HyMMS: A Hybrid Memory Management System for Game Development

Travis Woodward

September 16, 2012

MSc Computer Science project report

Department of Computer Science and Information Systems,
Birkbeck College, University of London

Contents

1 Abstract

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

The goal of this project is to design, and implement, a Memory Management system to meet these demands, whilst also exploring the possibility of adapting modern Automatic Memory Management techniques for use in a Game Development environment.

Supervisor: Dr Szabolcs Mikulas

2 Introduction

C++ has long been the de-facto standard development language for the Games industry[2]. It is chosen primarily for its performance characteristics: its speed of execution, its small memory overhead, and the ability it provides to manipulate low level resources in order to squeeze ever greater performance from the platforms being targeted by the developers[3]. As a result, the vast majority of 3rd party software libraries designed for Game development are written in C++[3].

For smaller teams of developers, these software libraries offer the opportunity for them to reuse the work of others in order to make best use of the limited human resources at their disposal. Functionality that could take a small team weeks or even months to develop can be available to them simply by linking to a library and including the appropriate header files in their source code. For a small team developing any non-trivial game, 3rd party software libraries are indispensable[4].

For this reason, C++ is an attractive choice as a development language for smaller teams. However such a choice does not come without a cost. C++, for all of it's performance advantages, can be a difficult and cumbersome language to write software in. One area in particular where C++ is at a disadvantage is in memory management[3].

In C++ memory is managed 'manually'. This means that the developer is in full control of when memory is allocated for some object within the program, and when that memory is released so that it can be reused[6]. This manual memory management can easily lead to errors[5], the details of which were discussed in section 2.2 of the project proposal.

These errors can often be difficult and time consuming to debug[5]. For small teams this detracts from other work that the programmers need to do, such as adding features to the game itself, and so will either lower the quality of the finished product or else will lengthen development time which increases costs. Further evidence that this is a common problem can be found in the availability of several products on the market designed solely to help developers track down memory bugs in C++[1].

There are of course alternatives for smaller development teams. Increasingly smaller teams are turning to other, more modern languages such as Java or the .NET family of languages to use in the development of their games[7]. There are fewer 3rd party libraries available for these platforms[8], and it may require compromises in the design of the game as a result, but for these teams the increase in productivity that comes from the language features of a modern language is worth this trade-off.

One of the major attractions of a modern language over C++ for these teams is automatic memory management in the form of Garbage Collection[1]. As set out in section 2.4 of the project proposal, Garbage Collection addresses many of the issues caused by manual memory management. However these Garbage Collectors have their own drawbacks in the context of game development, also described in section 2.4 of the project proposal. In particular, these collectors often struggle to provide the consistent frame rate that is vital for good performance in game applications[3].

To deal with these twin issues, we have designed a Garbage Collection algorithm that is specifically designed to produce a consistent frame rate, and then implemented this algorithm in C++. This allows users of the system access to the benefits of automatic memory management without inconsistent framerates, and to simultaneously retain access to the wealth of 3rd party software libraries written in C++.

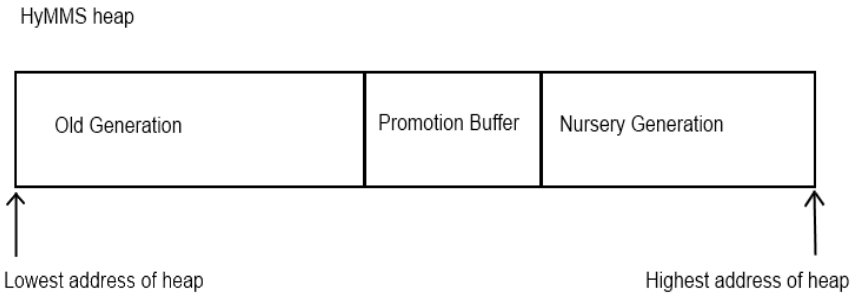
The formal requirements for this system were set out in Section 3 of the Project proposal. The next section will describe the core algorithm that the system uses in detail. This will be followed by a discussion of the design principles that guided the development of the system, and a description of the system's public interface. There will then be a description of the components of the system and how they fit together.

Next we will describe one of the most challenging areas of the implementation. We will then discuss the testing and evaluation of the system, including key performance metrics, a study of the limitations of the system, and what further improvements could be made were development to continue.

3 The Algorithm

The following is a description of the Garbage Collection algorithm we will use in the system. This algorithm has been specifically devised for this project, and is a hybrid of 3 separate families of Garbage Collector algorithm: the Mark Compact collector, the Incremental Collector, and the Generational Collector. A description of each of these families of algorithms appeared in section 2.4 of the project proposal.

The algorithm effectively splits the heap into 3 contiguous regions, which we shall call the Nursery Generation, the Promotion Buffer, and the Old Generation. At any given point during the execution of the program, the heap will be arranged like so:

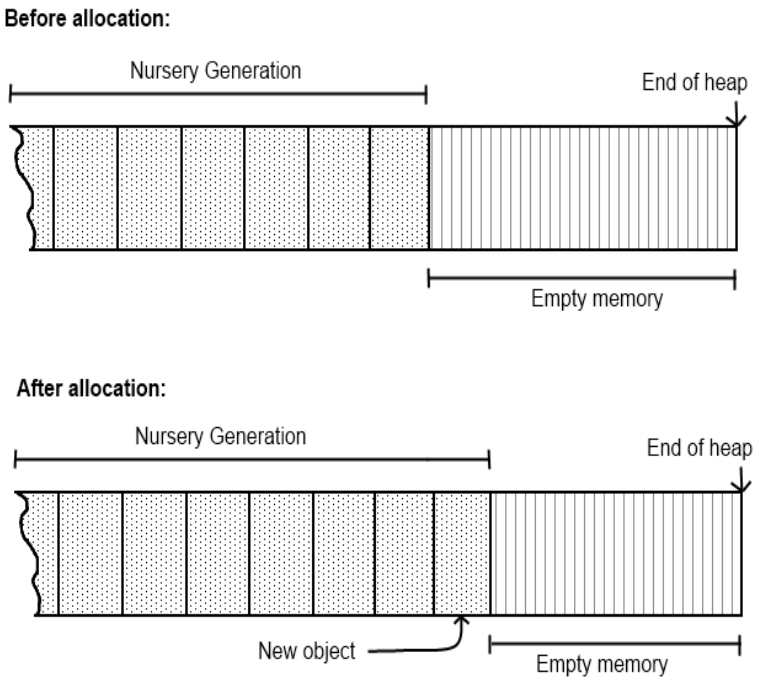


The algorithm dictates how objects are created on the heap, how they are moved between these 3 regions, and how and when they are destroyed. Throughout the algorithm objects are coloured either White, Black, Grey, or Green. This is an extension of Dijkstra's Tri-Colour abstraction[9], which was described in section 2.4.3 of the project proposal.

These colours determine how the algorithm will treat each object at each stage of the collection cycle. By changing the colour of objects the algorithm is able to keep track of the state of each object at any given moment and handle it appropriately.

3.1 Object creation

Objects are created in the Nursery Generation. They are created in the block of memory immediately following the most recently allocated object. In this way the Nursery Region effectively 'grows' as more objects are allocated (this known as sequential allocation[5]). New objects are created with a Green colouring.



3.2 Nursery Collection

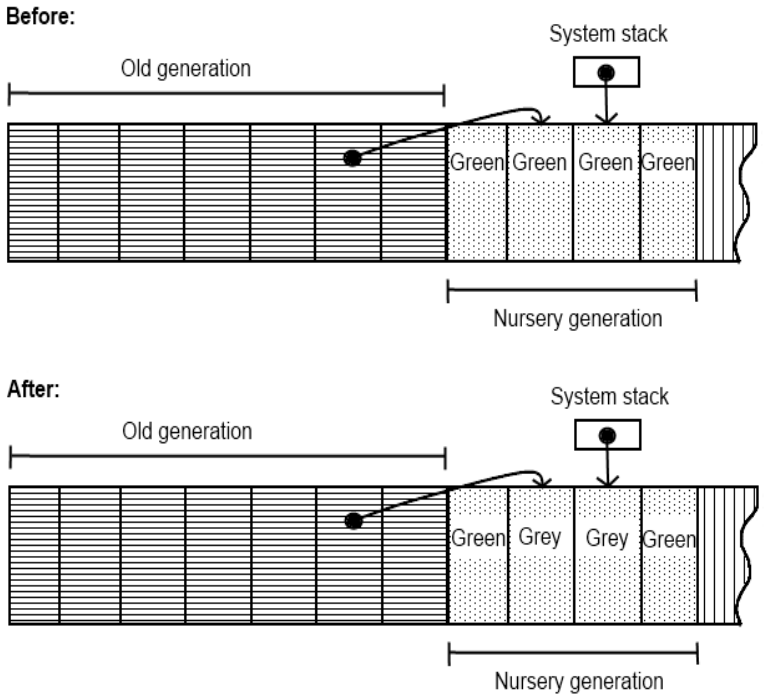
Periodically (every n frames, where n is defined by the user of the system) a Nursery Collection is performed. A Nursery Collection uses a simple Mark-Compact algorithm (described in section 2.4 of the project proposal) on objects in the Nursery generation. Therefore a Nursery Collection consists of the following steps:

- Tracing Nursery Roots
- Tracing Nursery Objects
- Nursery Compaction

Each of these steps is described below.

3.2.1 Tracing Nursery Roots

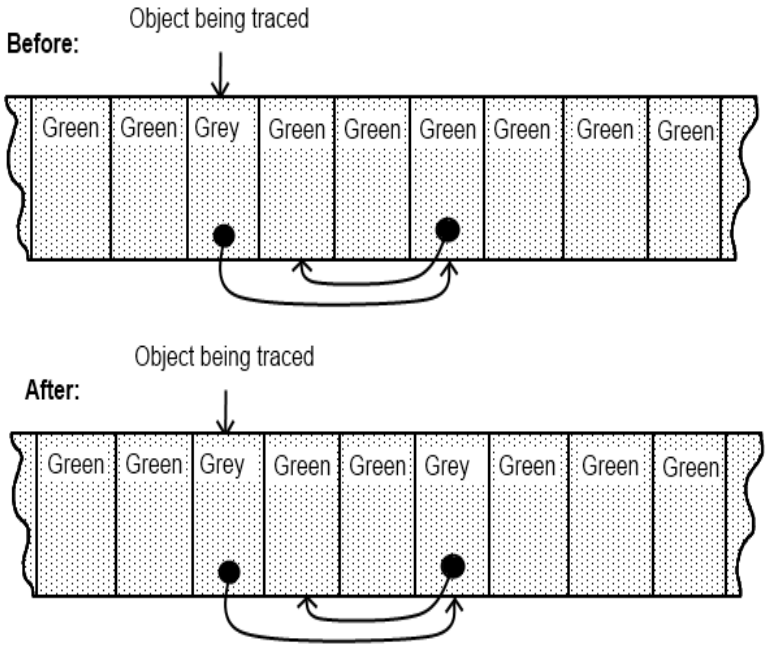
The 'Nursery Roots' are all of the references that point to an object in the Nursery generation that are not themselves member variables of an object in the Nursery generation (also known as 'inter-generational pointers'[5]). The diagram below illustrates:



Tracing the Nursery roots involves examining each Nursery root in turn and painting the object it points to Grey. Once this is done every object in the Nursery that is referenced from outside of the Nursery will be coloured Grey.

3.2.2 Tracing Nursery Objects

The next stage is to scan back and forth across the Nursery generation looking for Grey objects. When a Grey object is found we examine each reference that the Grey object contains, and paint the object that it points to (wherever in the heap that may be) as Grey.



When we have finished painting these objects Grey we paint the object we were examining as Black. We then continue to scan the Nursery generation for more Grey objects.

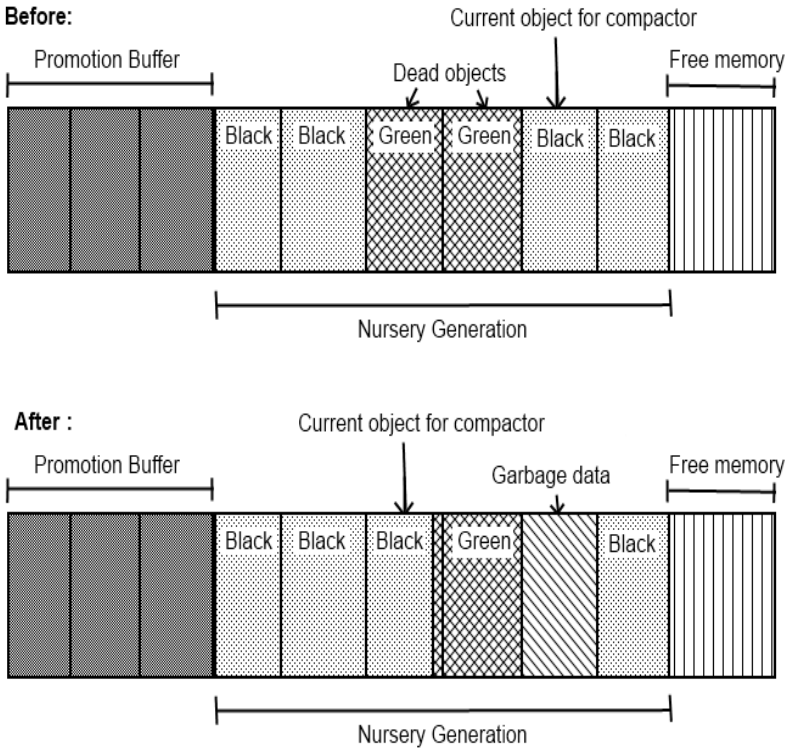
When we have made a full scan (in either direction) across the Nursery generation and found no Grey objects we can then move on to the next stage of the algorithm.

3.2.3 Nursery Compaction

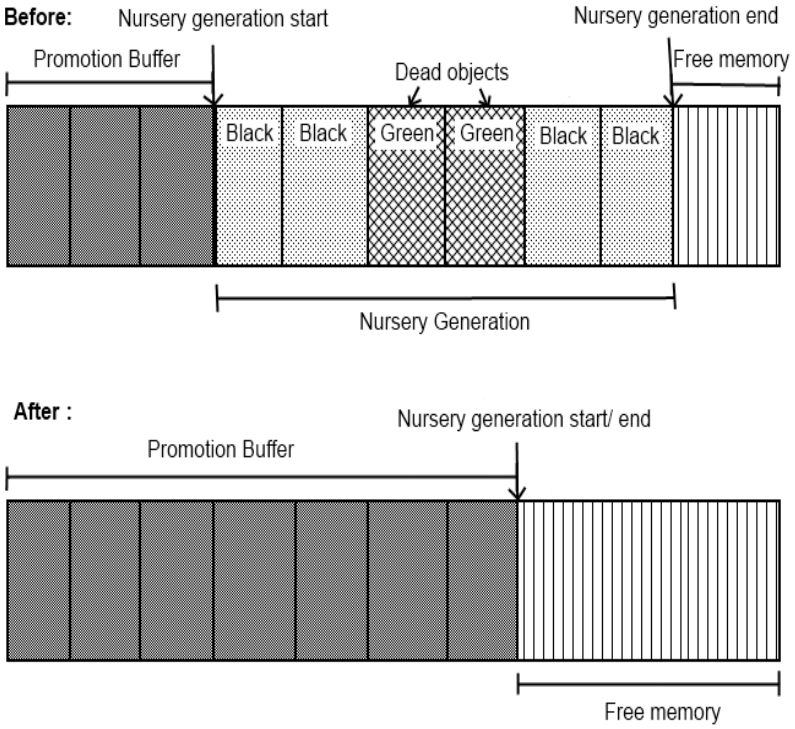
At this point all objects in the Nursery generation are either coloured Green or Black (since if any Grey objects remained the previous step would not have completed). The objects are then examined one by one in order (starting with the oldest which is at the lowest address due to the way objects are created - see section 3.1).

For each object, if it is found to be Green then this object is unreachable by the user application, or 'dead' (since if it could be accessed there would be some chain of references starting outside of the Nursery and ending with that object. If that were the case then the previous two steps would have resulted in this object being coloured Black). Any finalization of that object is performed (e.g. if that object owned an open file then that file should be closed), and the object is destroyed. The space occupied by that object is added to a running total of the space reclaimed so far during this compaction. This running total is initialized to the size of the interval between the end of the Promotion Buffer and the start of the Nursery generation.

If the object is coloured black then this object is reachable from outside of the Nursery generation, and so should be preserved. However it may be positioned after some other object in the Nursery that has been destroyed, leaving gaps in the heap. Therefore the compactor uses the cumulative total of space reclaimed so far, and subtracts this value from the live object's address to give the new address of the object[3]. The object is then copied to its new address. The end result of this 'sliding' process is that the live objects end up in a contiguous region of memory starting immediately following the end of the Promotion Buffer region. This process is illustrated in the following figure:



Finally, the algorithm changes its definition of the Promotion Buffer to include the recently compacted objects, and both the markers that denote both the start and end of the Nursery generation are set to the address at the end of the updated Promotion Buffer. This has the effect of setting the Nursery Generation to be empty, allowing the space used by dead objects to be overwritten when new objects are added to the Nursery Generation.



All three steps of the Nursery Collection phase of the algorithm are performed as a single atomic operation from the perspective of the user application. Therefore no new objects are created while this is occurring, and no references are created, destroyed, or set to point to a different object.

3.3 Incremental Collection

The remainder of the algorithm focuses on the Old generation and the Promotion Buffer. This part of the algorithm is broken down into a series of steps in much the same way as the Nursery Collection section of the algorithm. However in this case each step is broken down into multiple atomic operations.

Periodically the user application will give processor time to the algorithm, which in turn will run the Incremental Collection part of the algorithm. Each time it is given processor time, the Incremental Collection subroutine will run as many of the atomic operations that make up the current step in the subroutine as it can until it either runs out of time or until some guard condition is met which signals the subroutine should proceed to the next step.

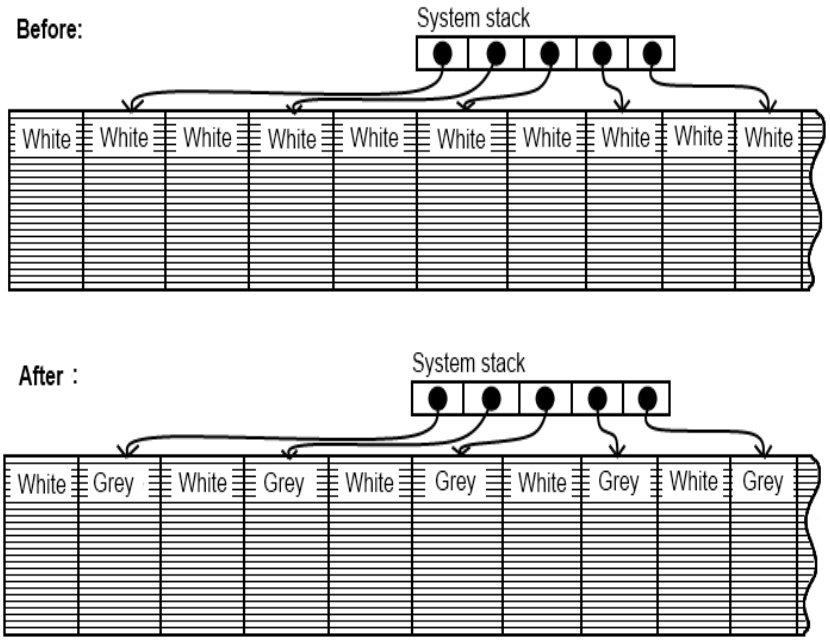
The steps of the subroutine and their guard conditions are listed in the table below:

Atomic step	**Guard condition**
Trace all roots	Once
Trace all non-Nursery objects	Until a full scan of the heap finds no Grey objects
Compact all non-Nursery objects	Until all Old generation & Promotion Buffer objects have been compacted
Nursery Collection	Once

Each of these steps are described below.

3.3.1 Trace all roots

All objects in the Old generation start each cycle of the Incremental Collection phase of the algorithm coloured White. The system iterates over all references that are not members of objects on the heap (such as those created on the stack, that are static, or that are global variables) and paints the objects that they reference Grey. This step occurs atomically and is only performed once before moving on to the next step.



3.3.2 Trace all non-Nursery objects

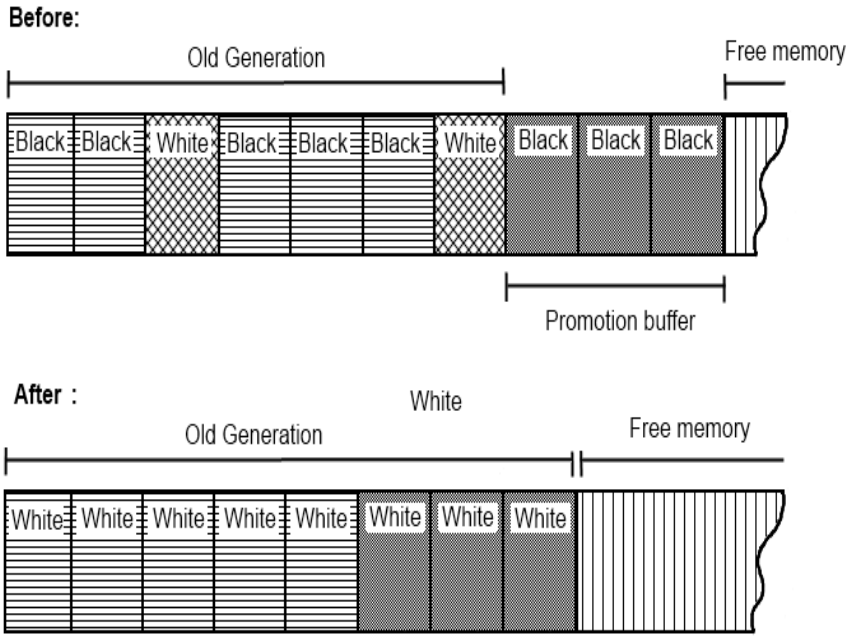
The Old generation is scanned back and forth, looking for Grey objects. When a Grey object is found each reference contained in that object is examined and the objects that they point to are painted Grey. Then the object itself is painted Black. The handling of each object is an atomic operation. When the a full scan of the Old generation is completed without locating a Grey object the guard condition is triggered and the execution passes to the next step.

3.3.3 Compact all non-Nursery objects

Starting at the lowest address of the heap, each object is examined. Any object that is coloured White performs any necessary finalization and is then destroyed, and has it's size added to the cumulative total of space reclaimed in the current compaction.

Any live object encountered has its new address calculated by subtracting the cumulative total from the object's address, and is then copied to that new address. It is then painted White, ready for the next cycle. Each check and delete/ move operation is atomic. Once all objects in the Old generation and the

Promotion Buffer have been compacted, the algorithm redefines the Old generation to be all of the objects that survived the compaction, and redefines the Promotion Buffer to be empty, beginning and ending at the address immediately following the last Old generation object.



3.3.4 Nursery Collection

The Nursery Collection step is simply a single execution of the Nursery Collection section of the algorithm as defined above. This is a single atomic step, and is only performed once.

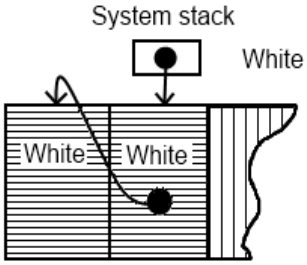
3.4 Additional Elements

The algorithm above is sufficient to ensure that no 'dead' object remains on the heap indefinitely (assuming that sufficient processor time is given to the algorithm). However, it is not sufficient to guarantee that a live object can never be incorrectly destroyed while it is still reachable by the user application. To address this, one additional element is required: the Write Barrier. A description of the Write Barrier and an explanation of its necessity is included below.

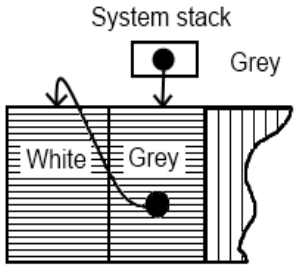
Also, the need to complete a Nursery Collection as the final step of the Incremental Collection requires some explanation, as this too is required to ensure that no live object is not incorrectly destroyed. An explanation of the need for this additional element of the algorithm also follows.

3.4.1 The Write Barrier

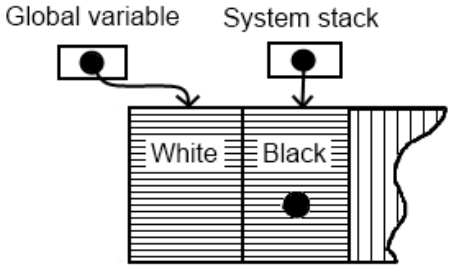
Consider the situation in which the Old generation of the heap consists simply of two objects as shown in the diagram below:



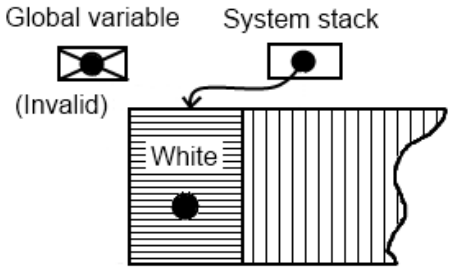
In this scenario, the second object is referenced by a root and the first object is referenced from the second object as illustrated. In the first stage of the Incremental Collection the roots are traced and the second object is painted Grey. Once the roots have been traced the algorithm is out of time, and control passes back to the user application.



The user application copies the reference to object 1 from object 2 and places it in a new root (say a global variable). It then sets the reference in object 2 to NULL. Then control is passed back to the algorithm. The Incremental Collection phase of the algorithm says that the next step is to scan the heap looking for Grey objects, which it does, and finds that object 2 is Grey. It examines the references in object 2. It finds one, but ignores it as it is NULL. It then paints object 2 black.



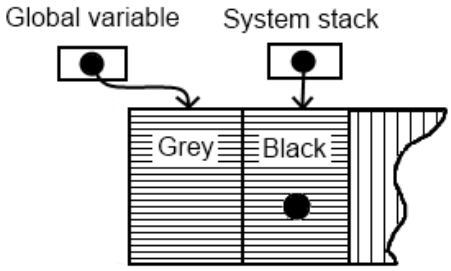
It will then scan back across the heap. Object 2 is Black and object 1 is White, and so it finds no Grey objects. It then moves onto the compaction stage, where it first examines object 1. It finds object 1 is White, and so destroys it. Clearly this is incorrect behaviour, since object 1 is still reachable from the global variable that the user application created.



This is possible because of the asynchronous nature of the Incremental Collection stage of the algorithm, allowing the user application to change the relationships between objects 'behind the back' of the algorithm.

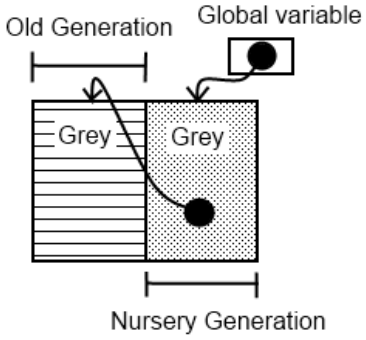
To solve this problem, we add an element known as a Write Barrier[10]. These are used commonly in Incremental Garbage Collectors. There are a number of subtly different Write Barrier algorithms. The one used here is a variation of the one devised by Dijkstra[11].

The Write Barrier is triggered whenever a new reference is created, or when an existing reference is set to point to a different object. When that happens the Write Barrier checks to see if the newly pointed to object is White, and if so paints it Grey. This prevents the scenario above, as when the user application created the new reference to object 1, object 1 would have been marked Grey. It would then have been traced and painted Black by the algorithm, surviving the compaction step.

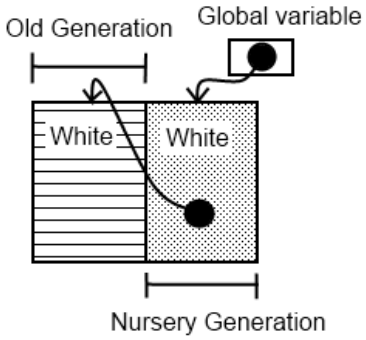


3.4.2 Nursery Collection as a part of Incremental Collection

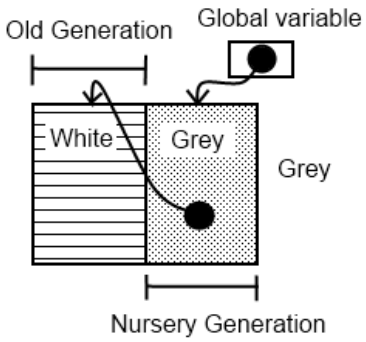
There is one final scenario in which an object might be collected incorrectly under the algorithm as laid out above. First, assume that there are two objects on the heap, one in the Old generation (object 1), and one in the Nursery (object 2), with object 2 containing a reference to object 1, and object 2 being referenced itself by a global variable (a root).



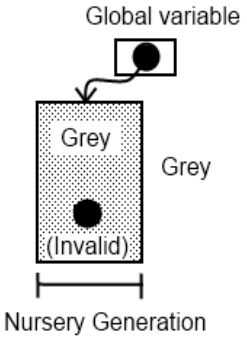
When the reference in object 2 was created, the Write Barrier would have painted object 1 Grey. Assuming that following this the Incremental Collection phase was given enough time to run to completion. As part of this object 2 would have been Compacted and painted White. Assume that we do not have the last step of the Incremental Collection section (i.e. the Nursery Collection).



Then consider the scenario that the Incremental Collection phase is again given enough time to run to completion, and that no Nursery Collection has taken place in the mean time. Since the reference in object 2 has not changed, the Write Barrier will not paint object 1 Grey a second time.



Although object 2 will be painted Grey when the roots are traced, as object 2 is in the Nursery it will not be traced as part of the Incremental Collection phase, meaning that the reference to object 1 will never be examined, and object 1 will not be painted Grey this way either. So when the Compaction step is performed object 1 will still be painted White and will be destroyed incorrectly.



The issue occurs when an object in the Nursery holds a reference to one in the Old Generation, and the Old Generation runs for a complete cycle without a Nursery Collection occurring. To prevent this, we simply add a Nursery Collection as the final step of the Incremental Collection phase of the algorithm, as described in the steps of the Incremental Collection above. In our example above, during the Nursery Collection object 2 will be traced and so object 1 will be marked Grey.

It can hopefully be seen that this algorithm satisfies both the conditions for the correctness of a Garbage Collection algorithm[5], that is both that no live object is destroyed, and that no dead object is allowed to remain in memory indefinitely.

4 Design

4.1 Design Principles

There were two main principles taken into consideration when designing the architecture and interface of the system. The first was to keep the interface as simple and familiar to the user as possible. Any requirement that, for example, required the user to make a special method call immediately following the creation of a new reference would likely lead to bugs in programs that could be at least as difficult to track down as those caused by manual memory management.

The second principle was that the system have good performance. Games by necessity have demanding performance requirements[3]. The game itself is likely to require a significant amount of the processor's time, and so any performance overhead on the part of the system will limit the features that the developers can include in the game. If performance problems consistently arise during development then the programming team may have to spend more time optimizing code or removing features than is saved by using the system in the first place.

There is a conflict between these two principles, as the simplest way to improve performance is to require the user to inform the system when it needs to perform a given operation, and to pass the system the information that it needs to perform that operation. This removes the need for the system to determine itself when it should perform that operation, and the need for it to correctly calculate the information it needs. However this would complicate the interface for the user and increase the likelihood of hard to find bugs occurring when the user inevitably makes a mistake.

Conversely, the less the user has to tell the system, the more the system has to calculate for itself. This could be a duplication of effort if the user application already holds the information for some other purpose, or the programmer can deduce what the information will be at compile time.

In order to balance these priorities, the following principle was adopted when making design decisions on the interface and architecture of the system: Start from the position that the interface should be minimal, and only add to it if is impossible or too expensive for the system to deduce when to perform some operation and the information it needs to perform it. The following subsection describes the public interface that was designed by following this principle.

4.2 Public Interface

The public interface is broken into two sections - functions called on the Garbage Collector itself, and the classes that are used by the user application to facilitate the use of the Garbage Collector. We shall start with the functions called on the Garbage Collector.

4.2.1 Garbage Collector functions

The GC class is the user's interface with the Garbage Collector. The GC class provides the following public static functions:

- CreateHeap(std::size_t heapSize)
- DestroyHeap()
- SetCollectorBudget(double timeBudgetInMS)
- SetLargeObjThreshold(std::size_t threshold)
- IncCollect()
- CollectAll()

Taking each of these in turn:

CreateHeap

This function takes the size the Garbage Collected heap the user wants to create as an argument, requests a block of memory from the operating system of that size, and initializes the various subsystems of the Garbage Collector.

DestroyHeap

This function handles clean-up of the various subsystems, performs any finalization of any objects that remain on the heap, and then frees the memory block used for the heap.

SetCollectorBudget

This function allows the user application to set, and to change, the amount of time the Garbage Collector runs for each time it is called before returning control of execution back to the user application. Whilst this could have been part of the CreateHeap function, it is quite possible that at different points in the user application it will be able to give up more processing time to Garbage Collection others, and so this function gives them the flexibility to change the collector's behaviour on the fly.

SetLargeObjThreshold

The implementation allows for particularly large objects to be stored off of the heap with a memory managed proxy serving as the interface between the system and the actual object. This is because objects on the heap are likely to be moved in memory by the compactor multiple times during their lifetime. For large objects this can be expensive, and so instead they are allocated to a single fixed location in memory and a smaller proxy allows the large object to still be memory managed.

This function allows the user to set the threshold for large objects on the fly. This is particularly useful, as any large subsystems created at the start of the game that persist throughout the game are unlikely to be moved much by the compactor at all, since they will only be preceded by other such subsystems[3]. In this case it makes more sense to keep these large objects on the heap in order to minimize the overhead from the extra level of indirection in accessing them.

However, temporary but large objects (such as 3D models that may only be needed for a single level of the game) would be expensive to keep on the heap, so instead we would want these to be treated as large objects. Hence the desired size at which an object is considered 'large' changes throughout execution, and so the need for this function.

IncCollect

This function runs as many atomic steps of the algorithm as it can before it has exhausted the allotted time budget, and then returns. This would normally be performed once per game loop (or frame) to evenly distribute Garbage Collection throughout the game's execution. However if there is a particularly demanding portion of the game which requires more processing power but allocates few or no new objects, it would be possible for developers to 'pause' Garbage Collection in these sections by not calling this function in that period of the game.

CollectAll

There will be times, for example immediately after a level has loaded, when it is convenient to run a full Garbage Collection of the whole heap in a single atomic step to clean out all of the no longer needed objects and compact together the newly created ones[3]. At these points the game developer does not care so much about how long the collection takes, since they are probably showing the player of the game a loading screen which is not expected to display smooth animations[1]. This function performs that operation.

4.2.2 Other interface elements

The other interface elements broadly fall into 4 categories:

- Base Class
- Container classes
- Handle classes
- Creation macros

A description of each of these interface elements follows:

Base Class

This consists of a single base class called Managed_Object. Any object that the user wants the system to manage the memory of must have this as a base class. This class holds all necessary state information related to the memory owned by the object and performs certain operations at the request of the system as part of the Garbage Collection algorithm. Apart from inheriting from this class the user application will have no direct interactions with it.

Container classes

Normal arrays and the standard library container classes cannot be managed directly by the Garbage Collector. This is due to the fact that they manage the memory of their elements themselves[6]. To help users get around this limitation, two container classes are provided as part of the system: Managed_Array and Unmanaged_Array. The former allows the user to create an array of Managed_Objects. The latter allows the user to create an array of normal objects that do not inherit from Managed_Object on the heap.

The system also provides a container class Handle_Array which is an array of SmartHandles (see below). However this is in fact a simple derivation of the Unmanaged_Array class.

Handle classes

This family of classes are used to replace pointers to objects that have their memory managed by the system. They are used much in the same was as Smart Pointers are in the C++11 standard[12] and Boost library[13]. The primary class in this family is the SmartHandle class. A SmartHandle variable 'handle' that references an object of type T (where T inherits from Managed_Object) is declared as SmartHandle< T> handle, as SmartHandle is a templated class.

A variable of type SmartHandle<T> can also reference an instance of Managed_Array<T>, with the subscript operator used to access the elements of the Managed_Array in the same way that the subscript operator can be used on a normal pointer to a normal array, e.g. if SmartHandle<T> handle references a Managed_Array<T> array, then handle[0] will return the first element of array.

In addition to SmartHandle, there is also an UnmanagedArrHandle class that can be used to reference and access the elements of an Unmanaged_Array in the same way as a SmartHandle is used to access the elements of a Managed_Array. Finally there is a HndlArrHandle which is the handle counterpart to a Handle_Array. HndlArrHandle is actually a derivative of UnmanagedArrHandle in the same way that Handle_Array is a derivate of Unmanaged_Array.

Creation macros

As part of the design of the public interface, it was the intention to try and minimize the requirements that are imposed on a user's code simply because they are using this system. One area where this is particularly true is around the constructor and creating new objects. Since part of the purpose of the system is to allow users to mix memory managed objects with non-managed objects in the same application (in order to use 3rd party software libraries) we could not simply override the global operator new() function and assume that every object is derived from Managed_Object.

Instead we chose to develop an alternative operator to operator new to signal to the user that they are creating an object that has its memory management handled by this system, which we have called 'gcnew'. The ideal language feature to provide such an operator is a new feature of the C++11 standard called variadic templates. These allow a templated function to be written that takes an unknown number of arguments. These arguments can then be 'forwarded' to an arbitrary function[12]. The syntax for using our gcnew operator if it were implemented using variadic templates would have been:

SmartHandle<T> handle = gcnew<T>(...constructor arguments...);

This syntax is familiar and unambiguous to the user, albeit slightly different to the normal C++ syntax.

Unfortunately Microsoft's C++ compiler does not currently support variadic templates[14]. Since Microsoft's game platforms (the Windows operating system and the Xbox 360) are among the most popular targets for game developers[8], for our system to rely upon a feature which the Microsoft compiler does not support would be an unacceptable design compromise. Therefore an alternative strategy was used.

In order to stay as close as possible to the familiarity of C++ syntax with our gcnew operator, this was implemented as a macro. The syntax for use of the macro version of gcnew is as follows:

SmartHandle<T> handle = gcnew(T)(...constructor arguments...);

The main draw back with this approach is that under normal circumstances a type inside parentheses denotes a typecast. This is a source of potential confusion for users of the system and those reading source code that uses this system who are unfamiliar with it. Also, in general over-reliance on macros is generally frowned upon in C++ development[6].

Nevertheless, the decision was taken to use macros to implement gcnew, as the only alternatives were to not support the Microsoft compiler, or else insist that all objects managed by the system are only constructed using a default constructor (thus bypassing the problem of how to 'forward' the arguments to the constructor). Neither of these was considered to be an acceptable design trade-off, and so the macro implementation of gcnew was used.

An additional creation macro gcnew_Pinned is also made available as part of the system. This has the effect of forcing the system to create the new object on the normal heap rather than the heap managed by our system, and using a proxy on the managed heap to ensure that it is correctly handled by the garbage collector. This is the same strategy as that we have employed for large objects.

The reason for supporting fixed or 'pinned' objects is that a 3rd party may need the address of a specific object in memory in order to call back into the user's game code. In our system, objects (and therefore their members) are likely to change their address frequently when they are moved by the compactor[5]. This would causes memory access errors when the 3rd party library tried to access the object it had a pointer to, as that object may have moved since the pointer was acquired.

By allowing the user to 'pin' an object to a specific location in memory we effectively bypass this issue.

4.3 Component architecture

In the previous subsection we described the public interface to the system, i.e. the elements of the system that users of the system will see. In this section we will discuss the counterpart to that: the internal components of the system that interact in order to provide the functionality that the user has access to via the public interface.

4.3.1 The Components

The major components of the system are listed here:

- Allocator
- Tracer
- Compactor
- GC
- GCTimer

Below we describe the primary responsibilities of each of these components.

Allocator

The Allocator's primary responsibilities are:

- Ownership of the actual memory that makes up the heap, including requesting it from the operating system and keeping track of where to allocate new objects
- Assignment of blocks of memory to new objects, including requests for off-heap memory in the case of large or pinned objects.

Tracer

The Tracer's primary responsibilities are:

- Tracing the Nursery roots during a Nursery collection
- Examining objects and tracing their member references (in both the Nursery and the Old generation)
- Painting objects Black that have had their member references traced
- Tracing the roots of the heap as a whole

Here by 'tracing' a reference we of course mean checking whether the object it points to is painted Green/ White, and if so painting it Grey.

Compactor

The Compactor's primary responsibilities are:

- Performing finalization on dead objects
- Destroying dead objects
- Calculating the new address of live objects
- Moving live objects to their new address
- In the case of Old generation live objects, painting them White after they have been moved

Most of these responsibilities are performed identically for all objects in the heap, no matter what section of the heap they are in.

GC

Although the GC class is a major part of the public interface of the system it is also the most significant component in the internal workings of the system. The GC class acts as the coordinator between the other components of the system, insulating them from one another (to the extent that is possible when the components must all make the same assumptions about the layout of the heap that they operate on). The GC class fills a broader coordination role rather than the more focused roles of the other components.

GCTimer

The final component described here is a timer. This actually more of a 'helper' component that a critical part of the system. It simply allows the other system components to get a high precision 'time stamp' of the current time, in order for them to determine when they have exceeded their time budget. This is made public in case users of the system wish to use this as the high precision timer for the rest of their game engine.

5 Implementation

The implementation phase of the project was conducted under a process of rapid iteration, with each iteration culminating in one of the milestones set out in section 4.2 of the project proposal. A full implementation log, detailing the implementation challenges of each milestone of the project, is included at appendix A. However we will here discuss the biggest challenge of the implementation and how it was overcome.

5.1 Registering SmartHandles

5.1.1 The problem

During the tracing phase of the algorithm, the Tracer component is presented with objects and must accurately determine where, within the storage allocated to that object, its member references lie and to which other objects each of those references point.

With full user cooperation this is a trivial matter to implement. In this case the system would simply require that for each object the user must go through all of its references and call a Register() function which adds them to some list within the owning object. Then the Tracer could simply use this list to find and trace each of the object's references.

However, to rely on the user application to do this would be an extremely poor design decision. The errors caused by a user forgetting to register a reference could be subtle. If at any point that reference is the only reference left in existence to some object, that object will be incorrectly destroyed. However, if this never occurs then the application will never give an error.

Worse, whether or not this situation occurs might depend on details of the Player's interactions with the game. In this scenario the error could be at least as difficult to track down as a conventional memory management bug, if not more so.

Therefore the system needs to find some way of either determining just by examining the object in memory where its references lie, or else finding some other way of registering references with the objects they are a member of with little or no cooperation from the user program.

5.1.2 Traditional solutions

By far the biggest obstacle to implementing a Garbage Collector in a language such as C++ is finding a way that for the Garbage Collector to efficiently identify and access all of the references in any given object[5].

Some 'conservative' Garbage Collectors for C++ (such as the Boehm-Weisser collector) accomplish this by examining each block of memory in the object that *could* contain a reference (specifically a plain pointer in the Boehm-Weisser collector). It then checks whether the value held in that block *could* in fact be a pointer by checking that it meets certain criteria (e.g. it holds some address which could plausibly be the address of an object)[15].

If the collector determines that a block of memory is a pointer then the block of memory at the address it holds is preserved and not collected. However it is quite possible that in fact that the block of memory did not hold a pointer at all, and instead held some other value which happened to be the same as some plausible address. Because the collector cannot know with certainty whether the block of memory contains a pointer or not it errs on the side of caution and 'conserves' the destination memory block just in case[10].

Another valid approach this problem is to make use of a C++ Reflection library. Reflection is a technique which allows the program to examine the structure of an object in memory to deduce information about its type and its members[16]. Reflection is supported natively by many modern programming languages, however it is not a language feature of C++, and so to use this we would need to rely on some 3rd party library that provides this functionality[16].

Whilst some reflection libraries can be fast at runtime, all require some co-operation from the user in order to build a database of the structure of all the different types in use within the application[16]. Also, since the Tracing of objects on the heap is already traditionally the slowest part of any Garbage Collection algorithm[10] anything we do to slow down this phase of the algorithm will only worsen the bottlenecking effect it has on the algorithm.

However, if a fast, memory efficient reflection library could be found that requires no or minimal cooperation from the user application, then this would have been given serious consideration if the eventual solution had not been devised.

5.1.3 Address based automatic registration

The implemented solution to this problem is based on the fact that the address of all object members lies inside the bounds of the memory allocated to the object itself[6]. In other words, if the address of object obj is denoted &obj, then the address of m (&m), where m is a member of the object obj obeys the following rule:

$$\&obj \leq \&m < (\&obj) + sizeof(obj)$$

Therefore, given the address of a reference and that of an object on the heap, it is trivial to determine whether that reference is a member of that object, in which case the reference can then register itself with that object. However, first we need to obtain the address of the reference and identify the candidate objects that it could be a member of. We also need to decide *when* the registration will occur.

The last problem is the simplest to solve. In order for the algorithm to operate correctly, there can never be an occasion when the algorithm is allowed to run whilst a reference is unregistered, as this could lead to objects being incorrectly destroyed. Therefore references that are members of objects must be registered before they are set to point at an object.

The logical time to attempt to register the reference is on construction, as this is necessarily before the reference has been set to point at any object. We can also guarantee that, if the reference is a member of some object, then the part of the object relating the the Managed_Object base class will have already been constructed at the time the reference is constructed. This is true since the part of the object relating to the base class is always fully constructed prior to any part relating to the derived class[6].

If we are performing the registration of the reference within the reference's constructor, then obtaining the address of the reference is trivial as the constructor has access to the implicit *this* pointer, giving us the reference's address[6]. It only remains to determine which object the reference is a member of and call the registration function of that object.

One method would be to simply search through every object on the heap. Intuitively this seems like it would be prohibitively expensive. However, this method is not as inefficient as it seems on first consideration.

If we consider when member references will first be created we realise that, provided the reference class (in our case SmartHandle) has a default constructor, this is guaranteed to be called after memory is allocated for the owning object, but before that object's own constructor is called[6].

Because the Allocator allocates memory linearly, we are also guaranteed that the object that the reference is a member of will be one of the closet objects to the end of the heap, if not the closest, as objects will lie in order of age on the heap[5]. Based on this, we can see that linearly scanning the heap from top to bottom ('youngest' to 'oldest') is likely to very quickly find the object that owns the current reference. Therefore this was an acceptable approach to take for registering references.

It should be noted that because of this, the order in which member variables are declared in user application classes may have an affect on performance, and it would be a sensible optimisation to place all reference members at the start of the list of member variables, which ensures that the owning object would be the first object checked.

Up until now we have assumed that the reference being created is in fact a member of some object on the heap. If all references performed the above registration process then those that are allocated elsewhere (e.g. in a global variable or on the stack) would scan through the entire heap before deducing that they are root references.

Instead we perform an initial check to see whether the address of the reference is within the bounds of the heap (and therefore a member of some object). If it is not then we can register it as a root immediately. Otherwise we continue with the member registration process.

6 Testing

6.1 Testing Methodology

Two phases of testing were performed on the system:

- Unit and Integration testing
- Simple demo game

6.1.1 Unit and Integration testing

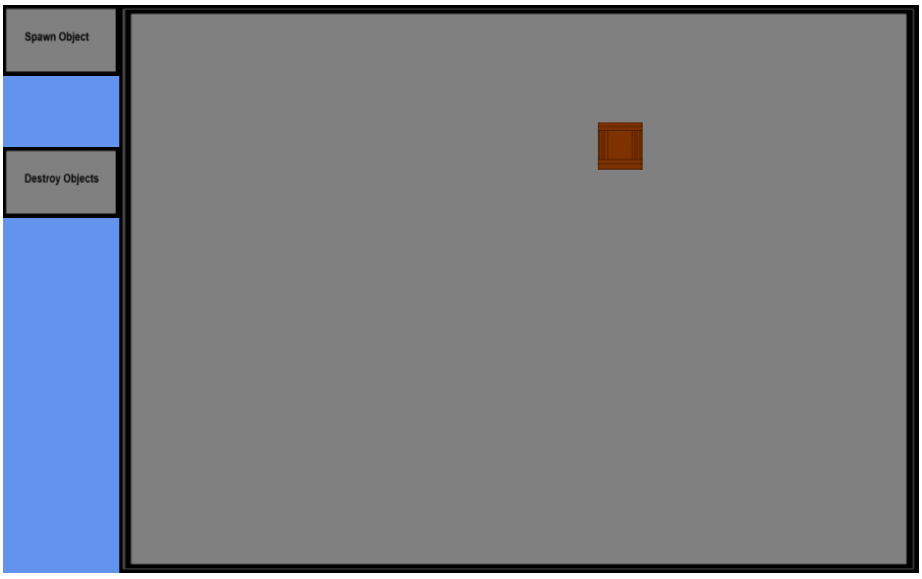
Unit and Integration testing took place throughout development using the built in native unit testing framework in the Visual Studio 11 beta. Each component was tested in isolation, or with as little interaction from other components as possible where those object required a shared heap to operate on.

Integration tests were performed between the various components, testing that they cooperated correctly, culminating in a set of tests that test normal operation and error handling across the whole system.

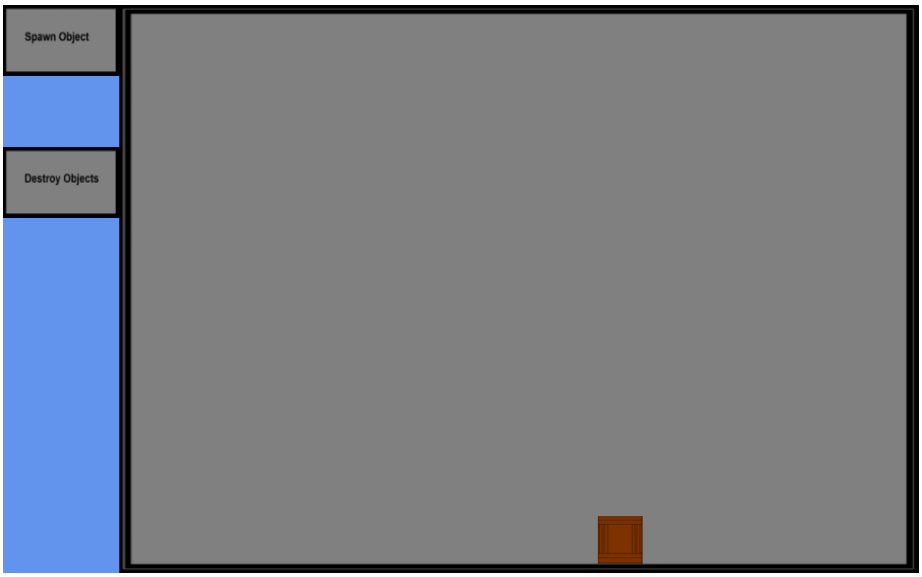
With each iteration of development all previous unit and integration tests were run to ensure no bugs had been introduced to previously tested code (with the exception of redundant tests that tested temporary functionality required only for a previous milestone). The final unit and integration test suite is included with the source code to this project.

6.1.2 Simple demo game

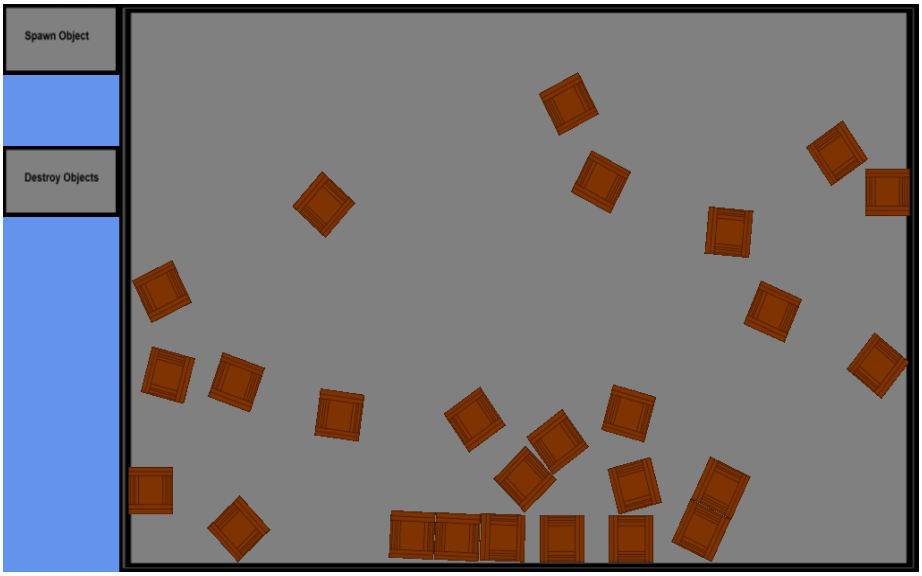
In order to test the system in practice, and to measure performance, a simple demo application was developed. The application is a simple 2D rigid-body dynamics simulation. The demo allows the player to click on a button to create new 'crates' represented by small squares with an image applied to them.



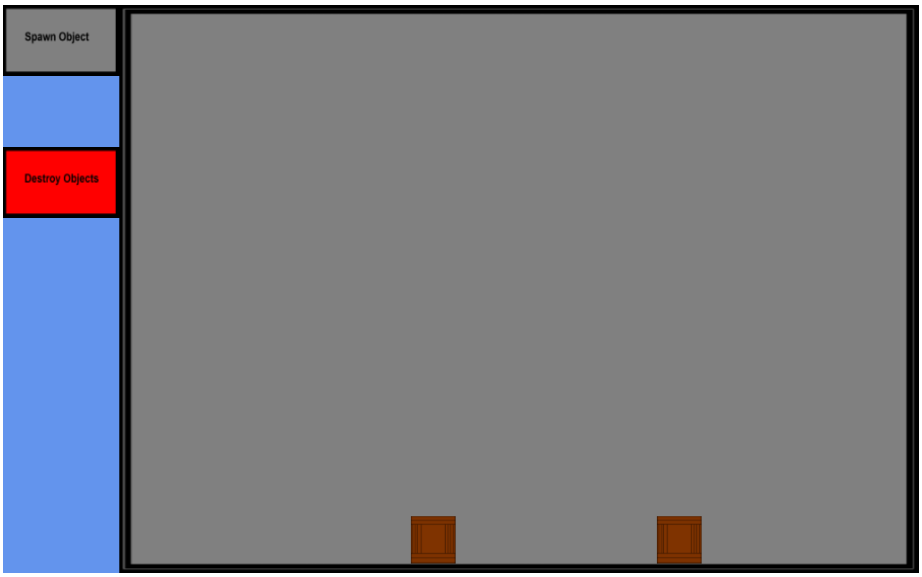
A simulated force of gravity makes the crates fall towards the bottom of the screen.

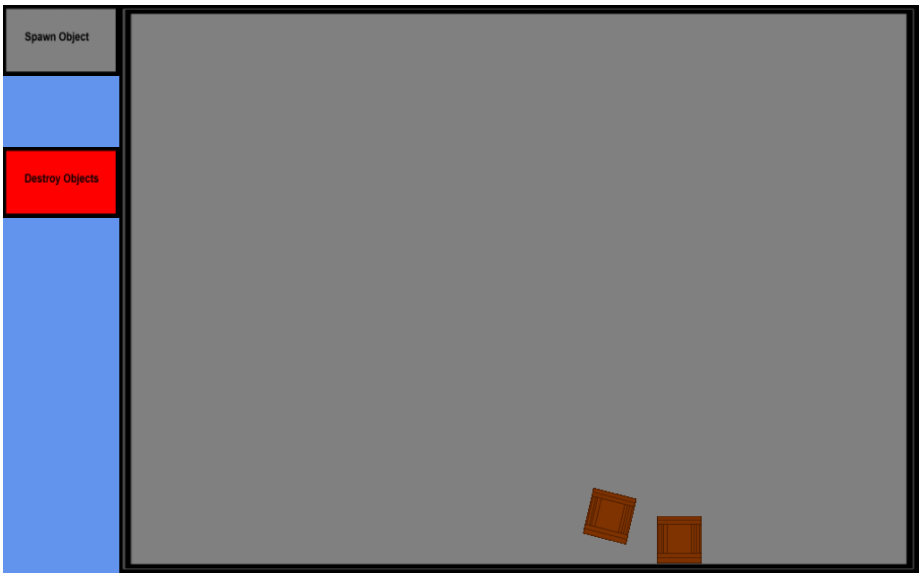


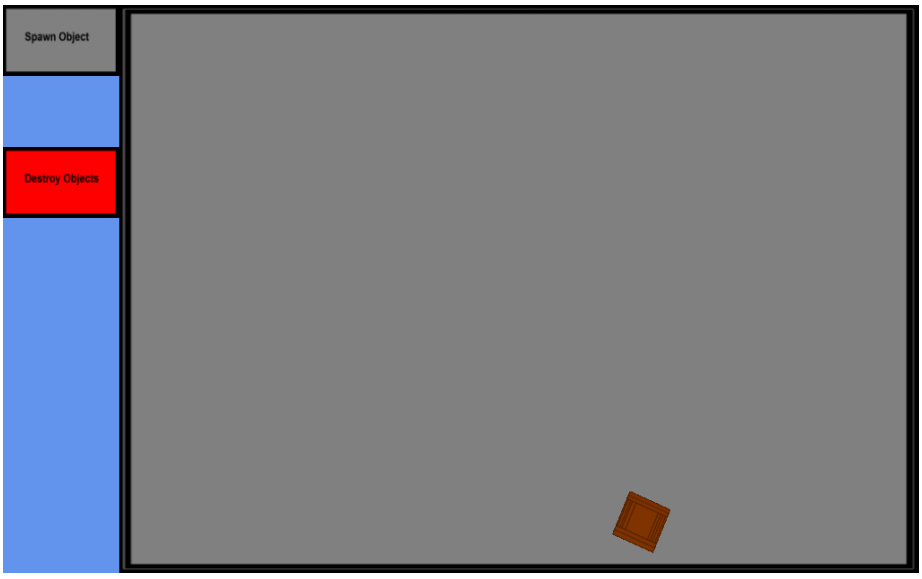
These crates may be moved around the screen by the user clicking on them and dragging across the screen. The crates also react to collisions with one another.



The crates are confined to a rectangular region of the window. The player may create new crates up to a maximum of 50 crates at any one time. A second button allows the player to toggle whether a crate being moved by the player should interact normally with other crates or whether it should instead destroy objects that it touches.




Spawn Object
Destroy Objects


Spawn Object
Destroy Objects

The demo game was implemented 3 times in order to compare and contrast performance:

- In C++, using manual memory management
- In C++, using HyMMS
- In C#, using the built in .NET Garbage Collector

The purpose of implementing the game using manual memory management is to provide a performance base line. We would expect this to be significantly faster than using a Garbage Collector.

The purpose of implementing the game using C# is to give us a comparison to an existing high performance Garbage Collector. The desktop version of the .NET Garbage Collector is significantly faster than the implementation available on the Xbox 360, but unfortunately it is not possible for members of the public to run code written in C++ on the Xbox 360, and so we cannot use that as a comparison. However, the desktop implementation will give us a good benchmark for Garbage Collector performance.

The two C++ implementations of the demo use the 3rd party Box2D library[17] to provide the rigid-body simulation and the C# implementation uses a C# port of the library[18]. In order to keep comparisons fair we include performance metrics that discount the execution time used by the 3rd party libraries.

Full source code for all three demo games is included with this project.

7 Performance Evaluation

7.1 Methodology

There were two primary methods for gathering performance data from the demo games. The first was to insert performance measuring code into the demo application itself and have this record various performance metrics, before writing this information out to a log file at the end of execution.

The second method was to use the game benchmarking application Fraps[19] (which is commonly used by the PC gaming media to measure the performance of games). For each benchmark the details of the 3rd party software used to gather it will be listed.

This accurately measures the time taken to render each frame of the game on the screen, and can be set to run for a set number of seconds from the press of a pre-set key, enabling more consistent measurements to be taken.

All performance metrics were recorded on the development PC which has the following specifications:

- CPU: Intel Core2Quad Q9400 @ 3.0GHz
- RAM: 4GB DDR2, CAS 5
- GPU: Nvidia GTX 275, 896MB VRAM

Performance metrics are likely to vary across different systems.

7.2 List of Performance metrics

The following table details all of the performance metrics gathered from the demo games:

Performance Metric	**Measurement method**
Frametime	Fraps
Frametime (excluding physics)	In demo measurements
Garbage Collection time/ frame	In demo measurements, HyMMS only
Time between full collections	In demo measurements, HyMMS only
Nursery Collection time	In demo measurements, HyMMS only

Here Framerate refers to the amount of time elapsed between one frame being rendered to the screen and the next, Garbage Collection time per frame refers to the amount of time spent on Garbage Collection per frame. Unfortunately a way could not be found to measure this for the .NET Garbage Collector for comparison. Nursery Collection time refers to the amount of time each Nursery Collection takes to complete.

For the demo performance metrics the simulation is started and 25 boxes are added and allowed to come to rest. Performance metric collection then takes place for precisely 30 seconds, during which time the player manipulates, destroys, and creates crates in no particular pattern in an attempt to simulate normal play. At the end of the 30 seconds the performance metric is recorded.

Only one performance metric is recorded during each run, and any performance measurement logic in the demos are disabled when not being used. This is repeated 5 times for each performance metric.

This is obviously not a sufficient sample size or rigorous enough methodology for any statistically valid results. However none of the samples have been optimised in any way, and neither has the HyMMS library. Therefore any attempt at a rigorous comparison study would be meaningless. Nonetheless, the data does serve to provide a ballpark figure for the performance of games developed using HyMMS as opposed to other commonly used memory management systems. A brief summary and high level analysis of the performance results follows.

7.3 Results and Analysis

The full data for each performance metric is included with this project. A summary of the results is described in the table below. Each result listed is the mean time in milliseconds across all runs of that Performance Metric:

Performance Metric	**Result**
Frametime (Unmanaged Demo)	0.3263
Frametime (HyMMS Demo)	0.3094
Frametime (.NET Demo)	0.6855
Frametime - no phys. (Unmanaged Demo)	0.2627
Frametime - no phys. (HyMMS Demo)	0.2448
Frametime - no phys. (.NET Demo)	0.2430
Garbage Collection time per frame	0.0048
Time between full collections	0.6164
Nursery Collection duration	0.0004

The results can roughly be divided between those that provide a comparison of the performance between the different Demo applications and those that describe the performance of the HyMMS Garbage Collection system itself.

7.3.1 Comparative Results

When considered in isolation the normal Frametime figures paint an interesting picture. In particular, they show that both of the Demo implementations written in C++ were taking on average less than half the time that the Demo implemented in C# using the .NET framework to execute each frame.

Although published data on the comparative performance of C++ vs C# using .NET suggests that C++ is significantly faster[20], such data is in the context of small, comparable, optimized programs. In our case, while the demo specific code is very similar between the two languages, there are significant differences in the graphics frameworks used to display the two Demos.

The C++ Demos used a framework written on top of OpenGL which was based on tutorial code from an introductory text on OpenGL[21]. By comparison, the the C# version of the Demo was written on top of Microsoft's XNA framework[24] which is highly optimized and has been used for numerous commercial games on the PC, Xbox 360, and Windows Phone 7 platforms[22][23].

Given these factors, the results are surprising. However, all becomes clear when these results are looked at in combination with the Frametime results that do not include the time taken by the Box2D physics framework each frame.

In this performance metric the C# Demo implementation was slightly faster than the C++ implementations. Given the difference between the two performance metrics for the C# implementation we can deduce that the C# port of the Box2D library is significantly slower than it's C++ counterpart. This means that the raw frametime metric cannot tell us anything about how performance of HyMMS compares to the performance of the .NET Garbage Collector.

We still have difficulty drawing valid conclusions about the relative performance of HyMMS compared to the .NET Garbage Collector from the performance metric that does not include the execution time of the Box2D library. This is due to the underlying weaknesses in the graphics framework of the C++ implementations compared to the Graphics framework used by the C# demo set out above.

We can however see that HyMMS's (unoptimized) speed appears to be at least in the same order of magnitude as that of the mature, optimized .NET Garbage Collector in the context of this Demo.

Also, the large difference in performance between the C++ and C# implementations of Box2D illustrate again part of the reason that a library such as HyMMS would be useful. Box2D is a very commonly used 3rd party library among independent game developers, and yet its .NET implementation is significantly slower than its original C++ counterpart. This is a microcosm of the landscape for 3rd party libraries for game development. In general C++ libraries for use in games tend to be more mature and better optimized[1].

The difficulties that prevent us from making a valid comparison between the .NET implementation of the Demo and the C++ implementations do not exist between the C++ implementation using manual memory management and the implementation using HyMMS.

In both comparative performance metrics the HyMMS implementation is marginally faster than the implementation using manual memory management. However, the margin is so small that this no conclusion can be drawn about which implementation is faster (the difference in both cases is less than 20 microseconds).

Also, the implementation in both cases is not at all optimized. For example, it is common practice when working with either manual or automatic memory management to avoid allocating memory as much as possible, as it is usually expensive. One way that this is accomplished is by pooling objects where appropriate.

The GameObject instances in the Demo (which represent the crates) would be ideal for pooling (kept in memory and reused, rather than continually creating and destroying), but instead all implementations allocated new memory for these objects when they were created and discarded their memory when deleted.

However, these results again illustrate that there is no significant performance penalty in the context of the Demo application from using the HyMMS system. Since the primary goal of the HyMMS system is to bring the benefits of automatic Garbage Collection to game development in C++ without a significant performance impact, these results, as far as they can, indicate that this goal has been achieved.

7.3.2 Garbage Collector results

The remainder of the performance metrics were designed to provide an insight into the performance of the HyMMS Garbage Collector itself. The first of these metrics, Garbage Collection time per frame, shows how much of the frame time on average is spent on Garabage Collection. As described in the design section, the user program can set a time budget for the collector which limits this figure, but in our simple Demo this budget is never hit.

Indeed, when considered together with the frametime performance metric for the HyMMS demo this result indicates that Garbage Collection takes less than 2% of the frame execution time in this simple demo. We can expect that in a more complex application this proportion would increase, but this metric gives us an indication that there is no inherently large performance overhead in the Garbage Collection algorithm devised for HyMMS.

The next metric was intended to give an indication of how long dead (unreachable) objects are likely to persist in memory (since an object may persist for up to two collection cycles if it is orphaned after being marked Grey during the start of a collection cycle). Based on this metric, dead objects in the demo application may persist for a maximum of 1.233ms (2 x 0.6164). Taken together with the frametime metric this indicates that a dead object could persist for a maximum of 4 frames before being deleted in the demo.

Again, this metric is likely to increase in a more complex application since the time taken for a collection will be relative to the number of objects on the heap. However we would expect that any inherent inefficiency in the algorithm would have presented itself in this demo, which does not appear to be the case.

The final performance metric was designed to test what proportion of Garbage Collection time is spent collecting the Nursery Generation. As we can see from the result, the time spent on collecting the Nursery region is an order of magnitude less than the amount of time spent on Garbage Collection in total each

frame. This is likely due to a low allocation rate in our demo application. Whilst a game that allocates more objects per frame would show the benefits of a Nursery generation more explicitly, this metric does indicate that even where the Nursery Collector does very little work there is no real overhead in its inclusion in the algorithm.

However one can easily envisage a game application that frequently allocates objects each frame (e.g. if we had written our own 2D physics library instead of using a 3rd party library then objects such as collision contacts would have been allocated each frame and persisted only for that frame[2]). Therefore to be useful for game development in general it is important that HyMMS includes a Nursery collector to prevent these objects for persisting too long in memory.

There is one obvious omission from the range of performance metrics, given that one of the stated goals of the HyMMS system was to spread the overhead of a the Garbage Collector evenly across multiple frames. An obvious performance metric to have illustrated this would have been the variance in time taken by the Garbage Collector each frame. However there was little point in calculating this metric.

The reason for this is that the primary feature of the system used to 'smooth' the amount of time taken by Garbage Collection each frame is the time budget which caps the amount of time taken by the collector each frame. Since this feature was not engaged by this demo (due to the minute amount of time taken by the collector each frame) this metric would not give a fair indication of HyMMS's capability with regards to distributing Garbage Collection costs across multiple frames.

Given more time a better test would have been to develop a more demanding demo application that would have required the developer to limit the amount of time spent on Garbage Collection each frame.

8 Evaluation

8.1 Requirements revisited

8.1.1 Essential Criteria

We shall examine each of the essential criteria set out in section 3.1 of the project proposal in turn to determine whether or not HyMMS meets those criteria.

Allocation times that do not depend on the size of the object or on heap occupancy

HyMMS uses a sequential allocator that simply allocates new objects at the top of the heap. This is significantly faster than an allocation system that keeps track of free blocks of memory and searches through them to find one large enough to hold the new object[25]. The time take to allocate a new block is not in anyway dependent on the size of the request or the size or number of other objects are on the heap.

There are two exception to this: if the allocation request exceeds the remaining space left on the heap then it triggers a full collection, which would be slow, and if the request is for an object over the large object threshold set by the user then it will be allocated using the standard C++ allocator. However both of these are exceptional circumstances, and are necessary to avoid problems such as a heap overflow or slow compaction times due to copying large objects.

Therefore in normal operation, HyMMS meets this criterion.

No dependencies on the implementation of the compiler

HyMMS is written to be as compiler agnostic as possible. However, when working with low level systems there is always the possibility that some exotic compiler may break convention in a way that is unexpected. In almost all cases HyMMS depends only on the rules of the C++ standard.

There is one exception to this however: the C++ standard is silent on the 'orientation' of data in the heap, i.e. whether fields are laid out starting in ascending or descending memory addresses from their base address[6]. However research did not uncover any known C++ compilers that allocate memory for fields in descending address order, and so HyMMS also relies on this assumption.

It should be noted however that the timing module used to measure the time taken by the collector so far relies on conditional compilation to choose between two different implementations depending on whether the user is targeting a Microsoft platform such as the Windows operating system or the Xbox 360, or whether they are targeting a Unix based system. This is due to Microsoft providing a different, non-standard API for access to high precision timing information[26].

Low heap fragmentation

As HyMMS uses a compacting Garbage Collection algorithm, it necessarily minimizes heap fragmentation[10]. Again the exception to this is the case of large or pinned objects which are allocated off of the heap. These should be exceptional in normal use of the system.

Predictable locality of reference

Because HyMMS allocates object sequentially, if an object requests memory for a child object as part of its construction that object will itself be allocated close to the parent object, with only other objects that are also children of that object between them. This should maximise the locality of reference for accessing member objects, which is the best an automated system can hope to do without intervention from the user of the system[25].

Finalization of objects

As part of the compaction process HyMMS calls the destructors of dead objects to allow finalization to be performed. The destructor is the conventional place for C++ programmers to place finalisation code when using manual memory management, and so using it here for the same purpose helps to minimize the work required to convert an existing game to use HyMMS.

A developer definable upper bound on time spent on Memory Management each frame

This was built into HyMMS explicitly and is provided as part of the public interface.

Therefore we can safely assert that HyMMS meets the Essential Criteria for the project.

8.1.2 Desirable Criteria

Unlike the Essential Criteria, which were relatively conservative in setting out the requirements for a memory management system for game development, the desirable criteria were significantly more challenging. Given the time constraints, not all of these were attempted.

Automatic Memory Management

HyMMS is a Garbage Collected system and so provides Automatic Memory Management. However it does not mandate that all objects use the Garbage Collector, and can be used side by side with manual memory management techniques.

Minimal cache misses

HyMMS provides good locality of reference (see above) which should minimize cache misses due to accessing objects by the game application[25]. It also access objects linearly during the tracing phase of the algorithm rather than using a marking-stack as is common among other Garbage Collection algorithms[5]. This, depending on the hierarchy of objects and the order in which they are arranged in memory, could lead to significantly fewer cache misses.

Similarly, the marking of an object does not require accessing of the actual object itself, only the of the reference table, which is likely to already be in cache memory due to it being used for every object access by the game application and for painting other objects during the tracing phase.

However due to time constraints the demo application was not tested for cache misses and compared with the number of cache misses in the manually managed implementation of the demo. Therefore although the system was designed with minimizing cache misses in mind, it is not possible to determine from the tests performed how successful it is at meeting this criterion.

Memory Alignment

Support for correct memory alignment was added to the system relatively late in the development process, and so had to be retrospectively added to components that had already been implemented such as the Allocator and Compactor. However, on researching memory alignment it became clear that its importance was more significant than was realised at the time of writing the project proposal.

This was due to the fact that some platforms have a very high performance penalty for accessing mis-aligned data, and other game platforms (such as the PlayStation 2) will crash on trying to access mis-aligned data[3]. Therefore memory alignment support was added. In hindsight, support for correct memory alignment should have been an essential rather than desirable criterion.

A separate area of the heap for large objects to avoid expensive copying of large data structures common in Games

This criterion was partially implemented. Whilst large objects are allocated off of the Garbage Collected heap, they are not allocated on a specific separate area of the heap, but instead are allocated using the standard C++ allocator.

This was mostly due to time pressures, and a simple augmentation of the system would be to alter the large object handling code to instead allocate memory from a large pre-allocated block of memory of a size specified by the developer.

Compatibility with third party software libraries

Compatibility with third party libraries is fully supported by being able to use Garbage Collected and manually managed objects side by side. It is also supported through the addition of pinned objects, allowing the third party libraries to hold normal C++ pointers to a Garbage Collected object if need be. There are some limitations to the current implementation of pinned objects however, which are discussed in the Limitations section.

Debug meta-data, a feedback mechanism, and thread safety

These features were not implemented due to time pressure and, on the part of thread safety, complexity. All of these do however remain extremely useful features to include in a system like HyMMS.

8.2 Limitations

As well as lack of some features as described in the subsection above, there are a number of other limitations to the HyMMS system which need to be outlined for potential users of the system. In particular the system lacks support for multiple inheritance. One reason for this is the fact that the order in which members appear in memory in the case of multiple inheritance is not dictated by the C++ standard, and does vary from compiler to compiler[6].

In the case of interface inheritance (i.e. where one of the base classes contains no data members) the system could potentially be adapted for this purpose, but this possibility has not been explored at this time. Whilst the lack of support for arbitrary multiple inheritance is not likely to be a major hindrance to the use of the HyMMS system in game development (due to it being generally discouraged because of the complexities it brings)[1], not supporting arbitrary interfaces could well be an issue.

Interfaces are one of the key elements of polymorphism and are commonly used in many industries, including the games industry[3]. Currently if a developer needs a class to inherit an interface, that class cannot be managed directly by the HyMMS library, and the user would instead need to create their own proxy for objects of that class.

A further limitation is the use of pinned objects. In the current implementation, pinned objects are owned directly by their proxies. This means that if there are no other managed objects that reference the proxy then both the proxy and the pinned object will be collected. However, it could be that some third party object is still referencing the pinned object, and so it would be incorrectly collected.

There is no way to arbitrarily solve this problem as it will depend on the assumptions that the 3rd party library make about objects that it holds pointers to.

In particular it is possible that the 3rd party library will assume that the object will exist as long as it needs it to. In this case the only solution for users of the HyMMS system is to ensure that the proxy is properly referenced for as long as the 3rd party library may need to access it. This is much the same as needing to keep a pointer to that object so that it could be deleted in a manual memory management environment.

It is worth mentioning again that the HyMMS system will not operate correctly in a multi-threaded environment. Whilst non-managed objects could of course use multi-threaded code, it would be impossible without substantial modifications to the algorithm for some Managed_Objects to be manipulated by one thread while other Managed_Objects are manipulated by another.

Similarly the reference table implementation that allows handles to access objects is not currently designed to deal with concurrent accesses from multiple-threads. Therefore using the current version of the HyMMS library in a multi-threaded environment could lead to undefined behaviour.

A small development team (the primary target of the HyMMS library) is less likely to use multi-threading heavily in their games, but it is by no means uncommon for them to do so[1]. If a game is already built around a highly multi-threaded design then it will not be a suitable candidate for conversion to use the HyMMS library for memory management.

8.3 Future work

Potential future work to further develop the HyMMS library should start with further, more detailed analysis of the current implementation to deduce which steps within the algorithm are the most costly. It would also benefit from being used as part of the development of a larger game. This would underline which of the afore mentioned limitations are the biggest obstacles to HyMMS becoming a commercially viable software library. It would also highlight which of the additional unimplemented features are the most likely to be missed by a team using the library.

Further, attempts should be made to target other platforms with HyMMS. Although HyMMS was designed to be compiler and platform agnostic, this project has only tested it fully on the platform of a PC running the Microsoft Windows operating system. However HyMMS could potentially be of use on all platforms that are targeted by developers using C++ to develop games.

9 Conclusion

The purpose of this project was to develop a Memory Management system that is suitable for use in a Game Development environment. The HyMMS library is such a system. Beyond that, it is a proof of concept that shows that it is possible to develop a Garbage Collector that could feasibly be used in a Game Development environment, particularly for smaller development teams with limited programmer time.

Whilst the success of this project should not be overstated, the development of HyMMS has resulted in a system on which a stable, commercial quality library could be built. With further work to overcome its current limitations, HyMMS could help smaller developers spend significantly less time finding and fixing memory management issues, allowing them to focus on their real goal: making the game itself.

Appendices

A Implementation Log

This log details the various challenges and design decisions that were faced during each of the implementation iterations. Each iteration ended in the completion of one of the milestones set out in section 4.1 of the project proposal.

A.1 Smart Handles

A.1.1 Design decisions

At this early stage the SmartHandle class was intended to be a simple abstraction of accessing objects indirectly via a table of pointers. This was key to being able to arbitrarily change the address of an object on the heap without making all references to that object invalid[3].

Design cues were taken from the design of existing smart pointers such as the std::shared_ptr class in the C++11 standard[12]. These tend to be templated classes, taking the type of the object they point to as their single template parameter. e.g. a std::shared_ptr p that points to a variable of type T would be declared as:

std::shared_ptr<T> p;

These smart pointers also override the de-reference operator (*) and the member access operator (->) to allow the programmer to use them in the same way that they would use a normal pointer[12]. Without the template parameter this would not be possible as both operators need to return a reference of the same type as the object being pointed to.

The SmartHandle class followed this same design, so that a SmartHandle h that references an object of type T is declared as:

SmartHandle<T> h;

It also overrides the de-reference and member access operators.

A.1.2 Implementation Challenges

The main implementation challenge was how to best implement the indirect access to objects that the SmartHandle provides. Since a single table of pointers was to be shared among all SmartHandle instances it might at first seem that the logical place to store this table would be as a private static member of the SmartHandle class.

However, as the SmartHandle class is templated, this would have had the affect of creating a separate table for each instantiation of the templated class (i.e. there would be one table per type of object that the programmer uses a SmartHandle to reference).

As well as this, the reference table would also be used in later iterations to store the colour of each object on the heap, and this colour would be manipulated by other components of the system such as the Tracer and the Compactor.

The table could have been made a global variable, however this would have allowed all objects to have direct access to the table, which is undesirable as it opens up the possibility of a whole range of bugs if the table is manipulated directly.

Instead, the GC class was created and the table was made a private static member of that class. The GC class was intended to the fulfil the coordination role across all of the components to enable them to cooperate without relying on each other's implementations. This is an example of the Mediator design pattern[27].

The GC class itself will contain only static members and methods, rather than being a true object in its own right. Whilst a Singleton pattern[27] could have been used, this would require all SmartHandle instances to obtain a pointer to the singleton GC instance in order to access the reference table. This would be a wasteful overhead since there are likely to be a very large number of SmartHandle instances in the user application.

The SmartHandle class was necessarily made a friend of the GC class in order to provide it access to the reference table without allowing arbitrary access by other objects.

A.2 Managed_Object class

A.2.1 Design decisions

User's of the system should only interact with the Managed_Object class by having an object that they want to be handled by the HyMMS library inherit from it. The base class should then hold all state information necessary for the rest of the system to correctly manage the memory of any object that inherits from it.

It should also provide any functionality that has to be performed on an actual instance of an object (e.g. registering references that are members of the object, see section 5.1).

In its initial implementation the Managed_Object class only keeps track of the index of the instance's address in the reference table.

A.2.2 Implementation Challenges

The biggest challenge of implementing the Managed_Object class initially was to decide where the responsibility for registering a new Managed_Object with the reference table should lie. There were two feasible options:

- Inside Managed_Object's constructor
- Inside the constructor or assignment operator of the first SmartHandle instance to reference the new object

Registering from the Managed_Object constructor

Registering the new object inside its constructor automatically would have the undesirable side effect that any object inheriting from Managed_Object created on the stack or in a global variable would be registered with the reference table, but wouldn't actually be on the heap. However, this problem can easily be mitigating by checking that the address of the object lies within the bounds of the heap before proceeding with registration.

This option also has the downside that it leaves open the possibility that a programmer could create a new Managed_Object on the heap but never reference it with a SmartHandle. This would mean that the object would be destroyed in the first Garbage Collection after its creation. However, since one of the assumptions of the system is that programmers use SmartHandles in place of normal pointers, we cannot protect against programmers purposefully violating this rule.

Registering from the constructor of assignment operator of the first SmartHandle instance to reference the new object

This option has the advantage that we can guarantee that on registration the object is referenced by at least one SmartHandle instance. However it has the dangerous side effect that a Managed_Object could be created on the heap and persist without any accompanying information in the reference table. This would mean that other components would have to first check whether an object was registered before going on to lookup or set information such as its colour or address, from the reference table.

It would also mean that the SmartHandle class would essentially be responsible for a part of the Managed_Object's initialization, which is not ideal.

Overall it seemed most logical to ensure that, if a Managed_Object has been created on the heap, it also has an entry in the reference table, and so the first option was selected.

As part of this milestone the reference table and the SmartHandle class were updated to only work with objects of a class derived from Managed_Object.

A.3 Custom Allocator and Compactor

A.3.1 Design decisions

The initial design of both of these components and their interfaces was straight forwards and didn't require any particularly interesting design decisions to be made.

A.3.2 Implementation Challenges

The first implementation challenge related to these components was to determine how the Compactor should parse the heap. That is, how, given the current object, the Compactor should be able to find the next object in the heap, and how it should can determine the size of the current object in order to move it to its new location on the heap.

Whilst it was possible that a pointer to the next object on the heap could have been created and maintained within each Managed_Object instance, this would not solve the issue of determining the object's size (see discussion on alignment below). Therefore the simplest solution was to store the object's size as a member of Managed_Object.

The only place the size of the whole object (not just the part relating to the Managed_Object class) is known is within operator new. Therefore in order to cache this value and store it within the instance of Managed_Object it was necessary to override Managed_Object::operator new.

Then, in order to give the Managed_Object constructor access to the size of the object being created, Managed_Object::operator new caches it's 'size' argument in a static member variable of the Managed_Object class. This can then be accessed from within the Managed_Object constructor and stored within the Managed_Object instance.

The second (and somewhat larger) implementation challenge occurred from the realisation that correct memory alignment is somewhat more important than originally thought. The impact of attempting to access misaligned memory can have severe impacts on performance, and on some platforms can lead to a crash.

As this fact did not come to light until after the initial implementations of the Allocator and Compactor, support for proper memory alignment had to be retrospectively added to their implementations.

This problem stems from the fact that when a CPU reads a contiguous block of data from memory into its registers of size x bytes, it can only access blocks of memory that have an address that is a multiple of x[3]. For example, if we had a data structure that needs to be read into registers in 16 byte segments, the address of that data structure must be of the form 0x...0, i.e. a multiple of 16.

If a memory block is not correctly aligned the resulting behaviour is CPU dependent. Some modern CPU's, such as those using the intel Nehalem architecture[28], can read non-aligned memory with little or no performance penalty. Many desktop CPUs have to combine the results of multiple reads and use bit level manipulation to copy the correct value into their registers[3]. This is significantly slower than a simple read. As mentioned in subsection 8.1.2, on some console CPUs such as the one in the Playstation 2 an attempt to read misaligned data will result in a crash[3].

The solution to this problem involved changes to many elements of the components implemented so far.

Firstly the system had to obtain the alignment of the object being created in order for the Allocator to ensure that it returns a correctly aligned block of memory. Then sufficient data needed to be calculated and stored in the Managed_Object part of each object in order for the Compactor be able to correctly parse the heap and for it to be able to ensure that when it moves an object it moves it to an address that meets that object's alignment criteria.

The first problem is solved by using a macro to replace the call to operator new that the programmer uses to create new objects. This not only allows the system to retrieve the extra information that it needs to correctly handle alignment requirements, but also signals to the programmer that they are creating an object who's memory is managed by the Garbage Collector.

The macro itself is defined below:

#define gcnew(TYPE) new(__alignof(TYPE))TYPE

Where

__alignof(TYPE)

is a macro that returns the alignment requirement of a given class. This macro is built into Microsoft platforms, and is replicated for non-Microsoft platforms in HyMMS with a definition that calls the Unix C++ function alignof(), which returns the same result. This macro calls an overload of Managed_Object::operator new, which in turn passes the alignment requirement to the Allocator.

The gcnew macro is used as described in subsection 4.2.2.

The second problem is solved by storing any offset that is needed due to alignment requirements in the previous object on the heap. In this way when the Compactor examines an object it can add together the fields giving the size of the current object and the offset of the next object in order to get the address of the next object on the heap.

Also, in order for the Compactor to safely move objects on the heap it must know the alignment requirements of that object, so this is stored as a field of the Managed_Object class in the same way as the object size, by the overloaded operator new caching it in a static member of the Managed_Object class in order for it to be accessible in the Managed_Object constructor.

A.4 Incremental Compactor

A.4.1 Design decisions

This milestone involved the extension of the Compactor to allow it to perform compaction incrementally. Because the design mirrored that of the single step compactor, no interesting design decisions were necessary during this milestone.

A.4.2 Implementation Challenges

The main implementation challenge for this milestone was in obtaining microsecond precise timing information which could be used to accurately measure how long the collector had been running at any given point and to determine when it had exceeded it's time budget.

The challenge came when attempting to implement this in a cross platform way. Prior to the C++11 standard there was no standard way of obtaining microsecond precise timing information in C++[12]. On platforms derived from Unix such as Linux and Mac OS X this is exposed through the gettimeofday() function[29][30].

Under Windows this functionality is provided through the QueryPerformanceCounter() function[26]. Under C++11 there is a high_resolution_clock class within the std::chrono namespace[12]. However, in Microsoft's most recent compiler at the time of implementation, high_resolution_clock has not yet been implemented to microsecond precision[31].

In order to meet the requirement of being compiler and platform agnostic, two different implementations, combined with conditional compilation were required in a GCTimer class in order to abstract away the different methods of obtaining microsecond precise timings.

A.5 Registration of SmartHandles

This implementation of this milestone has already been detailed in subsection 5.1.

A.6 Simple Mark Compact Collector

A.6.1 Design decisions

At this stage in the implementation the only major component required for Garbage Collection was the Tracer component. One of the major design decisions in implementing the Tracer was how to best store the colour markings of each object. Whilst storing these within the object instance would have been viable it would have meant that an object's memory must be accessed and/ or written to when checking or changing its colour.

This would mean a cache miss every time an object's colour is checked or changed[5], and given that when tracing a single object multiple child objects need to be checked and/ or painted, this could be very costly.

The alternative was to store colour information in the reference table alongside the object's address. This has the major advantage that all colour information is located close to each other in memory, increasing the chances of the colour information for an objects already being in the cache when it comes to checking or marking it[5].

Also, since the appropriate line in the reference table would need to be loaded into the cache in order to access the actual object, this is highly likely to produce fewer cache misses than storing colour information in the object instance itself.

A.6.2 Implementation Challenges

The implementation of the Tracer was relatively straight forward, and no particularly difficult challenges presented themselves. Similarly, as the implementation of the simple Mark Compact collector required only simple changes to the public interface and coordination of the various components, this too was straight forwards to implement.

A.7 Incremental Mark Compact Collector

Because an incremental Compactor had already been developed at this point the only component that requires extending is the Tracer. Whilst this required adding state information to the Tracer class there were no particularly interesting design decisions to make. Similarly, as the main functionality of tracing an object was already implemented, all that remained was to adapt the Tracer to keep track of its current state and to treat the tracing of each object as an atomic operation.

The Write Barrier also had to be implemented. However since the only place in the system that is aware of a reference being created, destroyed, or modified is the SmartHandle class, there was no real decision to make on where to implement the Write Barrier. Therefore the Write Barrier was implemented as a helper function called from the constructor and assignment operator of the SmartHandle class.

There were also no particularly challenging implementation details in this milestone.

A.8 Nursery Generation

A.8.1 Design decisions

The main interesting design decisions in this milestone were around how to best use the functionality already implemented in the single step Tracer and Compactor methods for Nursery Collection. There were various potential design patterns that could have been used, such as the Decorator pattern[27], in order to add the extra functionality to the existing Compactor to cater for the Nursery Generation.

However, since the Compactor is an internal component of the system and will not be extended by others we can instead revert to functional decomposition in order to share code between the single step collector and the Nursery Collector.

A.8.2 Implementation Challenges

There were a number of implementation challenges in this milestone. With the introduction of a new colour marking, all components had to be checked to ensure that they correctly handle the fact that there is a fourth colour. Particular care had to be taken where conditions had relied on an object *not* being a particular colour, as that assumption might no longer be valid.

However a bigger challenge was altering the Write Barrier in order to correctly register and de-register the roots of the Nursery generation. The Write Barrier required significant modification to ensure that it correctly handled each of the permutations of whether or not the newly referenced object is in the Nursery generation and whether or not the previously referenced object was in the Nursery generation.

As well as this it also had to detect whether the SmartHandle being created, destroyed, or manipulated is itself a member of a Nursery object (in which case it cannot be a Nursery root). In order to do this the function used to register member references had to be extended so that it set a flag in the each reference to denote whether or not the owning object is in the Nursery Generation. The Compactor also had to be extended iterate over member references of objects being moved to the promotion buffer and unset this flag.

A.9 Large Object Handling

A.9.1 Design decisions

Since Large Object handling was intended to be largely invisible to users of the system once they have set the threshold that triggers it, there were no design decisions to take from an interface perspective. From an implementation perspective, the main consideration was how to continue to automatically manage the memory of the large objects when they are no longer on the heap.

One option was to extend the functionality of all of the other components to handle objects that are not on the main heap, and to have the Compactor check each of the large objects to see if they are alive as part of the compaction phase of the algorithm.

Whilst this would have been feasible, it would have required significantly extending all of the other components, and would have been likely to introduce several bugs into the system that would have needed to be tracked down and fixed.

The other option was to employ the Proxy pattern[27] and have a fully memory managed object on the heap that, when destroyed, would destroy the large object it was the proxy of as part of its finalization. This was a much more elegant solution, as it meant for most of the other components that large objects on the heap was treated no differently to any other unmanaged resources.

A.9.2 Implementation Challenges

The one issue that could not be resolved through the use of a proxy was the registration and tracing of member references in the large object. Firstly member references would need to check through each of the large objects in order to determine whether they are members of those objects. Secondly, the Tracer needs to be able to determine that an object on the heap is a proxy and if so trace the large object that it is proxying for rather than the proxy itself.

It would have been exceptionally expensive to search through the large object list for every SmartHandle that is not a member of an object on the heap, as many of these would actually be root handles and not members of large objects. Instead the GC class keeps track of the addresses of the upper and lower bound of system memory in which the large objects are stored, and uses this as a filter to determine whether handles are likely to be members of large objects or not. If they pass this filter the handles must then search the large objects list.

This solution is not ideal, but it is acceptable as by definition there are likely to be very few large objects in existence[3], and so a handle should be able to iterate over them fairly quickly.

In order for the Tracer to determine whether or not an object is a proxy, a flag was set in the same byte used to store the colour flag of the object in the reference table.

A.10 Pinned Objects

Pinned objects were not an original milestone set out in the project proposals. However, it became clear during development that many 3rd party libraries require pointers to objects in fixed locations in memory, and so there was a need for the programmer to be able to create objects with fixed addresses.

One option would have been to have users of the system implement their own proxy for such pinned object as the need arises.

However, since good cooperation with 3rd party libraries is one of the essential criteria for the project and one of the main reasons that developers might want to use the system, it seemed logical to implement this as a built-in feature.

A.10.1 Design decisions

The main design consideration for this milestone was what the user interface should be for allowing programmers to signal that an object should be pinned. There were two main ways that this could be achieved:

- A Pinned_Object base class that inherits from Managed_Object
- Include the functionality required for pinning in Managed_Object and add a gcnew_Pinned creation macro to signal that an object should be pinned

The main advantage of having a separate base class is that it provides type safety for pinned objects, ensuring that a pinned object is not accidentally 'unpinned' through copying where that is not intentional.

The main disadvantage is that the programmer needs to carefully design their classes around which objects need to be pinned. It also means that the programmer must be able to tell at compile time whether or not an object will ever need to be pinned.

In some scenarios that might be difficult, for example if a library needed a pointer to a GameObject, but only one at a time, then the programmer would need to make all GameObjects pinned (which would be a large and unnecessary overhead) or else to create a separate PinnedGameObject class with matching conversion operations.

The main advantage of integrating pinned object functionality into the Managed_Object class is that it allows any object instance (rather than type) to be pinned.

The main disadvantage is that this method cannot exploit type safety to protect against the passing a pointer to an unpinned object to a 3rd party library. However, it is impossible for the library to completely protect the programmer wherever an interaction with manual memory management is concerned, and so this is an acceptable trade-off for the additional flexibility.

A.10.2 Implementation challenges

There were no real implementation challenges in delivering this milestone, since much of the functionality for pinning an object had already been implemented to support large object handling.

A.11 Managed_Array

Another element that was missing from the original system description in the project proposal was any kind of container class for managed objects. Neither C-style arrays nor the container classes in the C++ standard library are suitable for objects managed by the HyMMS library by default.

However arrays and other containers are commonly used in a game development environment, as games often involve collections of similar objects that need to be iterated over and have some action performed on them[1] (e.g. a list of all objects in the scene, or a list of events to trigger).

Therefore it was logical to implement a basic array for Managed_Objects on top of which other more complex container classes could be built by users of the system as needed.

A.11.1 Design decisions

The were two major design decisions that were made when implementing the Managed_Array class.

The first of these was to decide what subset of the normal array/ container functionality to provide to the user. The most commonly used functionality of linear containers such as c-style arrays and std::vector is access to elements via the subscript operator[6], i.e. if arr is a c-style array then arr[i] returns the i^{th} member of the array.

As this is the most commonly used functionality for containers, this was the bare minimum that had to be included in the interface of Managed_Array. In addition to the subscript operator, a commonly used function of std::vector is the size() member function[6]. This returns the number of elements currently stored in the vector. This is not functionality that c-style arrays provide, however programmers that use c-style arrays often find that they must manually keep

track of the number of elements in the array in order to iterate over them safely[6].

Therefore it seemed sensible to provide this functionality to users of Managed_Array. There were two other prominent features of std::vector that might have been useful to programmers using Managed_Array. These are stack/ queue operations and iterators.

Stack and queue operations are simply convenient operations for adding and removing members to the front or end of the array. However the implementations of these operations usually require resizing the array in memory. Whilst this would be possible in HyMMS, it would have been difficult to implement efficiently in the time constraints available. These are also operations that a programmer could implement themselves by extending Managed_Array (whereas this would have been impossible with the subscript operator).

The primary benefit of iterators is to provide a consistent interface for users writing generic code to iterate across an arbitrary container class[6]. However, since we are only implementing a linear container class there is little benefit to offering the functionality of an iterator to users. Again this is something that programmers using HyMMS could easily extend Managed_Array to provide.

The second design consideration was how the user should interact with the Managed_Array object itself. There were two options:

- to follow the approach of std::vector, where an object of type std::vector which is used to access the elements of the vector, or
- to follow the approach of c-style arrays where the user applies the subscript operator to a pointer of the same type as the elements of the array

These two approaches would require the user to use a different syntax. Since the Managed_Array object itself would likely be created on the heap and referenced through a SmartHandle, if we took the std::vector approach then the syntax for accessing an element of the Managed_Array would be:

```
// Create a Managed_Array<T> with 10 elements

SmartHandle<T> array = gcnew(Managed_Array<T>)(10);

// Initialize a new variable of type T with the value 100

//assume it has a constructor that accepts an int argument

T temp = T(100);
```

// Set the first element of the Managed_Array to the value of temp

(*array)[0] = temp;

This last line is the code that is used to access the member of the Managed_Array. This is slightly cumbersome, as the user needs to remember to apply the de-reference operator (*) to the SmartHandle before applying the subscript operator ([]).

If instead we use the c-style array approach we get the following:

// Create a Managed_Array<T> with 10 elements

SmartHandle<T> array = gcnew(Managed_Array<T>)(10);

// Initialize a new variable of type T with the value 100 // assume it has a constructor that accepts an int argument

T temp = T(100);

// Set the first element of the Managed_Array to the value of temp

array[0] = temp;

This is clearly much cleaner, and avoids the potential for errors if the de-reference operator is omitted. Therefore the c-style of access was chosen, and the subscript operator was integrated into the SmartHandle class.

A.11.2 Implementation challenges

One challenge was ensuring that the elements of the Managed_Array were handled correctly by the collector. This required special code paths for handling Managed_Arrays rather than normal Managed_Objects in a number of the components, in particular the Tracer component.

The elements of the array, which are Managed_Objects in their own right, may be referenced either directly through a SmartHandle in the normal way, or they may be accessed via the base Managed_Array object and the subscript operator. If an element is referenced directly by a SmartHandle then the system will treat the element the same as any other Managed_Object. However, if the element is not referenced by any SmartHandle but the base Managed_Array object *is* referenced, then the element is still reachable and so should not be collected.

Therefore the Tracer component needs to treat a Managed_Array object as though it has SmartHandle members that reference each of its elements, and so paint each element Grey if the Managed_Array object is Grey when it traces it.

A bigger challenge was the implementation of the subscript operator. A naïve implementation would simply have multiplied the index of the requested element by the size of the type of the elements in the Managed_Array. This would then be added to the address of the first element to get the address of the requested element.

However, due to the incremental nature of the compactor, it is possible that the at any point in time only some of the elements of the Managed_Array will have been compacted, leaving a 'gap' in the array. As only the Compactor holds information about its current state, it needed to be extended to detect when it was compacting an array, and then it needed to keep track of how many elements of the array it has compacted so far.

It then needed to expose this information to the SmartHandle class, along with the size of the 'gap' between elements (which the Compactor already tracks as this is equal to the total memory size of all 'dead' object encountered so far).

A.12 Unmanaged_Array

The main downside of the Managed_Array class is the requirement that elements be Managed_Objects. It may be that the programmer needs to use arrays of objects that cannot be Managed_Objects (e.g. because they are part of a 3rd party library). However, they either need to keep track of these arrays manually, or else they will need to implement their own managed proxy to have their memory managed by the HyMMS library.

Another reason to want to be able to keep an array of objects on the heap that are not Managed_Objects is the fact that SmartHandles are not Managed_Objects. It is very common, both in game development and elsewhere in software development, to need to use arrays of references[1][6]. As SmartHandles are part of the HyMMS system, it would be perverse if an array of SmartHandles could not be kept on the managed heap.

A.12.1 Design decisions

The design decisions for this class mirrored those of the Managed_Array class. For consistency the functionality provided by the Unmanaged_Array class was kept the same as that of the Managed_Array class. There was however a different decision with regards to the subscript operator.

Whilst the decision was still taken that the subscript operator should apply to the handle pointing to the Unmanaged_Array (as with Managed_Array), this could not be implemented directly in the SmartHandle class as with the Managed_Array subscript operator. This is because SmartHandles cannot reference unmanaged objects.

Instead a new handle class was created - UnmanagedArrHandle - and the subscript operator was implemented on that instead.

A.12.2 Implementation challenges

The main implementation challenge was how to represent the unmanaged elements on the heap. However the solution to this was ultimately very straight forwards. Unmanaged objects are already allowed on the heap as members of Managed_Objects.

Therefore the simplest approach was to trick the HyMMS library into seeing the array elements as members of the Unmanaged_Array object. This was implemented simply by setting the Unmanaged_Array's size so that the system sees it as encompassing the elements of the array. Then the system will register SmartHandles in the array as being members of the Unmanaged_Array object, and so trace and compact the array correctly.

The one caveat of using the Unmanaged_Array is that C++ pointers should not be kept to elements of the array. This is because the array will be moved in memory by the compactor. As such elements should only be accessed through the subscript operator on the UnmanagedArrHandle object that references the Unmanaged_Array. However this is the same as the requirement for using std::vector[6], and so this is something most C++ programmers will be familiar with[3].

A.13 Handle_Array

As mentioned in the previous subsection, it is expected that game developers will need to use arrays of references at some point in the development of most games. Using Unmanaged_Array and UnmanagedArrHandle, the syntax for this would be:

UnmanagedArrHandle<SmartHandle<T> > arr = gcnew(SmartHandle<T>)();

This is very cumbersome for code that is likely to occur relatively frequently. Therefore a simple alias for this is provided by the system in the form of the Handle_Array class (which is an alias for Unmanaged_Array<SmartHandle>) and HndlArrHandle, which is an alias for UnmanagedArrHandle<SmartHandle>. There the new syntax for the above would be:

HndlArrHandle<T> arr = gcnew(Handle_Array<T>)();

A.13.1 Design decisions

As this was only an alias there were no real design decisions to take here.

A.13.2 Implementation Challenges

Prior to the C++11 standard, it was not possible to use typedef with a templated type where the templated parameter was undetermined[6]. E.g. the following would cause a compilation error:

```
template<typename T>
typedef Unmanaged_Array<SmartHandle<T> > Handle_Array<T>;
```

C++11 adds a feature called template aliases which allows this using a different syntax[12]. However, once again, Microsoft's current compiler does not support this feature[14], and so it is not available to use without compromising one of our essential criteria.

Instead this aliasing was achieved here by making Handle_Array an empty subclass of Unmanaged_Array<SmartHandle>. This gives Handle_Array the full functionality of an Unmanaged_Array of SmartHandles, but allows us to use the simpler syntax outlined above. To illustrate, the full defintion of the Handle_Array class is:

```
template<typename T> Handle_Array
    : Unmanaged_Array<SmartHandle<T> > {};
```

The same approach is taken with HndlArrHandle and UnmanagedArrHandle<SmartHandle>.

References

[1] McShaffry et. al., 2012. *Game Coding Complete*. 4th Edition, Delmar

[2] Millington, I., 2007. *Game Physics Engine Development*. 1st Edition, CRC Press

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

[4] Martin, C. B., 2009. *The Independent Production of Culture: A Digital Games Case Study*. Games and Culture, Volume 4 Number 3, 276 - 295

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

[6] Stroustrup, B., 1997. *The C++ Programming Language*. 3rd Edition, Addison Wesley

[7] *2008 State of Game Development Survey*. Game Developer. 2008

[8] *Game Engines and Middleware in the West*. DeLoura, M., Computer Entertainment Developers Conference 2011

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

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

[11] Dijkstra, E. et al., 1978. *On-the-fly Garbage Collection: An exercise in cooperation*. Communications of the ACM

[12] ISO/IEC DTR 19768, *Draft Technical Report on C++ Library Extensions*. 2005

[13] Karlsson, B., 2005. *Beyond the C++ Standard Library: An Introduction to Boost*. 1st Edition, Addison-Wesley

[14] Lavavej, S. T., 2011. *C++11 Features in Visual C++ 11*. URL: http://blogs.msdn.com/b/vcblog/archive/2011/09/12/10209291.aspx [30 August 2012]

[15] Boehm, H. and Weiser, M., 1988. *Garbage Collection in an uncooperative environment*. Software: Practice and Experience Volume 18, Issue 9, pages 807-820

[16] Strobl, T., 2007. *Modern Concepts Applied to C++ - Object Persistence, Reflection, Events, Garbage Collection and Thread Safety in C++*. 1st Edition, Saarbrücken

[17] *Box2D: A 2D Physics Engine for Games.* URL: http://box2d.org [16 August 2012]

[18] *Box2D.XNA: a C#/XNA port of Box2D.* URL: http://box2dnxa.codeplex.com [18 August 2012]

[19] *Fraps: Real-time video capture & benchmarking.* URL: http://www.fraps.com [16 August 2012]

[20] Bruckschlegel, T., 2005 *Microbenchmarking C++, C# and Java.* URL: http://www.drdobbs.com/cpp/microbenchmarking-c-c-and-java/184401976 [31 August 2012]

[21] Astle, D. and Hawkins, K., 2004. *Beginning OpenGL Game Programming.* 1st Edition, Delmar

[22] Rao, A., *Maximizing Risk: The Building of Bastion.* Game Developers Conference China 2011, URL: http://www.gdcvault.com/play/1015264/Maximizing-Risk-The-Building-of [03 September 2012]

[23] Bedard, R., *Cubes All the Way Down: FEZ Technical Postmortem* Game Developers Conference 2012, URL: http://www.gdcvault.com/play/1015731/Cubes-All-the-Way-Down [03 September 2012]

[24] *XNA Game Studio 4.0 Refresh* MSDN Documentation, URL: http://msdn.microsoft.com/en-us/library/bb200104.aspx [17 August 2012]

[25] Blackburn, S. M. et. al, 2004. *The performance impact of garbage collection.* International Conference on Measurement and Modelingof Computer Systems, ACM SIGMETRICS Performance Evaluation Review, ACM Press

[26] *How To Use QueryPerformanceCounter to Time Code.* Microsoft Support Article, URL: http://support.microsoft.com/kb/172338 [06 September 2012]

[27] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* 1st Edition, Addison-Wesley

[28] Levinthal, D., 2008. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors.* URL: http://software.intel.com/sites/products/collateral/hpc/vtune /performance_analysis_guide.pdf [30 August 2012]

[29] Goldt, S., Meer, S., Burkett, S. and Welsh, M., 1995. *The Linux Programmer's Guide.* URL: http://tldp.org/LDP/lpg-0.4.pdf [07 August 2012]

[30] *Mac OS X Manual Page For gettimeofday*. Mac Developer Library, URL: https://developer.apple.com/library/mac/#documentation/Darwin /Reference/ManPages/man2/gettimeofday.2.html [07 August 2012]

[31] Lavavej, S. T., 2012. *Response to 'C++ <chrono> header's high_resolution_clock does not have high resolution'* URL: http://connect.microsoft.com/VisualStudio/feedback/details/719443/c-chrono-headers-high-resolution-clock-does-not-have-high-resolution [03 August 2012]