**Memory**

There are two types of memory, Mono memory and Unity memory.

**Mono memory**

Mono memory handles script objects, wrappers for Unity objects (game objects, assets, components, etc). Garbage Collector cleans up when the allocation does not fit in the available memory or on a *System.GC.Collect()* call.

Memory is allocated in heap blocks. More can allocated if it cannot fit the data into the allocated block. Heap blocks will be kept in Mono until the app is closed. In other words, Mono does not release any memory used to the OS (Unity 3.x). Once you allocate a certain amount of memory, it is reserved for mono and not available for the OS. Even when you release it, it will become available internally for Mono only and not for the OS. The heap memory value in the Profiler will only increase, never decrease.

If the system cannot fit new data into the allocated heap block, the Mono calls a “GC” and can allocate a new heap block (for example, due to fragmentation).

“Too many heap sections” means you’ve run out of Mono memory (because of fragmentation or heavy usage).

Use System.GC.GetTotalMemory to get the total used Mono memory.

The general advice is, use as small an allocation as possible.

**Unity memory**

Unity memory handles Asset data (Textures, Meshes, Audio, Animation, etc), Game objects, Engine internals (Rendering, Particles, Physics, etc). Use Profiler.usedHeapSize to get the total used Unity memory.

**Memory map**

No tools yet but you can use the following.

* Unity Profiler - not perfect, skips stuff, but you can get an overview. It works on the device!
* Internal profiler. Shows Used heap and allocated heap - see mono memory. Shows the number of mono allocations per frame.
* Xcode tools - iOS
* Xcode Instruments Activity Monitor - Real Memory column.
* Xcode Instruments Allocations - net allocations for created and living objects.
* VM Tracker (textures usually get allocated with IOKit label and meshes usually go into VM Allocate).

You can also make your own tool using Unity API calls:-

* FindObjectsOfTypeAll (type : Type) : Object[]
* FindObjectsOfType (type : Type): Object[]
* GetRuntimeMemorySize (o : Object) : int
* GetMonoHeapSize
* GetMonoUsedSize
* Profiler.BeginSample/EndSample - profile your own code
* UnloadUnusedAssets () : AsyncOperation
* System.GC.GetTotalMemory/Profiler.usedHeapSize

References to the loaded objects - There is no way to figure this out. A workaround is to “Find references in scene” for public variables.

**Garbage collector**

* This fires when the system cannot fit new data into the allocated heap block.
* Don’t use OnGUI() on mobiles: it shoots several times per frame, completely redraws the view and creates tons of memory allocation calls that require Garbage Collection to be invoked.

**Creating/removing too many objects too quickly?**

* This may lead to fragmentation.
* Use the Editor profiler to track the memory activity.
* The internal profiler can be used to track the mono memory activity.
* System.GC.Collect() You can use this *.Net* function when it’s ok to have a hiccup.

**Allocation hiccups**

* Use lists of preallocated, reusable class instances to implement your own memory management scheme.
* Don’t make huge allocations per frame, cache, preallocate instead
* Problems with fragmentation?

**Preallocating a memory pool.**

* Keep a List of inactive GameObjects and reuse them instead of Instantiating and Destroying them.

**Out of mono memory**

* Profile memory activity - when does the first memory page fill up?
* Do you really need so many gameobjects that a single memory page is not enough?
* Use structs instead of classes for local data. Classes are stored on the heap; structs on the stack.
* Read the [Understanding Automatic Memory Management](https://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html) page.

**Out of memory crashes**

At some points a game may crash with “out of memory” though it in theory it should fit in fine. When this happens compare your normal game memory footprint and the allocated memory size when the crash happens. If the numbers are not similar, then there is a memory spike. This might be due to:

* Two big scenes being loaded at the same time - use an empty scene between two bigger ones to fix this.
* Additive scene loading - remove unused parts to maintain the memory size.
* Huge asset bundles loaded to the memory
* Textures without proper compression (a no go for mobiles).
* Textures having Get/Set pixels enabled. This requires an uncompressed copy of the texture in memory.
* Textures loaded from JPEG/PNGs at runtime are essentially uncompressed.
* Big mp3 files marked as decompress on loading.
* Keeping unused assets in weird caches like static monobehavior fields, which are not cleared when changing scenes.