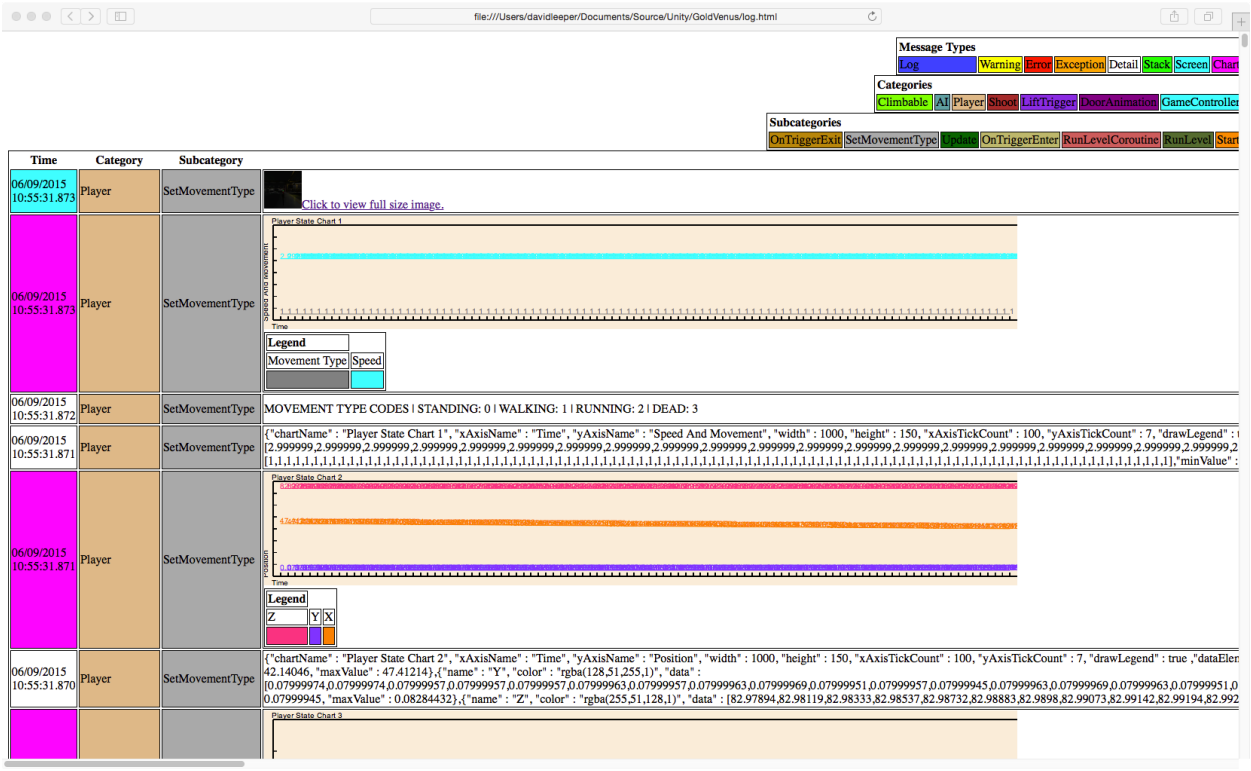
Logger

For Unity

Version 1.1

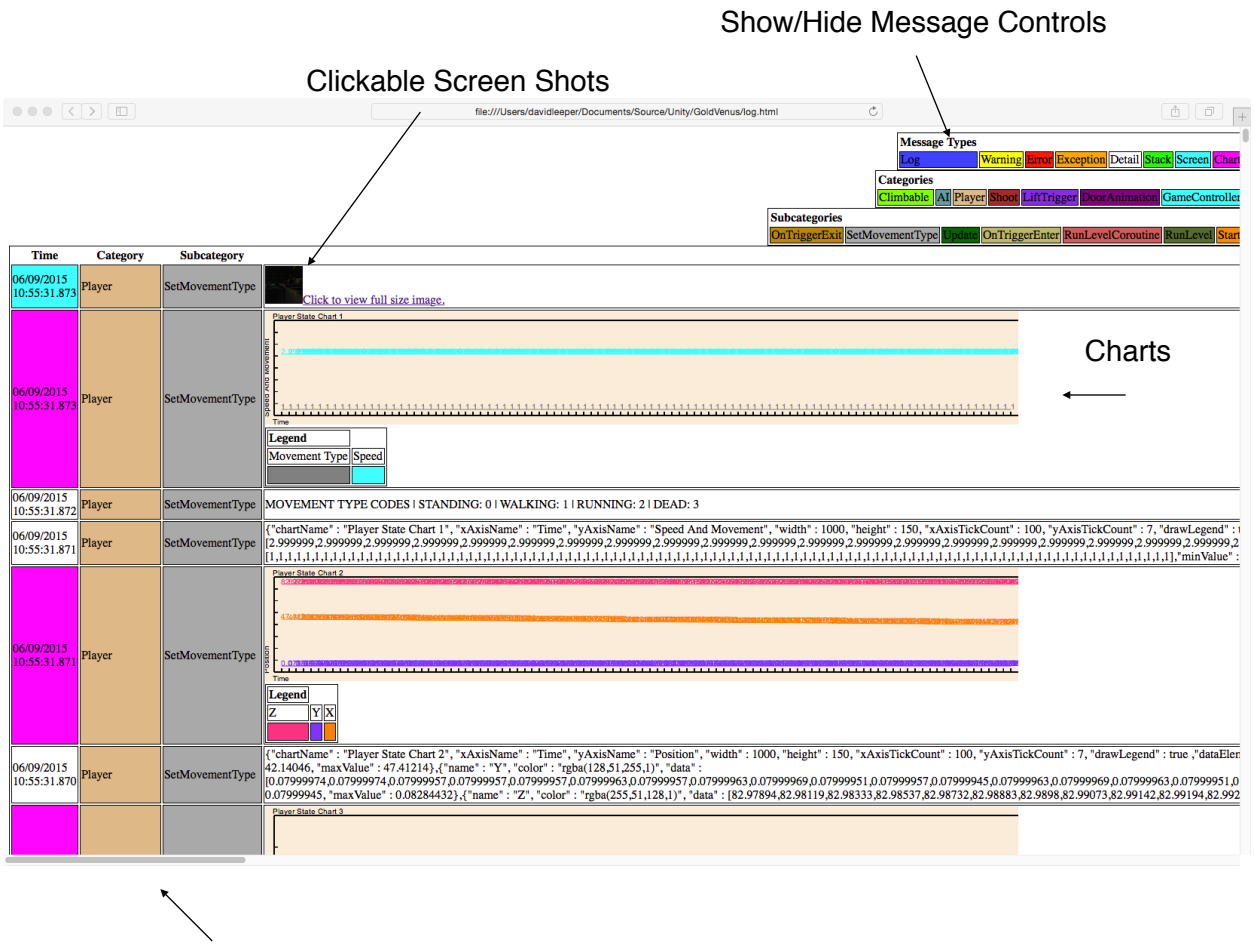


Written by David Leeper, author of the Logger code.

Introduction

Logger for Unity provides a rich drop-in replacement for Unity's logging facility. The package provides:

- Rich media HTML logs that can zipped up and sent via email, or stored in DropBox or source control and retain full functionality.
- Error, exception, warning, logging, and detail messages.
- Stack trace logging.
- Screen shot logging.
- Chart logging.
- The ability to group logging message by type, category, and subcategory.
- The ability to show and hide all messages of a given type, category, or subcategory.
- The ability to also log to Unity's standard console logger.
- Most recent entries always at the top of the log.
- Easy to use API.
- Rolling Logs
- Ability to turn off logging certain messages types throughout the program by setting a boolean value in the logger class.



Sample Log

The best way to get familiar with what the Logger can do is to open up the sample log in a browser. You'll need a browser that supports HTML 5, which all of the modern browser do.

Play around with the buttons to see how the hide and show various messages, click on the screen shot icons to open up the full image, and check out the charts. Notice the relationship between the Message Type, Category, and Subcategory columns of the chart on the left and the button controls at the top right of the screen.

The log contains a complete walk-thru of a simple game level. Notice that the newer messages are at the top of the log and the older messages are at the bottom.

I'm confident you'll be impressed with the power of the Logger and eager to incorporate it into your games.

Coding With The Logger

Just add Logger.cs to your project to start using the logger. All logging except charts can be done by making calls to static methods on the Logger class. For charts, you'll need to set up the data and then pass it to the logger. For screen shots, you'll need to set up a subfolder to store the images. We'll cover how to do this in just a moment.

Message Types

Logger supports the standard message types of Error, Warning, Log Message, and Detail. Additionally, it supports Stack Traces, Charts, and Screen Shots.

Categories And Subcategories

The logger methods have parameters for the category and subcategory of the message being logged. These parameters are text values that can be anything you want. I use the name of the class calling the logger as the category and the name of the method calling the logger as the subcategory.

Rolling Logs

Logger supports a single log of infinite length or up to three rolling logs of a predetermined size. The default is a single log of infinite length.

The `maxLogSize` member determines how large a log can be. A value of -1 indicates an infinite sized log. Any other value is taken as the maximum number of bytes the log can contain. For example, a value of 1000000 indicates a maximum size of 1 MB.

The `logPath1`, `screenshotSubfolder1`, and `logFile1` members indicate the path, file name, and screen shoot subfolder of the first log. The `logPath2`, `screenshotSubfolder2`, and `logFile2` members indicate the path, file name, and screen shoot subfolder of the second log. The `logPath3`, `screenshotSubfolder3`, and `logFile3` members indicate the path, file name, and screen shoot subfolder of the third log.

If you do not wish to use a second or third log, just set their corresponding members to an empty string. This is the default.

Examples on how to set up the various types of logs are provided below.

```
// Set up a single log of unlimited size. This is the default.
Logger.maxLogSize = -1;
Logger.logPath1 = ".";
Logger.logFile1 = "Log.html";
Logger.screenshotSubfolder1 = "screens";
Logger.logPath2 = "";
Logger.logFile2 = "";
Logger.screenshotSubfolder2 = "";
Logger.logPath3 = "";
Logger.logFile3 = "";
Logger.screenshotSubfolder3 = "";
```

```
// Set up a single log of 1 MB size.
Logger.maxLogSize = 1000000;
Logger.logPath1 = ".";
Logger.logFile1 = "Log.html";
Logger.screenshotSubfolder1 = "screens";
Logger.logPath2 = "";
Logger.logFile2 = "";
Logger.screenshotSubfolder2 = "";
Logger.logPath3 = "";
Logger.logFile3 = "";
Logger.screenshotSubfolder3 = "";
```

```
// Set up a two logs of 1 MB size.
Logger.maxLogSize = 1000000;
Logger.logPath1 = ".";
Logger.logFile1 = "Log1.html";
Logger.screenshotSubfolder1 = "screens1";
Logger.logPath2 = ".";
Logger.logFile2 = "Log2.html";
Logger.screenshotSubfolder2 = "screens2";
Logger.logPath3 = "";
Logger.logFile3 = "";
Logger.screenshotSubfolder3 = "";
```

```
// Set up a two logs of 1 MB size.
Logger.maxLogSize = 1000000;
Logger.logPath1 = ".";
Logger.logFile1 = "Log1.html";
Logger.screenshotSubfolder1 = "screens1";
Logger.logPath2 = ".";
Logger.logFile2 = "Log2.html";
Logger.screenshotSubfolder2 = "screens2";
Logger.logPath3 = ".";
Logger.logFile3 = "Log3.html";
Logger.screenshotSubfolder3 = "screens3";
```

☞ When Logger roles from one log to the next, it first deletes any existing log and screen shots for the new log. For example, if the log roles from log1 to log 2, the existing log 2 and its screen shots are deleted before any messages are written to log 2.

Logger Class Parameters

☞ The defaults for these values set up a single log of unlimited size in the current directory. The log file will be named log1.html. Screenshots will go in a subfolder named “screens1”.

☞ If a subfolder you specify for storing screen shots does not exist it will automatically be created by the Logger.

The Logger class has a few public static variables you use to tune the logger to your needs.

```
// Log file settings
public static long maxLogSize = -1; // -1=unlimited log size, 1000000=1 MB
public static string logPath1 = "./Logging";
public static string screenshotSubfolder1 = "screens1";
public static string logFile1 = "log1.html";
public static string logPath2 = "";
public static string screenshotSubfolder2 = "";
public static string logFile2 = "";
public static string logPath3 = "";
public static string screenshotSubfolder3 = "";
public static string logFile3 = "";
// Log message switches
public static bool logToHTML = true;
public static bool logToConsole = true;
public static bool logMessages = true;
public static bool logWarnings = true;
public static bool logErrors = true;
public static bool logExceptions = true;
public static bool logDetails = true;
public static bool logStackTraces = true;
public static bool logScreenShots = true;
public static bool logCharts = true;
```

The proper usage of these variables is explained below.

VARIABLE	USE
maxLogSize	The maximum size of a log file. Used to create rolling logs. A value of -1 indicates a log of infinite size. A value of 1000000 indicates a log of 1 MB.
logPath1	Indicates where the first log should be written.
screenshotSubfolder1	The name of the folder used to store screen shots for the first log. Must be a direct subfolder of log path. The folder will be created if it does not exist.
logFile1	The name of the first log file. The name should have an html extension, as the log is HTML.
logPath2	Indicates where the second log should be written. Leave this value blank if no second log is desired.
screenshotSubfolder2	The name of the folder used to store screen shots for the second log. Must be a direct subfolder of log path. The folder will be created if it does not exist. Leave this value blank if no second log is desired.
logFile2	The name of the second log file. The name should have an html extension, as the log is HTML. Leave this value blank if no second log is desired.
logPath3	Indicates where the third log should be written. Leave this value blank if no third log is desired.
screenshotSubfolder3	The name of the folder used to store screen shots for the third log. Must be a direct subfolder of log path. The folder will be created if it does not exist. Leave this value blank if no third log is desired.
logFile3	The name of the third log file. The name should have an html extension, as the log is HTML. Leave this value blank if no third log is desired.
logToHTML	Set to true if the Logger should write log messages to the HTML file.
logToConsole	Set to true if the Logger should write log messages to the Unity console. Note that screenshots and charts cannot be written to the console.
logMessages	Set to true if the Logger's Log() method should be active.
logWarnings	Set to true if the Logger's LogWarning() method should be active.
logErrors	Set to true if the Logger's LogError() method should be active.
logExceptions	Set to true if the Logger's LogException() method should be active.
logDetails	Set to true if the Logger's LogDetail() method should be active.

`logStackTraces`	Set to true if the Logger's LogStackTrace() method should be active.
`logScreenShots`	Set to true if the Logger's LogScreenShot() method should be active.
`logCharts`	Set to true if the Logger's LogChart() method should be active.

Sample Logging Calls

A few sample logging calls are shown below. Notice the methods are static and each takes a category and subcategory as the first and second parameters.

```
using General;
Logger.Log(“LiftTrigger", "OnTriggerEnter", "Player in lift.");
Logger.LogError("Chase", "Awake", "Could not find enemy's NavMeshAgent.");
Logger.LogException("Chase", “Awake", oException);
Logger.LogDetail("Player", "SetMovementType", movementTypes);
Logger.LogScreenShot("Player", “SetMovementType");
Logger.LogChart("Player", "SetMovementType", chartSpeedMovement);
Logger.LogStackTrace("Player", "SetMovementType");
```

As you can see, the Logger class is in a namespace called General.

Chart Setup

Before we cover how to set up a chart for Logger, let's take a look at the elements of a chart.

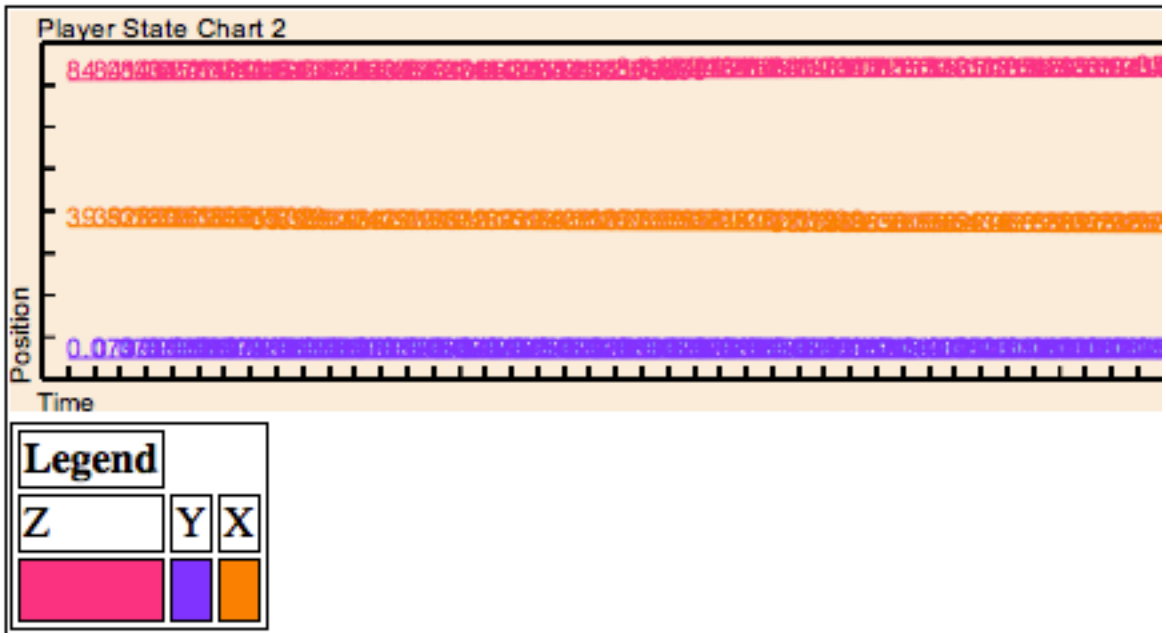


CHART ELEMENT	NOTES	DEFAULT
`chartName`	Called **Player State Chart 2** in the above example.	Chart
`xAxisName`	Called **Time** in the above example.	X Axis
`yAxisName`	Called **Position** in the above example.	Y Axis
`width`	The width of the chart in pixels.	450
`height`	The height of the chart in pixels.	150
`xAxisTickCount`	The number of ticks to be drawn along the X axis of the chart.	10
`yAxisTickCount`	The number of ticks to be drawn along the Y axis of the chart.	5
`drawLegend`	A boolean value indicating if a Legend should be drawn for the chart.	FALSE
`chartDataElements`	An array of objects that hold the values for the chart. The above example has three elements: X, Y, and Z.	None

The `chartDataElements` array is a collection of `ChartData` objects. The `ChartData` class holds the values for a single element of a chart. Each `ChartData` object provides the following information:

CHART DATA ELEMENT	NOTES	DEFAULT
`name`	The name of the element. Used to draw the Legend. The above example has elements name X, Y, and Z.	Data
`color`	The color used to draw the element in the chart. The above example uses red, orange, and purple.	Black
`data`	An array of floats that store the values of the element. Each value will be plotted along the Y axis of the chart, with successive values moving along the X axis. This is an array of floats.	None

Example code for setting up a chart is provided below.

```
using General;
public class Player : MonoBehaviour
{
      private ChartData[] chartDataPosition;
      private Chart chartPosition;
      private int dataIndex = 0;
```

The `ChartData` array is used to store one `ChartData` object for each element of the chart.

```
      void Awake(){
            chartDataPosition = new ChartData[3] {
                  new ChartData("X", 100,
                        new Color(1.0f, 0.5f, 0.0f, 1.0f)),
                  new ChartData("Y", 100,
                        new Color(0.5f, 0.2f, 1.0f, 1.0f)),
                  new ChartData("Z", 100,
                        new Color(1.0f, 0.2f, 0.5f, 1.0f))};
            chartPosition = new Chart(chartDataPosition, true,
                  1000, 150, 100, 7,"Player State Chart 2", “Time",
                  "Position");
      } // Awake
```

In the `Awake` method of the class, we initialize the `ChartData` array with three new `ChartData` objects. The `Chart` object is then initialized with the `ChartData` array.

Notice the constructors of these classes are used to set all the data in the class. The parameters for these constructors will be discussed at the end of this chapter. Alternatively, you can just use the versions of the constructors with no parameters and then set each public member value.

```
      void OnUpdate(){
            chartDataPosition[0].data[dataIndex] =
                  GetPlayer().transform.position.x;
            chartDataPosition[1].data[dataIndex] =
                  GetPlayer().transform.position.y;
            chartDataPosition[2].data[dataIndex] =
                  GetPlayer().transform.position.z;

            dataIndex++;
            if (100 == dataIndex){
                  Logger.LogDetail("Player", “SetMovementType",
                        chartPosition.ToJSON());
                  Logger.LogChart("Player", "SetMovementType",
                        chartPosition);
                  chartPosition.ClearData();
                  dataIndex = 0;
            } // if
      } // OnUpdate
} // class
```

In the `OnUpdate` method we add the data to each chart element. In this case we're adding the Player's X position to the first element, the Y position to the second element, and the Z position to the third element.

Then we check to see if the number of entries is 100. If it is we print a detail of the chart's data by calling `Logger.LogDetail` and passing it the chart data in JSON form using the ToJSON() method. This is completely optional, but shows how to get a text representation of the chart.

Then we log the actual chart itself using `LogChart` and passing it the chart object. Note that you do not have to place chat logging in the `OnUpdate` method. It can go anywhere in the code you'd like to use it. After the chart is logged, we clear its data to start fresh for future logs.

Constructors

Both the `ChartData` and `Chart` classes have constructors that accept parameters that allow you to initialize the chart.

Note that the values passed to these constructors all have corresponding public data members. So if you want, you can call the empty constructors and set each parameter manually.

Chart Data Constructor

```
public ChartData(string inName, int inDataSize, Color inColor)
```

PARAMETER	PURPOSE
inName	The name used to identify the chart data in the chart's legend.
inDataSize	The number of elements in the data array.
inColor	The color used to draw the chart element.

Chart Constructor

```
public Chart(ChartData[] inChartData, bool inDrawLegend = false,
      int inWidth = 450, int inHeight = 150, int inXAxisTickCount = 10,
      int inYAxisTickCount = 5, string inChartName = "Chart",
      string inXAxisName = "X Axis", string inYAxisName = "Y Axis")
```

PARAMETER	PURPOSE	DEFAULT
inChartData	An array of ChartData objects contained in the Chart.	None
inDrawLegend	The number of elements in the data array.	FALSE
inWidth	The pixel width of the chart.	450
inHeight	The pixel height of the chart.	150
inXAxisTickCount	The number of ticks down along the X axis.	10
inYAxisTickCount	The number of ticks down along the Y axis.	5
inChartName	The name of the chart.	Chart
inXAxisName	The name of the X axis.	X Axis
inYAxisName	The name of the Y axis.	Y Axis

Other Public Methods

Chart Data Public Methods

METHOD	PURPOSE
`ToJSON`	Returns a JSON string representing the chart data.
`ClearData`	Clears the data array from the ChartData.
`GetMinValue`	Returns the minimum value in the ChartData data.
`GetMaxValue`	Returns the maximum value in the ChartData data.

Chart Public Methods

METHOD	PURPOSE
`ToJSON`	Returns a JSON string representing the chart.
`ClearData`	Clears the element array from the Chart.
`GetMinValue`	Returns the minimum value in the Chart.
`GetMaxValue`	Returns the maximum value in the Chart.

How Data is Stored in the ChartData class.

The data array of the `ChartData` data class is of type float. So all values you wish to chart must be converted to float values.