GameMonkey Script Garbage Collection

GM currently supports two different types of GC (Garbage Collection), traditional *scan and sweep (atomic)*, 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 thread 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 *virtual 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 does not perform *memory compaction*. This reduces CPU time, but may leave memory in a fragmented state, possibly reducing memory access 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 (unless configured to do so), 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 begin 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 (by default) 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.

The current version of the garbage collector allows two memory limits to be set, the *hard* upper limit and the *soft* lower limit. The soft limit should be less than the hard limit and both should be well above the required memory to run the application. If the soft limit is set too high, it may not finish collecting before the hard limit is encountered causing a full collect to occur. If the soft limit is set too low, the collection will restart more regularly than necessary. To calibrate the limits, you should determine the maximum amount of memory the virtual machine will ever need during execution and set the hard limit as far above this value as possible. The soft limit should then be set well above the required limit also, but some percentage below the hard limit, so there is plenty of time for the collector to complete its operation before the hard limit is reached. You may be able to accomplish this by setting the hard limit too low during development and observing how high the memory grows with auto-size enabled. You may observe the garbage collection statistics counters to make sure collection does not occur too regularly.

Note that the current version of the GC may show two collection starts at a time. This is because the collector goes through phases and actually takes two passes to collect all garbage. There is also a warning counter that may be monitored in conjunction with the other stats counters. In the current version the warning counter will rise with the restart counter. What should be monitored is not the exact number of warnings or restarts but the rate at which they increase. For example, a very poorly configured soft and hard limit can cause the collector to restart continuously. Future versions may improve the statistics measures to help tune memory management in a more understandable manner. If careful memory management or performance tweaking is not an issue, just enable the auto-size option and don't be concerned about garbage collection at all.

Performance Summary

The following configuration parameters effect performance while the GC is active:

1. The amount of memory the *virtual machine* is allowed to use, set as the *hard limit*. The more memory available, the less frequent GC will occur. This limit should be well above the amount actually required by the application or it will cause GC to run continuously, assuming there actually is enough memory in the physical system to continue execution.
2. The *soft limit* setting. This memory limit should also be set well above the amount required by the application, but some way below the *hard limi*t. The difference needs to be enough memory to allow the application to run smoothly while the GC phases occur. If this limit is set too close to the *hard limit*, the incremental GC will stop if the *hard limit* is reached and a full collect will take place immediately, possibly slowing the application.
3. The number of objects the GC tries to collect per frame. The GC will trace at least this many objects per frame as long as objects are available. This number effects how much work the GC does per frame while it is active and how many frames the collection will take to complete. This

number needs to be high enough to complete collection between the soft and hard memory limits and low enough to take an acceptable amount of CPU time per frame.
4. The number of objects reclaimed by the GC per frame. Once the GC has determined that an object is garbage, it is set aside to be reclaimed. The *destruct* function is called during the reclamation phase and memory is recovered. This number needs to be high enough to complete collection between the *soft* and *hard* memory limits and low enough to take an acceptable amount of CPU time per frame.

The following factors effect performance while the GC is active:
1. The number of objects in the entire *virtual machine*. This determines the amount of time taken to complete a full GC cycle.
2. The rate at which objects are created and destroyed, or the lifespan of the objects. The higher the turn over, the quicker memory will be used up and sooner GC will start again. Note that the implementation is quite efficient in that objects allocated while GC is inactive and destroyed before the GC becomes active have a minimal GC cost. They do however accumulate in memory until they are reclaimed.
3. The number of *root* objects to scan. These are things like thread stacks, types and globals. The root objects are always traced atomically. Note that these roots may not necessarily cause a performance issue by themselves.
4. The number of references in a single object. A *table* or *user object* may contain hundreds or thousands of references to other objects. The tracing of these objects occurs in one go. Although the code supports the ability to break a single object trace into multiple phases this has not been implemented (since it has never been needed). As only one level is traced at a time, the object reference tree as a whole will not influence performance. (The *reference tree* refers to the full set of child or member objects).
5. The *write barrier* is called at *set* and *get* operations that work with object references. This adds a tiny overhead to the *virtual machine* performance. When the GC is not active (because there is free memory available below the *soft limit*) the cost of the *write barrier* is minimal.

Interaction with application owned objects

The application may own objects that are not linked to others (such as the *global table*) within the v*irtual machine*. These objects must be traced like *root* objects when the GC starts a new cycle. The v*irtual machine* allows a callback function to be set where this tracing should occur. Failing to do this will cause the GC to reclaim objects that are still in use by the application but inaccessible to the VM.

When to disable the Garbage Collector

The GC may be temporarily disabled when required. When disabled, if the GC is currently doing work, it will pause, if it is currently inactive, it will not start. The main reason to disable the GC is when an application is allocating multiple objects and they are not immediately connected to the root (via other objects etc.). If the GC is configured to run on every allocation and it is not disabled, it may begin collecting newly allocated objects before they get a chance to be linked properly.

Future improvements to the current implementation:

1) Implement partial Trace() function. The Incremental GC is designed to allow an object to trace only part of its (immediate) children in one frame, and resume work in the next frame. It facilitates this by returning a completion flag, 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 (immediate) children are merely added to the process, they are NOT traced recursively as in the traditional GC.
2) Improve the GC counters so they are more understandable. This would simplify GC configuration and tweaking, and possibly improve automatic memory management.