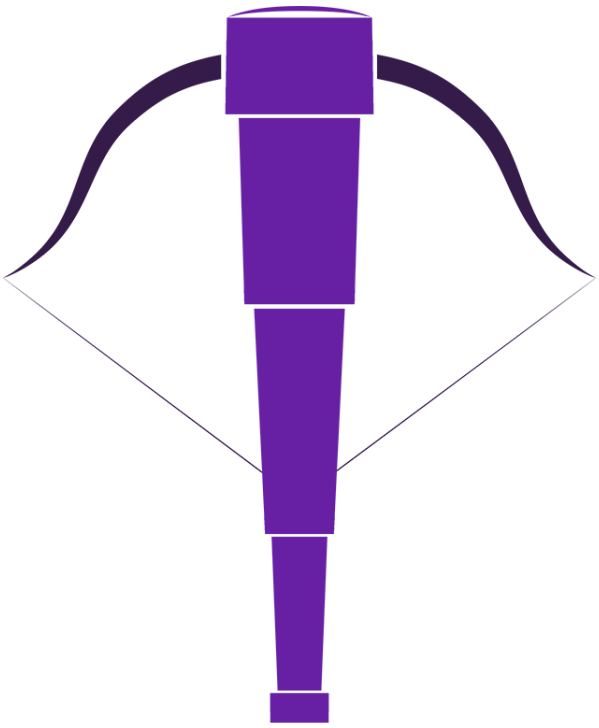
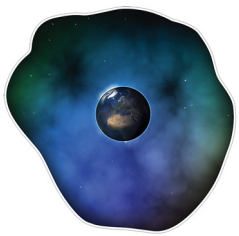


AQHAT

Auto Quality Hardware Assessment Tool



EAGER AMOEBA®

CONTENTS

INTRODUCTION

Hello and thank-you for purchasing AQHAT!

AQHAT is a Unity plugin that analyses device information and scores it's potential performance, this score can then be used to set quality upon launch or configure anything that requires you know the rough performance of a device. An example script is included in order to show you how to use this performance score in order to set your quality setting.

AQHAT caches data after it's first run, to ensure the script is not unnecessarily ran.
This cache is ignored if we detect a hardware deviation that would effect the scoring system.

We have tested AQHAT on; UWP, Windows, Linux, Android, Amazon Android, OSX, IOS and tvOS.

If you have any issues or improvements for us, please let us know via our websites contact page - http://eageramoeba.co.uk/Contact/

To keep updated on further developments and any other Unity plug-ins we produce, follow us on facebook, twitter, YouTube and instagram via @EagerAmoeba

THE EPIC OF AQHAT

The name AQHAT is an abbreviation of 'Auto Quality Hardware Assessment Tool'. Co-incidentally it's also the name of an obscure figure in Canaanite mythology.

The Epic of Aqhat is an incomplete epic poem telling the story of a son of Danel (Prince Aqhat) who was granted a magical bow by the gods, he was later killed for this bow by the goddess of love and war. It was then lost in the sea while being transported to her by the servant that killed Aqhat. The rest of the tale is lost due to the discovered tablet being only partially intact, however it is assumed to detail a resurrection story that will explain the droughts during summer months in the eastern Mediterranean. The idea being the drought is caused by the death of Aqhat and his resurrection ushers in the fertile months for crop production.

We like this idea and have used aspects of the legend in our branding design for this product, it would be wrong to use the same name as an abbreviation and not reference the magical bow that was owned by said prince. If we were to stretch this we would say it parallels our product by helping you easily hit your quality target for each device as a magical bow would never miss it's target.

ASSET CONTENTS

1. A data folder containing 5 files. The are device/hardware/IOS configurations and free sample Passmark data.
2. A documentation folder containing instructions and explanations of the methods and systems involved.
3. A prefabs folder containing a sample implementation of the methods.
4. A scenes folder containing a debug/test scene.
5. A scripts folder containing CSVReader.cs, DeviceRating.cs and AutoQuality.cs.
6. README.txt

HOW IT WORKS

AQHAT uses Unity's SystemInfo data combined with data contained in several txt and csv files to approximate the device's potential performance.

The potential performance is measured by a score that is used relative to other devices and the individual configuration of each studio/individual. For example, with the default configuration scores and iPod Touch 6 at 2.13. A PC running16GB memory, AMD FX 8350 and an Nvidia 970 GTX scores at 10.9. Lastly, a cheap android phone (Discovery A8) scores at 1.02.

To help make this reading more accurate AQHAT can make use of benchmark data in CSV format or an experimental feature that scans hardware names and approximates a score based upon the name and version number (guided via a configuration file detailed later on).

LIMITATIONS

AQHAT is limited in a number of factors. The most obvious one is the experimental mode detecting name and version number of hardware devices, the featured is configured for a number of devices but obscure configurations could be missed.
AQHAT will not take into account limiting factors such as remaining device storage or degrading technology.
Lastly, devices that include a large difference in score between any of the three areas (RAM, GPU and CPU) could produce an inaccurate score. For example, 512GB of ram in combination with high-end GPU and CPU components will elevate the score, above where it realistically should be with the serious RAM bottleneck.

QUICK START (EXAMPLE METHOD)

AQHAT requires a certain amount of configuration in order to be used correctly. In order to set it up correctly you will need to know the minimum score required to run your program at the lowest Unity quality setting. An example prefab has been included to show you how to use the score to set the Quality.

1. Open the sample scene "testScene" found in the prefabs folder.
2. Check that all settings in the module DeviceRating.cs attached to AutoQuality.cs are correct. This can be left as-is if you are unsure.
3. Run this scene on all relevant devices and note down their scores, both media and total. These are lines one and two. Alternatively, limit each hardware configuration reading using the cap and mock settings.
4. Using the noted down scores, set the "Quality Bands" array in accordance with your quality settings defined within unity. Each array entry should be the lowest score needed to hit this quality setting.
5. Save the "AutoQuality.cs" module as a SEPARATE prefab to the one found in the AQHAT folder.
6. Use the prefab you've saved in the first scene your project accesses.
7. Make sure the asset is loaded and the script runs BEFORE any other assets are loaded.
8. Enjoy!

Scripts

AQHAT contains three scripts;
AutoQuality.cs – The example script, this script uses the score found in DeviceRating.cs and sets the quality based upon this.
CSVReader.cs – Free script from the Unity community library to read and parse CSV files. Provided under creative commons sharealike license.
DeviceRating.cs – The main bulk of AQHAT. When ran this script analyses your device, assigns it a score and then caches the results.

PLEASE NOTE:
For optimum results, make sure all scripts are ran BEFORE you load any game objects. Otherwise you will re-load assets you've already loaded upon changing quality.

DeviceRating.cs will allow the access of several static values for use within your own Automatic Quality script. These are as follows:

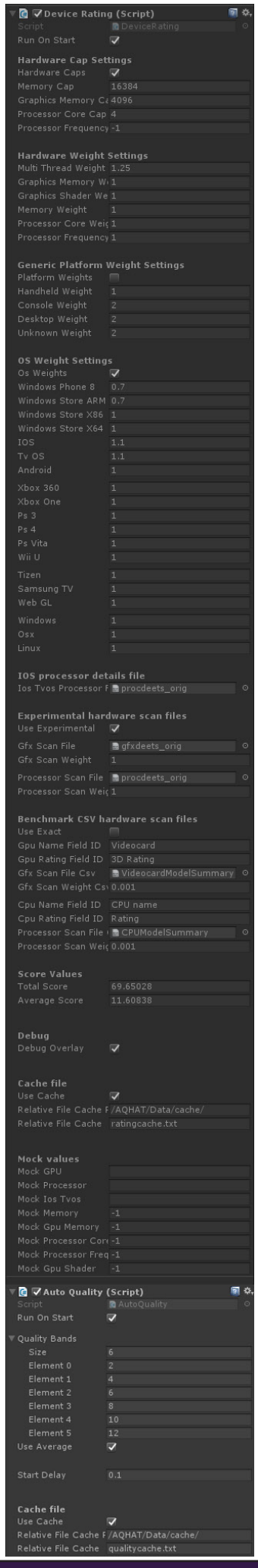
```
public static float TotalScore;
public static float AverageScore;
public static float GFXScore;
public static float ProcessorScore;
public static float GFXScoreNoScan;
public static float ProcessorScoreNoScan;
public static float MemoryScore;
public static float ScanProcessorScore;
public static float ScanGFXScore;
static string fileCachePathStStatic = "";
public static float processorCores;
public static string processorName;
public static float processorFrequency;
```

There is also a static method included in order to wipe the cache for both scripts, this can be access via the following lines:

```
AutoQuality.wipeCache();
DeviceRating.wipeCache();
```

In order to access either of these scripts, you need to include the eageramoeba namespace within your own script. For example, place this at the top of your script file in order to access the static score values:

```
using eageramoeba.DeviceRating;
```



DeviceRating.cs

This is the main bulk of AQHAT. When ran this script analyses your device, assigns it a score and then caches the results.

By default, this script is set to run on start. However this can be disabled and then ran later with the static method ‘RunMe’. Use this line of code to call this method:
DeviceRating.RunMe();

Cap settings

AQHAT uses 4 values within SystemInfo in order to determine each score. These values can be capped on an individual basis. Doing so puts a hard limit on the amount that can be read from each value.

For example setting the memory cap to 16384 will not allow any more than 16GB of memory to be taken into account.
Setting the Processor Core Cap to 4 will not allow more than 4 cores to be take into account.

Setting the cap to -1 means that there will be no cap on that particular value. E.g setting Processor Core Cap to -1 means that all cores will be taken into account.

This can be enabled/disabled using the “Hardware Caps” boolean.

Default values are as follows:

```
public bool hardwareCaps = true;
public float memoryCap = 16384;
public float graphicsMemoryCap = 4096;
public float processorCoreCap = 4;
public float processorFrequencyCap = -1;
```

Weight settings

AQHAT allows you to weight certain values, this means that certain values will be multiplied by the weight setting. This can be used to increase or decrease the score value. You can at current weight the OS, generic device (mobile, desktop, console) and hardware results.

Each weight category can be disabled with it’s relevant boolean value.

The hardware weights will multiply that specific hardware’s score by the set value.
For example, a processor frequency score of 2 with a weight of 1.5 will result in a score of 3. A weight of 0.5 will result in a score of 1.

The OS and generic platform weights multiply the total score by the set value of the device is found to fall into these categories. Generic platform is applied before OS weight.
For example, if OS weights and generic device weights are enabled and set to 1.5 for Android and 1 for Mobile then and android phone with a Total score of 10 would be modified to score 15.

Default values are as follows:

```
public float multiThreadWeight = 1.25f;=
public float graphicsMemoryWeight = 1f;
public float graphicsShaderWeight = 1f;
public float memoryWeight = 1f;
public float processorCoreWeight = 1f;
public float processorFrequencyWeight = 1f;

public bool platformWeights = false;
public float handheldWeight = 1;
public float consoleWeight = 2;
public float desktopWeight = 2;
public float unknownWeight = 2;

public bool osWeights = true;
public float windowsPhone8 = 0.7f;
public float windowsStoreARM = 0.7f;
public float windowsStoreX86 = 1f;
public float windowsStoreX64 = 1f;
public float IOS = 1.1f;
public float tvOS = 1.1f;
public float android = 1;
public float xbox360 = 1;
public float xboxOne = 1;
public float ps3 = 1;
public float ps4 = 1;
public float psVita = 1;
public float wiiU = 1;
public float tizen = 1;
public float samsungTV = 1;
public float webGL = 1;
public float windows = 1;
public float osx = 1;
public float linux = 1;
```

IOS/tvOS processor details

Unfortunately, IOS and tvOS does not allow you to gain access to the processor details of the device. To get around this we have compiled a txt file containing details of all known tvOS and IOS devices. We use the SystemInfo.deviceModel variable to match with the hardware strings of each device, when a match is gained the processor name, number of cores and the frequency is inserted into the system.
To access these values, you can use the static variables;

```
DeviceRating.processorCores
DeviceRating.processorName
DeviceRating.processorFrequency
```

The syntax of each entry is this file are as follows;
Example:

```
iPhone1,1/1/412/Samsung 32-bit RISC ARM 11;
```

Breakdown:

```
Device Model/Processor Cores/Processor Frequency/Processor Name;
```

Hardware scan

In order to increase the accuracy of the score we have included two methods to scan the hardware and add to the score based upon benchmark data or detected hardware, this looks at the GPU and the CPU.
Two methods have been included and can both be enabled separately, enabling both means both scores will be added to the overall score. We're recommend using one ourselves but some may want to do this. The first method (experimental scan) scans the hardware name, then uses this combined with the version number to approximate a score. The second uses benchmark data to add score based upon a weighted version of the benchmark data's calculated score. In either method tm and ® symbols are stripped out. We recommend only using experimental scan when benchmark data is unavailable, for example due to price, size or performance.

Experimental scan

The experimental scan uses two txt files containing data on most well-know CPU and GPU brands. The relevant hardware name is then compared to the results in this text file and a score is estimated.
This is done by using the version number, certain character combinations and matched words within the hardware name. The version number will largely dictate the score, this can the be weighted with the overall weight value for the GPU and CPU.

By default, the weight is set to 1 for both GPU and CPU.

There are two files, gfxdeets_orig.txt for the GPU and procdeets_orig.txt for the CPU.

These files contain entries separated by ';'.

An example entry is as follows:

```
Geforce/0/4=0.1/4/C/1080.1070.1060.Titan.GeForce2.GeForce3.GeForce4.Ge-
Force 256,GTX=0.5,GT=0.5,Ti=0.5,M=-3;
```

Each entry is split by commas, the first entry contains details of the hardware to look at and subsequent entries contain character sequences that when matched add score (found after the equals sign) if a match is found in the first comma split.

Within the first comma separated entry, it is separated by '/' and each value is then separated as follows:
0 – Contains words or character strings that require a full or partial match. Can be multiple values if each value is separated by a full stop.
1 – Dictates which number to start the version number scan.
2 – Dictates which number to end the version number scan and what weight the score will carry.
3 – Dictates how many numbers can be within the string after all characters have been removed.
4 – Dictates whether value 0 has to be contained (C), fully matched (M), or Contain Match (CM) with multiple values separated each by a full stop.
5 – Contains words or characters separated by a full stop that cannot be included within the hardware entry.

Step-by-step using the card GeForce 8800 and the sample entry this works as follows.
0 – The parser scans the card name for a match of GeForce, as dictated by number 5 set to contain (C).

1, 2 and 3 - The most complicated part of the parser would be the version number, as detailed in values 1-3. The way this works is that the characters are stripped from the string and only the numbers are left. For example, the example entry is looking for a length of 4 after the characters are stripped out. A GeForce 8800 would be stripped to simply '8800'. A Geforce GTX 970 would be rejected at this point. This value then sets the base score for the parser by using the first one-two values, 88. This is then weighted by using the weight as found in 2 after the '=' sign, which is 0.1. Therefore the score thus far would be 8.8

5 – In this example, none of the values separated by a full stop will be found in the example.

After this, the rest of the entries are parsed. None of these entries would match so no score is added after this point, creating a final score of 8.8. If the card name were to contain GTX however, this would match both GTX and GT thus adding 1 score to the result, creating a final score of 9.8. If it were to contain M 3 score would be deducted from the result, creating a final score of 5.8

Benchmark CSV Data

The benchmark data scan uses CSV data and the GPU/CPU name to add score to the result. This is done via using two fields, one for the benchmark score and one for the device name.

The device name is stripped of both tm and ® symbols then compared to each entry using the device name field. Once a match is found, the benchmark score is then weighted via the weight value and added to the total score.

Sample data from Passmark that can be acquired for free has been included in this asset, we've stripped any data that would cause a conflict (including irrelevant markers and tm/® symbols).

By default, the weight is set to 0.001 for both GPU and CPU.

Unity 4 and below 5.3

Unity 4.5 is supported by AQHAT, however 2 things are not by Unity when using anything below 5.3.

The first, is providing processor frequency details, the second is detecting graphics multi-threading.

For processor frequency, we've provided 3 estimation variable you can set yourselves. This is where you can set a processor frequency per device platform type. These variables will not appear when on a version of Unity that does not need them.
They are as follows:

```
public float handheldProcessorEstimate = 1200;
public float consoleProcessorEstimate = 1900;
public float desktopProcessorEstimate = 2600;
```

For multi-threading, we have set this to be detected as false for Unity 4 as it is only supported in Unity 5 and above.

DEBUGGING

In order to debug the code we have provided several options, the first being a boolean that allows a debug overlay containing relevant data to be enabled.

The second is a set of mock variables that will spoof device names. This include GPU name, CPU name, the IOS/tvOS device name and 5 variables allowing you to override the detected hardware specs (GPU memory, memory/RAM, processor count and processor frequency).

For the first 3 values, leave these blank to keep disabled.
For the last 5 values, set these to -1 or less to keep disabled.

The IOS/tvOS device name will not kick in OS override and only changes the device name, the only system affected will be when IOS data is retrieved (hence the variable name).

If you want to spoof a particular OS, the easiest way is to switch platform in the build settings.

In order to spoof performance and compare it to score, we recommend using a virtual machine to limit hardware specification whilst running your game with the debug overlay enabled.

An example scene has been included to help with this, called ‘testScene.unity’.

Example debug screens, using default values found in prefab:

AutoQuality.cs

In order to make this module usable out of the box, we have included an example script 'AutoQuality.cs'.
This script works by placing a required score on each quality setting (within unity's default quality setting), then iterating through each and checking if the total or average score is less than until a match is found. Each value is the amount required to exceed that setting. For example, you have 6 quality settings, your second is set to 4 but your first is set to 2. A device scoring 1 will fall within the first quality band as it is less than 2 but a score of 2.1 will fall within the second as it is more than two but less than 4.

By default, this script is set to run on start. However this can be disabled and then ran later with the static method 'RunMe'. Use this line of code to call this method:

```
AutoQuality.RunMe();
```

You should make the number of quality bands match the number of quality settings you have defined within Unity. Any more or less and this could negatively effect the results of this module.

There are several other options you can configure in this script, such as;
Run on start – Allows you to define if this script should be ran on the start of the script.
Use average – Allows your to define if the script should use the average/mean score or the total score
Start delay – The initial function is set by an Invoke to allow the DeviceRating.cs script to finish, this defines the delay that this has. By default it is set to 0.1 seconds.

Lastly, if you set the quality manually later you can use this function to make the AutoQuality cache reflect this. So your user setting will be used upon launch of the application.
Run this line of code after setting the quality yourself:

```
AutoQuality.writeQuality(QualitySettings.GetQualityLevel());
```

CACHE

In order to reduce startup times on subsequent runs as this is a heavy script, AQHAT caches data from the first run and will use this unless it detects a change in the OS/hardware configuration.

This is achieved largely by the use of txt files stored under Application.persistentDataPath then within the required directory and file-name as set within the script.
This method has been used to be compatible with UWP apps as PlayerPrefs does not work with the UWP platform. However, tvOS doesn't work with this method, consequently AQHAT reverts to using PlayerPrefs on tvOS.
By default the sub-path is set to '/AQHAT/Data/cache/' and the file-name to 'ratingcache.txt' for DeviceRating.cs and 'qualitycache.txt' for AutoQuality.cs.
The cache can be disabled via the boolean 'useCache' on both AutoQuality.cs and DeviceRating.cs.

In order to reset the cache (can sometimes be required if your configuration was initially wrong or spoof data was cached), please use the static method 'wipeCache'. This can be called via the following line of code for DeviceRating.cs:

```
DeviceRating.wipeCache();
```

The following line for AutoQuality.cs:

```
AutoQuality.wipeCache();
```

EXAMPLE PREFAB AND SCENE

An example prefab and scene are included for your use, containing the default values for the system.
By default debug mode is enabled.

DONATIONS AND CONTRIBUTIONS

While our license allows you to use this is any many projects as you like, we would appreciate those that have found success in multiple projects to contribute a small amount to help with the upkeep of AQHAT.
We want to keep updating this software with new configurations for emerging hardware manufacturers and products but this takes time away from our other efforts.

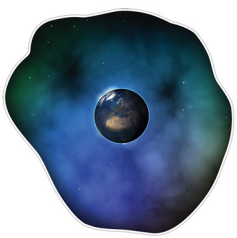
To this end, if you create an extensive experimental configuration that we may find useful we would appreciate you getting in touch via our website and letting us know.

If you find any bugs or ways to optimize our script, please get in touch and let us know.

Our contact form can be found here - http://eageramoeba.co.uk/Contact/

Lastly, if you feel like contributing a small amount of money, you can do this via paypal here - http://paypal.me/EagerAmoeba

Thanks for purchasing AQHAT, we hope it serves you well!



EAGER AMOEBA®