Unity DebugTools - v0.3.0

Author: Aaron Walwyn **Website:** www.aaronwalwyn.com/DebugTools/ **Contact (email):** me@aaronwalwyn.com **Current Version:** Debug Tools v0.3.0 *alpha*

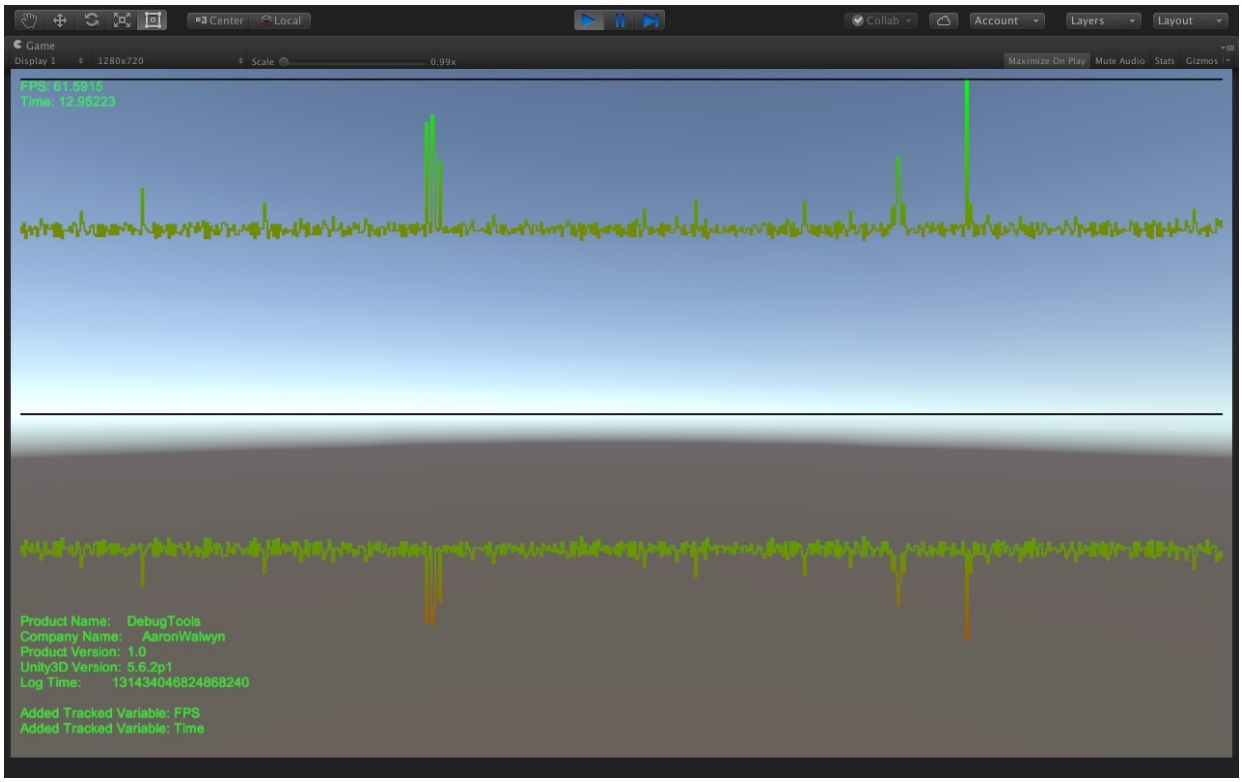
	5.3	5.4	5.5	5.6	2017
Windows x64			X	X	
Windows x86			X	X	
Android arm-v7					
Android x86					
iOS					
macOS (OS X)			X	X	
Linux					

The package is most likely compatible with all of the above systems and cpus however it has not been officially tested as of yet.

Introduction

DebugTools is a unity extension designed to help developers output useful inforamtion in a visual way to help debugging and testing of games and applications.

It is currently in a early alpha stage compromising of a few limited features such as logging to an onscreen console, tracking varaibles on screen and drawing graphs on the screen. The asset will remain in alpha/beta while the major features are developed and updated. Feature requests and feedback are always appreciated.



Getting Started

The simpilest way that you can get started with the asset is to drag and drop the `[ Debug ]` prefab located at `/Assets/DebugTools/Prefabs` into your scene (preferably the first scene that will we ran). When ran this will create all the necessary components programmatically.

Alternatively if you would like to start the DebugTools from a script you can instead call the method `DebugTools.Create ();` at any point in the application.

To make use of the DebugTools functionality in other scripts you should be sure to include `using DebugTools` at the top of those scripts.

Controls

```
,               - Show / Hide Console Output
.               - Show / Hide Variable Tracker
/               - Show / Hide Graph Output

In a future version these key binding will be adjustable
```

Logging

```
1  DebugTools.Log(object obj, toConsole=false, toFile=false);
```

Logging is easy with DebugTools. Similarly to Unity's native `Debug.Log(object obj)` you can instead call `DebugTools.Console.Log(object obj)` and the `.ToString()` of the object will be called and printed to the on screen console output. Additionally you can set the optional parameter `toConsole` to true so that the message is also printed to the Unity editor debug console or the parameter `toFile` which will log the message to the application log file for review after runtime.

The console class also integrates the ILogger interface allowing you to use all the same Log overloads which can bee seen below.

```
Log(LogType logType, object message)
Log(LogType logType, object message, UnityEngine.Object context)
Log(LogType logType, string tag, object message)
Log(LogType logType, string tag, object message, UnityEngine.Object context)
Log(object message)
Log(object message, bool toConsole=false, bool toFile=false)
Log(string tag, object message)
Log(string tag, object message, UnityEngine.Object context)

LogWarning(string tag, object message)
LogWarning(string tag, object message, UnityEngine.Object context)

LogError(string tag, object message)
LogError(string tag, object message, UnityEngine.Object context)

LogFormat(LogType logType, string format, params object[] args)
LogFormat(LogType logType, UnityEngine.Object context, string format, params object[] args)

LogException(Exception exception);
LogException(Exception exception, UnityEngine.Object context);
```

The log file can be found in the persistent data path for your application, for more info see the Unity3D documentation.

Tracking Variables

```
1  DebugTools.Track (string key, object obj);
```

Tracking a varaible in the DebugTools package is also easy, simple call the method above with a unique key and pass in an object of which the `.ToString()` method will be called and tracked in the top corner of the screen.

Currently there is no way to remove tracked variables, this will be available in an update to the package.

Creating Graphs

```
1  DebugTools.CreateGraph (string key, float min, float max, out
   DebugTools.Graph graph)
```

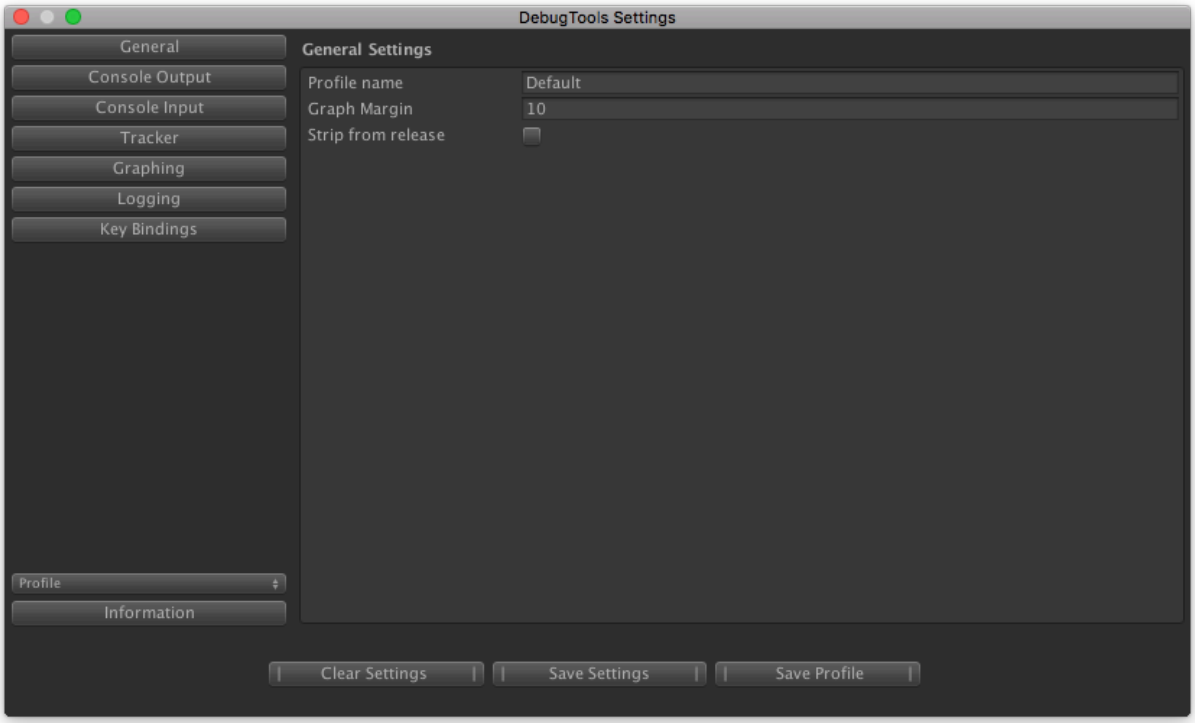
Lastly the functionality to create graphs has also been included in the package, to create a graph you must first create a `DebugTools.Graph` as a **global** object then call the method above, passing in the minimum and maximum values for the parameter as well as the graph object just created.

The value of the graph will start as the average of the minimum and maximum value, to update the value you can set the `latestValue` variable of the graph.

Adjusting the default settings

DebugTools uses a range of default varibles which determine how the different elements are drawn, updated and controlled. These values are all adjustable by developers via the built in unity editor, however they are also adjustable by updating a settings file while your application is deployed (So you can make changes without having to re-build the application).

To get started with editing the settings open the custom editor window at `Tools/DebugTools/Settings` and you should be presented with the screen below. This panel allows you to adjust settings for each of the different components of DebugTools. Please note that some panels don't currently have any settings but will do in future.

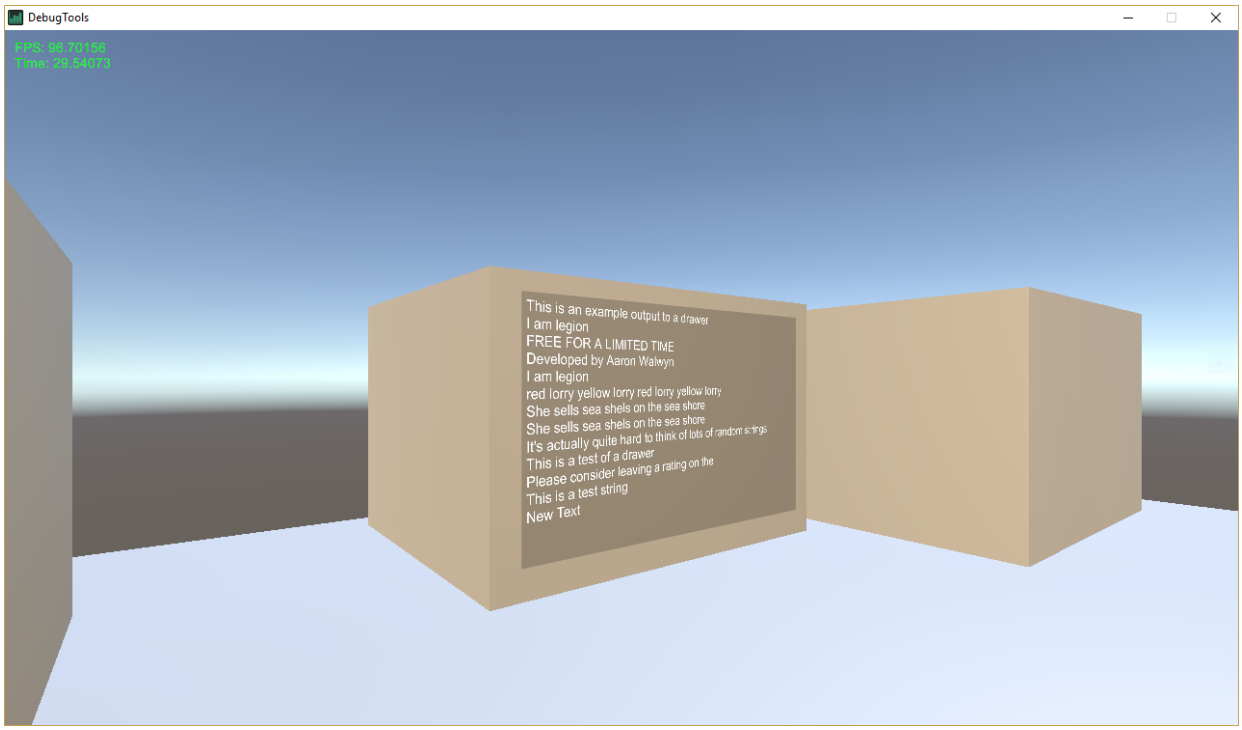


The first time the settings panel is opened a settings file will be created is the folder `Assets/StreamingAssets/DebugTools` which contains a xml represenation of all the currently set variables. This will be included in your application and read at runtime if it exists. You can modify this file while your application is deployed to adjust these variables without having to rebuild the application. (Currently the xml does not contain any guidence on valid values but will do in future).

Please note that alothough profiles can be saved it is not currently possible to switch profiles in the editor. This functionailty will be avalaible in the next version of DebugTools

Creating Custom Loggers

DebugTools also ads the option for you to create your own custom loggers on top of the default console output included by default. This is especially helpful when developing for VR where screen space UI can be problematic and world space UI is preferential.



This can easily be done by creating a new object that instead inherits from the `MonoDrawer` object, which itself inherits from `MonoBehaviour`.

```
public class ExampleVariableDrawer : MonoDrawer {

  public Text outputText;

    // Use this for initialization
  void Start () {
    //The drawer name must be set before the
RegisterDrawer() function
    //is called and is not case-sensitive eg. (exampledrawer
=== ExAmPleDrawER).
    drawerName = "exampledrawer";
    RegisterDrawer();
  }

  //Here you can implement your own custom way of outputing
the file be
  //it to a UI element or log file
  public override void DrawVariable(string key, object
message) {
    outputText.text = message.ToString() + "\n" +
outputText.text;
  }

  public override void DrawVariable(string key, string
format, params object[] args) {
    if (args.Length  <= 0) {
      DrawVariable(key, (object)format);
    } else {
      StringBuilder sb = new StringBuilder();
      sb.AppendFormat(format, args);
      DrawVariable(key, sb);
    }
  }
}
```

To write to the custom logger from elsewhere in your application you use the function `DrawManager.DrawVariable(string key, object message)` to draw something to the output. for example:

```
DrawManager.DrawVariable("exampledrawer", "This is a test of
the draw manager");
```

Licence

Use of this package is subject to the terms of the Unity's standard Unity Asset Store End User License Agreement. More information is available here.