

GameMonkey Script Garbage Collection

GM supports two different types of GC (Garbage Collection), traditional scan and sweep, and incremental. These modes are enabled by #defines. It is recommended you use the incremental mode unless you have a good reason not to as it should reduce or eliminate spike drops in performance that could otherwise occur with the standard GC method.

How does it work?

Traditional GC

Traditional GC is very simple and effective, but has the drawback that when the GC runs, and that could be anytime during execution, it may hog the CPU for an unpredictable period of time. The algorithm looks something like this:

GC is triggered by reaching some memory usage threshold.

Recurse over all objects, starting with root objects like threads stacks and globals.

Mark object as valid.

Mark objects children recursively.

Iterate over all objects in entire machine.

If object is not marked, remove and delete it.

All objects must do to work with the GC is implement two functions.

- 1) Mark() Called to call mark() on all its children.
- 2) Destruct() Called to free this objects memory.

Incremental GC

The GC works because each object we care about is connected to another in some way. For example, globals are connected to the machine, general objects are connected to each other in containers and structures. When the GC walks through the objects, anything that is not connected to another, all the way back to a root object must be garbage, otherwise it would be referred to by some other object.

Incremental GC tries to perform a small amount of GC work regularly, perhaps each game frame, so even if it adds a tiny amount of overhead, the performance spikes are reduced as it never has to make passes through the entire set of objects. The method used in GM could technically be known as a Soft Realtime Incremental Garbage Collector via a Write Barrier. The method is based on a tri-color marking scheme and the newly allocated objects are marked 'black' which means they are not considered in the current cycle. The configuration choices of the GC are designed to maximize performance. The implementation is based on "Non-Compacting Memory Allocation and Real-Time Garbage Collection" by Mark S. Joshnstone 1996. I would encourage all interested in real time garbage collection schemes and also memory allocation strategies to read this document.

The GC executes only when a cycle has been triggered by reaching a memory usage threshold. It executes only when a thread has yielded, not on every allocation, which means that a non-yielding thread could potentially accumulate a large amount of allocated memory before the GC gets a change to run. The implementation also assumes that that GC is configured to do enough work on average to eventually complete. Because newly allocated objects are marked black, this is guaranteed to occur, at the expense of more memory used. It is possible that the programmer may need to tweak a) the amount of object to be traced each frame, and b) the memory threshold to being the GC cycle. These factors control the efficiency of the scheme. Because the GC

implementation is 'soft', certain constraints are relaxed in the interest of average performance. Namely that the GC will not set a hard limit on allocated memory, and that when the GC does work, it is not guaranteed to take less than a certain period of time. 'Hard' realtime GC implementations have calculated worst cases where they trace limited blocks of memory and the worst case is that block entirely consists of pointers to other objects requiring future tracing.

All objects must do to work with the GC is implement two functions.

- 1) Trace() Called to add its children to the GC process, and optionally only do this to a limited number of child objects.
- 2) Destruct() Called to free this objects memory.

Future improvements to the current implementation:

- 1) Perhaps set a soft memory limit at which GC should start a cycle, and a hard memory limit at which the GC should perform a complete atomic execution, freeing all garbage to prevent the hard limit from ever being exceeded. The programmer would tweak the soft limit in an attempt to prevent the hard limit from ever being reached. The programmer uses his/her knowledge of the kind of work the program is doing to tweak this value.
- 2) Implement partial Trace() function. The Incremental GC is designed to allow an object to trace only part of its children in one frame, and resume work in the next frame. It does this by returning a completion flag or not, and by storing a 'context' in a structure passed between the GC and the object type code. I have not implemented this because I don't believe it is necessary in the kinds of programs I use GM for. The current worst case is that a object has MANY children and ALL those children must be added to the trace process. Note that the children are merely added to the process, they are NOT traced recursively as in the traditional GC.
- 3) There is one outstanding issue with the current implementation of the Incremental GC in GM. Due to the way it works, at no point in the GC execution cycle is the minimum amount of memory known, so automatically calibrating the memory threshold at which to start the next GC cycle is difficult to calculate. I tried to estimate the limit, but when the estimate was too low, the GC fired prematurely, and when I estimated too high, even by a tiny margin, the GC fired later and later. Due to the way it works, if no code was executed, but the GC was allowed to complete, it actually has to do two complete cycles to ensure all garbage is collected. This is due to the configuration of the GC which is for maximum performance as the expense of a small amount of extra memory used. Perhaps there is a solution to this issue other than what was suggested in improvement (1).