**Best Practices [draft]**

# **1. Setting Up Development Environment**

## **1.1. Git**

### **1.1.1. .gitignore**

Unity Project folder contains generated files, *Temp* folder with temporary files and *Library* folder with assets imported for the specific platform — these must be ignored in a *.gitignore* file. We recommend to use this [*.gitignore*](https://github.com/github/gitignore/blob/master/Unity.gitignore) file as a starting point.

### **1.1.2. Unity Project Configuration**

Before doing the first commit, make sure that the project uses the following settings in *Editor Settings* window *(Edit > Project Settings > Editor)*:

1. Asset Serialization: Force Text
2. Version Control: Visible Meta Files

This will force Unity to create a *.meta* file for each asset and use YAML to serialize files in the project. Read more about YAML format in the [Unity Manual](https://docs.unity3d.com/Manual/FormatDescription.html).

### **1.1.3. Git LFS**

[Git LFS (Large File Storage)](https://git-lfs.github.com/) makes working with large files more efficient. Depending on your project size you probably would want to set it up. Note that this will only work if you use GitHub or a git server that supports the Git LFS API.

### **1.1.4. Git Hooks**

As it was mentioned earlier for every asset in the project Unity creates a *.meta* file.

**FORGETTING TO COMMIT THIS FILE WITH THE NEWLY CREATED ASSET WILL RESULT IN BROKEN REFERENCES FOR EVERY OTHER TEAM MEMBER.**

To prevent this we recommend setting up pre-commit hooks, like the ones found [here](https://github.com/3pjgames/unity-git-hooks).

### **1.1.5. Merging YAML Files**

Generally we discourage the practice of several people working on the same asset at the same time. A proper workflow includes splitting the project into Scenes, Prefabs and ScriptableObjects — so merge conflicts are rare.

But we also have a nice tool which makes merging Unity assets much easier. It is called *UnityYAMLMerge* and is shipped with every Unity version. You can read more about setting up Git to use this tool instead of general text merge app at the [Unity Manual](https://docs.unity3d.com/Manual/SmartMerge.html).

## **1.2. Cache Server**

*The Cache Server* stores imported assets and speeds up initial import of project data, as well as platform switching within a project. *Unity 5.5* and later ships with built-in *Cache Server* which can be enabled in *Editor Preferences*. Within a team development environment we recommend to set up an instance of *Cache Server* on a remote machine in the network.

The latest version of *the Cache Server* is being maintained on [GitHub](https://github.com/Unity-Technologies/unity-cache-server/) as an open source project, and in time of writing the most up-to-date build is available on [the following branch](https://github.com/Unity-Technologies/unity-cache-server/tree/feature/streams-refactor) or can be installed using *NPM*:

npm install unity-cache-server@beta

Note, that NPM is bundled with a *NodeJS* install and *the Cache Server* requires *NodeJS v8.9.4*.

# **2. Assets**

## **2.1. Textures**

Textures often represent a large, if not the largest, fraction of a title’s memory usage. As such, the appropriate [per-platform compression settings](https://docs.unity3d.com/Manual/class-TextureImporterOverride.html) should be applied.

### **2.1.1. Color Format**

Visually examine whether a *16-bit* color format will suffice instead of a *32-bit* color format.

### **2.1.2. Compression**

Experiment with the texture compression settings per asset while checking the resulting size (shown in the inspector) to balance size and visual fidelity. For example, RGBA4444 may be sufficient for textures that require alpha gradients, whereas RGBA5551 may be acceptable for textures with cutout alpha areas. Textures without an alpha channel are good candidates for RGB565.

Use *Crunch compression*, if applicable. *Crunch* is a lossy compression format on top of DXT or ETC Texture compression. Textures are decompressed to DXT or ETC on the CPU and then uploaded on the GPU at runtime. *Crunch compression* helps the Texture use the lowest possible amount of space on disk and for downloads. *Crunch Textures* can take a long time to compress, but decompression at runtime is very fast. Read more about how *Crunch compression* works in [*this blog post*](https://blogs.unity3d.com/2017/12/15/crunch-compression-of-etc-textures/).

Note that *PVRTC* texture format on *iOS* devices and *ETC* format on *Android 4.x* devices require square textures. When compressing a non-square texture, two behaviours can happen:

* If not used by a sprite, and the compressed memory footprint is smaller than the one uncompressed, the texture will be resized based on the [npotScale](https://docs.unity3d.com/ScriptReference/TextureImporterSettings-npotScale.html) factor.
* Otherwise the texture will not be resized and marked as uncompressed.

### **2.1.3. Read/Write Enabled**

Make sure that *Read/Write Enabled* setting is unchecked if texture data doesn’t need to be accessed on CPU to avoid duplication.

### **2.1.4. Mipmaps**

Look into disabling *Mipmaps* on UI textures and other textures which do not benefit from having *Mipmaps*.

## **2.2. Meshes**

Model import settings provide a lot of scope for improving memory footprint and due to the decrease in size of the model data it will also help with loading times. The most pertinent settings for both memory and runtime costs are:

### **2.2.1. Read/Write Enabled**

This should almost be universally disabled. By having this setting enabled Unity will keep two copies of the mesh in memory and as such double the memory footprint of the model. This will also affect loading times due to the double allocation. The only time to keep it on is if you need to use the mesh as a mesh collider.

### **2.2.2. Import Blendshapes**

This should be disabled for every model that does not have blendshape information that will be used.

### **2.2.3. Normals and Tangents**

If normals and tangents are not used by the material for rendering then set them to *None* to save the cost of vertex information.

### **2.2.4. Optimise Game Objects**

This should also be enabled wherever possible. This setting changes how *Animators* structure their state machine and data internally, allowing for more optimal processing via multithreading. It also eliminates the creation of an internal *Transform* hierarchy representing a model’s structure, removing the need to copy data between the bones and *Transforms*. This data would usually be copied on the main thread, so removing this is a significant gain. Finally, this setting ensures that mesh skinning can be multithreaded; without it, the skinning is performed on the main thread. The Transform hierarchy is also flattened, making calculation of global rotation and position properties less expensive.

For those transforms which actually need to be updated directly, add them to the list of *Extra Transforms to Expose* in the model import settings.

### **2.2.5. Animation Type**

Set *Animation Type* to *Generic* for models that do not require *Animators*.

### **2.2.6. Animation Compression**

Set *Animation Compression* to *Optimal*, which will choose the most appropriate method of keyframe reduction and compression for the data.

High-level information on animation compression can be found in the [Unity Manual](https://docs.unity3d.com/Manual/FBXImporter-Animations.html). There is also a good third-party [blog post](http://nfrechette.github.io/2017/01/30/anim_compression_unity5/) about Unity’s animation compression.

## **2.3. Prefab Links to Source FBX Files**

Presently, when an FBX is used within a Prefab, the link to this FBX is broken. This means that changes made to the FBX will not propagate to the Prefab containing an instance of this *FBX*.

However, in the meantime, the [Unity FBX Exporter](https://assetstore.unity.com/packages/essentials/fbx-exporter-beta-101408) can be used to solve this problem. It requires *Unity 2017.1+* and (optionally) *Maya 2017+* or *3ds Max 2017+*. It is still considered to be in beta and is evolving rapidly. You can find documentation and community support for the package on [the Forums](https://forum.unity.com/threads/new-fbx-exporter-in-2017-2.499889/).

The *FBX Exporter* package supports a minimal workflow for converting a GameObject hierarchy into a *Linked Prefab*. A *Linked Prefab* is automatically updated whenever the FBX files change on disk.

Use *Convert To Linked Prefab (GameObject > Convert To Linked Prefab Instance)* to replace the GameObject hierarchy with an instance of a Prefab that is linked to an FBX Model. The *Linked Prefab* contains a *FbxPrefab* script component that merges FBX Model changes into the *Linked Prefab*. *Convert To Linked Prefab* exports each selected GameObject hierarchy as a *.prefab* and a *.fbx* file.

Whenever an FBX file is changed on disk and the FBX file resides within the *Assets* folder of the Unity project, the built-in *FBX Importer* runs and updates the geometry, the linked prefabs is updated accordingly.

Note if a merge conflict occurs between a changed FBX file and a *Linked Prefab*, the FBX file takes precedence.

This workflow also allows you to have FBX changes be applied automatically to manyprefabs using this FBX. Take for example, you have a *Red Crystal* collectible Prefab and a *Green Crystal* collectible Prefab. Both these Prefabs can have a reference to the same Crystal FBX through the Fbx Prefab script that comes with the exporter. When the FBX changes on disk, the changes will automatically be applied to both Prefabs using this given FBX.

## **2.4. Audio Assets**

To make sure that audio doesn’t contribute to high Memory usage and has minimal CPU overhead, use the following settings:

1. Force audio clips to *mono* when they do not require stereo sound.
2. Small (<200KB) audio clips should be set to *Decompress on Load*. Internally, Unity allocates a 200KB playback buffer for audio decompression. Setting audio clips smaller than 200KB to remain decompressed in memory after load saves memory at playback time.
3. Large clips (>1MB) should be set to *Streaming*. Note that only one audio clip can be streamed at any given time.
4. Other clips should be set to *Compressed into Memory*.

Various platforms also have preferred compression settings:

1. On Android, *Vorbis* should be used for all clips to fully utilize the underlying built-in hardware decompression.
2. On iOS, the format should be set to *ADPCM* for shorter clips, and *MP3* for longer clips.
   1. Audio clips using *MP3* take advantage of the iOS hardware decoder, and as such, we advise that longer audio clips use this format.
   2. *ADPCM* offers a fixed 3.5:1 compression ratio and is very inexpensive to decompress. For very short sound effects, *ADPCM* is a more suitable choice.

## **2.5. Resources Folders**

In Unity all the folders named *Resources* are built into the game and all the assets from these folders can be loaded by name using [Resources.Load()](https://docs.unity3d.com/ScriptReference/Resources.Load.html) API.

While this is rather convenient, we do not recommend using *Resources* folders for games beyond a prototyping stage, other than in very simple use cases. Use of *Resources* folder can lead to increased load times and memory usage, and when used in conjunction with *Asset Bundles* can lead to an increased risk of duplicated assets at runtime.

More detailed information about *Resources* folder can be found in the [*Asset Bundles and Resources Best Practice Guide*](https://unity3d.com/learn/tutorials/topics/best-practices/guide-assetbundles-and-resources?playlist=30089).

## **2.6. Removal of Redundant or Unused Assets**

In general, any unused code, references, or plugins should be removed from the project. While the compiler will attempt to strip unused code from the final assembly, references are left alone. The *Assembly Browser* in MonoDevelop or the *Object Browser* in Visual Studio can be used to inspect a library’s reference dependencies to identify duplicates. Doing so will also help with improving overall compile times.

Similarly, it may be worth checking for duplicate dependencies across the plugins the project utilizes (e.g., like JSON parsing). This can be somewhat mitigated with access to the plugin source to remove duplicate dependencies. More information about optimizing application binary size can be found in the [IL2CPP Build Size Optimization](https://support.unity3d.com/hc/en-us/articles/208412186) knowledge base article.

Removing redundant assets from the project will reduce build size and deployment times, improving your workflow iterations. We suggest reviewing the [Asset Auditing best practice guide](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity4.html) for a more complete summary of our recommendations.

## **2.7. Asset Auditing Tooling**

Asset import settings are sometimes changed accidentally or unknowingly, which can be avoided by automating the audit process. We recommend looking into creating tools that leverage the [*AssetPostprocessor API*](https://docs.unity3d.com/ScriptReference/AssetPostprocessor.html) to automate or force-check asset import settings to conform to the provided guidelines.

Mark Harkness, one of our Enterprise Support engineers has been working on [a tool for asset auditing](https://github.com/MarkUnity/AssetAuditor) which you can use in your project.

## **2.8. Build Size**

To stay within limits of *Over The Air* download in App Stores, it is crucial to minimize the final build size of the game. Asset auditioning is a very important step in achieving this goal.

Typically, the application size is often more of an issue on iOS than Android, due to the inclusion of 64-bit code as well as 32-bit code. For this reason, we just wanted to highlight the news that [as of September 19th, Apple have increased the Over-The-Air limit from 100MB to 150MB](https://developer.apple.com/news/?id=09192017b).

### **2.8.1. Build Report Tool**

The most effective tool for discovering size reduction opportunities is Unity’s prototype *Build Reporting Tool*. This tool is a small webpage that connects to the Unity Editor and displays statistics on included resources and generated code size.

In the time of writing, the tool was available on Unity’s file server: <http://files.unity3d.com/build-report/>

To use the tool, first create a Unity build (development or release) via the Unity Editor. After the build is complete, keep the Unity Editor open. The Editor caches information related to the most recently completed build, but this information is flushed when the Editor is closed.

After completing the build, open the above link in a web browser and click the *Connect* button. This will establish a local HTTP connection to the Unity Editor, which will then transmit the raw build report to the web app. The web app will then visualize the data received from the Unity Editor.

# **3. Asset Bundles**

Currently the best approach to splitting a game into manageable parts and reducing initial download is using *Asset Bundles*. Please refer to [The Guide to Asset Bundles and Resources](https://unity3d.com/learn/tutorials/topics/best-practices/guide-assetbundles-and-resources?playlist=30089) in [Best Practices section](https://unity3d.com/learn/tutorials/s/best-practices) to learn more on this topic.

## **3.1. Grouping assets into Bundles**

A good strategy for deciding what assets to group into a single bundle is to look at which assets are commonly used in tandem. For instance, a character model and its character-specific animations might be found in a single bundle, but commonly-used animations may be bundled separately. If the team is considering a bundle strategy that involves a multitude of small bundles, keep in mind that bundles incur the following estimated memory overhead:

1. 8KB for header data
2. 16KB for loading buffers, which persist after loading completes
3. 72KB for SerializedFile dependency list, if bundle has dependencies
4. 50KB+ varied for Type Tree data

## **3.2. Tools**

### **3.2.1. Asset Bundle Browser**

[Asset Bundle Browser](https://github.com/Unity-Technologies/AssetBundles-Browser) allows developers to manually specify which asset goes to which bundle and helps to find potential issues with duplicated assets across bundles. More detailed information can be found the [Asset Bundle Browser manual](https://docs.unity3d.com/Manual/AssetBundles-Browser.html).

### **3.2.2. Asset Bundle Graph**

TBD

## **3.3. Assets**

One of the biggest issue with *Asset Bundles* is the ease of packing duplicated data into different bundles. This is especially bad when the data being duplicated is textures because textures tend to be the heaviest assets in a game. Usually this is due to each bundle pulling in all implicit dependencies. Developers should explicitly group dependencies into bundles, in accordance with [Section 4.3 of the Asset Bundle Best Practice Guide](https://unity3d.com/learn/tutorials/topics/best-practices/assetbundle-usage-patterns?playlist=30089). In some cases developers may want to intentionally duplicate assets in multiple bundles for ease of loading and asset management. In these cases just verify your intended duplicate assets using the *Asset Bundle Browser* tool and be cautious not to duplicate asset in memory via loading from two separate bundles at the same time.

### **3.3.1. Shaders**

Shaders should be stored in their own bundle, or in a set of bundles, in order to leverage the dependency system to avoid duplicate runtime instances. We recommend to avoid using any built-in shaderin assets that are bundled, as these shaders will be duplicated at *Asset Bundle* build time and runtime. If a built-in shader must be used, download it from the Unity website and include it in your project via *Asset Bundles*.

## **3.4. Usage**

In general, there are a few rules of thumb about Asset Bundles usage:

1. Don’t use the IgnoreTypeTree option, as it will increase risk of incompatibility.
2. For versioning, don’t rely on the hash that Unity generates for the *Asset Bundle*, as they are only estimates. Although slower, using CRC for this purpose is preferred.
3. When unloading *Asset Bundles*, you should avoid using [AssetBundle.Unload(false)](https://docs.unity3d.com/ScriptReference/AssetBundle.Unload.html), as it increases the user’s burden of tracking and destroying loaded objects, which [AssetBundle.Unload(true)](https://docs.unity3d.com/ScriptReference/AssetBundle.Unload.html) would handle automatically.

### **3.4.1. Compression**

#### **WebGL**

For WebGL games, it is generally most optimal to compress *Asset Bundles* with the *LZ4* algorithm instead of *LZMA*. If additional compression is desired, it is possible to configure the CDN serving the *Asset Bundles* to further compress them with *gzip* or *Brotli* compression, which can then be transparently decompressed by the browser before the *Asset Bundle*’s bytes are delivered to Unity.

However, as the WebGL standard requires support for *DXT* texture compression, it is possible to compress all textures with the *Crunch* algorithm and export them to in *DXT1* or *DXT5* format. Textures compressed with *Crunch* generally are already near-optimally compressed and recompression with *LZMA* would not result in appreciably improved compression ratios.

Therefore, it is generally advisable to separate *Crunch*-compressed textures into their own uncompressed *Asset Bundles*.

### **3.4.2. Loading**

There are two primary benefits to choosing UnityWebRequest for all asset bundle downloads. The first improvement is that UnityWebRequest can stream downloaded bytes to disk, instead of keeping all bytes in memory until the bundle has finished downloading. This prevents a spike in native memory that can be dangerous for memory pressure, depending on the size and total number of bundles being downloaded in parallel.

The second benefit is that when shipping bundles with *LZMA* compression (which is the default), UnityWebRequest will recompress the asset bundles to *LZ4* compression to improve load times from the cache. Please read [this Unity Manual page](https://docs.unity3d.com/550/Documentation/Manual/AssetBundleCompression.html) for more details about *Asset Bundle* compression and downloads. For unencrypted *Asset Bundles*, developers can start to receive this benefit using UnityWebRequest.

#### **WebGL**

While it is possible to use the built-in WWW / UnityWebRequest APIs to transmit HTTP requests and download Asset Bundles on WebGL, it is preferable to use[*CachedXMLHttpRequest*](https://assetstore.unity.com/packages/essentials/tutorial-projects/cachedxmlhttprequest-71538) plugin from Asset Store, which is maintained by Unity’s WebGL team. This plugin uses the CachedXmlHttpRequest API, which permits the browser to natively handle sending and receiving data and is generally more efficient than the transpiled versions of Unity’s built-in HTTP communication APIs.

## **3.5. Other topics**

### **3.5.1. Encrypting Asset Bundles**

If the bundles must be encrypted (and therefore decrypted on load), we recommend:

1. Using *uncompressed* bundles or *LZ4 compression* to avoid allocating memory for decompressed content.
2. Loading the bundle with [AssetBundle.LoadFromStream()](https://docs.unity3d.com/ScriptReference/AssetBundle.LoadFromStream.html) API (available since *Unity 2017.2*) to skip writing decrypted data to disk or storing it in memory.
3. For encryption, use *SHA256* or higher**,** which can be found in the standard .NET cryptography libraries.

## **3.6. New Asset Bundle API**

In Unity’s 2018 releases, the build pipeline for *Asset Bundles* is being upgraded. A new, C#-driven system will be released in experimental mode in *2018.2*, and initial tests show it as 10% faster than the existing system in large-project use cases. This upgrade also includes a few new low-level C++ APIs.

[The Addressables system](https://docs.google.com/document/d/1hPLNLdrF0qAvjEJTpKf-cuO_d4FCV0H2cqBeP1Zo6mA/edit) is going to make working with *Asset Bundles* much easier and less error-prone.

# **4. Profiling**

## **4.1. Profiler**

*The Unity Profiler* built into the Editor is a convenient way to observe spikes in CPU usage, GC collection frequency, and the relative performance cost of rebuilding UI canvases. It’s critical to note that while profiling in the Editor is convenient and allows for rapid iteration, it is no substitute for profiling on device, which will always have different performance characteristics.

Note that Unity shows in *the Profiler* only data it can gather, this means that 3rd-party plugins consuming CPU and allocating memory are invisible for it.

### **4.1.1. Timeline View**

*The Timeline View* allows you to see if the main thread is waiting on other threads, such as the *Render Thread* or *Job Threads*. This can help you track down performance problems. Read more about using *the Timeline View* in the [Unity Manual](https://docs.unity3d.com/Manual/ProfilerCPU.html).

### **4.1.2. GC Allocations**

Managing the size and usage of the *Managed Heap* is important for performance in a Unity project. This tool allows you to identify where *Managed Heap* allocations are occurring in your code, which is extremely useful. Using the *GC Alloc* column to pinpoint and eliminate as many *Managed Heap* allocations as possible is highly recommended for any Unity project.

Enabling *Deep Profiling* is particularly useful when identifying GC allocs, as this can help to see exactly where the allocations are coming from. However, please note that at present *Deep Profiling* is available only within the Unity Editor.

Please note that when profiling a game that is running within the Unity Editor, you may see some allocations that do not occur when the game runs outside of the Unity Editor and should be discounted. For example, [GetComponent()](https://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html) causes an allocation in the Unity Editor only.

An alternative to *Deep Profiling* is to use [Profiler.BeginSample()](https://docs.unity3d.com/ScriptReference/Profiling.Profiler.BeginSample.html) and [Profiler.EndSample()](https://docs.unity3d.com/ScriptReference/Profiling.Profiler.EndSample.html) APIs to hone in on where the allocations are coming from.

### **4.1.3. User Thread Profiling**

Starting with Unity *2017.3* developers can now use the [Profiler.BeginThreadProfiling()](https://docs.unity3d.com/2017.3/Documentation/ScriptReference/Profiling.Profiler.BeginThreadProfiling.html) API to declare custom threads for the Unity profiler to measure.

### **4.1.4. Profiler Scripting**

For various purposes it might be needed to access profiling data in the editor. There are API’s for this hidden in the UnityEditorInternal namespace. These API’s can be used freely without reflection but are subject to being changed without notification between Unity versions. The following snippet demonstrates the how to get the statistics information from the profiler:

var s = ProfilerDriver.GetAllStatisticsProperties();

for (int i = 0; i < s.Length; i++) {

Debug.Log(s[i] + " " + i + " " +

UnityEditorInternal.ProfilerDriver.GetFormattedStatisticsValue(

UnityEditorInternal.ProfilerDriver.lastFrameIndex, i)

);

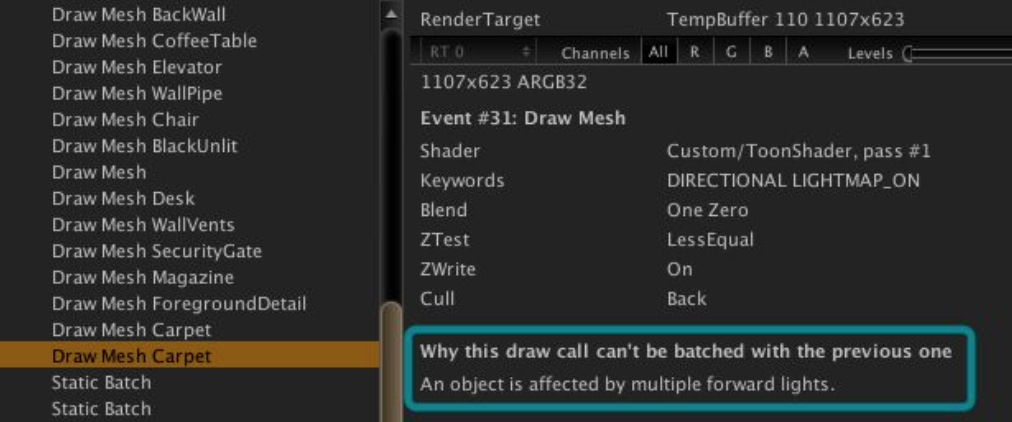
}

This would represent all the statistical information that would be displayed in *the Profiler.* You could distill this down to only the pieces of information that you wanted to record by noting the index for the information that desire and only accessing it.

## **4.2. Frame Debugger**

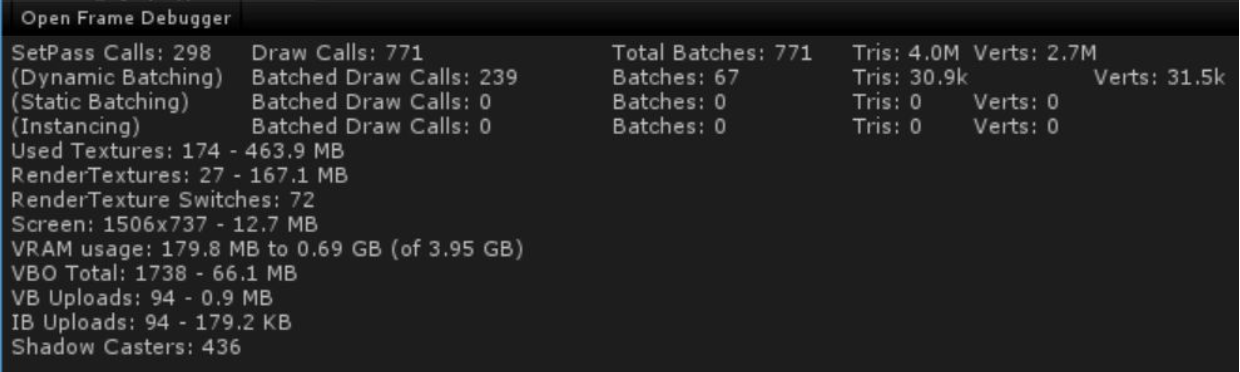
*Draw Call Batching* is an effective technique used by Unity to reduce GPU state changes, minimizing resource-intensive operations in the graphics driver. In order to be batched, GameObjects must meet certain requirements (e.g., they must share the same material). The *Unity Frame Debugger* can be used to iterate through the various draw calls used to construct a single frame.

Since *Unity 5.6* *the Frame Debugger* will also display the reason why the currently-selected draw call could not be batched with the previous one:



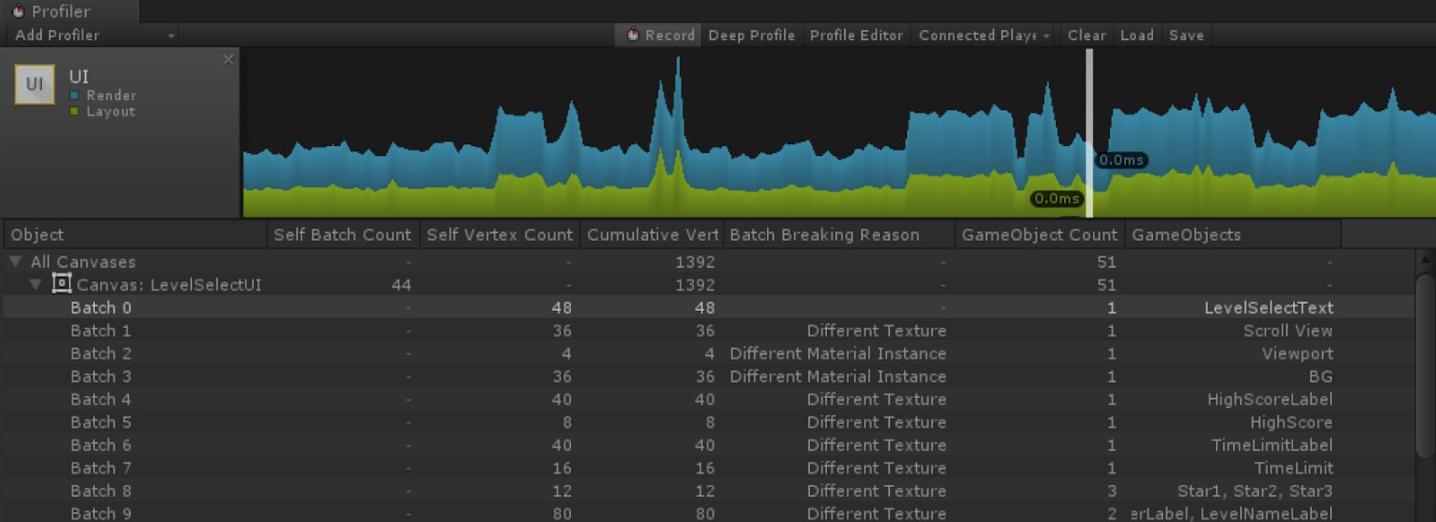
Read more on why Unity might start a new batch in [this blog post](https://blogs.unity3d.com/2017/04/03/how-to-see-why-your-draw-calls-are-not-batched-in-5-6/).

The *Rendering* section row in *the Unity Profiler* will also display the total number of batched draw calls for the current frame:



## **4.3. UI Profiler**

*Unity 2017.1* adds a new UI profiling component in *the Unity Profiler*. *The UI Profiler* can be used to identify reasons why UI draw calls could not be batched:



In general, game titles should aim to have as few batches as possible to reduce the cost of draw calls and GPU state changes.

Available graphs:

1. *Layout*— shows how much time was spent on generating layouts for UI elements. The auto layout system provides ways to place elements in nested layout groups such as horizontal groups, vertical groups, or grids. It also allows elements to automatically be sized according to the contained content. The auto layout system is flexible, but comes for a price. Layout components are relatively expensive, as they must recompute the sizes and positions of their child elements each time they are marked dirty.
2. *Render*- shows how much time it took for UI to render.

### **4.3.1. Batch breaking reasons**

Each batch has information about why UI has terminated it and started a new batch. Here is the list of possible reason for that:

1. *NotCoplanarWithCanvas*— UI element has a different Z coordinate and that caused a break. In these cases UI don’t know if something should be batched or not due to size dependencies.
2. *CanvasInjectionIndex*— Whenever you have nested canvases and UI needs to display canvas between other UI elements, batch will be broken. For example if nested canvas needed to display the popup of a dropdown combobox, we have to break the previous batch to ensure that the popup is on top of the drop down itself and its siblings.
3. *DifferentMaterialInstance*— Whenever UI elements has a different material, we need to create a new batch with that material to render it separately.
4. *DifferentClipRect or DifferentRectClipping*— When we encounter element which is clipped by different clipping rect or it is a first element that uses clipping rect, this breaks the batch.
5. *DifferentTexture or DifferentA8TextureUsage*— Whenever UI element has a different texture, we need to create a new batch with that texture to render them separately.

## **4.4. Memory**

The *Memory* section of *the Unity Profiler* window is the only tool which is able to correctly display the current size and usage of the managed heap. To find this information, look at the *Mono Used* and *Mono Reserved* sizes in *the Unity Profiler*.

Unfortunately, it is of limited use when diagnosing memory issues on iOS builds, because *the Unity Profiler* does not reflect actual memory pressure on the device.

### **4.4.1. ShaderLab Allocations**

When shaders compile at runtime they can expand into many variants due to #pragma multi\_compile or shader\_features. Developers can determine if their shaders are taking large amounts of memory at runtime by investigating their editor log at build time.

Here is an example of what developers will want to look for:

Compiled shader 'TEST Standard (Specular setup)' in 31.23s

d3d9 (total internal programs: 482, unique: 474)

d3d11 (total internal programs: 482, unique: 466)

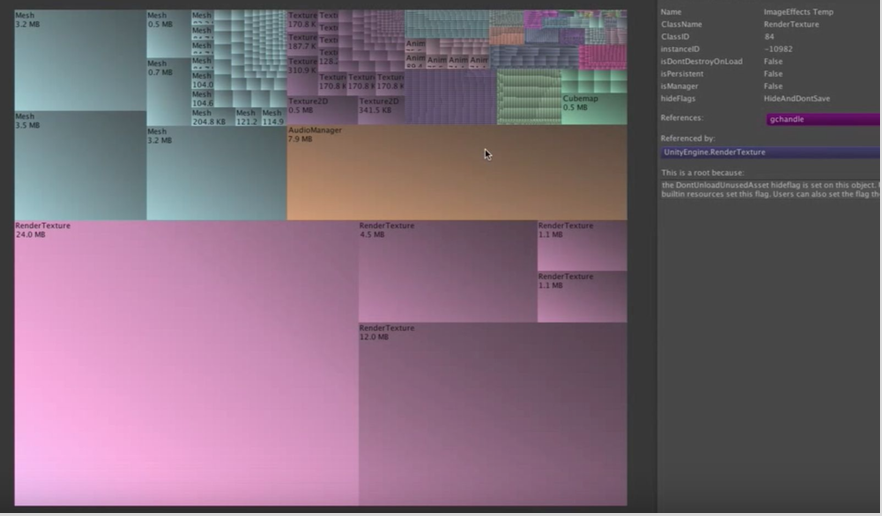
metal (total internal programs: 482, unique: 480)

glcore (total internal programs: 482, unique: 454)

In this example a shader expands into a lot of variants (due to #pragma multi\_compile or shader\_feature), *482* of them are actually included into game build. This shader will be compressed and will be roughly the sum of *compressed* sizes: 0.14 + 0.12 + 0.20 + 0.15MB = 0.61MB. At runtime, the compressed data is kept in memory *(0.61MB)*, and the data for the currently used graphics API (e.g. Metal) is uncompressed *(2.56MB in this case)*.

## **4.5. The Memory Profiler**

One of the less known tools is *Memory Profiler* which is available on [BitBucket](https://bitbucket.org/Unity-Technologies/memoryprofiler).



This tool will only track the memory allocated by Unity engine, and from user scripts when using the *IL2CPP* back-end. If any third party tool uses their own allocator, this tool will not be able to track it.

Additionally, it currently does not track allocations performed on worker threads, so be mindful if your title directly creates threads to perform certain tasks. This will be addressed in a future version of the tool.

Note that *Unity 2017.3* has added support for showing memory allocations from user scripts when using the Mono back-end. However all relevant allocations should be visible with older Unity versions when using *IL2CPP* builds.

Also the *ClassID* value shown in the upper right for the currently selected object no longer matches the native *class IDs*, which are documented in our [*Class ID Reference*](https://docs.unity3d.com/Manual/ClassIDReference.html). This should be ignored, and will be removed in a future version of the tool. The class name is correct, however.

# **5. Debugging**

TBD

## **5.1. Android**

### **5.1.1. Crashes**

*Dave Hampson*, one of our support engineers wrote [a tool](https://bitbucket.org/davehampson/symbolicate) that will allow you to symbolicate the release crash dumps that you get from Google.

It does not give you the full line numbers from the call stack but it will allow you to see the method names and give you a much better idea of what could be going wrong. Full instructions for its use are on the link previously provided.

# **6. Optimization**

## **6.1. Improving Startup and Load Times**

A popular strategy to reduce (perceived) loading time on startup is to have your first scene be a completely empty scene, with a single script that synchronously loads the *“real”* first scene on Start. As only resources for the first scene *(which should be kept to a minimum)* are loaded, this approach gives you the ability to display a small level of interactivity while the next scene is loaded.

### **6.1.1. Scenes & Prefabs**

During a loading operation, two major operations consume time — either on the loading thread or on the main thread, depending on whether the load is asynchronous or not. One is *file I/O*, primarily consisting of read and seek operations as the *SerializedFile* system loads bytes and seeks the file pointer to load specific properties or Objects.

The other is *deserialization* time, where the incoming byte stream is converted into a properly-structured Object in memory. This is denoted by the invocation of a Transfer function on some base class: MonoBehaviour::Transfer is used to deserialize the properties of a MonoBehaviour, Transform::Transfer deserializes the properties of a Transform, and so on.

Some savings can be achieved by minimizing the number of GameObjects, Transforms and MonoBehaviours inside a given Scene or Prefab.

When scenes are loaded additively and asynchronous, one simple way to improve the loading times is to adjust the value of [Application.backgroundLoadingPriority](https://docs.unity3d.com/ScriptReference/Application-backgroundLoadingPriority.html). This can be set to high value during loading screens so that the content can be loaded as fast as possible. During gameplay it can then be returned to a lower value so that loading will not adversely affect gameplay. Another place where the async loading can be controlled is from the *Quality Settings*. The two parameters [asyncUploadTimeSlice](https://docs.unity3d.com/ScriptReference/QualitySettings-asyncUploadTimeSlice.html) and [asyncUploadBufferSize](https://docs.unity3d.com/ScriptReference/QualitySettings-asyncUploadBufferSize.html) can be used to tweak how much time async loading operation can use to integrate assets into Unity main thread.

### **6.1.2. Script Initialization**

When a scene is loaded, prior to the first update of the screen:

1. [Awake()](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html) is called for all MonoBehaviours (even disabled) on active GameObjects,
2. [OnEnable()](https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnEnable.html) is called for all enabled MonoBehaviours,
3. After all [Awake()](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html) and [OnEnable()](https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnEnable.html) methods for all MonoBehaviours are executed, before the first [Update()](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html) call all [Start()](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html) methods are called.

We recommend to avoid having heavy logic in these methods to minimize perceived loading time.

### **6.1.3. Audio**

Delaying audio files loading could be beneficial to loading time. In order to achieve that, the audio clip must be saved with the flag [AudioClip.preloadAudioData](https://docs.unity3d.com/ScriptReference/AudioClip-preloadAudioData.html) set to false and the audio clip can be loaded later using [AudioClip.LoadAudioData()](https://docs.unity3d.com/ScriptReference/AudioClip.LoadAudioData.html). If you want the loading to be done asynchronously, you will also need to set the [AudioClip.loadInBackground](https://docs.unity3d.com/ScriptReference/AudioClip-loadInBackground.html) flag to true in order to avoid blocking the main thread.

### **6.1.4. Physics Colliders**

A small saving in loading time might be had by checking the *Prebake Collision Meshes* option in the *Player Settings*. Normally, Unity’s loading thread will build a PhysX mesh representation for a collider. This does not impact the main thread’s performance, but does increase loading time. Prebaking the collision meshes will reduce loading time, at the expense of larger asset bundles, since mesh data will now include the PhysX mesh representation. Note that collision meshes shared between objects, including those which are scaled differently, will only be computed once, so the saving in loading time will depend on the number of unique collider meshes and their complexity.

### **6.1.5. Asset Bundles**

Leveraging *Asset Bundles* effectively will greatly assist with improving load times as well as reducing memory impact. Proper usage of *Asset Bundles* typically requires significant planning of resource layout and management. Amortizing *Asset Bundle* loading – that is, loading only assets required for the first scene – is also an effective strategy to reduce startup times.

## **6.2. Scripting Optimizations**

The [Understanding Optimization](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity.html) guide provides a general overview of the process of optimizing scripting performance. This section includes additional important topics.

### **6.2.1. Consider When to Derive from MonoBehaviour**

There are many cases when game code classes should not be derrived from MonoBehaviour. When making architectural decision about what functionality should go into classes derived from MonoBehaviour you should make sure that you understand performance implications. Here are several examples.

#### **Pure Data Classes**

*Pure Data* classes (also known as *Value Objects*) are classes which are only used to store data. For CPU performance reasons, pure data classes should not derive from MonoBehaviour. Comparing two UnityEngine.Objects (MonoBehaviour, GameObject, etc.) to each other, or to null, is slower than one may expect. Unity overrides the == operator for UnityEngine.Objects operands that does not compare memory addresses. Instead, the == operator accesses the native object associated to the managed wrapper object using the wrapper object’s instance ID.

As this roundtrip can be costly, it’s preferable to only inherit from MonoBehaviour when needed. For pure data classes, static singleton classes or ScriptableObjects will often suffice for such needs.

#### **Classes Which Don’t Rely on Transforms Hierarchy**

TBD

#### **Update() Method Usage**

To call [Update()](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html) method on a component derived from MonoBehaviour Unity needs to cross the border between *Native* code and *Managed* code which incurs some overhead. If the game uses hundreds or thousands of such components, this overhead starts to be very noticeable. You can find more detailed description of how this works in [*the following blog post*](https://blogs.unity3d.com/2015/12/23/1k-update-calls/).

Generally we recommend to create a custom global *Update Manager* to tick all required components.

#### **Abstract MonoBehaviour Extensions**

Working on a components derived from MonoBehaviour in a modern IDE is a bit uncomfortable because an IDE usually fails to understand the usage of special Unity methods like [Start()](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html) and [Update()](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html). There’s a temptation to create an *abstract class* with all the special Unity methods as virtual functions to make IDEs happier. The issue with this approach is that **ALL** classes derived from MonoBehaviour in the project will have [Update()](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html) methods and will be ticked by the engine (see the previous section).

### **6.2.2. Accelerometer**

Most of Unity titles don’t use *Accelerometer*, but by default it is set to be active in the *Player Settings* at *60hz*. Disabling this will not only save CPU time but will also save some battery life in the title. It is easy to track down accelerometer usage in the *Instruments Time Profiler*.

### **6.2.3. Assembly Definition Files**

*Unity 2017.3* introduced [*Assembly Definition Files*](https://blogs.unity3d.com/2017/11/22/unity-2017-3b-feature-preview-assembly-definition-files-and-transform-tool/) which allow a project to define its own *managed assemblies* and assign user scripts to them on a per-folder basis. In turn, this should result in faster iteration times, since only those assemblies actually affected by script changes will be built.

Note that, while having multiple assemblies does grant modularity, it can also increase the application’s binary size.

### **6.2.4. Runtime Mesh Generation**

Updating and [*uploading*](https://docs.unity3d.com/ScriptReference/Mesh.UploadMeshData.html) mesh data is expensive. There are strategies to decrease CPU load on these operations.

For example, since iOS devices are based on an *unified memory scheme*, working on data shared between CPU and GPU is hard to get right. One recommended strategy to avoid being blocked on CPU while GPU still has a handle over the vertices is *to double buffer* meshes that are changed at high frequency. This should alleviate the issue by working on a different mesh each frame. For further information about how vertex data is handled on iOS devices, we recommend the read of [*Apple's Best Practice Guide For Working With Vertex Data*](https://developer.apple.com/library/content/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html).

Another area of optimization would be to do less work each frame:

1. Avoid calling [Mesh.UploadMeshData()](https://docs.unity3d.com/ScriptReference/Mesh.UploadMeshData.html). The internal implementation in this context would force the mesh to be recreated every frame and vertices/indices buffers to be reallocated or resized. *Double-buffering* the generated mesh would remove the need to sync the data between frames thus the need to call this function.
2. Assign bounds manually. Calling [Mesh.RecalculateBounds()](https://docs.unity3d.com/ScriptReference/Mesh.RecalculateBounds.html) is expensive and could potentially trigger heavy temporary allocations. If you are in complete control of the vertices you create, you could hold track of the min-max bounds and simply assign the result to [Mesh.bounds](https://docs.unity3d.com/ScriptReference/Mesh-bounds.html).
3. Do not update *invisible/off-screen* objects. In order to alleviate the work on the CPU, it is recommended to remove mesh generation for elements that are not on screen.

### **6.2.5. C# Features**

#### **A. Closures and Anonymous Methods**

A closure is generated when an *anonymous function or lambda* references one or more variables contained within its enclosing scope, that is, they aren’t explicitly passed to the function as parameters nor are declared inside the function itself. In this case, the function is said to *‘close over’* the variables.

In such cases, the compiler will generate a class containing the code from the anonymous function or lambda, as well as references to the variables being captured. The instantiation of this class results in a GC allocation. When the code containing the variables is out of scope, the variables themselves will not be destroyed until such time as all closures referencing them are destroyed.

As method references are allocated on the heap, the use of both closures and anonymous methods should be minimizedas much as possible (especially on a per-frame basis). Read more about technical reasons for avoiding these in [*Understanding the managed heap*](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html) section of [*Understanding optimization in Unity*](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity.html) guide.

#### **B. LINQ**

*LINQ*is a common cause of managed-memory allocations and non-performant code. Due to its ease of use, it is fairly easy to unintentionally create allocations and slow-running loops. Thus, it should be avoided as much as possible.

#### **C. Strings**

*Microsoft* has published [*extensive best practice guidelines*](https://docs.microsoft.com/en-us/dotnet/standard/base-types/best-practices-strings) on the usage of strings in C#.

Prefer integer or bitmask comparisons over string comparisons when possible, as the latter is far less efficient. If you do need to compare strings, ensure that you use *ordinal comparisons* when comparing strings that do not contain user input. *Ordinal comparisons* perform simple byte-by-byte checks, akin to C/C++'s native string comparison methods, instead of requiring expensive culture-sensitive multibyte comparisons. By default, C# string comparisons perform culture-sensitive comparisons.

#### **D. Properties**

*Using public variables in place of properties*will reduce time spent for stackframe allocation and teardown. If properties are preferred for code safety reasons, #define directives can be used to manage preferred code paths in release and development builds. For example:

#if DEVELOPMENT\_BUILD  
 private int health;  
 public int Health { get { return health; } }  
#else  
 public int Health;  
#endif

#### **E. Boxing**

The managed runtime allocates memory when boxing occurs, which will need to be collected later by the garbage collector. This will introduce CPU performance hitches, as all script execution code is paused during garbage collection passes. Furthermore, the allocations added from boxing further fragment the managed heap, increasing memory usage by the application. For these reasons, care should be taken to minimize boxing operations as much as possible.

For more information about boxing, managed memory, and Unity’s Garbage Collector, please refer to the section on [*Understanding the managed heap*](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html)in the best practice guide [*Understanding optimization in Unity*](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity.html).

#### **F. Foreach Loop**

Historically, we considered foreachwasteful, as it created GC garbage every time it was used. The negative impact of these allocations, particularly if generated every frame, generally outweighed the convenience of the language feature. **Unity 5.6 has solved this problem for some containers, but not all.**

|  |  |  |
| --- | --- | --- |
| ***Container*** | ***Initial iteration*** | ***Subsequent iterations*** |
| Value[] | No | No |
| List<> | No | No |
| ArrayList | Allocates | Allocates |
| Dictionary | Allocates | No |

Furthermore, all calls to <container>.GetEnumerator() will always allocate. Foreach statements can usually be easily rewritten to use a cheaper for-loop.

The *"hidden"* allocations that occur when using foreach normally occur due to boxing (or constructing) the enumerator or the values themselves in the case of value types.

### **6.2.6. Unity APIs**



Some Unity APIs are either slow or allocate Managed memory. Usage of these APIs should be avoided in performance critical areas. Please consult the best practice guide: [*Understanding Optimization in Unity*](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity7.html)for details.

#### **A. GameObject.AddComponent()**

[AddComponent()](https://docs.unity3d.com/ScriptReference/GameObject.AddComponent.html) can be a very expensive operation to perform during frame time sensitive moments. [AddComponent()](https://docs.unity3d.com/ScriptReference/GameObject.AddComponent.html) will have variable cost depending on the type of component getting added. Adding a component can be slow for the following reasons:

1. Unity must allocate the memory required for the Component.
2. Next, Unity messages all of the components of the GameObject to let them know that a new component has been added. Each component will then have its own code which may run depending on what type of component was added and how they might interact. This cost will scale with the total number of components on the GameObject as well as the types of interactions of those components (e.g.a *Rigidbody* component needs to know if a *Collider* has been added).
3. Lastly, Unity will need to run any code to appropriately awake the new component.

Note that adding MonoBehaviours has additional overhead not listed here when added at runtime.

We recommend instantiating objects from prefabs. If there is no choice that makes sense other than composing an object at runtime, we recommend that you create a template object that you can clone.

#### **B. GameObject.GetComponents()**

As of *Unity 5.3*, there are overrides of the [GetComponents()](https://docs.unity3d.com/ScriptReference/GameObject.GetComponents.html) and [GetComponentsInChildren()](https://docs.unity3d.com/ScriptReference/Component.GetComponentsInChildren.html) APIs which accept pre-allocated List objects. The input List will be filled with the results of the API, and will be expanded if the number of results exceeds the List’s capacity. Usage of these variants should be preferred, as it permits the reduction of spurious GC allocations via collection reuse.

#### **C. Object Instantiation**

One of the aspects which usually impacts performance is on-demand object instantiation. It is important to pool or keep frequently created and destroyed objects in memory for the following reasons:

1. To prevent thrashing the *Mono Heap*.
2. To prevent the instantiate function calls from slowing down the frame.
3. To prevent the need to call [Resources.UnloadUnusedAssets()](https://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html) regularly during runtime, which will cause garbage collection to occur. [Resources.UnloadUnusedAssets()](https://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html) can then just be reserved for moments in the application lifecycle where frame time is less important, such as when changing scenes.

This is usually very relevant for UI performance. If possible, it is best to keep the core set of UI objects and only instantiate the smallest set of objects necessary.

In general, it is a best practice to avoid assigning names to runtime-generated GameObjects outside of Editor builds, as they’re unnecessary unless the [GameObject.Find()](https://docs.unity3d.com/ScriptReference/GameObject.Find.html) API is utilized.

#### **D. Transform Changes During Object Instantiation**



When instantiating a GameObject, developers will want to use the version of the API that accepts a position and rotation. Not only does this remove the need for you to access that instantiated GameObject’s Transform, but it is potentially two fewer calls into engine code.

[Instantiate](https://docs.unity3d.com/ScriptReference/Object.Instantiate.html)(Object original, Vector3 position, Quaternion rotation);



In *Unity 5.4* we altered the representation of transforms in memory so that each root transform’s entire child hierarchy is stored in compact, contiguous regions of memory. When instantiating new GameObjects that will be instantly reparented into another hierarchy, consider using the new GameObject.Instantiate() overloads which accept a parent argument.

[Instantiate](https://docs.unity3d.com/ScriptReference/Object.Instantiate.html)(Object original, Transform parent);

Using this overload avoids the allocation of a root transform hierarchy for the new GameObject. In tests, this speeds up the time required for an instantiate operation by about 5-10%.

#### **E. Enabling and Disabling Animators**



The default behavior for *Animator* components is to only retain their data buffers and bindings while active and enabled. Thus, when disabling an object with an *Animator* component, the whole internal state of this component is cleared and is later recreted, resulting in CPU spikes.

However, this behavior will be overridable in *Unity 2018.1*. In that version, setting the [Animator.keepAnimatorControllerStateOnDisable](https://docs.unity3d.com/2018.1/Documentation/ScriptReference/Animator-keepAnimatorControllerStateOnDisable.html) property to true, then enabling and disabling the *Animator’s* GameObject will cause the *Animator* to initialize its data buffers and avatar bindings, but then disable it and prevent it from ticking without freeing the animation state.

#### **F. Coroutines**

Each time a coroutine is started, an object will be allocated on the *Managed Heap* to track the state of the coroutine. This object is also used to track the state of any variables in the coroutine’s local scope which the compiler determines it necessary to persist across invocations of the coroutine. *(Generally, this is locally-scoped variables whose values must persist across one or more yield calls)*

As such, when writing coroutines, it is important to minimize the number of memory-intense locally-scoped variables. Aggressively nulling-out reference types in coroutines are one way of accomplishing this.

Infinitely looping (such as *yield return null*) or long-running coroutines should universally be eliminated from the codebase. Any managed memory allocated within a long-running coroutine will never be garbage-collected while the coroutine is running. Infinite coroutines with any managed memory allocations will therefore appear to leak memory. Any coroutines which require infinite loops can generally be extracted in an *Update* callback, or otherwise handled via a global *Update Manager*.

Note that:

1. Instances of [WaitForSeconds](https://docs.unity3d.com/ScriptReference/WaitForSeconds.html) can be cachedand reused in coroutines throughout the lifetime of the game.
2. 0 is not considered equivalent to null; the integer value will be boxed in order to coerce it into a reference type. Therefore, writing yield return 0 will allocate one integer value each frame. This should be avoided.
3. When disabling a GameObject, all of its coroutines are stopped and destroyed.

#### **G. FindObjectOfType()**

[FindObjectOfType()](https://docs.unity3d.com/ScriptReference/Object.FindObjectOfType.html)is a very expensive operation, as it requires traversing through the scene graph. As such, it should be used sparingly and have its value cached.

#### **H. Constant Properties**

Unity exposes a number of constants on its classes for the convenience, such as [Vector3.zero](https://docs.unity3d.com/ScriptReference/Vector3-zero.html). These are implemented as properties that return a specific, constant value. Because accessing a property always requires a method call, there is a small but significant cost associated with accessing any of these properties. Where necessary to optimize tight inner loops, create static readonly values to replace Unity’s constant-valued properties.

#### **I. Finding an Object by Tag**

[Component.CompareTag()](https://docs.unity3d.com/ScriptReference/Component.CompareTag.html)performs a string comparison every frame. We recommend using a different way to compare fields at runtime.

#### **J. Changing Transform Position**

When modifying both position and rotation, use [Transform.SetPositionAndRotation()](https://docs.unity3d.com/ScriptReference/Transform.SetPositionAndRotation.html)instead of individually setting these properties separately to reduce the number of instructions issued.

If possible, use local positions instead of world positions*(Unity traverses the hierarchy up and down on each world position request/change because internally only local position is stored)*.

#### **K. Camera.main**

Calling [Camera.main](https://docs.unity3d.com/ScriptReference/Camera-main.html)will actually do a search via tag under the hood, similar to [GameObject.FindWithTag()](https://docs.unity3d.com/ScriptReference/GameObject.FindWithTag.html). It is much more efficient to cache the result and access the main camera via the cached reference. The only caveat here is to know when the main camera changes to update the cached reference.

#### **L. Logging**

[Debug.Log()](https://docs.unity3d.com/ScriptReference/Debug.Log.html) calls are very expensive and can be observed in *the Unity Profiler*. A large portion of CPU time will go to generating the stack traces on log messages. Unity will generate stack traces by default but you can disable this behavior and manually set the stack trace type using [Application.SetStackTraceLogType()](https://docs.unity3d.com/ScriptReference/Application.SetStackTraceLogType.html).

The three options available are:

1. [Full](https://docs.unity3d.com/ScriptReference/StackTraceLogType.Full.html) — this will generate the full *Call Tree* for the logged messages. Both managed and native code symbols will be part of the *Call Tree*.
2. [ScriptOnly](https://docs.unity3d.com/ScriptReference/StackTraceLogType.ScriptOnly.html) — this will only generate a *Call Tree* for the managed part of the code.
3. [None](https://docs.unity3d.com/ScriptReference/StackTraceLogType.None.html) — this will only output the message that the user supplied to the logger.

In most cases we recommend using the None option. You can however change the log type at runtime if you are logging an exception and require the full call stack information. In that situation you will need to use the variants of [Debug.Log()](https://docs.unity3d.com/ScriptReference/Debug.Log.html) to specifically call out that you are logging an exception.

You should subscribe to [Application.logMessageReceivedThreaded](https://docs.unity3d.com/ScriptReference/Application-logMessageReceivedThreaded.html) event which is fired when a log message is triggered in Unity. This callback supplies the [logtype](https://docs.unity3d.com/ScriptReference/LogType.html) so you can turn on and off full stack trace logging only when an exception type message has been generated.

We recommend wrapping calls to [Debug.Log()](https://docs.unity3d.com/ScriptReference/Debug.Log.html) into another method, and adding the [Conditional(“DEBUG”)] attribute to the wrapper method. This will cause all log messages to be fully compiled out of *Release* builds.

## **6.3. Memory Usage**

Reducing memory usage is critical to minimizing [the growth of the managed, non-compacting heap, reducing garbage collection](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html), avoiding OS eviction, and improving general CPU performance.

### **6.3.1. Mono Heap**

**One thing to always remember is that Unity Mono Heap never shrinks.** We generally recommend to:

1. Pool frequently used objects to reduce Heap fragmentation.
2. Assess large allocations which can inflate Mono Heap.

### **6.3.2. Garbage Collection**

As a general guide, based on experience across many projects, it is recommended that an acceptable level of per-frame GC allocations would total around *100 bytes* on average. Ideally, a title should strive for zero or as low as possible. Typically, a high level of GC allocations leads to more frequent and more obvious CPU spikes when the garbage collector is invoked, impacting the fluidity of the game.

GC allocations can be identified from the *CPU Usage* area of *the Profiler* during runtime but there are also tools like [JetBrains Rider](https://www.jetbrains.com/rider/) and the JetBrains [extension](https://blog.jetbrains.com/dotnet/2014/06/06/heap-allocations-viewer-plugin/) for Resharper that can help to identify these from code analysis.

Please read the section about [*Garbage Collection and Managed Heap*](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html) in the best practice guide [*Understanding Optimization in Unity*](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity.html), which details common causes for unintentional GC Allocations. This guide will give you a better understanding of the costs of GC allocations on memory and CPU performance at runtime.

### **6.3.3. Resources.UnloadUnusedAssets() and GC.Collect()**

[Resources.UnloadUnusedAssets()](https://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html) and [GC.Collect()](https://msdn.microsoft.com/en-us/library/xe0c2357(v=vs.110).aspx) can be called to help free unused memory, though this will likely introduce performance hitches. A potential compromise is to call these functions when compute cycles are available during non-user interactive game sequences, such as loading screens.

#### **Unloading Scenes**

Calling [UnloadScene()](https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.UnloadSceneAsync.html) destroys GameObjects associated with the scene but does not unload assets. In order to do so, you need to call [Resources.UnloadUnusedAssets()](https://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html) when the scene has been unloaded.

### **6.3.4. HideFlags**

Improper use of [HideFlags](https://docs.unity3d.com/ScriptReference/HideFlags.html) may result in assets not being cleaned from Memory even after an [Resources.UnloadUnusedAssets()](https://docs.unity3d.com/ScriptReference/Resources.UnloadUnusedAssets.html) pass. Assets marked as [HideFlags.HideAndDontSave](https://docs.unity3d.com/ScriptReference/HideFlags.HideAndDontSave.html) or [HideFlags.DontSave](https://docs.unity3d.com/ScriptReference/HideFlags.DontSave.html) are never unloaded unless explicitly released by [Object.DestroyImmediate()](https://docs.unity3d.com/ScriptReference/Object.DestroyImmediate.html).

We recommended that the team audit all usages of [HideFlags](https://docs.unity3d.com/ScriptReference/HideFlags.html) in the project to ensure that objects with these flags set are explicitly destroyed after use.

### **6.3.5. DontDestroyOnLoad**

Be careful when using [DontDestroyOnLoad](https://docs.unity3d.com/ScriptReference/Object.DontDestroyOnLoad.html) flag because it implies that you are responsible for destroying the object when the scene is unloaded. It is important to use this flag sparingly and smartly to avoid memory leaks.

### **6.3.6. Particle Systems**

Each Particle System uses a few Kb of RAM for its internal buffers (3.5Kb for the simplest Particle System and growing to 25+Kb when more modules are used). Particle systems retain the memory buffers used to evolve their simulations, even when disabled; keeping a large number of inactive particle systems in the scene can rapidly consume large amounts of memory.

The first optimization should be to reduce the number of particle systems present in the scene.

Once the number of particle systems in the scene has been optimized, further savings on each individual particle system may be achievable by:

● Altering the number of particles (default is 1000 - see [*Particle System - Main module*](https://docs.unity3d.com/Manual/PartSysMainModule.html))

● Using billboards instead of meshes (see [*Particle System - Renderer module*](https://docs.unity3d.com/Manual/PartSysRendererModule.html))

● Reducing the number of modules active on each particle system

### **6.3.7. Changing Quality Settings at Runtime**

We advise to use [QualitySettings.masterTextureLimit](https://docs.unity3d.com/ScriptReference/QualitySettings-masterTextureLimit.html) at runtime to specify the base *mipmap* level appropriate to the device. On lower-end devices this improves both memory consumption and GPU performance.

Usually UI texture atlases do not have *mipmaps* and so are not affected by the use of [QualitySettings.masterTextureLimit](https://docs.unity3d.com/ScriptReference/QualitySettings-masterTextureLimit.html). There are two approaches to this:

Firstly, if all of the UI textures’ import settings have *Generate Mip Maps* checked under the *Advanced* section, then the resulting texture atlas(es) will also generate *mipmaps*. Any such textures which are not set to generate *mipmaps* will be packed separately into a different ‘group’ — this can be observed in *the Sprite Packer* tool.

Secondly, you could implement a [*custom sprite packer policy, as described here*](https://docs.unity3d.com/Manual/SpritePacker.html). This involves creating an Editor script somewhere in the game project, with the example default policy code shown on the documentation page being a good starting point. In this default policy code, inside the [OnGroupAtlases()](https://docs.unity3d.com/ScriptReference/Sprites.IPackerPolicy.OnGroupAtlases.html) method is a loop which sets up the sprite entries. There is already a line which sets up the entry.settings.generateMipMaps property, currently assigning it to the value obtained from the texture importer. Changing this line to force entry.settings.generateMipMaps = true will ensure that any atlases generated using this sprite packer policy are mipmapped. Again, the results can be seen within *the Sprite Packer* tool, where you can choose to pack or repack specific atlases with your custom policy.

## **6.4. Physics Performance**

### **6.4.1. PhysX**

Unity uses *NVIDIA PhysX* library. There are cases when physics performance can degrade due to the number of simulated objects and the nature of the default PhysX broadphase *sweep-and-prune (SAP)* algorithm.

#### **PhysX Broadphase**

The *SAP* algorithm internally maintains 3 lists, 1 per axis, of AABB extents for all objects within the scene. What this essentially reduces to, for each list, is a pair of values per object, representing its minimum and maximum bounding values on that axis. These lists of values are kept in sorted form, in order to be performant during sweep operations. However, when a new object is added to the scene, or an object moves, it is possible that one or more axis lists will have to be at least partially re-sorted.

Even worse, if the game world has a large range in both X and Z, but relatively little in Y, an object insertion or movement is very likely to alter the list ordering in Y. If there are tens of thousands of objects in the list, spanning a relatively small range, even with just a small movement of a dynamic object, we might force the re-sorting of hundreds or thousands of objects which are a great distance away from the dynamic object.

One other possible cause of CPU spikes here could be that one or more static colliders were moved, likely by accident. Although perhaps not very likely, it is recommended nonetheless that the team audit any physics objects which can be moved during the game, and check that none are marked as static.

In *Unity 2017.3* it is possible that switching to use PhysX’s *multi-box-pruning (MBP)* algorithm could mitigate this to some degree. The MBP algorithm requires a little more setup, but allows for specifying multiple bounding regions to split up the scene.

#### **PhysX Midphase**

The PhysX midphase is relevant only for potential collisions involving a static mesh collider. In such cases, there is a cost associated with traversing the mesh collider’s bounding volume tree to determine a set of individual triangles which may be involved in the collision with the other object. This then results in an expanded number of potential collider pairs, one to check each triangle against the other object.

Therefore, this phase is a factor of the collision mesh’s complexity and density, along with its overall size. In addition to reducing the overall triangle count or optimising overly dense areas, particularly large meshes could be subdivided into multiple meshes. This will push a little more work onto the cheaper AABB checks in the broadphase, to potentially reduce the work done in the midphase.

### **6.4.2. Colliders**

We generally discourage the usage of *MeshCollider*, usually it is much more performant to use compound colliders (i.e. groups of primitive colliders, such as boxes and spheres) to approximate a mesh’s shape.

If you have to use a mesh as collision shape, you can use a simpler optimized mesh for this purpose instead of the detailed model mesh.

### **6.4.3. Non-allocating APIs**

When using [Physics.RaycastAll()](https://docs.unity3d.com/ScriptReference/Physics.RaycastAll.html) method Unity allocates memory for returned objects every call. It is recommended to use the non-allocating method [Physics.RaycastNonAlloc()](https://docs.unity3d.com/ScriptReference/Physics.RaycastNonAlloc.html) instead. Similarly, check for calls to any other physics methods which may generate GC allocations and refactor to use the *NonAlloc* versions instead. These methods are listed [in the Unity Scripting Reference](https://docs.unity3d.com/ScriptReference/Physics.html).

### **6.4.4. Transforms Synchronization**

Setting [Physics.autoSyncTransforms](https://docs.unity3d.com/ScriptReference/Physics-autoSyncTransforms.html) to false prevents Unity from automatically syncing any transform changes with the physics system. This way objects with Rigidbodies can be freely moved via the Transform component without it affecting the physics system. It should be noted that any changes that are made will be batched up and applied to the physics system during the next fixed update call.

### **6.4.5. Manually Simulating Physics**

Setting [Physics.autoSimulation](https://docs.unity3d.com/ScriptReference/Physics-autoSimulation.html) to false lets you manually use [Physics.Simulate()](https://docs.unity3d.com/ScriptReference/Physics.Simulate.html) to advance PhysX state. Note that this method is available even outside of the Play Mode.

### **6.4.6. Heavy Logic in FixedUpdate**

[FixedUpdate](https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html) can be executed several times per frame. The code which controls the execution looks like so:

globalTime += Time.deltaTime;

while (globalTime >= Time.fixedDeltaTime)

globalTime -= Time.fixedDeltaTime;

Simulate(Time.fixedDeltaTime);

RunFixedUpdatesOnScripts();

If you look closely, you can see that this logic can easily spiral out of control because of just one frame taking longer to execute or render. In this case Unity will have to run [FixedUpdate](https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html) more than once, and if some script contains heavy logic inside this method, the next frame will take even longer to render, hence Unity will be executing more and more [FixedUpdate](https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html) methods to keep up.

It is recommended to carefully adjust *Fixed Timestamp* or execute physics simulation manually.

### **6.4.7. Data Storage**

Physics data is stored as component data and passed to PhysX during initialization. Components are serialized into the scene/prefab of the GameObject to which they're attached.

## **6.5. Animation Performance**

When using [Animator.Update()](https://docs.unity3d.com/ScriptReference/Animator.Update.html) method to manually update an *Animator*, the *Animator* component should be disabled in order to prevent multiple updates. Note that manual updating of an *Animator* will always execute on the main thread, and can't take advantage of the multi-threaded processing utilised by those which are automatically updated.

As mentioned earlier, performance of the *Animators* can be improved by enabling the *Optimize Game Objects* option on the model.

## **6.6. Audio Performance**

#### **WebGL**

[*Streaming Audio Source for Unity WebGL*](https://assetstore.unity.com/packages/essentials/streaming-audio-source-for-unity-webgl-75612) plugin on Asset Store, maintained by Unity’s WebGL team, provides support for playing audio via native JavaScript APIs. This allows the browser to manage playback buffer allocation & data streaming on its own, which is generally more efficient than using Unity’s built-in Audio Sources on the WebGL platform.

## **6.7 Terrain Performance**

TBD

## **6.8. Light Baking Performance**

Our Lighting Team wrote a great [*tutorial*](https://unity3d.com/learn/tutorials/topics/graphics/introduction-precomputed-realtime-gi?playlist=17102) on how to bake lightmaps with Enlighten.

## **6.9. Rendering Performance**

### **6.9.1. Frame Rate**

#### **WebGL**

In general, Unity WebGL titles should avoid setting the [Application.targetFrameRate](https://docs.unity3d.com/ScriptReference/Application-targetFrameRate.html) property. When set, Unity must switch from using the efficient [requestAnimationFrame](https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame) API, which allows Unity to update at the browser’s native refresh rate, to scheduling its updates via the [setTimeout](https://www.w3schools.com/jsref/met_win_settimeout.asp) API.

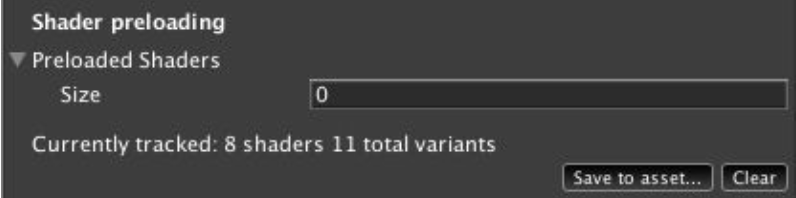
Browsers do not guarantee that a callback registered via [setTimeout](https://www.w3schools.com/jsref/met_win_settimeout.asp) will be called at the exact interval requested. It is possible that, on some machines, the browser will invoke Unity's update callback at irregular intervals, leading to the perception of a choppy frame rate.

### **6.9.2. Shaders**

For a mobile title, *we recommend zero usage of the Standard Unity shader*. This shader is not optimized for mobile, and as mentioned previously, built-in shaders will be duplicated across bundled assets.

When a shader variant is used for the first time, Unity may be observed spending a non-trivial amount of time in Shader.CreateGPUProgram() method, which can lead to annoying CPU spikes.

To alleviate this, you can *“record”* the active shaders used in the scene, then save it as asset in the *Graphics Settings*:



The [ShaderVariantCollection](https://docs.unity3d.com/ScriptReference/ShaderVariantCollection.html) can include all the shaders required for the application. When using this feature, Unity will compile all the shaders found in the collection before the first scene load. This operation ensures they will be ready for use ahead of time.

Alternatively, you may also call [Shaders.WarmUpShaders()](https://docs.unity3d.com/ScriptReference/Shader.WarmupAllShaders.html), but this function offers less granular control.

### **6.9.3. Batching**



Batching is used to save CPU time by merging several meshes together and considering them as a single item to avoid work duplication. Currently the batching system divides the batchable objects into two rings: a close-up one and a far-from-view one. This allows drawing close-up meshes first and reducing overdraw by using Z-culling for further meshes.

A game should take advantage of batching by using the fewest number of materials, textures, and shaders as possible. You should look at atlasing set piece textures together, and make this a part of the artists’ authoring pipeline.

In order for two meshes to be batched, the following must be true:

1. Both meshes use the same instance of the same material.
2. Both materials reference the same shader and texture.



You should verify that all non-movable geometry is marked as *Static* in the Editor, to automatically take advantage of *Static Batching*. If props are loaded dynamically, you may manually use the [StaticBatchingUtility](https://docs.unity3d.com/ScriptReference/StaticBatchingUtility.html) to add those pieces of geometry to a static batch.

It might also be helpful to make use of runtime mesh combining, if there are many smaller meshes that share materials and lightmap indices. Take care to ensure that meshes are not combined in a way that may negatively impact culling.

### **6.9.4 Instancing**

*Instancing* is used to save GPU time by keeping in GPU memory vertices of a draw call and reusing these for several draw calls but changing parameters. GPU instancing is activated on per-shader basis and removes GPU state changes between draw calls. GPU instancing is meant to draw a lot of objects that uses the same data.

This in-depth article from *GPU Gems 2* is a good resource to understand the difference between instancing and batching: [*Inside Geometry Instancing by Francesco Carucci*](https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter03.html).

### **6.9.5. Accessing textures from scripts**

Copying texture data from GPU to CPU is a costly operation. It usually leads to spikes on CPU while waiting for the work to be done — Gfx.ReadbackImage(). In order to use functions like [ReadPixels()](https://docs.unity3d.com/560/Documentation/ScriptReference/Texture2D.ReadPixels.html) it is important to set *Read/Write Enable* flag in the [*Texture importer*](https://docs.unity3d.com/560/Documentation/Manual/class-TextureImporter.html).

The common technique to access textures from CPU or to change the render textures of the camera is to override the [OnRenderImage](https://docs.unity3d.com/ScriptReference/Camera.OnRenderImage.html) function in script and call [Graphics.Blit()](https://docs.unity3d.com/ScriptReference/Graphics.Blit.html). If your [OnRenderImage](https://docs.unity3d.com/ScriptReference/Camera.OnRenderImage.html) function just calls [Graphics.Blit()](https://docs.unity3d.com/ScriptReference/Graphics.Blit.html) then it is important to remove the function to avoid adding unnecessary hooks in the render loop.

However, [Graphics.Blit()](https://docs.unity3d.com/ScriptReference/Graphics.Blit.html) may not be the most suitable option for you:

1. Use [Graphics.BlitMultiTap()](https://docs.unity3d.com/ScriptReference/Graphics.BlitMultiTap.html) on materials associated with several texture fetching algorithms like bloom or blur.
2. Use [Graphics.CopyTexture()](https://docs.unity3d.com/ScriptReference/Graphics.CopyTexture.html) when you want to copy CPU and GPU textures over without any specific post-process. This is more efficient than using [Graphics.Blit()](https://docs.unity3d.com/ScriptReference/Graphics.Blit.html).

### **6.9.6. Multithreaded Rendering**

On Android, you can enable *Multithreaded Rendering*to help increase the graphics performance of your game running on high-end Android devices with multiple cores. However, note this may hurt performance on low-end devices due to the overhead cost of creating and combining multiple render task operations. This setting is also a package flag, and thus cannot be changed at runtime.



On iOS, if the *Graphics API*in the *Player Settings* is set to *Metal*, Unity will offload graphics rendering tasks to worker threads running on other CPU cores. If *OpenGL ES 2/3* is selected, no multithreaded rendering will occur. By default, the option is set to *Auto*, in which Metal will be automatically used for Metal-capable devices (iPhone 5s or newer).

## **6.10. UI Performance**

When building UI general best practices include:

1. Disable rendering of offscreen element.
2. Ensure only one [EventSystem](https://docs.unity3d.com/ScriptReference/EventSystems.EventSystem.html) exists in the hierarchy at any time.
3. Prefer direct scripting over Animators.
4. Avoid nested Layout Groups.

You can get more details on UI optimization in the [*Best Practices guide*](https://unity3d.com/learn/tutorials/temas/best-practices/guide-optimizing-unity-ui).

### **6.10.1. Minimize Canvas Rebuilds**

*Canvas* rebuilds are a common cause of high CPU utilization, and can be identified In *the Unity Profiler* by searching for *CanvasBuildBatch*.

To minimize the number of Canvas rebuild operations, components should be split up into multiple Canvases, based on update frequency.That is, objects that aren’t updated very often should reside on a single Canvas, while objects that are updated more frequently should exist on a separate Canvas. Canvas render frequency can be observed by searching for instances of *OnWillRender* in *the Unity Profiler*.

### **6.10.2. Disable Canvas Components to Toggle Visibility**

The easiest way to toggle visibility of UI is to disable the parent *GameObject* of that UI hierarchy. In doing so, the mesh data for the *Canvas* is discarded. Enabling the *Canvas* later on requires the data for the *Canvas* and its child *Canvases* to be rebuilt. To avoid this, the target UI could be placed onto its own *Canvas*, then disable or enable the *Canvas* component component as needed. Doing so will keep the mesh data in memory.

Take caution with this approach, as the *Canvas* will be still marked for any *Graphic Raycaster* target. Further, the associated MonoBehaviour will also continue to remain active (this can be controlled by using a global *Update Manager*, or adding checks to the *MonoBehaviour* itself).

### **6.10.3. Raycast Optimization**

All *Canvas Renderers* when created have *Raycast Target* property enabled by default. This can result in an unnecessarily large cost for touch events. We recommend disabling the *Raycast Target* option on any *Canvas Renderers* that do not require it, as this will reduce the CPU cost for touch events. Please read the [*Raycast Optimization*](https://unity3d.com/learn/tutorials/topics/best-practices/fill-rate-canvases-and-input?playlist=30089)section of the [*UI Best Practice guide*](https://unity3d.com/learn/tutorials/temas/best-practices/guide-optimizing-unity-ui) for more details.

In addition, each *Canvas* by default has a *GraphicRaycaster* component. *GraphicRaycasters* are UI components that should be carefully reviewed and used only as necessary. For each frame, *GraphicRaycasters* perform intersection tests for all of the interactable UI elements per top-level *Canvas* against processed input, then dispatch the appropriate UI event handler as needed. The CPU performance cost of these tests grows linearly with the number of interactable UI elements. Furthermore, the computational costs of graphic raycasting are also still present when Canvas’ alpha is set to zero.

As such, care should be taken to ensure that *Graphic Raycasters* are removed from Canvases that don’t require them.

### **6.10.4. Scroll Rect**

Unity UI *ScrollRect* component is a well-known source of performance problems. The *ScrollRect* implements the *ILayoutElement* and *ILayoutGroup* interfaces, leading to the performance problems.

We recommend writing your own scrollable-list UI component which would hide or disable UI components when they are no longer visible in the scrollable list.

### **6.10.5. Font.RequestCharactersInTexture**



If the game has dynamic UI text elements, sometimes CPU spikes ending with [Font.RequestCharactersInTexture](https://docs.unity3d.com/ScriptReference/Font.RequestCharactersInTexture.html) may be observed in *the Unity Profiler*. This method is called on-demand as new text is loaded and displayed. It will regenerate the dynamic font texture and add the characters needed. To avoid [Font.RequestCharactersInTexture](https://docs.unity3d.com/ScriptReference/Font.RequestCharactersInTexture.html) getting called at undesirable times, we advise to execute it manually ahead of time with the set of required characters.

### **6.10.6. Text Mesh Pro**

#### **Text Mesh Rebuilds**

Much like Unity’s built-in *UIText* component, making changes to the text displayed by the component will trigger costly canvas rebuilds. Minimizing changes to a *TextMeshProUGUI* component’s text field and ensuring that it is parented to an appropriate Canvas will keep *TMP* performance as efficient as possible.

#### **Fonts and Memory Usage**

Given that there is no dynamic font feature in *TMP*, one must rely on fallback fonts. Understanding how fallback fonts are loaded and used is crucial to optimizing memory when using *TMP*.

Glyph discovery in *TMP* is done recursively – that is, when a glyph is missing from a *TMP* *Font Asset*, *TMP* iterates through the *Font Assets* fallback fonts, and then through their fallback fonts.If the glyph is not found, *TMP* moves on to *TMP Settings* and iterates through the *Default Sprite Atlas* for associated unicode characters, followed by its fallback font list, and finishes with the *Default Font Asset*.

*TextMesh Pro’s Font Assets* are loaded when they are referenced in a scene or project. They are principally referenced by *TextMeshPro Text* components, by the *TMP* *Settings*, and also by *Font Assets* themselves, as fallback fonts. If *Font Assets* are referenced in the *TMP Settings* asset, those *Font Assets* and all their fallback *Font Assets* will recursively be loaded when the first scene with a *TMP Text* component is activated. If the default sprite sheet asset is referenced, that will also be loaded.

Additionally, when a *Font Asset* is referenced by a *TextMeshPro* component in a given scene and has not been loaded via *TMP Settings*, then the referenced *Font Asset* and all of its fallback *Fonts Assets* will recursively be loaded once the component is activated. It is important to keep this process in mind when working on a project with many fonts, particularly if available memory is an issue.

For the above reasons, having all localized language *Font Assets* loaded by the *TMP Settings* upfront would be detrimental to memory pressure. Should this be a necessary requirement, we suggest using *Asset Bundles* to load *Font Assets* in a modular way. When the application starts, a bootstrap step should be included to verify the user’s locale and to setup font asset fallbacks for each font asset:

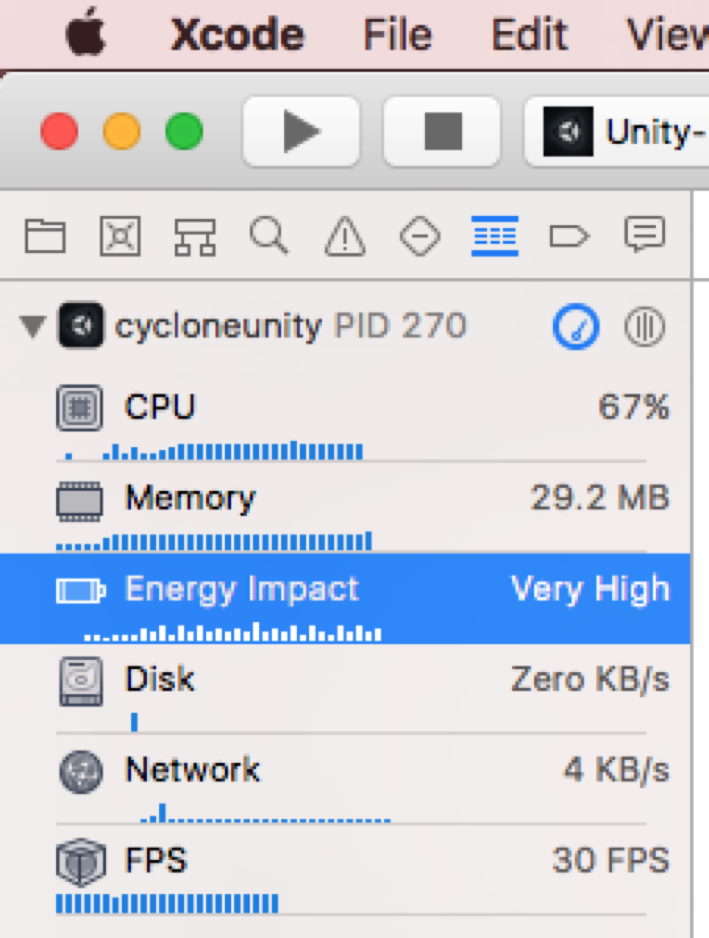
1. Create an *Asset Bundle* for base TMP Font Assets (e.g., minimal Latin glyphs for each font).
2. Create an *Asset Bundle* for needed fallback TMP Font Assets per language (e.g., one Asset Bundle for TMP Font Assets for each font needed for Japanese).
3. Load your base *Asset Bundle* in the bootstrap step.
4. Based on locale, load the needed *Asset Bundle* with fallback fonts.
5. For each font in the base *Asset Bundle*, assign fallback font assets from the localized font Asset Bundle.
6. Continue bootstrapping your game.

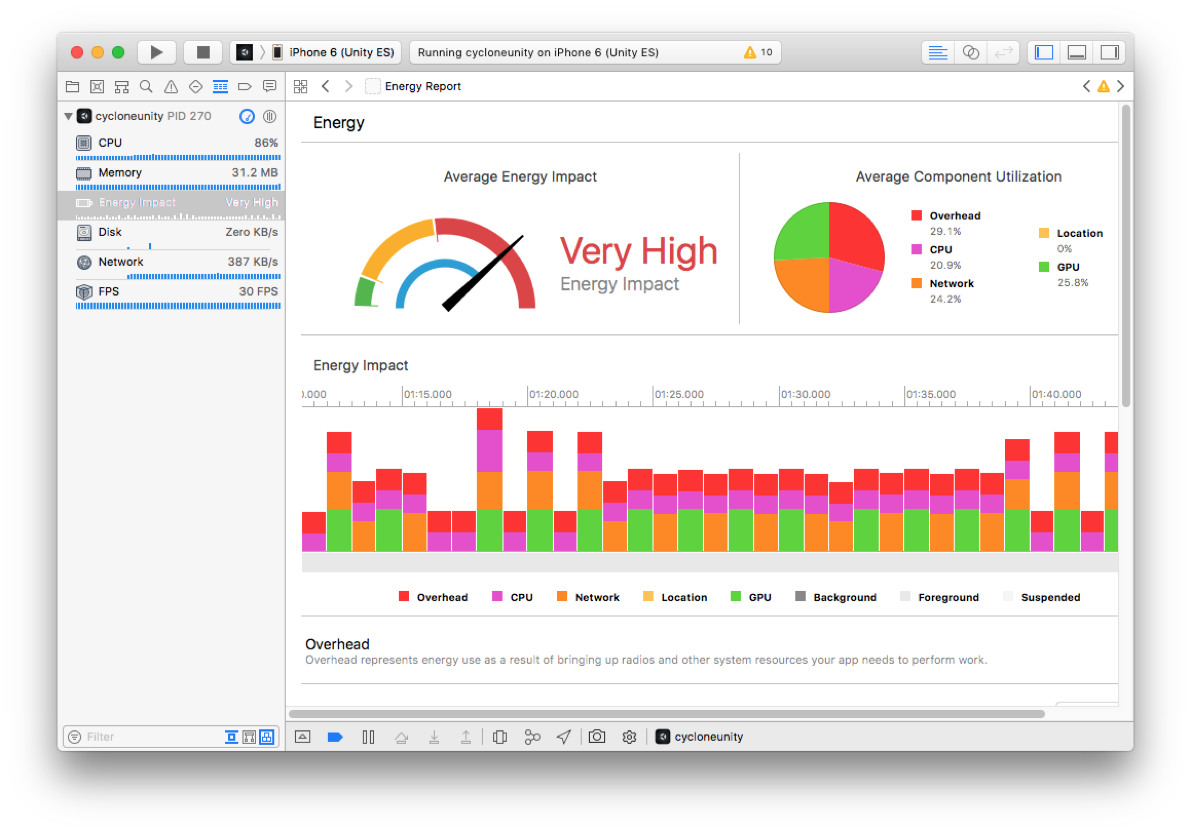
The *Default Sprite Asset* reference may also be removed from the *TMP Settings* if no images are used, for additional modest memory savings.

## **6.11. Thermal Performance**

It is difficult to answer accurately what contributes the most to device heating: CPU or GPU. Neither Android or iOS give us very detailed low-level information in this area.

Currently, the best estimation for this on iOS devices is to run the game directly from *Xcode*, and then switch to the *Debug Navigator* pane. Select the *Energy Impact* timeline to reveal, in the main view, a real-time graph of estimated energy usage. CPU and GPU usage are highlighted separately, and whilst it’s hard to say just how accurate these figures are, in relative terms they seem to be a reasonable guideline.





# **7. Internal Systems**

## **7.1. Serialization**

The best public description of the serialization system, at present, is [this Unite talk](https://www.youtube.com/watch?v=N-HJvfVuKRw).

# **8. Miscellaneous Topics**

## **8.1. Custom Command Line Arguments**

Though Unity does not provide a direct API for doing this it can be easily achieved through using standard C# calls. This helper function shows how it can be done:

private static string GetArg(string name)

{

var args = System.Environment.GetCommandLineArgs();

for (int i = 0; i < args.Length; i++) {

if (args[i] == name && args.Length > i + 1) {

return args[i + 1];

}

}

return null;

}

As an example you could use this given the following instruction supplied to the command line:

Windows:

Unity.exe -customArgOne "arg one" -customArgTwo “arg two”

OSX:

Unity.app/Contents/MacOS/Unity -customArgOne "arg one" -customArgTwo “arg two”

You would then be able to call:

string s = GetArg("-customArgOne");

Debug.Log(s);

s = GetArg("-customArgTwo");

Debug.Log(s);

This would output *arg one* and *arg two* in the console window of the editor.

# **9. Native Profiling Tools**

## **9.1. Intel GPA**

You can get the latest version of Intel GPA at this site: <https://software.intel.com/en-us/gpa>

Once everything is installed on your machine you need to run the *Graphics Monitor* tool. This tool will add an icon to your system tray. From it, you can select the *Analyze Application…* option. This will open a window that allows you to set up the application you want to analyze. The command line option should be the path to the executable plus any command line options you may want to pass to the application.

Once you have set this up you can then run the application. Once the application is running there are various commands which can be issued to the running application to change behavior and identify what may be causing any issues. The most useful one of these is the option to capture a frame which is done via *Ctrl+Shift+C*.

Once the frame has been captured you can then access the data via other applications. This is accessed again via the system tray icon. This time however you want to use the *Graphics Frame Analyzer for DirectX…* option. When the tool opens it will provide you with a list of frame capture and you can then pick the ones that are relevant to the project that you are working on.

## **9.2. RenderDoc**

[*RenderDoc*](https://renderdoc.org/) is a free open source graphics debugger for Windows and Linux. *RenderDoc* support is integrated into the Engine. You can find more info about it in the [*Unity Manual*](https://docs.unity3d.com/Manual/RenderDocIntegration.html).

## **9.3. iOS Profiling Tools**

[*Xcode Instruments*](https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/index.html), provided by Apple, is a versatile performance profiler that allows you to better understand how your title is using memory, and what areas performance is bound to. When you use the tool, you will need to add a specific Instrument to retrieve specialized views. We recommend using the latest version of *Xcode* for the most reliable and best performance profiling experience.

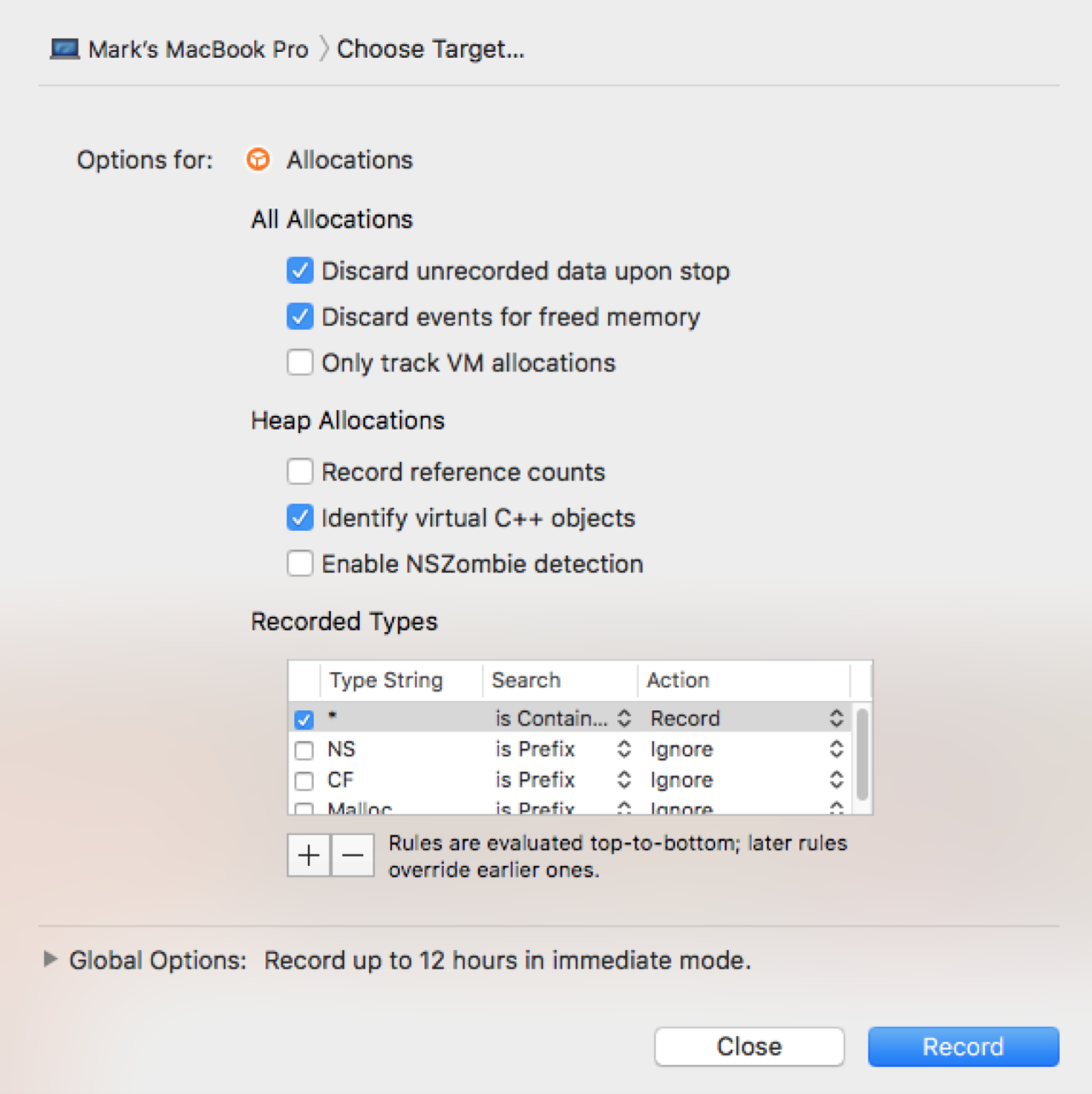
### **9.3.1. Allocations Instrument**

The *Allocations*instrument can be used to track native memory allocations — that is, anything that ends up mapping to malloc(). If the project makes use of any native plugins, their memory allocations will appear here. The call graph view is especially useful to determine the critical path.

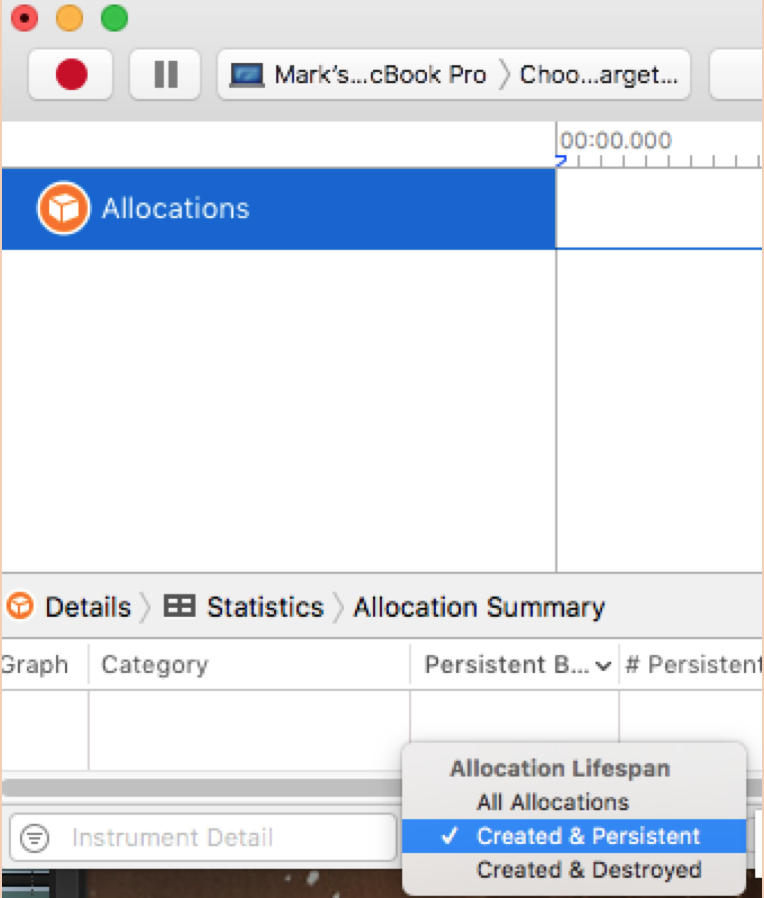
The *Allocations* tool is useful for looking at the native allocations within a Unity project. This tool provides a full call stack for each allocation, so you can see the script code that triggered the allocations within the native parts of Unity. This is especially useful as some of these allocations may not seem obvious from a scripting perspective, such as the memory that is allocated as part of *IL2CPP* every time a method is called for the first time.

If you are using other libraries, the Allocations tool can be used to see what those other libraries are allocating at a native level. This is useful because native allocations from other libraries are hidden to Unity and cannot be seen within *the Unity Profiler*. The reason for this is that Unity does not provide an exposed native allocator at this time and it does not track other native allocations.

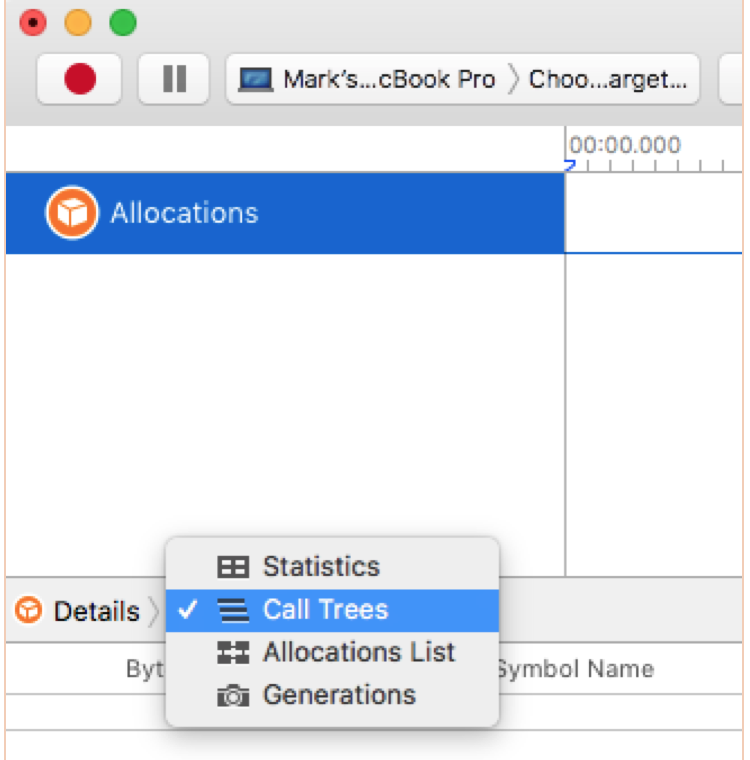
The recommended settings for exploring memory allocation in the *Allocations* tool are highlighted in the following screenshots.



Discarding events for freed memory is important. This means that only what is left in memory is visible in the timeline.



Only created and persistent allocations should be visible, otherwise you may end up chasing down an allocation that is no longer in memory.



Using the *Call Tree* view will allow you to see the call trees for each allocation. Using this view, you will be able to pin down what is taking up the most memory and where it was allocated from.

#### **Generations**

The *Allocations* tool can also be used to mark generations in memory. This is a useful way to view allocations over time, and to track down memory issues with leaking objects.

As an example workflow, load your application into the main menu and select *Mark Generation*. This will display everything that has been loaded into memory and not unloaded up to that point. Play a level in the game and then go back to the main menu, and select *Mark Generation* again. The display will update with a new generation. This new generation will show everything that was loaded into memory and not unloaded since the last generation: that is to say, everything that has not been unloaded after playing the level.

### **9.3.2. VM Tracker Instrument**

The *VM Tracker*instrument is the tool to use to check an app’s *Dirty Memory*, which refers to RAM allocations that cannot be paged to reduce memory pressure. Please refer to [this document](https://docs.google.com/document/d/1fg29PBVzjjQIyrvPsrZ5gMtEgaHxwTKKDAisHUw5oUU/edit) for more details on VM Tracker and how iOS memory is structured.

Reasonably safe limit values of *Dirty Memory* for different iOS devices (from what we have seen in the wild):

* 180Mb for 512Mb devices,
* 360Mb for 1Gb devices,
* 1.2Gb for 2Gb devices.

Note that even if your application falls under these recommended limits, eviction from iOS is still possible. Going beyond this further increases the chance of eviction on iOS.

### **9.3.3. Time Profiler Instrument**

The *Time Profiler*instrument enables you to look at CPU usage, and track any CPU core contentions, across all script and native Unity engine code. It can also be used to do a symbol search to find common problematic areas, or non-performant code patterns:

1. Identify boxing operations in your project by searching ::Box(
2. Find out how much of your frame time is spent rebuilding and updating dirty UI Canvas objects by looking for ::WillRenderCanvases.



[*The best practice guide on Profiling*](https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity1.html) is a good resource to better understand how to dissect your call stack.

### **9.3.4. Xcode Debug Panel**

*The Debug Panel* in *Xcode* displays a memory value that is close to the fully accurate value that is displayed in *VM tracker*. As such, the *Debug Panel* can be used to get a quick understanding of memory usage without using more in depth tools. Be aware that in general we see this value within 5% of the normal value in VM tracker but it should never be taken as fully accurate or within that tolerance.

The value displayed in the Debug Panel is the same value that can be fetched from the operating system on a full release build. [Here](https://oc.unity3d.com/index.php/s/HThKw8xHsi7A0pl) is a link to project that shows how to poll this value from code.

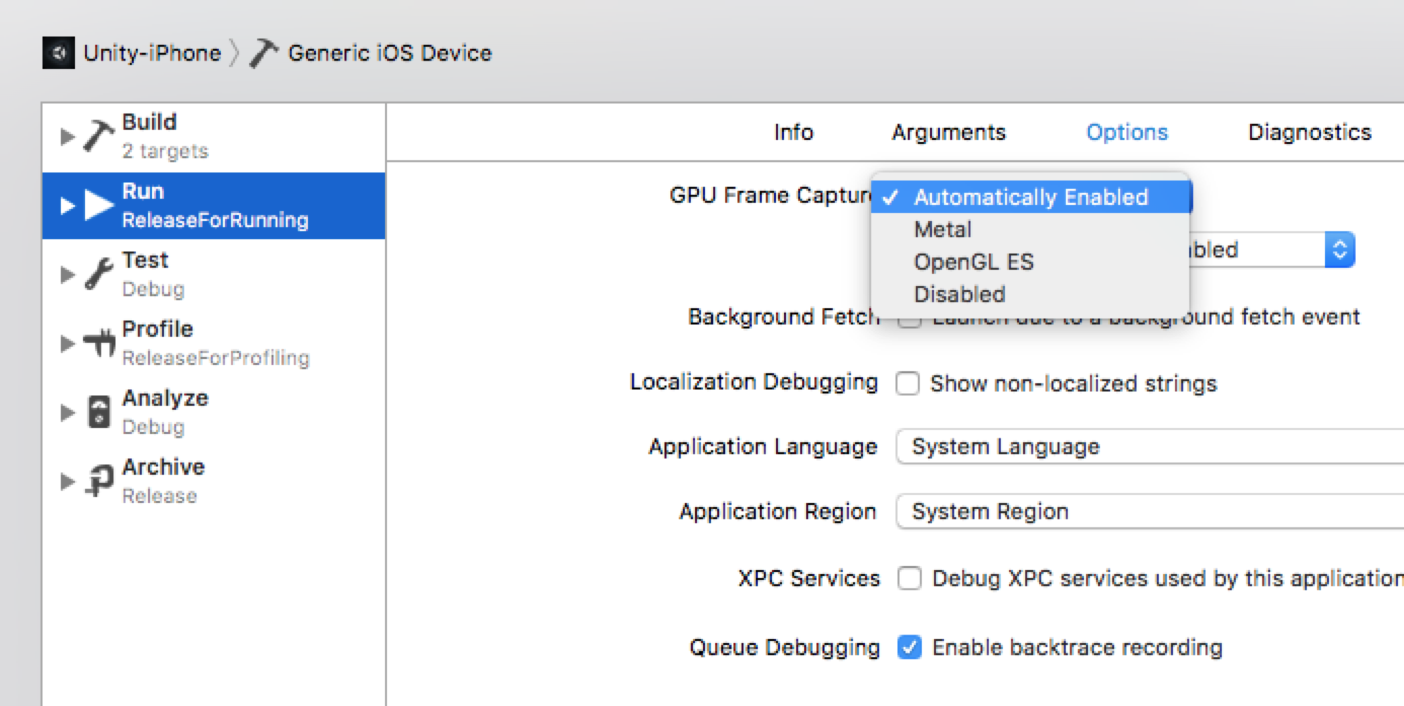
### **9.3.5. Xcode Frame Debugger**



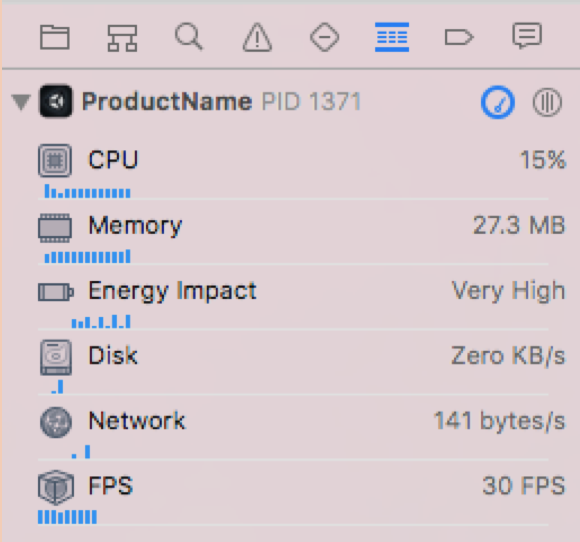
*Xcode Frame Debugger*is a useful tool to capture a frame of a complex scene in your title. Once a frame is captured, use the *Frame Navigator* (View by Performance) to view bottlenecks in your shaders.

*The Xcode Frame Debugger* has multiple uses when it comes to optimising Unity projects.

To set up the *Xcode Frame Debugger*, first check that the Scheme for the project is set correctly for the graphics API that the project is using. To do this, go to *Product -> Scheme -> Edit Scheme*. This will open a pop up window where you can modify the graphics capture settings for the appropriate API from the drop down.

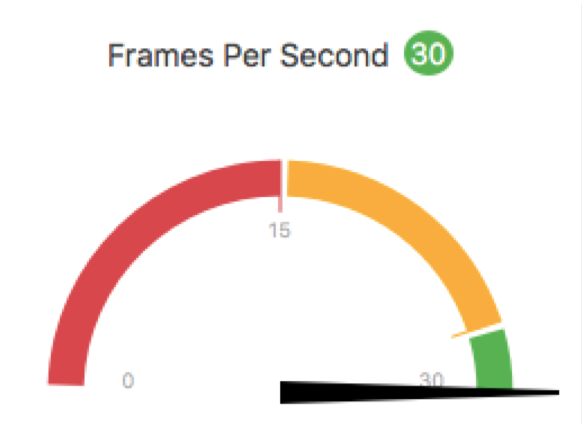


Once this has been set you can run the project from Xcode as normal. Once the project is running, you should then select the debug panel. The icon for selecting this is highlighted in the screenshot below.



To view the FPS counter, you should then select FPS.

If everything has been set correctly you will see the FPS counter working. If it is not working, you may have set the incorrect gfx API for capture. It has also been known that mismatching versions of iOS on the device and the version of Xcode being used can lead to this feature not working.



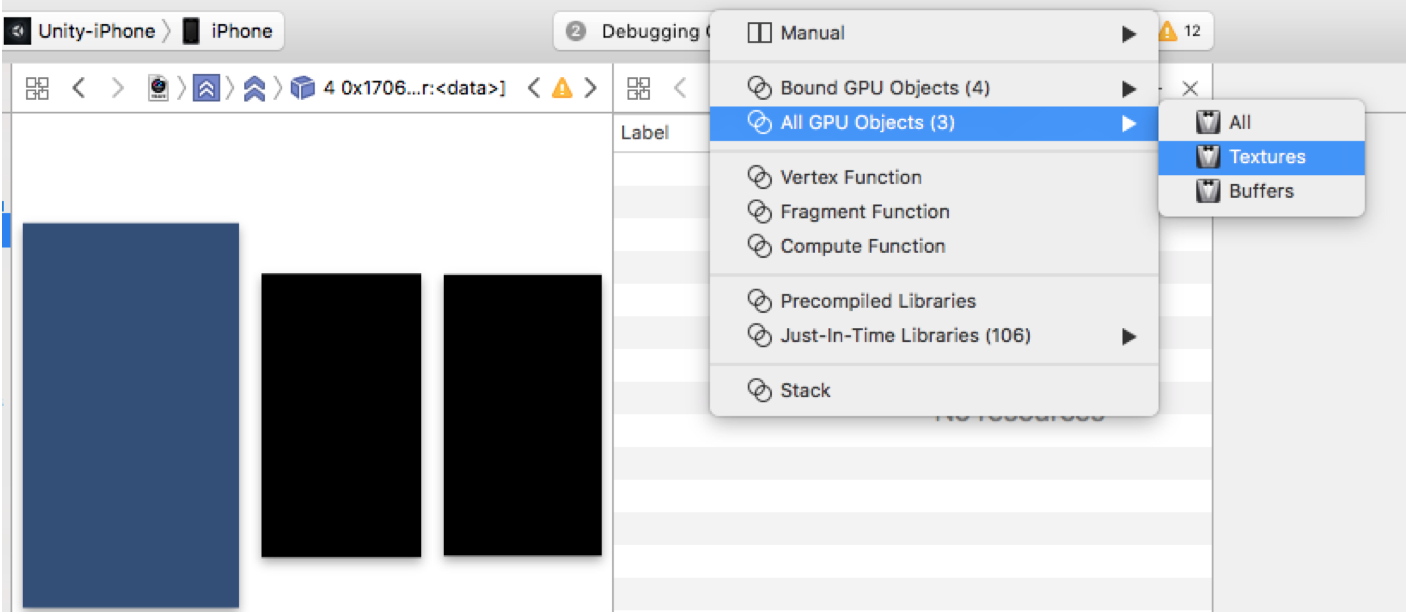
Once you have established that the FPS counter is working, you can then carry out a frame capture. Do this by clicking the camera icon near the bottom of the screen.

Once the frame capture is complete, you can then explore draw calls to determine the best course of action in GPU optimisation.

#### **Using the Xcode Frame Debugger to See Loaded Textures**

Using the *Xcode Frame Debugger,* you can see all of the textures that are currently loaded into memory, not just the ones that are being used by the GPU. This is due to the shared memory architecture of iOS devices.

To see all of the textures in memory, use the selection menu (highlighted below).



This allows you to explore for duplicate assets along with checking the runtime formats and sizes of all textures being used in the project.

#### **Shader & Frame Performance**

A recent addition to *the Xcode Frame Debugger* is the ability to get an overview of the frame being debugged. From the screenshot you can see where this new section can be found. It is useful to be able to see exactly how long the frame took to render, and where time was spent within that frame.

