change this default behavior.

The second way to create a log message is as follows:

```
1     Logger logger1 = new Logger("MY_LOGS_DIR", "logfile1");
2     logger1.Log("PLAYER COLLECTED 30 COINS!", "PLAYER EVENT");
```

This will do the same as before, but will also append a custom title to the log message for better readability.

2.2.3 Advanced Options

Logging to the Unity Debug Console

By default, loggers only write to their specified log files. However, some log messages can also be written to the Unity Debug Console:

```
1  Logger logger1 = new Logger("MY_LOGS_DIR", "logfile1");
2  logger1.Log("PLAYER COLLECTED 30 COINS!", "PLAYER EVENT", logToConsole: false);
3  logger1.Log("Spawning enemy . . .", logToConsole: false);
4  logger1.Log("PLAYER DIED", "PLAYER EVENT", logToConsole: true);
```

The example above logs 3 messages. The first message has a title and is only written to **logfile1.txt**. The second message has no title and is also only written to the log file. The third message, however, is also copied and written to the Unity Debug Console and will be visible in the Unity Editor as well as in the log file.

i NOTE

Log messages written to the Unity Debug Console are not buffered in-memory – they are written to the console immediately.

i NOTE

Be mindful of using the optional `logToConsole` parameter when logging – Unity's `Debug.Log()` method is costly, and can degrade performance in case many loggers are used.

Buffered Messages

For better performance when using many loggers and/or logging many messages, log messages are buffered in-memory by default. Once a logger has buffered enough messages, they are flushed (written to) the respective log file.

You can change this behavior on each individual logger during construction. This allows you to have loggers for important messages that need to be flushed immediately, as well as loggers with less important messages that can be flushed periodically for better performance.

```
1  Logger logger1 = new Logger("MY_LOGS_DIR", "logfile1");
2  Logger logger2 = new Logger("MY_LOGS_DIR", "logfile2", logsBeforeFlush: 1);
3
4  logger1.Log("Spawning enemy . . .");
5  logger2.Log("PLAYER DIED", "IMPORTANT EVENT");
```

The example above creates 2 loggers, one with the default flush value, and another that will flush messages immediately. After the code above has finished executing, only **logfile2.txt** will have been created.

i NOTE

Buffering thresholds can also be changed after the logger has been created, using the public `logger.LogsBeforeFlush` property.

i NOTE

The logger main component contains the script `FlushBufferedMessagesOnExit`. This script is responsible for force-flushing all buffered but not yet written log messages of all loggers on application exit. Furthermore, the `~Logger()` finalizer method in the `Logger` class will also flush all remaining messages, as long as the Garbage Collector runs.

Deleting Leftover Log Files

By default, leftover log files from previous program executions are not deleted. This means that on application restart, if a logger is created again with the same log file, the new messages will be appended to the old log file.

You can change this behavior on each individual logger during construction. This allows you to have loggers for important messages that need to be stored for longer periods, as well as loggers with less important messages that can be deleted after the application has ended.

```
1 // Some previous code has created logfile1.txt and logfile2.txt.
2 // They currently exist in the logger directory and contain messages from
3 // the last time the application has run.
4
5 Logger logger1 = new Logger("MY_LOGS_DIR", "logfile1", logsBeforeFlush: 1);
6 Logger logger2 = new Logger("MY_LOGS_DIR", "logfile2", deletePreviousLogs: true);
7
8 logger1.Log("Spawning enemy . . .");
9 logger2.Log("PLAYER DIED", "IMPORTANT EVENT");
```

In the example above, `logger1` retains the default behavior of not deleting leftover log files, whereas `logger2` will do the opposite.

After the code above has finished executing, `logfile1.txt` will contain its old messages AND the message from `logger1`, and `logfile2.txt` will have been deleted by `logger2`. Once `logger2` has logged enough messages to fill its buffer and flushes to its log file, `logfile2.txt` will be recreated and will contain only the new messages from `logger2`.

i NOTE

Even if `logsBeforeFlush` has not been modified, the logger will still delete the previous log file when calling `Log(deletePreviousLogs: true)`. Said otherwise, the logger will still delete the leftover log file even if no flushing is required yet.

2.2.4 Examples

```
1      import ProjectGamedev.Logging;
2
3      // The most basic usage.
4      Logger logger1 = new Logger("logs", "logfile");
5      logger1.Log("Starting game . . .");
6
7      // Many loggers can write to the same file.
8      Logger enemyLogger1 = new Logger("logs", "enemies", logsBeforeFlush: 3);
9      Logger enemyLogger2 = new Logger("logs", "enemies", logsBeforeFlush: 1);
10     enemyLogger1.Log("Goblin spawned.", "[INFO] [ENEMY]");
11     enemyLogger2.Log("Orc spawned.", "[INFO] [ENEMY]");
12
13     // All possible options.
14     Logger complex = new Logger(
15         new DirectoryInfo("path/to/logdir"),
16         "complex_logfile",
17         deletePreviousLogs: true,
18         logsBeforeFlush: 7
19     );
20     complex.Log(
21         "Player discovered new area",
22         "[INFO]",
23         logToConsole: true
24     );
25
26     // Some methods for inspecting the logger state, if necessary.
27     // Discussed further in the Advanced section below.
28     Logger log = new Logger("dir", "file");
29     string fullAbsPath = log.GetLogDirAbsPath();
30     bool logfileCreated = log.LogFileCreated();
31     DirectoryInfo logdir = log.GetLogDirectory();
32     FileInfo logfile = log.GetLogFile();
```

2.3 Tests

The automated tests used during the development of this tool are included under **Project GAMEDEV\Tools\Logger\Scripts\Tests**.

You can safely remove the entire **Tests** folder if you have no need for the tests. Alternatively, you can run them using the Unity Test Runner package[3], as well as expand them in case you make custom modifications to the Thread-Safe Logger package.

Section 3

Advanced

Below are advanced details on how the Thread-Safe Logger package works. Read this in case you need implementation specifics.

3.1 Thread-Safe IO

The Thread-Safe Logger package uses our Thread-Safe IO[4] package under the hood for concurrent IO access. For detailed information on how the IO thread safety works, please consult the Thread-Safe IO documentation[5] (both included in the package and available on our website).

3.2 Log Buffer

The heart of the Thread-Safe Logger package is **Logger**'s internal **LogBuffer** class. It is responsible for buffering all messages for all logger objects in a thread-safe **ConcurrentDictionary** collection, as well as for flushing messages to their log files in a thread-safe manner.

3.3 Flushing Buffered Messages On Exit

As discussed above, there are 2 components that ensure buffered messages are written to their log files when the application exits.

Firstly, if the main component is attached to the active scene, flushing will be invoked immediately when receiving the **OnApplicationQuit** Unity event. Furthermore, **OnApplicationPause** and **OnApplicationFocus** are also monitored, as iOS, Windows Store Apps, Windows Phone 8.1, and WebGL builds do not always trigger the **OnApplicationQuit** event.

Secondly, the **LoggerBuffer** class implements a custom finalizer (C# jargon for a destructor), which is called by the Garbage Collector right before all class objects are destroyed. When this happens, all buffered messages (if any exist) are flushed to their log files. However, C# finalizers are non-deterministic – it cannot be determined when

the finalizer will be called, only that it eventually will.

3.4 Logger State

As briefly showcased in the example above, the `Logger` class provides several methods for inspecting its state at runtime.

Log Directory Path

```
1     Logger log = new Logger("dir", "file");
2     string fullAbsPath = log.GetLogDirAbsPath();
```

This method returns the full absolute path of the logger's log directory as a string. Thread-safe.

Log File Created

```
1     Logger log = new Logger("dir", "file");
2     bool logfileCreated = log.LogFileCreated();
```

This method returns whether the log file for this logger has been created yet. Equivalent to `File.Exists(GetLogFile().FullName)`, but is thread-safe.

Log Directory Info

```
1     Logger log = new Logger("dir", "file");
2     DirectoryInfo logdir = log.GetLogDirectory();
```

This method returns a `DirectoryInfo` object that contains various information on the logger's log directory. Thread-safe.

WARNING

Although the method above is thread-safe, the `DirectoryInfo` object returned has methods that are not. Consult with the official C# documentation to see which methods of the `DirectoryInfo` class are thread-safe and take precautions when using those that are not.

Log File Info

```
1     Logger log = new Logger("dir", "file");
2     FileInfo logfile = log.GetLogFile();
```

This method returns a `FileInfo` object that contains various information on the logger's log file. Thread-safe.

⚠ WARNING

Although the method above is thread-safe, the `FileInfo` object returned has methods that are not. Consult with the official C# documentation to see which methods of the `FileInfo` class are thread-safe and take precautions when using those that are not.

Section 4

Support

In case you experience any problems with the tool, cannot manage to set it up properly, or want to simply propose an improvement, do not hesitate to contact us at contact@project-gamedev.com, or fill out the form at project-gamedev.com/contact.

Best of luck with your game development journey!

Bibliography

- [1] Unity Script Reference. `Application.persistentDataPath`. <https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html>
- [2] Microsoft Learn. `DateTime.Now` Property. <https://learn.microsoft.com/en-us/dotnet/api/system.datetime.now?view=netstandard-2.1>
- [3] Unity Script Reference. Unit Testing. <https://docs.unity3d.com/2021.3/Documentation/Manual/testing-editor-test-runner.html>
- [4] Project G.A.M.E.D.E.V. Thread-Safe IO.
<https://www.project-gamedev.com/concurrent-io.html>
- [5] Project G.A.M.E.D.E.V. Documentation. Thread-Safe IO. API & Documentation
<https://docs.project-gamedev.com/concurrent-io/api.html>