A Memory Management System for Game Development

Travis Woodward

April 10, 2012

Contents

1 Abstract

General purpose Memory Management systems are often poorly optimised for use in game development. Games place very specific demands on a memory management system (discussed in Section 3) which the native memory management systems implemented within the runtime environments of popular programming languages (e.g. std::malloc and std::free, or the .Net Garbage Collector) are not properly equipped to meet[1].

In this project I intend to fully define those requirements, and design a Memory Management system to meet them, whilst also exploring the potential to adapt modern Automatic Memory Management techniques for use in a Game Development environment.

2 Background

2.1 Performance needs of the Games Industry

Game developers have always pushed the performance boundaries of whatever platform they develop for. Whether they are targeting games consoles, desktop computers, mobile devices, or web browsers, they are always faced with limited resources with which to meet their customer's expectations of improved graphics, richer, more interactive environments, more intelligent AI, and more realistic physics.[2]

Within these confines, very few trade-offs can be made that sacrifice performance in order to simplify development. For this reason games developers often develop in C/C++, for both it's native performance characteristics and in order to manually manage memory to avoid the performance penalty that comes with Garbage Collection.[1]

2.2 Issues with the standard C++ Allocator

2.2.1 Locality of allocation

Standard Allocators such as that in the C++ runtime often manage free blocks of memory using a linked-list structure often referred to as a 'free list'. It then uses a first-fit algorithm to find a block of memory large enough to meet the requested allocation.[3]

However, Games often allocate the majority of the resources that they need for a set portion of the program (e.g. a level) at once, and often related resources are allocated at the same time, and are also likely to be accessed at approximately the same time during execution.[4]

A first-fit algorithm gives no guarantees about the locality of objects allocated at the same time, meaning that it is very possible for two closely related objects that are allocated one after the other to be stored in completely different memory locations, leading to a cache miss every time this pair of objects is used by the Game.

2.2.2 Fragmentation

In most languages that use Manual Memory Management, when an object's memory is freed it is returned to the Allocator for reuse. If the program creates objects that have a large variety of different sizes (as is particularly common in Games) it is likely that the next object to be stored in that location in memory will be smaller than the deleted object[5]. If the remainder of this block of memory is smaller than the space required by the types of objects required by the program this space will remain vacant.

If this happens in a large enough proportion of cases we can reach a situation where a significant portion of the heap is empty, but the Allocator is unable to find any contiguous blocks of memory large enough to fulfil an allocation request. This is known as *memory fragmentation*.

2.2.3 Dangling pointers

When an object is manually deleted, there is no guarantee that there are not references, or pointers, to that object still stored within active objects. Nor is there any guarantee that those objects will not attempt to access the now deleted object via those pointers. If the object has since been overwritten (or partially overwritten) with the data of a new object, this could easily lead to an invalid memory access, or worse, 'garbage' data being returned and used by other systems in the Game, leading to a later crash or incorrect behaviour in a different part of the program[6].

2.2.4 Memory leaks

If the last pointer to a particular active object is deleted or overwritten, then that object becomes unreachable, and can no longer be deleted, meaning that it's memory will never be returned to the Allocator for reuse. This object will remain in memory until the Game terminates and all of the memory that the Game used is reclaimed by the operating system. If this happens too often then the heap could potentially become full of these unreachable objects, causing the Game to run out of usable memory.

2.3 Reference Counting

One method of Memory Management that is widely used in Games is reference counting[1]. Reference counting works by keeping track of the number of references, or pointers, pointing at each object. When the last reference is removed from a given object, that object is deleted automatically (as it would otherwise be unreachable) and it's memory is returned to the Allocator as usual.

This is commonly implemented in C++ by way of Smart Pointers[4]. These are pointers wrapped in classes which allow the developer to use them in much the same way as normal pointers, but that also take care of the reference counting and deletion of objects when their last reference has been removed.

Reference counting does not make any demands on where the Allocator creates new objects, and so cannot address the issues of locality or fragmentation. It does however eliminate the problem of dangling pointers (assuming that all references are wrapped inside smart pointers). Since objects are only deleted when the last reference to that object has been removed, we cannot have the case of a reference pointing to an object that has been deleted (since then the object's reference count could not be zero)[6].

In general purpose applications and research, the overhead of incrementing and decrementing reference counts on each pointer operation is considered to be too expensive to be efficient for most applications[7]. However, the overhead of reference counting is predictable given that it is a part of normal pointer operations, and it can be evenly distributed across multiple frames without knowledge of the implementation of the Memory Management system[4].

Whilst the overhead of reference counting is deemed acceptable in Game Development there is an issue with this method of Memory Management. Reference counting is unable to collect cyclic references[8]. For instance, if an object A references object B and object B references object A, then even if all other references to these two objects are deleted, the reference count for each will still be greater than zero, and so neither will be deleted, causing a memory leak.

2.4 Tracing Garbage Collection

Tracing Garbage Collection works on the following principle: If we start from those objects that we know to be reachable, e.g. those that are referenced from the stack, static, or global objects, and follow each pointer that they hold to other objects on the heap, then these objects must also be reachable. If we repeat this process until we exhaust all of the pointers of every object we have visited, then we will have visited precisely those objects which are reachable. Any other objects on the heap, therefore, must be unreachable, and their memory may be reclaimed[9].

Clearly this eliminates the possibility of unreachable objects filling up the heap (i.e. memory leaks), provided the Garbage Collector runs often enough in relation to the rate at which objects become unreachable. It also addresses the issue of dangling pointers: an object is deleted only if it is no longer reachable from a live object, meaning that the only objects that may still hold pointers to this object are those which are themselves unreachable, and so will never attempt to follow those pointers.

Based on the definition of a tracing Garbage Collection above, they do not inherently solve the issues of locality of allocation or heap fragmentation. However, some tracing Garbage Collection algorithms do indeed address one or both of these issues. We shall explore one family of tracing Garbage Collectors below, and discuss how it interacts with these two issues. We will also discuss it's shortfalls, and then look at how it can be improved upon using other Garbage Collection techniques in order to improve it's suitability for Game Development. Other families of collector will be discussed in the project itself to examine why they are less suitable than the family described here.

2.4.1 Terminology

Before we describe these algorithms we need to briefly define some key terms that we will be using to describe the way in which these algorithms work. The first of these terms is the 'Root set'.

These are the objects that are directly referenced from areas of memory that are known to be reachable. These areas include the program stack, global variables, and static variables belonging to classes. In each case, if a pointer to an object exists in one of these areas of memory then the user program will always be able to access that pointer, and so will be able to reach the object it points to. We define these objects to be the 'Root set' or 'Roots'[4]. Every tracing Garbage Collector must in some way derive it's set of reachable objects from the Root set.

The second of the terms we need to define is the 'Object Graph'. This the term we use to describe the generic structure of objects on the heap. Objects on the heap can be modelled as a directed graph, with the objects themselves as the nodes of the graph, and the pointers being the edges, with the direction of the edge flowing from the object holding the pointer to the object being pointed to[4].

The third term we need to define is the 'Mutator'. This is the name given in academic texts on Garbage Collection to the user program, or in our case, the Game[4].

2.4.2 Mark Compact Garbage Collection

Mark Compact Collection is a straight forward algorithm based on the definition of a tracing Garbage Collection outlined above, combined with a 'Compactor' which de-fragments the heap by moving the active objects in such a way that they fill all of the empty space vacated by the unreachable objects[10].

The first part of the algorithm is the Mark stage. The Mark stage starts with the root set, and traverses the object graph, visiting each reachable object. When an object is visited it is 'marked' in some way, usually by setting a specific bit flag either in the object's header or in some separate table stored in the memory belonging the Garbage Collector. When the collector comes across an object that has already been marked it ignores the object and continues to traverse the other unmarked objects[6].

Once the collector has finished marking the objects of the heap, it then moves on to the Compact stage. There are two forms of Compact stage used in this family of collectors: Sliding and non-sliding. Non-sliding compactors move objects from one end of the heap to fill gaps left by dead objects at the other end of the heap. Sliding compactors instead move each object as far as it can go in

the same direction one after another, effectively 'sliding' them to eliminate any gaps between objects[6]. In this way at the end of a collection all live objects lie in a contiguous region of memory.

In a Mark Compact system, after a collection the heap is divided into two regions, one containing live objects, the other containing free space. This means that the Allocator can satisfy allocation requests simply by allocating the first x bytes of memory after the end of the live object region, where x is the size of the allocation request[11]. Clearly from this definition Mark Compact Collectors deal explicitly with the issue of heap fragmentation, and guarantee that objects allocated at the same time are placed (at least initially) near each other in the heap.

It should be noted that sliding compactors maintain any locality of objects in memory from one collection to the next, whereas non-sliding compactors may move an object in order to fill a gap in the heap, but neighbouring objects may be moved to somewhere entirely different on the heap, and so it does not maintain the locality of objects at allocation[4].

It should also be noted that non-sliding compactors pre-suppose that objects are of the same (or at least similar) size, otherwise we may end up not actually solving the fragmentation problem at all, which would remove one of the main benefits of this algorithm over other families of collectors[6]. This can be dealt with to some extent if objects are usually of one of a small number of different sizes, as each size of object can be managed in a separate region of the heap, however objects in Games do not usually follow this pattern[2]. It seems clear given this information that a sliding compactor would be better suited to our needs.

While the main strength of Mark Compact (it's automatic de-fragmentation of the heap) comes from it's compaction stage, this is also the cause of it's major weakness. Because Mark Compact moves objects to different locations in memory, it must deal with the fact when it moves an object, all references to that object will immediately become invalid[2]. This issue is where most of the complexity of Mark Compact algorithms come from, and part of the reason that they has fallen out of favour in recent years.

There are several approaches to updating the pointers in the heap to the new location of an object. These will be discussed in full in the project itself. However, for our purposes we actually look to a solution currently used for dealing with this issue in standalone memory compactors already employed in the Game Development industry.

This method involves replacing all pointers in the heap with handles. These handles, instead of holding the address of the object itself, hold an index into a table stored at a fixed location at one side of the heap (or possibly statically allocated)[2]. Each object also holds the index of itself somewhere in it's header. In this way, when the compactor moves the object, it only need update the address in the appropriate table entry to ensure that all objects with references to this object will be able to access it at its new location.

This method comes with a significant memory overhead, effectively requiring space for an additional pointer and an additional index per object. It also places a performance penalty on accessing objects, as such an operation would require two reads rather than one to deal with the extra level of indirection, and since the handle table will likely be stored at a very different address to both the handle and the object being accessed, potentially leading to extra cache misses. However, despite these drawbacks, this method is currently used in the Games industry to deal with moving of objects during heap de-fragmentation[2], so for our purposes we may assume that this level of memory overhead and performance penalty is acceptable.

The second major drawback with the algorithm described is that currently an entire collection would need to happen atomically. In a large and complex Object Graph (as is commonplace in Game Development) this is likely to take several microseconds, particularly on low powered hardware such as mobile devices. During this time, the Mutator may not run, else it might change the connectivity of Object Graph behind the collector's back, leading to live objects being incorrectly collected[12].

In Game Development, in order to reach the standard performance benchmark of 60 frames per second(fps) the time between one frame being rendered to the screen and the next being rendered must be less than 16.66ms. Even to reach the lower acceptable benchmark of 30 fps the delta between frames must be less than 33.33ms[1]. In either case a full Garbage Collection may take up most of this budget, or may possibly even exceed it. This will cause frames to not be rendered, making the Game appear to stutter, which is not an acceptable outcome in Game Development. There are however families of Garbage Collectors that are designed to avoid this 'stop-the-world' atomic approach to collection.

2.4.3 Incremental Garbage Collection

The principle of Incremental Garbage Collection is that the Garbage Collection can be broken down into small chunks that can be interleaved with the user program[13]. Most incremental collectors use some variation of Dijkstra's tricolour abstraction, whereby the object graph is partitioned into 3 sets: Black, Grey, and White.

White objects are those that have not been visited by the Garbage Collector since the previous Garbage Collection. Grey objects are those that have been visited themselves, but their children have not all be visited yet. Black objects are those that have been visited and all of their children have also been visited. Hence Black objects may not reference White objects, only Grey objects and other Black objects[13].

The collector starts from the root set and marks all of the objects directly referenced from the root set as Grey. The collector can then pick any Grey object, colour each of it's children Grey, and then colour the object itself Black. It does not matter which Grey object the collector starts from or chooses next. When there are no further grey objects in the heap the collector has finished, and any white objects are unreachable[13].

One element of this algorithm that helps it to run incrementally is that there is no need for a stack to help traverse the object graph. The collector may simply scan sequentially through the heap to find the first Grey object and continue from there, making it easy for the algorithm to resume if interrupted by the mutator[13]. However, precautions need to be taken to ensure that this algorithm operates correctly. In particular there is an issue with the mutator changing the structure of the heap in between the collector starting it's collection and completing it. For example, if a reference contained inside an object that has been coloured Black is then replaced with a reference to a White object, then the collector may never visit that White object, since it's parent is not white or grey. At the end of the collection this object would be collected even though it is live, leaving a dangling reference in the parent[4].

There are two main methods that are used to handle this issue. They are called Read Barriers and Write Barriers. Read barriers operate on the principle that if an object accesses any white object during collection, the object accessed must also be live (since otherwise it couldn't have been accessed), and should be coloured grey. However, Read Barriers are generally seen as very expensive, since a very large number of reads generally take place, and the extra cost per read could lead to a large performance penalty overall. The alternative is Write Barriers. Write Barriers operate on the slightly different principle that when a pointer in a live object is overwritten with a pointer to a different object, then this new object must be live, and so should be coloured grey. Since pointer writes are less common than pointer reads this is considered much cheaper than a Read Barrier[6].

Both of these methods have a potential downside, in that they make no guarantee about whether their parent object has been visited by the collector yet. Indeed, it is possible that after the child has been coloured Grey the parent will become unreachable before being reached by the collector. This would mean that we could have unreachable objects marked as Black or Grey. This in itself

is not an issue per se, as they would still be collected at the next collection, but in the mean time they will remain in memory, as what is known as 'floating garbage'[6].

2.4.4 Generational Garbage Collection

One further method of amortizing the cost of Garbage Collection across multiple frames is that of Generational Garbage Collection. Generational Garbage Collection is based on the observation that a large proportion of allocated objects only live for a very short time. This means that there is potentially performance to be gained by collecting objects that have been allocated recently (i.e. 'young' objects) more frequently and only rarely collecting older objects, since by the assumption above the majority of garbage collected will be found among this young generation[14].

Generational algorithms divide objects in the heap into two or more partitions or 'Generations' based on age. Younger Generations are collected more often than older generations. Objects that survive a given number of collections in one generation are 'promoted' to the next oldest generation[6].

One of the major issues with Generational Garbage Collection is that of inter-generational references. In order to determine the liveliness of objects in the youngest generation we need to determine which are referenced from either the root set or objects in older generations. However, we wish to find these references without doing a full trace of the heap, as then we would lose any benefit of using a Generational Collector. The way this is normally achieved is through a Write Barrier. When the address of an objects in the younger generation is written to a reference, this is trapped by the Garbage Collector and stored, usually in a list or vector. These references into the younger generation effectively form it's root set[4].

However, the relationship between generations is two way, meaning that in order to collect an older generation independently of a younger generation it is necessary to trap references from the younger generation to the older generation. However, since we are assuming that the majority of objects in the younger generation will quickly become unreachable this means that many of these young-to-old references will also be short lived, which could lead to floating garbage in the older generation. The normal way to deal with this is for the younger generation to also be collected whenever the older generation is collected[4].

There is also a balance to be struck as to how many collections objects should need to survive before being promoted. A simple answer is to promote objects after a single collection. This means that the collector does not need any meta data about the objects to determine how many collections they have survived.

However it could lead to objects that are created just before a collection being promoted before they have a chance to become unreachable[6]. They would then have to wait for a larger collection to have their memory reclaimed.

On the other hand, choosing a larger number of collections as a criteria for promotion leads to a requirement to store extra data about these objects to determine how many collections they have survived so far. It can also make the promotion mechanism more complicated, since these objects need to be picked out from other live objects that have not yet survived the required number of collections, but still need to be retained, leading to fragmentation in the younger generation[15].

There are two downsides of Generational Garbage Collection as a whole. The first is that if the application does not fit with the assumption that many objects live for a short period of time, then it can become very inefficient, with the majority of objects being promoted out of the younger generation only to later become garbage in the later generations[16]. In Game Development many small objects are created on a per-frame basis, for uses as diverse as graphical particle effects to bounding geometry within the physics engine[2]. Therefore for our purposes this assumption should hold.

The second downside is that there will still be longer pause times when the older generations are collected, as these will require younger generations to be collected as well. For Game Development, this means that, although frame misses due to Garbage Collection would be much less frequent than with a stop-the-world collector, they would still occur. Depending on whether or not the developer can ensure that these collections happen at moments in the Game when a drop in framerate will not be noticeable (e.g. when a level is loading), this may still be unacceptable for a commercial Game[2]. However this could be mitigated by creating a hybrid system that collects the older generation incrementally, preventing the longer pauses from occurring.

3 Requirements

The requirements for any Memory Management system to be used in Game Development can be split into two groups: Essential and Desirable Criteria. Essential Criteria are defined as those that any system lacking them would be entirely unsuitable for Game Development. Desirable Criteria are defined as those aspects of a Memory Management system that would either improve the performance of the Game using them, or that would decrease development time by removing the potential for certain types of error. Not all Desirable Criteria will be implemented in this project.

3.1 Essential Criteria

The Essential Criteria for the project are defined as follows:

- Allocation times that do not depend on the size of the object or on heap occupancy
- No dependencies on the implementation of the compiler, e.g. a reliance on the existence of an object header, the location of fields in relation to the start of an object etc.
- Low heap fragmentation
- Predictable locality of reference
- Finalization of objects
- A developer definable upper bound on time spent on Memory Management each frame

3.2 Desirable Criteria

The Desirable Criteria of the project are defined as follows:

- Automatic Memory Management
- Minimal cache misses
- Memory alignment
- A separate area of the heap for large objects to avoid expensive copying of large data structures common in Games
- Compatibility with third party software libraries
- Debug meta-data to allow tracing of allocations, identification of objects that remain live significantly after their last use etc.
- A feedback mechanism to increase the amount of time spent reclaiming memory smoothly when heuristics show that the heap is low on memory.
- Thread safety

4 Implementation Plan

4.1 System Design Outline

The system is to be a hybrid between a Generational and Incremental Garbage Collector that uses a Mark Compact algorithm with sliding compaction as a basis for collection. The system will be built from a number of components listed below:

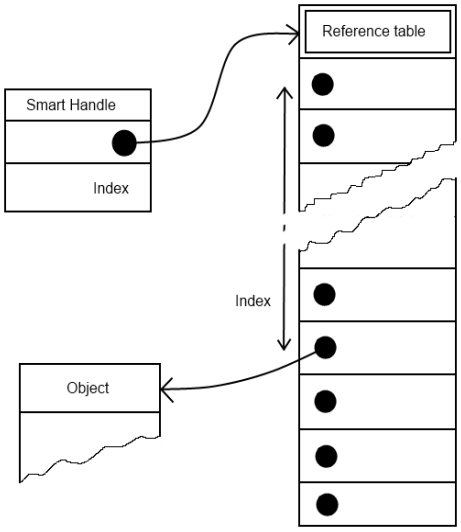
- Smart Handles
- Common Base class
- Allocator
- Compactor
- Tri-Colour Tracer
- Nursery generation
- Older generation
- Large Object Handling
- Additional Features

We will briefly describe each of these and how they interact to form the Memory Management system.

4.1.1 Smart Handles

Handles encapsulated within objects will be used in place of any pointers or references to objects being managed by the system. In order to minimize compaction costs these 'Smart Handles' will hold an index into a table to the side of the main heap which in turn will hold the memory address of the actual object to be referenced. These smart handles will also override the de-referencing operator (*) to allow developer using the system to use the Smart Handles in the same way they would use normal pointers.

The Smart Handles will also be used to facilitate the housekeeping required by other parts of the system, such as implementing write barriers and for locating all of the references within an object during the tracing stage of a Garbage Collection cycle.



A second type of Smart Handles will also be included in the design. These are used for references that reside on the stack or as Global or Static variables, i.e. references that will form the root set of the Object Graph. These will register themselves with the collector so that they can be found for use during collections. This avoids the system being dependent on the specifics of how the compiler allocates space for objects on the stack, as Global variables, or as Static variables.

Although the use of Smart Handles results in an extra cost for de-referencing operations, as handles are currently used in the Games industry to aid in de-fragmentation of manually managed heaps, the extra cost can be assumed to be acceptable.

4.1.2 Common Base class

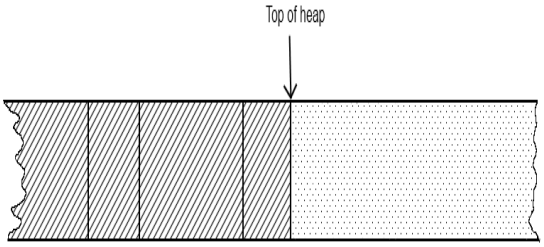
A common Base class called Managed_Object will be the base for all objects who's memory is managed by the system. This is required in order to remove any dependency on the implementation of the compiler. This class will allow the Tracer (see below) to find all of the references held in the class by providing the facility for Smart Handles to register themselves with the object on creation.

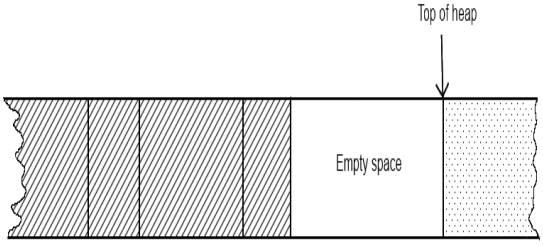
4.1.3 Allocator

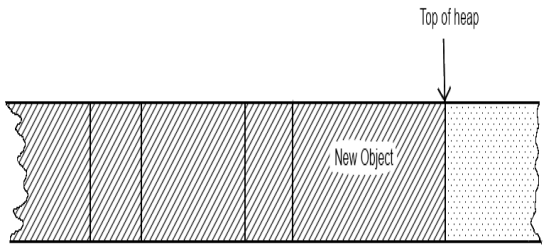
The Allocator will be a standard Allocator as commonly used in Mark Compact systems as described in section 2.4.2. On initialization it will request a large (developer defined) block of memory from the operating system to be used to store the heap for the system. From that point it will fulfil allocation requests

simply by increasing the pointer that marks the top of the heap by the size of the allocation request and returning the previous address of this pointer.

The following diagrams illustrate the process of allocating memory for a new object.







The Allocator will also be responsible for checking that the system is not about to run out of memory and taking the appropriate action (e.g. calling for a full, uninterrupted collection of the heap to reclaim all memory from Garbage).

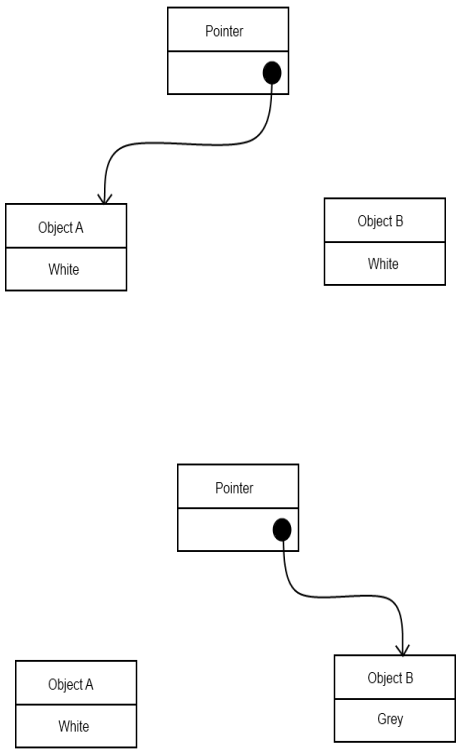
4.1.4 Compactor

A sliding compactor will be used to simultaneously de-fragment the heap and to reclaim space occupied by dead objects. Any dead objects will have their destructor called to allow them to release any unmanaged resources, and will have it's entry in the reference table used by the Smart Handles set to null. The compactor will be able to run incrementally.

4.1.5 Tri-Colour Tracer

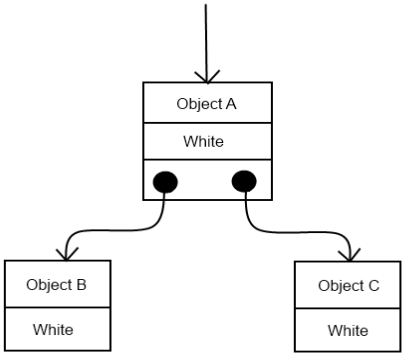
As described in section 2.4.3, Incremental collection is often implemented using a Tri-Colour algorithm that marks objects objects as either White, Black, or Grey depending on whether the Tracer has visited them and all of the objects that they reference.

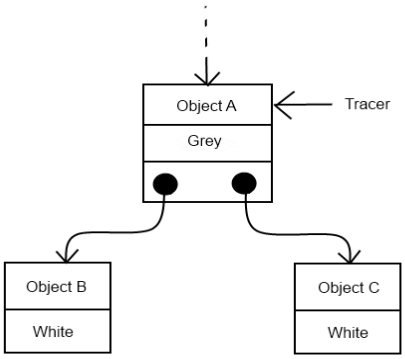
In order to protect against the Black-White reference invariant being broken, write barriers will be used, implemented within the Smart Handles mentioned above. This write barrier will colour any object that has it's reference written into a Smart Handle as Grey. The diagram below shows the effect of the barrier on an object when a pointer is overwritten with it's address:

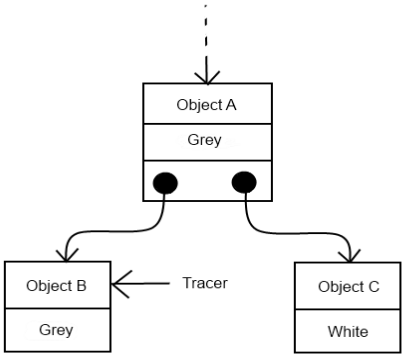


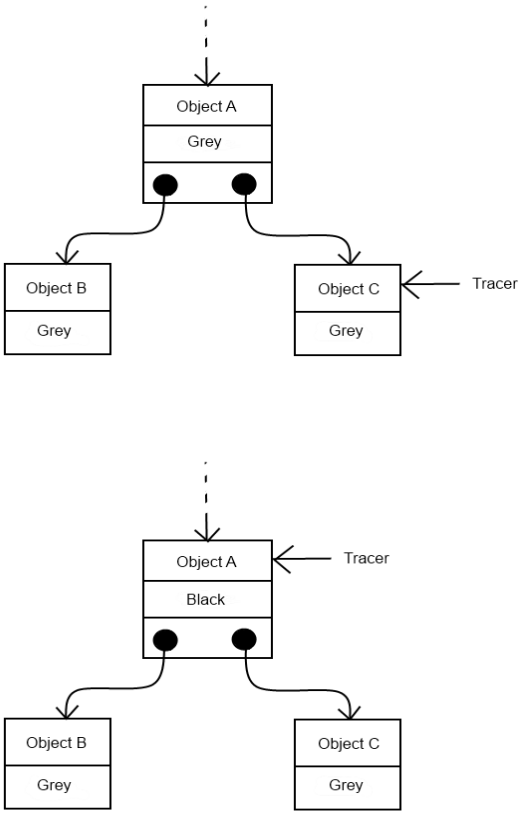
The Tracer will also make use of the Managed_Object base class in order to reach each of the references stored within each object.

The following diagram shows the initial state of 3 objects in the object graph, followed by diagrams showing the path of the Tracer and how it alters the colour of these objects:


Object A
White
Object B
White
Object C
White


Object A
Tracer
Grey
Object B
White
Object C
White


Object A
Grey
Object B
Tracer
Grey
Object C
White

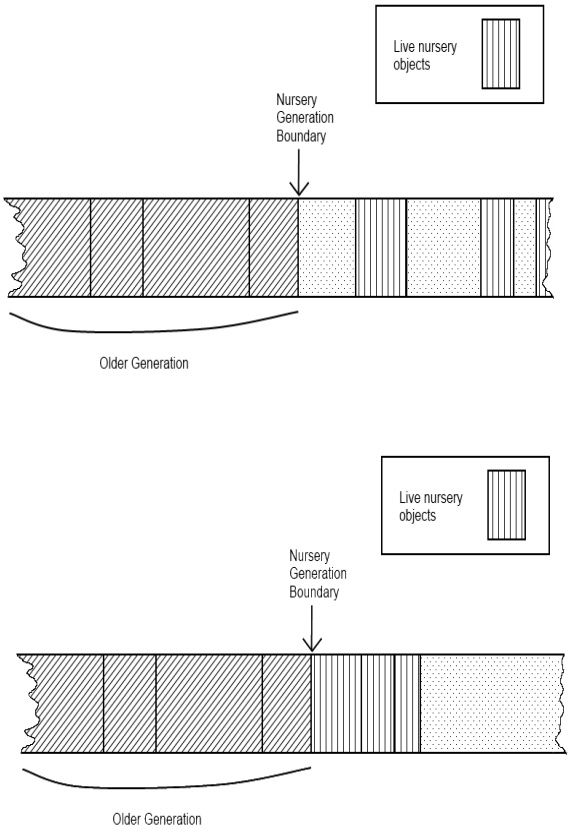


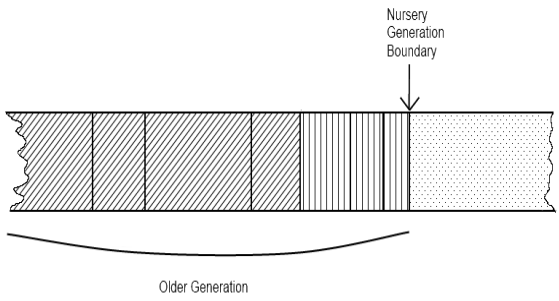
4.1.6 Nursery Generation

Newly allocated objects will be marked as being in a 'Nursery Generation'. This Nursery Generation will be managed separately from the rest of the heap. At set intervals (measured in whole frames) a full Mark-Compact cycle on the objects in the Nursery Generation will be completed. Surviving objects in the Nursery Generation will be slid to the bottom of the Nursery Generation by the compactor. Due to the fact the Allocator works simply by updating a pointer this implies that the bottom of the Nursery Generation is also the top of the remainder of the heap. Therefore the surviving objects will be promoted out of the Nursery Generation simply by moving the pointer that marks the boundary between the Nursery Generation and the rest of the heap.

The following diagrams show the process of Nursery object promotion. In the first the Nursery Generation is uncollected, and so is fragmented by dead

objects. In the second the Nursery Generation has been collected, with the live objects being compacted up against the older generation. In the final diagram the pointer to the boundary between the generations is bumped so that the older generation now includes the newly promoted objects:





The root set of the Nursery Generation will be determined by use of a write barrier to register all references to objects within the Nursery Generation from elsewhere in memory.

In order to prevent objects in the rest of the heap from being incorrectly reclaimed by only being reachable through the Nursery Generation, the normal write barrier will also apply to objects in the Nursery Generation.

The collection cycle of the Nursery will not be time bound, and must be completed atomically. However, the frequency with which the Nursery Generation is collected can be tuned to the number of allocations made since the last Nursery Generation collection.

4.1.7 Older Generation

Objects that have been promoted from the Nursery Generation will reside in an Older Generation that will be managed by an Incremental collector made up of the incremental Tri-Colour tracer and the incremental compactor, both described above. Objects newly promoted from the Nursery Generation will be marked as black. The write barrier described above will ensure that the no black-white references invariant is maintained.

A time budget (in milliseconds) will be set by the developer. Each frame the total CPU time spent on collection activity will not exceed this budget, with the exception of the case whereby a Nursery Generation collection exceeds the budget, or the Allocator cannot fulfil an allocation request and must call a full collection. In the case whereby a Nursery Generation collection is completed within the time budget, the Older Generation collector will use the remainder of the time budget for that frame.

4.1.8 Large Object Handling

To avoid the cost of repeatedly copying large objects during compaction, a separate strategy will be used for dealing with large objects. There are various options for this strategy, including piecemeal copying of large objects, a separate, non-moving region of the heap, or simply requiring the developer to allocate them as non-managed objects and wrap a smart pointer to this object within a smaller managed object. These will be investigated during implementation.

4.1.9 Additional Features

There are other additional features that will be added to the design if time allows. These include the addition of a GUID to the Managed_Object class which will be stored both in the reference table and in Smart Handles referencing that object when in Debug mode. This will allow Smart Handles to verify that the entry in the reference table that they point to has not been overwritten to point to a new object. Other additional debug features include the logging of the line and call-stack where memory allocations were made to aid in tracking down extraneous allocations.

Additional performance features include the use of a feedback mechanism that uses heuristics such as heap occupancy (the percentage of the pre-allocated block of memory that has been allocated for use by objects) and average allocation rates to predict when the Allocator is in danger of running out of memory. The collector can then dynamically increase it's time budget to try and pre-emptively free more space without the cost of a full atomic collection of the entire heap which would cause a large drop in frame rate, instead spreading the extra cost across multiple frames.

4.2 Project Milestones

The project will be structured into several discreet stages, each of which will contribute in some way to the final system. The system will be written in a modular manner to allow the components at each stage to be used and built upon in later stages. If implementation takes longer than planned then at any milestone the project could be halted and still result in either a useful system in itself or at the least a set of modules that could used by another developer in their own Memory Management system. The planned stages of the project are described below:

4.2.1 gcnew() and gcdelete()

Since we are potentially intending to enable the developer to mix managed objects with unmanaged objects, we will refrain from overriding the Global new operator and instead create a new gcnew() function, copying the convention from other collectors such as the Boehm-Weisser collector and Microsoft's method for

allocating managed objects in C++/CLI. These will be updated during later milestones to cooperate correctly with new components in the system.

4.2.2 Smart Handle

This will ultimately handle the functionality described in the system design above. However it will initially only implement indirection via an object table. Other features will be added as required while implementing later milestones. If there is time left over in the period allotted to this milestone the debug features of Smart Handles described will be implemented at this stage. In it's initial form it can be used to reference objects of any type so that it might be used in a non-garbage collected system.

4.2.3 Managed Base Class

This will be the initial implementation of the Managed_Object class as described above. It will also be updated as needed to cooperate with new components developed in later milestones.

4.2.4 Custom Allocator and Compactor

These will implement the basic functionality required for automatic defragmentation of a non-garbage collected heap (described in their respective design sections above). At this stage the compactor will not be incremental. These subsystems, alongside the previous milestones, will form a viable Manual Memory Management system that provides automatic heap defragmentation.

4.2.5 Incremental Compactor

The compactor will be updated to run incrementally, unless explicitly called to run a full compaction atomically. Note that the functionality to run a full compaction atomically will be exposed to the developer to allow them to call a compaction at a convenient time, e.g. just after a level has loaded, to maximize performance.

4.2.6 Registration of Smart Handles

Smart Handles and possibly the Managed Base class will be updated to allow all Smart Handles belonging to an object to be traced by any future Garbage Collector. Options for different ways of implementing this (e.g. having all Smart Pointers for a given object form a linked list, or by associating a list of all of the indices held in those smart pointers with the base class or with the object's entry in the object table) will be explored during this phase of implementation.

4.2.7 Root Smart Handles

These will implement the functionality described above at the end of the Smart Handle design section.

4.2.8 Simple Mark Compact Collector

This milestone will involve implementing the Tri-Colour tracer (as described above), however at this stage it will not be incremental. It will also involve the creation of a management class that will coordinate the various Garbage Collection sub-systems. At this stage tracing of the object graph will run atomically, after which incremental compaction will take place using components build during earlier milestones.

4.2.9 Incremental Tracing

The tracer will be updated to be incremental. This will involve implementing the write-barrier within the Smart Handle class as described in the design section, and exposing functionality to the developer to set and update the time budget for the collector.

4.2.10 Nursery Generation

This will require the creation of a new sub-system to separately manage newly created objects, run a mark-compact over them (using the non-incremental versions of the tracer and compactor developed above) and promote the surviving objects by moving the pointer marking the boundary between generations, as described above. Some updates will be required to the write-barrier, tracer, and the management class.

4.2.11 Large Object Handling

As described in the design section, options for implementing this functionality will be explored during implementation.

4.2.12 Optional Features

As described above, there are several optional features that arise from the desirable criteria of the system. These will be implemented if time allows at the end of the project.

4.3 Development methodology and Testing

4.3.1 Unit Testing

As discussed previously, the system will be written in C++ due to this being the predominant development language of the Games industry. The components of the system will be developed using Test Driven Development. This will be facilitated by the Visual Assert plug-in for Visual Studio for the purpose of generating and running Unit tests, and by the Visual Assist X plug-in to aid with re-factoring. During development a test suite of Unit tests will be built up to test functionality.

4.3.2 Production Testing

The ideal testing method would be to use the system in a commercial quality Game. However such a Game would likely take longer to produce than the total amount of time allocated to the project. Instead, if time allows, a small, simple Game will be produced to demonstrate how the system would be used in practice. Again, if time allows, performance metrics will be used to compare between the memory system developed and Manual Memory Management to give an indication of the performance penalty that comes from using an Automatic Memory Management system.

4.3.3 Version Control

As the various components of the system will be enhanced or updated for each milestone of development Version Control is particularly important to ensure that if the project is required to fall-back to a previous milestone at any stage it can easily do so. For this project the source control software Kiln will be used (which is derived from the Mercurial source control software). This has been chosen due to it's integration with the Fogbugz issue tracking and project management software, as well as it's integration with Visual Studio.

4.4 Project Timeline

An initial estimated timeline of the project implementation is included below. Each milestone has been assigned an expected implementation time based on the expected complexity of the solution. Extra time has been allowed for the completion of milestones where the exact design of one or more of the components is still to be explored. Time has been factored in at the end of the project for writing up the project report (it is anticipated that much of the report will be written up at the completion of each milestone). Finally, downtime has been included in the timeline for examinations, including revision time.

Date	Milestone
16/04	Smart Handles
23/04	Managed Base Class
26/04	Allocator and Compactor
10/05	Exams/Revision
03/06	Incremental Compactor
11/06	Registering Smart Handles
25/06	Root Set Smart Handles
02/07	Simple Mark Compact
16/07	Incremental Tracing
23/07	Nursery Generation
06/08	Large object handling
27/08	Complete report write up

References

[1] McShaffry et. al., 2009. *Game Coding Complete*. 3rd Edition, Delmar

[2] Gregory, J., 2009. *Game Engine Architecture*. 1st Edition, CRC Press

[3] Kerinighan, B. & Ritchie, D., 1989. *The C Programming Language*. 2nd Edition, Prentice Hall

[4] Jones, R., Hosking, A., Moss, E., 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st Edition, CRC Press

[5] Wilson et. al., 1995. *Dynamic storage allocation: A survey and critical review*. International Workshop on Memory Management 1995

[6] Jones, R., Lins, R., 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 1st Edition, John Wiley & Sons

[7] Deutsch, P. & Bobrow, D., 1976. *An efficient incremental automatic garbage collector*. Communications of the ACM

[8] McBeth, J., 1963. *On the reference counter method*. Communications of the ACM

[9] McCarthy, J., 1960. *Recursive functions of symbolic expressions and their computation by machine*. Communications of the ACM

[10] Saunders, R., 1974. *The LISP system for the Q-32 computer*. The Programming Language LISP: Its Operation and Applications Information International, Inc. 32

[11] Blackburn, S., Cheng, P., McKinley, K., 2004. *Myths and realities: The performance impact of Garbage Collection*. International Conference on Measurement and Modeling of Computer Systems. ACM Press.

[12] Wilson, P., 1994. *Uniprocessor Garbage Collection techniques*. Technical report, University of Texas

[13] Dijkstra, E. et al., 1976. *On-the-fly Garbage Collection: An exercise in cooperation*. Language Hierarchies and Interfaces: International Summer School Springer-Verlag

[14] Foderaro, J. & Fateman, R., 1981. *Characterization of VAX Macsyma*. ACM Symposium on Symbolic and Algebraic Computation ACM Press

[15] Ungar, D., 1984. *Generation scavenging: A non-disruptive high performance storage reclamation algorithm*. ACM SIGPLAN Notices

[16] Zorn, B., 1993 *The measured cost of conservative Garbage Collection*. Software Practice and Experience