Follow your code path

Thank you for using FlowShow!

This is a complete guide of FlowShow usage, features, options and inner workings.
If you are in a hurry, start with the section **2.Quick Instructions** below for a targeted hands on.

Index

1.About FlowShow.................... 2

2.Quick instructions for basic usage.................... 3

3.The main FlowShow window.................... 4
- 3-1.Configuration.................... 5
- 3-2.Log Analysis.................... 9
- 3-3.Source code reports.................... 10

4.The log viewer window.................... 11
- 4-1.The log lines list.................... 12
- 4-2.Log viewer keyboard / mouse shortcuts.................... 13
- 4-3.Log viewer load/save/export.................... 14
- 4-4.Log viewer navigation options.................... 15
- 4-5.Log viewer search options.................... 15
- 4-6.Log viewer display options.................... 15

5.The log summary window.................... 16
- 5-1.The summary grid columns.................... 17
- 5-2.Top panel filters controls.................... 18
- 5-3.Bottom panel buttons.................... 18
- 5-4.Log summary with call graph structure window.................... 18

6.Technical details.................... 19
- 6-1.FlowShow source code.................... 19
- 6-2.FlowShow configurations/settings file.................... 21
- 6-3.Collecting source information from .net assemblies.................... 21
- 6-4.The IL Injection process.................... 21
- 6-5.Runtime log writing and the log stream.................... 24
- 6-6.Log stream reading.................... 25

7.Troubleshooting.................... 26
- 7-1.IL Injection / Installation errors.................... 26
- 7-2.Logging performance.................... 26
- 7-3.Where is .ToString() executions?.................... 26
- 7-4.Errors on Delete/Reset log.................... 27

8.Contact / support.................... 28

9.FlowShow History / Closing words.................... 29

1.About FlowShow

FlowShow is an '*extreme*' tracing tool, built to empower users with a better understanding of their projects code flow.

From frame zero, FlowShow captures and log every single method execution.
You can follow it in real time, directly inside the Unity Editor, on a **color highlighted viewer**.

FlowShow brings to light all the methods call chain, what called what, with **what parameters, return value, owner object, execution count, execution time** and more.

This level of detail can be helpful in hunting for bugs, finding performance bottlenecks, measuring code coverage for tests, gaining knowledge on external/legacy code and so on.

You choose exactly what methods to watch and how much to log.
Co-routines, threads, async/await, static, constructors, properties getter/setter, are all fully supported.

Getting too much log? You can **find and disable the logging of repetitive code loops** or **ignore methods by execution count**. Start/stop logging from script code, to **capture exact game portions**.

FlowShow also displays the summarized timings of methods execution, with local and total times - effectively, a profiler specific for tracked methods.
The log stream and the time track can be **reset at anytime**, even while game is playing inside Unity Editor, allowing the track of specific parts.

The log can be saved, loaded and exported as text file. Add exported text logs to git and be able to spot even tiny differences in the call chain (parameter, timings, etc.) introduced by new project versions.

FlowShow uses its own log engine, **built for speed**, persisted in binary file stream.
When extra logging performance is needed, **FlowShow can switch to full in-memory log** (very fast but space limited).

You can insert custom log calling FlowShow.Log(). It's also possible to redirect Unity Engine Debug.Log to FlowShow.

FlowShow needs **zero code changes to** work: no attributes to add, no classes to inherit.
All required instrumentation is injected directly inside .net DLL files via IL weaving. You can remove it all with one click command.

FlowShow is primarily designed to work while game is being played inside the Unity Editor. When building to a platform (via Unity Build Settings window) it's automatically uninstalled.
FlowShow can also be used for **editor scripts, windows and inspectors code**.

FlowShow uses Mono.Cecil library under MIT/X11 license; see Third-Party Notices.txt file in package for details.

2.Quick instructions for basic usage

Open FlowShow window with the Unity Editor menu **[Tools] -> [FlowShow]**.

Press [Select methods to install FlowShow], select the methods that you want to watch and press [Ok] button.

You can then press **[Play]** in Unity Editor. Every marked method execution will be logged, you can check it in the Log Viewer with [View Log] button.

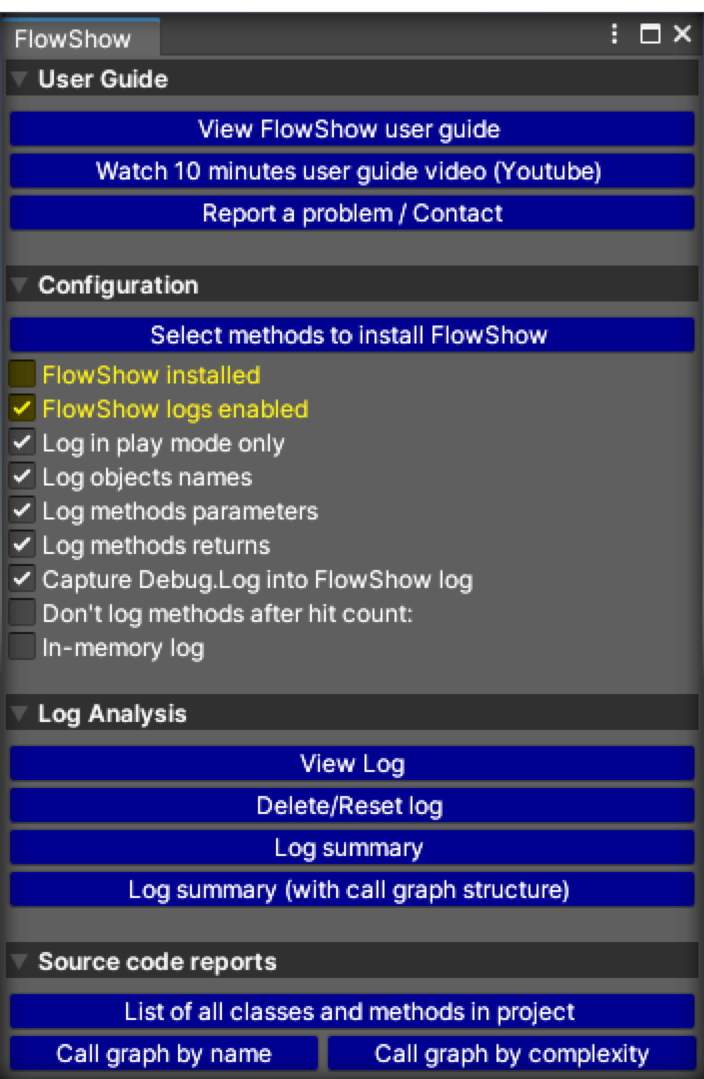
You can also get a consolidated timing view using the [Log Summary] button.

[Delete/Reset log] will both delete the log file and reset methods execution count and timings. You can reset log even while game is running inside the editor.

After finished, you can clear all FlowShow instrumentation unchecking [x] FlowShow installed check box.

3.The main FlowShow window

The main FlowShow window is accessed with the Unity Editor menu **[Tools]** -> **[FlowShow]**.



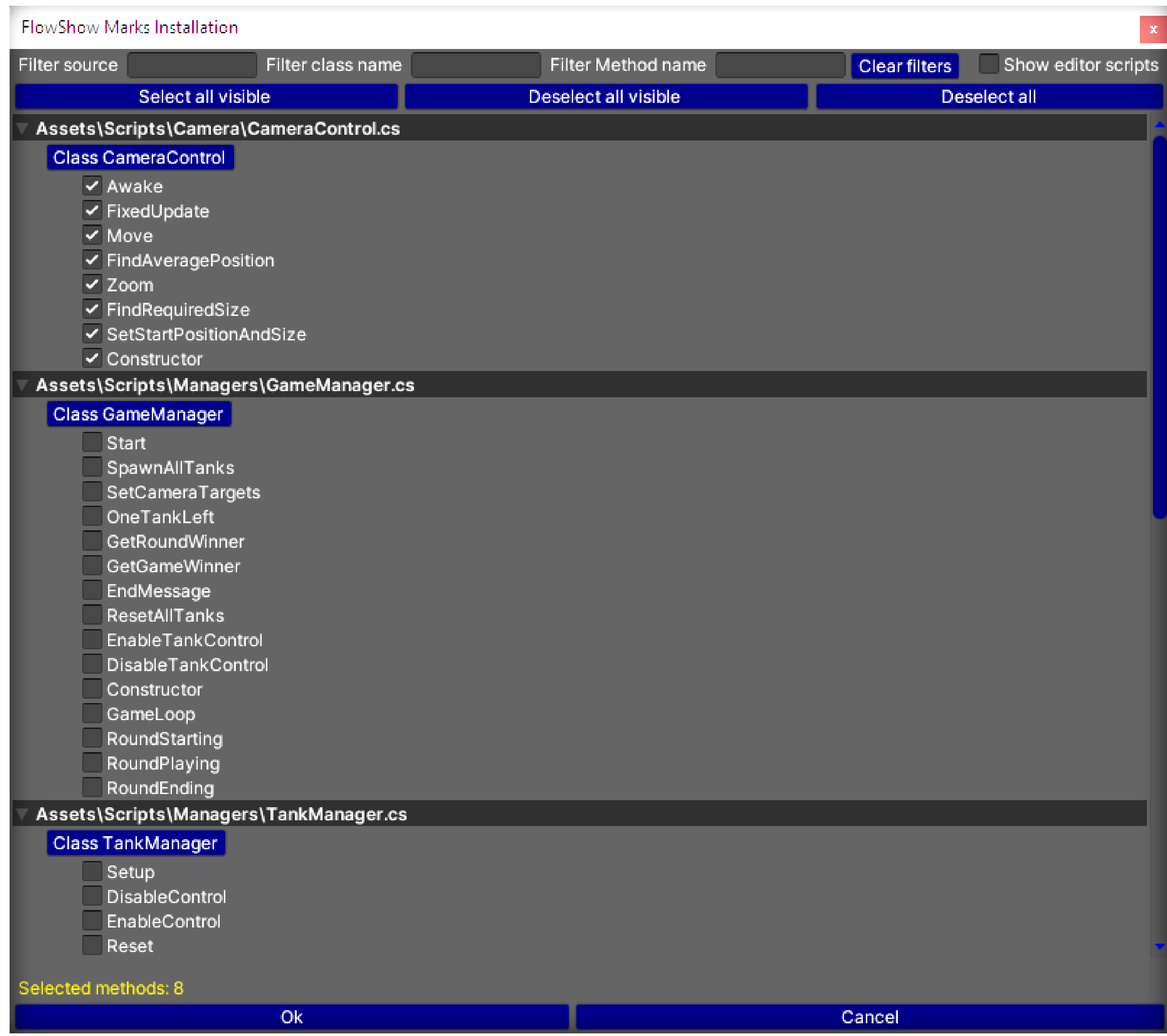
From this window, you can configure and execute the main functions of FlowShow.
The next topics in this chapter will discuss all sections in this window.

3-1.Configuration

FlowShow tracking/logging can be configured in a number of ways:

[Select methods to install FlowShow]

Clicking in this button will open the "FlowShow Marks Installation" window.



This window displays every method present in the project source code, grouped by source file and class.
Click on a source code header to open it in the IDE.
You can also click on a [class button] to toggle the selected state of all methods under it.
You can select/deselect individual methods by clicking in its checkbox [x].
Each select method will be marked to have its execution intercepted by logging, upon the FlowShow installation.
The "Filter source", "Filter class name" and "Filter method name" can be used to limit the displayed methods by some search criteria.
You can also check/uncheck "Show editor scripts" to show/hide methods of editor scripts.

[Select all visible] and [Deselect all visible] buttons will select/deselect all current displayed methods on the window.

[Deselect all] will clear the selection of every single method, currently visible on the window or not.

Please note that it is possible to have methods that are selected AND not displayed, for example, if filtering is active.

You can check the current count of selected methods on the message above the [Ok] button.
When finished selecting the desired methods, click [Ok] to start the FlowShow installation process.
If no method is selected, FlowShow will proceed to clear all its instrumentation.
[Cancel] will just close the window without any further change.

[x] FlowShow installed

Controls the FlowShow instrumentation installation.
When turned on, FlowShow will inject .net assembly code into project scripts DLL files, intercepting all (previously) selected methods execution start and exit.
While turned on, future compilations will also trigger a new FlowShow (re)installation.

When turned off, every FlowShow instrumentation will be removed from the DLL files.

Please note that direct calls to FlowShow.Log() and logs redirected from Debug.Log will be logged, even when FlowShow is not installed.
To prevent all logging, please use the global switch described below (FlowShow logs enabled).

This configuration can't be changed by script code or while game is running.

[x] FlowShow logs enabled

This is the global switch of the FlowShow logging.
When turned off, FlowShow will ignore all marked methods execution and all custom script logs (calls to FlowShow.Log() and redirected from Debug.Log).

It can be set by script:

```
FlowShow.Enabled = true/false;
```

This allows to control FlowShow to only log specific game moments.

[x] Log in play mode only

Set FlowShow to disable logging outside play mode.

It can be set by script:

```
FlowShow.LogPlaymodeOnly = true/false;
```

[x] Log objects names

Instruct FlowShow to include the object (method owner) description on log.
The object description is collected using object.ToString(), with length limited to 100.

It can be set by script:

```
FlowShow.LogObjectsName = true/false;
```

[x] Log methods parameters

Configure FlowShow to include method parameters name and value (if any) in logs.
Parameters values are collected using parameter.ToString(), with length limited to 100.
IList parameters are unpacked as [item, item, ...], limited to 10 first items.

It can be set by script:
```
FlowShow.LogMethodsParameters = true/false;
```

You can also write a custom parameter to text conversion using the event FlowShow.CustomParameterToText.
It will be invoked every time FlowShow needs to write a parameter in the log stream.
The signatures is:

```
delegate bool CustomParameterToTextDelegate(string methodName, object owner,
      string paramName, object paramValue, StringBuilder paramValueAsTxt);
```

When this event is filled, it is called on every parameter of every method call intercepted by FlowShow.
Idea is to provide a way to customize parameter to text conversion, before writing to log.

The event will receive the method name, object owner (if any), parameter name, the parameter value and a StringBuilder object.
Return `true` to sign that the content on StringBuilder object paramValueAsTxt must be use.
Return `false` to sign that the default parameter to text conversion must be used.

Please ensure that this method is minimal and highly optimized, as it will be executed for every parameter of every marked method execution.
For example:

```
bool CustomParameterToTextDelegate(string methodName, object owner,
      string paramName, object paramValue, StringBuilder paramValueAsTxt)
{
      // Quickly check if parameter type is Player class, return name
      if (paramValue is Player)
      {
            paramValueAsTxt.Append((paramValue as Player).Name);
            return true;
      }

      // Use default FlowShow parameter to text conversion
      return false;
}
```

[x] Log methods returns

Configure FlowShow to include method return value (if any) in logs.
The logged text will be the return?.ToString(), with length limited to 100.

It can be set by script:
```
FlowShow.LogMethodsReturn = true/false;
```

[x] Capture Debug.Log into FlowShow log

Replicate Unity Engine's Debug.Log, Debug.LogWarning, Debug.LogError, Debug.LogException into FlowShow log, while game is in Play mode. Calls to Debug.LogX in edit mode will be ignored.

It can be set by script:
```
FlowShow.CaptureDebugLog = true/false;
```

[x] Don't log methods after hit count

Some methods can reach thousands of executions very fast, turning into noise inside execution log.
This option allows to specify a maximum execution count filter, ignoring the log of executions beyond this mark.

It can be set by script:

```
FlowShow.MethodHitCountLimit = x;
```

0 (default) will turn it off.

[x] In-memory log

Enable/Disable the use of the in-memory log stream (instead of the default binary file).
The in-memory log can heavily increase the logging performance, mainly on machines without SSD disk.
The downside is that the log stream will be space limited.

Once filled this space, FlowShow will stop all logging (but will continue to mark execution count and timing).
However, it is always possible to reset the log stream, freeing it for new log data, even while game is running.
Please refer to section Delete/Reset log below.

To enable it, you have to choose a size in megabytes for the buffer (from 16 MB to 1024 MB (1 GB)) and click the [Apply] button.
When enabled, FlowShow will notify the current size and occupation of the in-memory log stream.
To disabled, uncheck it and click [Apply] button.

This configuration can't be changed by script code or while game is running.

3-2.Log Analysis

FlowShow offer some views and tools to work with collected log stream:

[View Log]

Opens the Log Viewer window.
In this window, the log stream will be displayed in a color highlighted list, along with many navigation, searching and log management options.
Please refer to the "Log Viewer window" section below for a detailed explanation.

[Delete/Reset log]

This command resets all the log data and method timings.
It can be safely used while game is running (even being a good strategy to focus log on desired game functionalities).

If a file is being used as log target, it will be deleted.
If the in-memory log is activated, the logs buffer will be cleared and the current occupation returned to near 0%.

All methods call count and execution timings will also be reset.

[Log summary] and [Log summary (with CallGraph structure)]

Opens the Log Summary window.
This window summarizes, in a grid view, the methods execution timings captured by FlowShow.
Please refer to the "Log summary window" section below for a detailed explanation.

3-3.Source code reports

Flow show can give some useful reports (text files) from the project source code structure.
They are captured from the analysis of compiled code on project DLL assemblies (instead of the actual runtime execution).
It can be useful to add these reports to version control and track what differences each project version causes.

[List of all classes and methods in project]

This is a basic list of all project source files, classes and its methods.

[Call graph by name] and [Call graph by complexity]

The call graph is a structure describing all the the possible methods call chain, split by entry points, captured from the compiled code on project DLL assemblies.
For example:

```
MethodA()
MethodB()
    MethodC()
        MethodD()

MethodX()
    MethodY()
```

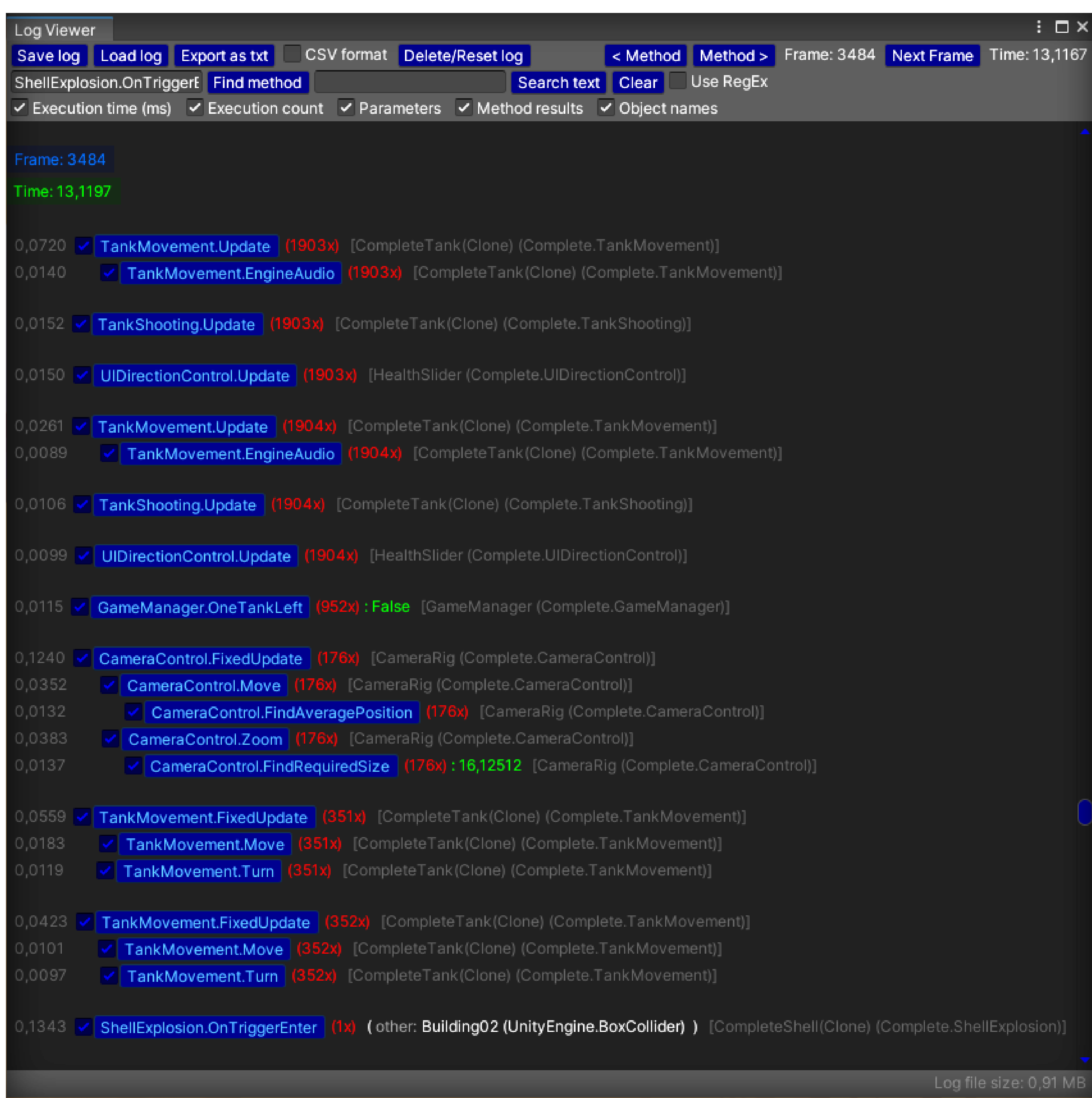
As methods A and X are never directly called from inside the source code, they became the split factor and "islands" entry point.
The call graph is the collection of all these call chain islands.

By itself, as a report, it is a useful way to understand code flow.
It's also used as an alternative structure of logs visualization inside FlowShow (Log summary with Call graph window).

You can get the call graph report ordered by name (name of the entry method) or by complexity.
The complexity is the length of each island inside the call graph.
In the above example, MethodA island has a complexity of **4** while MethodX island, **2**.

4. The log viewer window

This window displays the log stream in a color highlighted list with many navigation, searching and log management options.



The next topics in this chapter will discuss all sections in this window.

4-1.The log lines list

Method are listed in execution order and spaced according to their call hierarchy.
For example:

```
MethodA()
        MethodB()
                MethodC()
        MethodD()
```

Method A called method B, that called method C. After return of B, A called D.

You can navigate through the log stream using the scroll bar or using keyboard shortcuts. Please check the next section for a detailed list.

Enumerator methods (Unity Engine coroutines / C# async) or methods executed outside Unity Engine main thread, won't allocate space in hierarchy, using instead the current level.
For example:

```
MethodA()
        OtherThreadMethod() <- Start
        MethodB()
                MethodC()
        MethodD()

MethodA()
        MethodB()
                MethodC()
                OtherThreadMethod() <- Exit
        MethodD()
```

The reason is that their execution can expand beyond the call stack (as the calling method will continue its execution without awaiting the return).
Also, different from "normal" methods execution, this type of method will have its execution end displayed on the log.

Each method execution is displayed as a button in the log view. Click it to open its source code in IDE.
A typical log line of a method execution will be:

```
0,1343 [x] ShellExplosion.OnTriggerEnter (1x) ( other: Building02 (UnityEngine.BoxCollider) ) [CompleteShell(Clone) (Complete.ShellExplosion)]
```

- The first number `(0.1343)` is the execution time in milliseconds.
- The checkbox [x] enable/disable further logging of this method execution, useful to filter out methods that are turning into repetitive noise inside the log stream. **Hold SHIFT while pressing this check box** to also enable/disable the entire call chain below this method.
- The method button can be clicked to show method source in IDE.
- The second number **`(1x)`** is the current execution count of this method.
- Paramaters are displayed (if any) as (param: value, ...)
- Finally, the source object description

Aside from methods execution, FlowShow will also capture text log.

They can come from direct calls to FlowShow.Log() or from a redirect Debug.LogX message, when this option is turned on in FlowShow configuration.

Text logs will appear in the current level of the call chain, for example:

```
MethodA()
    MethodB()
        Hello World! <-- A text log message
        MethodC()
    MethodD()
```

So, MethodB logged this before executing method C (therefore, will appear below the call chain of MethodB).

FlowShow also emit automatically few log messages about some Unity Engine events (scene loaded and play mode state changed).

Finally, FlowShow will also record current game Time and Frame number information into the log stream.

At bottom panel, FlowShow will display the current log file size in megabytes.

If in-memory log is being used, FlowShow will display current buffer size in megabytes and current memory occupation %.

4-2.Log viewer keyboard / mouse shortcuts

Please, to be able to use the shortcuts, ensure that the Log Viewer window is currently the one with focus.

`(Up arrow / Down arrow)`
Navigate 1 line up / down in log stream.

`(Page down / Page up)`
Navigate 1 page up / down in log stream.

`(Mouse wheel movement)`
Navigate 1 line up / down.

`(Home)`
Navigate to the start of the log stream.

`(End)`
Navigate to the end of the log stream.

`(Ctrl + Up arrow)`
Navigate to the caller of the method execution displayed in the first line. If at hierarchy level 0, navigate to the previous entry method call.

`(Ctrl + Down down)`
Navigate to the next method execution of the same or lower level of the first displayed line.

`(F)`
Jump to the start of the next frame on the log stream.

`Click on a method button`
Open its source in IDE.

`Click on a method checkbox`
Enable/Disable further logging of this method execution.

`Shift + Click on the method checkbox`
Enable/Disable all methods below it in the call chain.

4-3.Log viewer load/save/export

[Save Log]

Saves the current log stream to a binary file.
It also saves the current methods execution count and execution timings.

[Load Log]

Load a .flowshow log file in the Log Viewer.
The file name will be displayed at the bottom panel.

FlowShow also loads methods execution count and execution timings.
If there is already current method execution present in FlowShow methods database, a dialog box will appear asking if it should be replaced.

[Close log file]

Only visible when a log file is currently loaded.
This will return the Log Viewer to display the FlowShow current log stream.

[Export as txt]

Exports the current log stream to a text file.
Keeping these files history can be helpful to check differences in the methods call chain (timings, parameters, etc.) through the project versions / commits.

If the current log is loaded from a .flowshow log file, exporting will be a one time operation, displaying the current progress on bottom panel.

However, if exported from FlowShow current (live) log stream, it will be an ongoing operation to keep on track with last log additions.
In this case, the exporting file name will be displayed on the bottom panel with 2 additional buttons:

[Stop] will cancel the ongoing log operation where currently is.

[Open] will open the exported text file on the default OS text editor (as read-only). The export operation will continue.

When log is reset, the ongoing operation is automatically stopped.

[x] CSV format

Change the format of the exported log text.

When not selected, the logs format will be the same as displayed in the Log Viewer.
When selected, format will be as CSV (Comma-separated Values), easily imported in spreadsheet software for custom analysis.

4-4.Log viewer navigation options

[< Method]

Navigate to the caller of the method execution displayed in the first line. If at hierarchy level 0, navigate to the previous entry method call.
Also available as shortcut `(Ctrl + Up arrow)`.

[Method >]

Navigate to the next method execution of the same or lower level of the first displayed line.
Also available as shortcut `(Ctrl + Down arrow)`.

[Next Frame]

Jump to the start of the next frame on the log stream.
Also available as shortcut `(F)`.

4-5.Log viewer search options

[Find method]

Find the next method call of the method(s) with same (or similar) inputted name.
This is an indexed search, able to quickly scan the entire log stream.

[Search text]

Search the entire log stream for the inputted text.
This is a slow search process as it is necessary to scan the entire log stream in detail.

[Clear]

Clears the [Find method] and [Search text] text boxes.

[x] Use RegEx

Enable/disable the use of Regular Expression for the [Find method] and [Search text] search process.

4-6.Log viewer display options

[x] Execution time (ms)
[x] Execution count
[x] Parameters
[x] Method result
[x] Object names

Controls what information the Log Viewer will display of the log lines.
This is only a display configuration (won't affect what is logged)

5.The log summary window

This window summarizes, in a grid view, the methods execution timings captured by FlowShow.
Clicking on a column will re-order the grid by this value, click again to inverse the order.

FlowShow Log Summary

Name (like) enabled/disabled Both execution count >= 0 All visible measured time: 00:00:00.3050000

		Name	Count <	Total (s)	Av.Tot (s)	Local (s)	Av.Loc (s)	%
✔	F	TankMovement.Update()	2550	0,12982	0,00005	0,09543	0,00004	31,25%
✔	F	TankMovement.EngineAudio()	2550	0,03438	0,00001	0,03438	0,00001	11,26%
✔	F	TankShooting.Update()	2550	0,05835	0,00002	0,05773	0,00002	18,90%
✔	F	UIDirectionControl.Update()	2550	0,03305	0,00001	0,03305	0,00001	10,82%
✔	F	GameManager.OneTankLeft()	1275	0,01720	0,00001	0,01720	0,00001	5,63%
✔	F	TankMovement.FixedUpdate()	476	0,02869	0,00006	0,01369	0,00003	4,48%
✔	F	TankMovement.Move()	476	0,00835	0,00002	0,00835	0,00002	2,73%
✔	F	TankMovement.Turn()	476	0,00665	0,00001	0,00665	0,00001	2,18%
✔	F	CameraControl.FixedUpdate()	238	0,03807	0,00016	0,01621	0,00007	5,31%
✔	F	CameraControl.Move()	238	0,01066	0,00004	0,00660	0,00003	2,16%
✔	F	CameraControl.FindAveragePosition()	238	0,00406	0,00002	0,00406	0,00002	1,33%
✔	F	CameraControl.Zoom()	238	0,01120	0,00005	0,00735	0,00003	2,41%
✔	F	CameraControl.FindRequiredSize()	238	0,00384	0,00002	0,00384	0,00002	1,26%
✔	F	TankMovement.OnDisable()	2	0,00007	0,00003	0,00007	0,00003	0,02%
✔	F	CameraControl.Constructor()	1	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	GameManager.Constructor()	1	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	ShellExplosion.Start()	1	0,00001	0,00001	0,00001	0,00001	0,00%
✔	F	ShellExplosion.OnTriggerEnter(other)	1	0,00013	0,00013	0,00013	0,00013	0,04%
✔	F	ShellExplosion.Constructor()	1	0,00005	0,00005	0,00005	0,00005	0,01%
✔	F	TankShooting.Fire()	1	0,00062	0,00062	0,00057	0,00057	0,19%
✔	F	CameraControl.Awake()	0	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	CameraControl.SetStartPositionAndSize()	0	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	GameManager.Start()	0	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	GameManager.SpawnAllTanks()	0	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	GameManager.SetCameraTargets()	0	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	GameManager.GetRoundWinner()	0	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	GameManager.GetGameWinner()	0	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	GameManager.EndMessage()	0	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	GameManager.ResetAllTanks()	0	0,00000	0,00000	0,00000	0,00000	0,00%
✔	F	GameManager.EnableTankControl()	0	0,00000	0,00000	0,00000	0,00000	0,00%

Enable all visible | Disable all visible | Reset times

The next topics in this chapter will discuss all sections in this window.

5-1.The summary grid columns

[Name]

Contains the methods description, as:

[x] [F] [Class.MethodName(param, param, ...)]

Each method is displayed as a button, click on it to open its source code in IDE.
When both Log Summary and Log Viewer windows are open in Unity Editor, you can click on the [F] (Find) button to search the Log Stream for the next execution of this method.
Each method also has a checkbox displayed on the left side. It enables/disables further logging of this method.

[Count]

Current execution count of the method.

[Total (s)]

Sum of total execution time of the method, in seconds.

[Av.Tot (s)]

Average of total time (total / execution count).

[Local (s)]

Sum of local execution time of the method, in seconds.
The local time is the total time minus time spent on other methods called inside it.
For example:

```
Clock
0      MethodA();                <- Local time: 1 | Total time: 7
1            MethodB();          <- Local time: 3 | Total time: 6
4                  MethodC();    <- Local time: 2 | Total time: 2
6                  MethodC();    <- Local time: 1 | Total time: 1
```

[Av.Loc (s)]

Average of local time (local / execution count).

[%]

Ratio of local execution time / all visible measured time (sum of all visible methods local time, displayed in the top-right corner of the window).

5-2.Top panel filters controls

You can limit the displayed methods by many filters.
Only visible methods contribute to the All measured visible time (displayed in top-right corner of the window)

Name (like)

Only show methods that contains the inputted text in its description (either in class name, method name or one of the parameters names).

enable/disable (Both / only enabled / only disabled)

Only show methods with the corresponding enabled status, or all when "Both" is selected.

execution count >= [0]

Only show methods with the execution count >= input value, or all when the input is zero.

5-3.Bottom panel buttons

[Enable all visible]

Enable all methods visible on this window by the current filter settings.

[Disable all visible]

Disable all methods visible on this window by the current filter settings.

[Reset times]

Reset all methods execution count and timings (to zero), visible or not.
Useful to track only the timing of specific game parts.

5-4.Log summary with call graph structure window

This the same basic Log Summary window but displayed with the call graph structure. Please refer to the call graph report for the explanation about this structure.

The window starts blank with an activation button. This is necessary because the call graph can take some time to be assembled.

Besides methods, this window also displays the call graph "islands" grouped timings.
Each island is displayed as a button with its entry method name. Clicking this button will enable/disable all methods below this island.
Sorting of this grid will be applied at island level (instead of individual methods).

Checking the grouped execution timing of the islands can help to identify performance expensive code loops hidden in the call chain.

6.Technical details

6-1.FlowShow source code

FlowShow source code is heavily commented and descriptive about every implementation aspect in technical details. This is a quick summary of each script purpose.

Editor scripts:

`Editor\FlowShowAssemblyInstaller.cs`
Handles installation of FlowShow instrumentation in .net DLL assembly files, using Mono.Cecil API

`Editor\FlowShowCallGraph.cs`
Creates and maintain the call graph structure from methods code flow

`Editor\FlowShowGUIHelper.cs`
UI helper methods for dealing with Unity Engine IMGUI

`Editor\FlowShowInstaller.cs`
A bridge between Unity Editor build pipeline and FlowShow IL assembly installation

`Editor\FlowShowLogExportTXT.cs`
Exports the log stream as text data

`Editor\FlowShowLogReader.cs`
Read and transforms raw log stream into indexed log lines

`Editor\FlowShowSelectedMethodsConfig.cs`
Save/Load marked methods for installation status into FlowShow settings file

`Editor\FlowShowSourceFile.cs`
Interaction with the project source at C# file level

`Editor\FlowShowSourceMethod.cs`
Interaction with the project source at method level

`Editor\FlowShowSourceReader.cs`
Coordinates the .net DLL assemblies reading, using the Mono.Cecil API

`Editor\FlowShowSourceType.cs`
Interaction with the project source at type declaration level

`Editor\FlowShowUtils.cs`
General helper methods

Editor window scripts:

`Editor\FlowShowWindow.cs`
The main FlowShow editor window, to enable Users to configure and execute all FlowShow available functionalities

`Editor\FlowShowWindowContact.cs`
Window to assist user on contacting FlowShow support via e-mail

`Editor\FlowShowWindowInstallation.cs`
Configuration window for the users to select methods for FlowShow instrumentation

`Editor\FlowShowWindowLogSummary.cs`
Displays a summary of all methods execution count and execution timings captured by FlowShow

`Editor\FlowShowWindowLogSummaryCallGraph.cs`
Displays a summary of all methods execution count and execution timings captured by FlowShow, grouped by call graph structure

`Editor\FlowShowWindowLogViewer.cs`
Displays the log stream in a color highlighted list with many navigation, searching and log management options

Runtime scripts:

`FlowShow.cs`
Main runtime static class of FlowShow logging, handles the logs capturing

`FlowShowLogWriter.cs`
Handles the writing of FlowShow log data into binary stream

`FlowShowSettings.cs`
Stores FlowShow settings and configurations in local text file

`FlowShowUnityInterface.cs`
Auto initialized GameObject script behavior to help FlowShow interfacing with some Unity Engine functionalities

6-2.FlowShow configurations/settings file

(Related classes: FlowShowSettings, FlowShowSelectedMethodsConfig)

FlowShow uses a custom binary file (FlowShow.config at project root) to read and write its settings.
Reason behind this is that FlowShow can need to read it from a different Thread of the Unity Engine main thread, preventing us on using PlayerPrefs/EditorPrefs classes.

The FlowShow static class maintain its own configuration in static fields, reading from the settings file at the static constructor.

Current selected (marked for installation) methods are also persisted in this config file.

6-3.Collecting source information from .net assemblies

(Related classes: FlowShowSourceReader, FlowShowSourceFile, FlowShowSourceType and FlowShowSourceMethod)

One of the fist FlowShow tasks is to collect information about the project, present in compiled .net IL code, inside DLL assembly files, using Mono.Cecil API.
FlowShow scans all referenced C# script files, declared types and methods.
Inside methods, it scans for IL .net body checking if FlowShow instrumentation is already present.
It also scans for Enumerable methods (Unity Engine co-routine / C# async), static status, parameters, return type, etc.

With all this information, an edit time database is built. This is where each selected (marked for installation) method status is stored in memory.

6-4.The IL Injection process

(Related classes: FlowShowAssemblyInstaller, FlowShowInstaller)

FlowShow captures the marked methods executions by code intercepting.
Using the Mono.Cecil API, it injects a call to FlowShow.Enter() at method start and a call to FlowShow.Exit() at method end.
This installation happens just after Unity Editor finished the compilation of the .net assembly DLL files (when FlowShow is activated and not in a final platform build).
FlowShow IL instrumentation correctly updates the PDB file of debugging symbols, allowing breakpoints to keep working as expected (more Mono.Cecil API magic).

After installation, a .net domain reload is performed.
FlowShow installation process requires Unity Editor in Debug mode (button on the lower right of Unity Editor window).

DLLs

The targeted DLL files are those containing any source file inside the assets project folder.
In most cases it will be Assembly-CSharp.dll and Assembly-CSharp-Editor.dll (if any editor method is marked), but it's not limited to (custom assembly definitions are supported).

DLL can contains reference to other DLLs. Sometimes, Mono.Cecil API asks about the location of a referenced DLL.
FlowShow captures this information from the projects .csproj files, at project root.
Please, ensure that these files are present and updated (you can use Unity Editor menu [Assets] -> [Open C# Project] to generate/refresh it).

Methods database

Beside interception calls, FlowShow also injects the initialization of the runtime methods database (FlowShow.MethodsDB, a MethodEntry[] array).

FlowShow uses this database for runtime information about current installed methods (like name, parameters, return, etc.), beside the storage of execution count and execution timings information.

Re-installation

Flow can install both into a "clean" DLL and over an already instrumented one.

The second case happens when the user changes the selected methods list and re-install FlowShow.

While FlowShow installation is configured to active, any code change (and subsequent recompilation) is handled as a clean DLL installation.

Uninstall

The removal of FlowShow instrumentation is just a request to Unity Editor to recompile all the assemblies.

Enumerable methods

Unity Engine co-routines and C# async methods uses a special kind of machine state class implemented by .net compiler. FlowShow detects it and install its instrumentation over the MoveNext() method of these classes.

Unhandled exceptions

For simplicity reasons, FlowShow does not install a try-catch handler between the Enter() and Exit() calls.

This means that any unhanded exception will prevent the reach to the Exit() call.

FlowShow address this issue in two fronts, please refer to the log stream section for a deeper explanation.

IL Weaving

Given a very simple method example:

```
int Inc(int number)
{
	return number + 1;
}
```

In .net IL code:

```
IL_0000: nop
IL_0001: ldarg.1
IL_0002: ldc.i4.1
IL_0003: add
IL_0004: stloc.0
IL_0005: br.s IL_0007
IL_0007: ldloc.0
IL_0008: ret
```

The objective is to insert the FlowShow.Enter() and FlowShow.Exit() at method start and end.
In C# code it would be something close to it:

```
int Inc(int number)
{
      FlowShow.Enter(this, new object[] { number }, 0);
      int result = number + 1;
      FlowShow.Exit(result, 0);
      return result;
}
```

"0" here is the index that this method is present in FlowShow runtime method database.

IL .net injected code:

```
Load "this" into stack
IL_0000: ldarg.0

Create a new object[] array to load the parameter values into it
This method has only 1 parameter (int number)
IL_0001: ldc.i4.1
IL_0002: newarr [netstandard]System.Object
IL_0007: dup
IL_0008: ldc.i4.0
IL_0009: ldarg.1
IL_000a: box [netstandard]System.Int32
IL_000f: stelem.ref

Loading the method id (0) parameter into stack. With this value FlowShow is able to link using its runtime method database.
IL_0010: ldc.i4.0

With the owner, parameters values and method id set, we cal call FlowShow.Enter()
IL_0011: call void ArcadiumPlayware.FlowShow.FlowShow::Enter(object, object[], int32)

IL_0016: nop
IL_0017: ldarg.1
IL_0018: ldc.i4.1
IL_0019: add
IL_001a: stloc.0
IL_001b: br.s IL_001d
IL_001d: ldloc.0

Here we change the previous last ret (return) instruction with nop, ensuring that any previous instruction jumping to the old ret falls into FlowShow.Exit injection below it.
IL_001e: nop

At method end, if method has a return, it will be loaded into the call stack before method exit. Just duplicate it to load as a parameter to FlowShow.Exit()
IL_001f: dup
IL_0020: box [netstandard]System.Int32
IL_0025: ldc.i4.0
IL_0026: call void ArcadiumPlayware.FlowShow.FlowShow::Exit(object, int32)

IL_002b: ret
```

For the full implementation details of .net IL weaving/inject, please check the FlowShowAssemblyInstaller source code.

6-5.Runtime log writing and the log stream

(Related classes: FlowShow, FlowShowLogWriter)

After instrumentation is completed and game is running inside Unity Editor, methods executions and other custom logs are captured by FlowShow.

This data is written into the log stream, either to a binary file or to the in-memory buffer.

The stream starts with the writing of the FlowShow runtime methods database.

This is necessary to make the log stream portable, as it is not granted that FlowShow will be installed (and installed with exact same methods selected) in future stream readers.

FlowShow writes each log event with a flag identifying the type (custom logs, Unity Engine redirected logs, time information, frame information, method execution start and method execution end).

Other log data are written in binary format - integer uses 7 bit encoding format (using few bytes for smaller numbers), texts are written using ASCII encoding.

FlowShow maintain some internal state to reflect current call chain hierarchy level, increasing at methods start, decreasing at methods end.

For example:

```
Call chain level
0 +1 MethodA()
1 +1   MethodB()
2 +1        MethodC()
3 -1        < C // end of C
2 -1   < B // end of B
1 +1   MethodD()
2 -1   < D // end of D
1 -1 < A // end of A
0
```

Method ends are never displayed in FlowShow views (except for enumerator and threaded executions) but they are present in the log stream.

They inform the end of a method in the call chain and informs about the returned value and total execution time (milliseconds).

As stated earlier, FlowShow IL instrumentation does not install a try-catch handler in each marked method.

Unhandled exceptions inside instrumented methods will break this call stack tracking as there will be no corresponding Exit() for the previous Enter() call

FlowShow address this issue in tho ways:

First case is when the exception rises unhandled up to the Unity Engine.

In this case, Unity Engine will catch it and log as error. FlowShow watch this log and proceed to reset all call stack status information and hierarchy Level to 0

Second and worst case is when some method in call chain swallows the exception in a try...catch block. For example:

```
MethodA()
     MethodB() <- Try...catch() block
          MethodC()
               MethodD() <- Exception raises
```

In this case, methods D and C will never get an Exit() call and no exception will surface.

FlowShow tries to fix it checking in every method Exit() for the registered method in the call stack level position.

If it differs from current method, FlowShow assumes some exception happened above it, resetting all methods above the CallStack.

6-6.Log stream reading

(Related classes: FlowShowLogReader, FlowShowLogExportTXT)

The FlowShowLogReader handles the log stream reading, either from a log file or from the in-memory log buffer.

A first reading pass scans the entire log stream, in multithread execution, indexing its raw content into an indexed log lines list.

Each index entry holds the position of the log line in the stream, along with some basic information about it (log type, hierarchy level, etc.).

Later on, clients can request for a full loading of a limited range of log lines, where the heavy lifiting of loading strings (log text, parameters value, return value, etc.) is done.

This class also support indexed and non-index search of many forms (next frame information, next method execution, general text search, etc.).

7.Troubleshooting

7-1.IL Injection / Installation errors

The .net IL injection process was extensively tested under many different scenarios.
Some types of methods (like constructors) requires specific injection strategies and is possible that some unforeseen situations triggers installation exceptions or results in invalid .net IL code.

FlowShow is programmed to check for any methods that got installation errors, removing those methods from the selected list and re-starting the installation process.
You will be notified about these errors.

Invalid .net IL code scenarios are trickier.
Chances are that .net/Unity Editor detects this invalid code in assembly with a helpful message about the offending method.
As a short term solution, unselect it, and re-install FlowShow.

You can then contact the FlowShow support (section "8.Contact information") informing about this error.
Please, if possible, include the offending method source code and/or the method .net IL body.

In all cases, you can always request total FlowShow instrumentation removal by uncheck the [x] FlowShow installed at the main FlowShow window.

7-2.Logging performance

You can improve logging performance in a number of ways:

- Activate in-memory log configuration
- Select fewer methods to receive logging information.
- Disable logging of object names, methods returns and (mainly) methods parameters.
- If you are using a custom parameter to text conversion, please ensure that it's minimal and highly optimized (as it will be called for every parameter of every instrumented method execution)
- Leave the [x] FlowShow logs enabled configuration turned off, turning it on in selected game parts.
- Use the execution count limit configuration.
- Disable the logging of lengthy, repetitive methods loops. This can be done in the Log viewer window or the Log Summary window.

7-3.Where is .ToString() executions?

FlowShow ignores .ToString() methods.
Despite some internal controls to avoid infinite recursion, .ToString() can be used inside the log writing procedures (on transforming owner objects, parameters and return values into text).

For that matter, FlowShow also ignores its own source code, to avoid a disastrous Klein Bottle type of situation.

7-4.Errors on Delete/Reset log

FlowShow writes the log file (**FlowShowLogs.flowshow**) in the Temp folder, inside the project folder.

Please ensure that the Unity Editor has write permissions on this file.

Antivirus and/or other scanning software can also lock the file, prevent FlowShow to delete/reset it. In this case, please add the project Temp folder as an exception.

8.Contact / support

Contact/Support e-mail is: l-tyrosine@arcadiumplayware.com

You can also use the [Report a problem/Contact] button, in the main FlowShow window.

Here you will be able to easily copy the last reported error by FlowShow (if any).
If you are running into an installation problem with a particular method, this tool also allow to extract and copy the .net IL body of it, helping a lot on the solution research.

Please feel free to also write about any doubts, suggestions or technical discussion about FlowShow inner workings.

9.FlowShow History / Closing words

This is the 6th generation of the FlowShow tool (or something close to it).
It started as a pascal/Delphi tool, back in 2000's, to help me find a way through endless legacy code base of large ERP systems.

After some early .net versions, I did write a (con)version back in 2015 targeting Unity Editor. Unfortunately, never managed to catch the time to finish it properly as a usable component.
One main issue was the necessity of source code instrumentation. While it worked automatically, I never really liked the idea of forcing the user to work/code between an instrumented block.

After many debugging sessions using endless Debug.Log in my small indie / game jams projects, I finally convinced myself how useful a proper FlowShow implementation would be for us, Unity 3D users.

In set/2021 I finally took the time to properly refine a Unity Editor targeted version will full .net IL injection support, using the amazing Mono.Cecil API.
I also expanded the core functionality of logging with some features on method timing, profiling, reporting, etc.

I really hope that you find it useful and help you and your team on the path to better coding and debugging; Following your code path!

Thank you.

- L-Tyrosine
Arcadium Playware

FlowShow and Arcadium Playware are not sponsored by or affiliated with Unity Technologies or its affiliates.
"Unity" is a trademark of Unity Technologies or its affiliates in the U.S. and elsewhere.