BACHELORARBEIT

im Studiengang Bachelor Informatik

Custom Memory Management in der Spieleentwicklung und C++

Ausgeführt von: Simon Dobersberger
Personenkennzeichen: 1010257008

BegutachterIn: DI Stefan Reinalter

Wien, 13. Januar 2013

FACHHOCHSCHULE
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Ort, Datum

Unterschrift

Kurzfassung

Das Thema 'Custom Memory Management in der Spieleentwicklung und C++' behandelt verschiedene Methoden für das Debugging und die Optimierung von Speicheroperationen. Im Rahmen dieser Bachelorarbeit werden folgende Themengebiete erläutert:

- Die Analyse der Speicherverwaltung innerhalb von C++ zeigt die Unterschiede und Aufgaben von Stack, Heap und Global Memory in Executables auf. Weiters ist die manuelle Anpassung des Speicheraufbaus eine essentielle Aufgabe im Bezug auf die effiziente Nutzung des Speichers auf Spielkonsolen und PCs.

- Die Verwendung von unterschiedlichen Speicherallokatoren erlaubt es dem Entwickler Speicher ohne Angst vor Speicherfragmentierung zu nutzen, die Performance zu verbessern und Ladezeiten zu verkürzen.

- Eine Analyse der Marktsituation zeigt auf, dass der verfügbare Speicher besonders auf Spielkonsolen verhältnismäßig gering ist. Memory Tracking hilft dem Programmierer dabei Speicherleaks, welche eine häufige Ursache für Out-of-Memory-Meldungen sind, zu entdecken. Weiters ist es für den Anwender nachvollziehbar, wieviel Speicherplatz verbleibt oder ob der Speicher beschädigt ist. Das Konzept eines Memory Trackers wird anhand eines Code-Beispiels in C++ behandelt.

Schlagwörter: Speicherverwaltung, Stack, Heap, Anpassung des Speicheraufbaus, Speicherallokatoren, Memory Tracking

Abstract

The topic 'Custom Memory Management in Game Development and C++' covers various methods for debugging and optimizing memory operations. Within the scope of this bachelor thesis, following subjects are introduced:

- The analysis of the memory management in C++ shows the differences and tasks of the stack, heap and global memory in executables in C++ programs. Furthermore, custom memory alignment is an important task for using the memory of gaming consoles and PCs efficiently.
- Using different types of allocators allows the developer to use memory without the fear of memory fragmentation and to save performance during load times.
- A market analysis also shows that the available memory of gaming consoles is especially low. Memory tracking helps the programmer to track down memory leaks, which are mostly the cause of out of memory exceptions. Moreover, it tells the user how much memory is left and if it is corrupted. This is shown by a code example of a memory tracker in C++.

Keywords: Memory management, stack, heap, memory alignment, custom allocators, memory tracking

Danksagung

An dieser Stelle möchte ich mich vielmals bei meinem Betreuer bedanken, welcher mir mit seiner fachlichen Kompetenz beiseite stand und im Zuge des Semesters einen tiefen Einblick in das Thema gewährte.

Inhaltsverzeichnis

1. Einleitung **1**

2. Analyse der Speicherverwaltung innerhalb von C++ **2**
2.1. Executables . . . 2
2.2. Stack . . . 2
2.3. Heap . . . 3
2.4. Memory Alignment . . . 3
2.4.1. Custom Memory Alignment . . . 5

3. Vergleich von Speicherallokatoren in der Spieleentwicklung **6**
3.1. Heap Allokator . . . 6
3.2. Stack Allokator . . . 8
3.2.1. Double-Ended Stack Allokator . . . 8
3.2.2. Stack-Based Resource Allocation . . . 9
3.3. Pool Allokator . . . 9
3.4. Single-Frame Allokator . . . 10
3.4.1. Double-Buffered Memory Allokator . . . 10

4. Memory Tracking **11**
4.1. Marktanalyse zur Größe des Speichers auf aktuellen Spieleplattformen . . . 11
4.2. Das Konzept eines Memory Trackers . . . 12
4.3. Betrachtung anhand eines Programmbeispiels . . . 13
4.3.1. MemoryManagement.h/.cpp und die MemoryManager-Klasse . . . 14
4.3.2. MemoryTracker.h/.cpp und die MemoryTracker-Klasse . . . 17

5. Zusammenfassung **21**

Literaturverzeichnis **22**

Abbildungsverzeichnis **23**

A. Anhang: Komplettes Codebeispiel aus dem Kapitel “Memory Tracking” **24**

1. Einleitung

Custom Memory Management ist ein essentieller Teilbereich in der Spieleentwicklung und aus den meisten aktuellen Spiele-Engines nicht mehr wegzudenken. Dies hat unter anderem folgende Gründe:

- Durch die Optimierung von dynamischen Speicherallokationen kommt es zu einer Verbesserung der Prozessorleistung sowie einer Verringerung des benötigten Speicherplatzes.
- Mithilfe eines Memory Trackers werden sogenannte Speicherleaks innerhalb des Programms aufgespürt, welche während der Laufzeit zwar Speicher anfordern, jedoch nie wieder freigeben. Durch die Beseitigung dieser Problematik werden Abstürze und eine Verschlechterung der Performance verhindert.

Für diese Thematik ist es wichtig, das Konzept der Speicherverwaltung zu verstehen und in C++ einsetzen zu können, da intensiv damit gearbeitet wird. Dieses wird daher im nachfolgenden Kapitel ausführlich analysiert.

2. Analyse der Speicherverwaltung innerhalb von C++

Für das Custom Memory Management in der Spieleentwicklung ist das Verständnis zur Aufteilung des Speichers in C++ erforderlich. Bei der Analyse dieser Speicherverwaltung geht zunächst hervor, dass dabei zwischen drei verschiedenen Bereichen unterschieden wird:

- Das fixe Speicherabbild in *Executables*[1]
- Der Stack
- Der Heap

Weiters ist die sogenannte *Memory Alignment* von bedeutender Wichtigkeit für die Effizienz, Funktion und Leistung der Speicherverwaltung.

2.1. Executables

Ein Executable wird in der Regel in mehrere verschiedene Segmente unterteilt. Diese sind je nach Betriebssystem unterschiedlich, haben aber zumindest unter UNIX- und Windows-System zahlreiche Ähnlichkeiten. Dazu zählt unter anderem die Speicherung der globalen und statischen Variablen, zu welchen auch die *Virtual Function Tables*[2] zählen. Zu finden ist deren Speicher im *Data Segment*, welches von Mike McShaffry in Game Coding Complete auch als *Global Memory* bezeichnet wird. Im Gegensatz dazu unterteilt Jason Gregory in Game Engine Architecture das Data Segment nochmals genauer in *Data Segment* und *Read-only Data Segment*. Letzteres wird auch häufig als *rodata* abgekürzt. In diesem Segment werden alle konstanten, globalen Variablen gespeichert, die innerhalb des Programms schreibgeschützt sind. Im sogenannten *BSS-Segment*, was für *block started by symbol* steht, werden ebenfalls globale und statische Variablen gespeichert. Allerdings sind diese nicht initialisiert. Der Linker speichert diese jedoch nicht mit Null-Werten in der Datei, sondern zählt stattdessen die Anzahl der benötigten Bytes. Diese werden vor dem ersten Funktionsaufruf vom Programm reserviert und mit Nullen bzw. den Standardwerten gefüllt.

2.2. Stack

Der Stack ist ein spezieller Bereich im Speicher der beim Start eines Programms vom Betriebssystem reserviert wird. Auf diesen werden nach und nach sogenannte *Stack Frames* gesetzt. Dabei handelt es sich um einen Speicherblock der bei jedem Funktionsaufruf erstellt wird. Speziell entsteht dieser bei der Verwendung der geschwungenen Klammern und definiert den *Local Scope*, welcher im Gegensatz zum *Global Scope* nur für den aktuellen Stack Frame gültig ist. Unter anderem werden dadurch lokale Variablen im Speicher verwaltet. Sobald das Programm die Funktion

[1] Executable = Die ausführbare Datei des Programms, welche vom Linker erzeugt wird, z.B. .exe oder .elf
[2] Virtual Function Tables = Tabelle für Zuordnung der virtuellen Funktionen innerhalb von Klassen

wieder verlässt, werden diese lokalen Objekte automatisch zerstört und freigegeben. Wie Jason Gregory erklärt, werden diese auch als *automatische Variablen* bezeichnet, da sich der Programmierer nicht um die Allokation und Deallokation kümmern muss. Da sie nach dem Austritt aus dem Local Scope jederzeit wieder im Speicher überschrieben werden könnten, ist darauf zu achten, dass diese vom Programm nicht nach außen hin referenziert werden.

2.3. Heap

Dynamische Allokationen behandeln den Speicher der während der Laufzeit mittels *malloc* bzw. *new* angefordert wird. Dieser wird auf dem sogenannten *Heap* reserviert, welcher vom Betriebssystem verwaltet wird. Ein Problem welches sich durch die Verwendung dieses Speicherbereichs ergibt, ist das Speicherleak. Es bezeichnet den Vorgang, wenn Speicher vom Programm angefordert, jedoch nie wieder freigegeben wird. Folglich sorgen Speicherleaks früher oder später für eine *Out-of-Memory-Exception*, eine Meldung die dann auftritt, wenn am Heap kein Platz mehr für neue Allokationen vorhanden ist. Um dieses Problem zu verhindern werden in der Spieleentwicklung Debug-Tools wie etwa der Memory Tracker eingesetzt, welcher in Kapitel 4 näher behandelt wird.

2.4. Memory Alignment

Aus Gründen der Performance wird die Position von Daten im Speicher nach einem speziellen Schema ausgerichtet. Dies ist unter anderem der Fall, wenn eine Struktur oder Klasse wie folgt definiert ist:

```
struct MemoryPacking
{
        int m_id;
        char m_char;
        int m_count;
        char m_reg;
        int m_points;
};
```

Die Annahme, dass die Variablen dicht gedrängt im Speicher liegen, ist nicht korrekt. Stattdessen werden die Speicherblöcke an ihrer Position angepasst, damit diese eine Speicheradresse besitzen, welche durch die Größe des Datentyps teilbar ist. Ein *char* kann sich daher an jeder Position im Speicher wiederfinden. Im Gegensatz dazu, ist ein Pointer mit einer Speichergröße von vier Bytes an einer Speicheradresse wie etwa 0x0, 0x4, 0x8, usw. positioniert. Sofern die Integer-Variablen in diesem Beispiel eine Größe von 32 Bit besitzen, wird der Speicher daher wie folgt angeordnet:

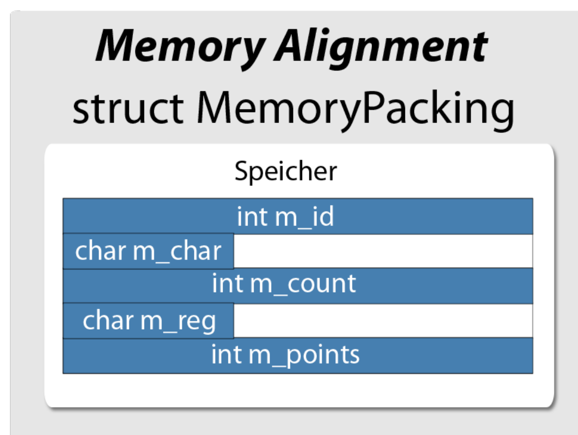

Abbildung 2.1.: Memory Alignment im Speicher

Diese Anordnung ermöglicht es der CPU effektiv auf die Speicheradressen zuzugreifen und wird vom Compiler durchgeführt. Moderne Prozessoren haben teilweise keine andere Möglichkeit auf Objekte zuzugreifen, weshalb zum Beispiel für eine schlecht positionierte Integer-Variable an der Adresse 0x40623173 die Speicherblöcke 0x40623172 sowie 0x40623176 ausgelesen werden. Anschließend werden zusätzlich noch Operationen durchgeführt, um nun auf die korrekten Bytes zuzugreifen.

> "Some microprocessors don't even go this far. If you request a read or write of unaligned data, you might just get garbage. Or your program might just crash altogether! (The PlayStation 2 is a notable example of this kind of intolerance for unaligned data.)"
> *Jason Gregory, Game Engine Architecture, 2009*

Es ist jedoch trotzdem möglich die Speicherblöcke unabhängig von Memory Alignment aneinanderzureihen. Dies geschieht mithilfe von Compiler-Anweisungen wie z.B. *pragma pack*, führt allerdings zu den angesprochenen Problemen.

Wird die Beispielstruktur nun entsprechend angepasst, lässt sich auf diese Weise Speicherplatz sparen:

```
struct MemoryPacking
{
        int m_id;
        int m_count;
        int m_points;
        char m_char;
        char m_reg;
};
```

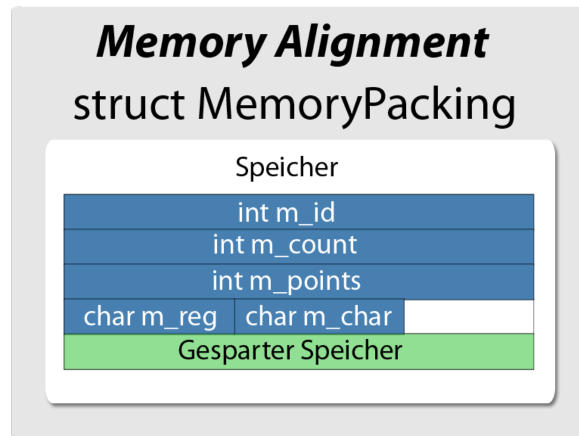

Abbildung 2.2.: Sparsames Memory Alignment im Speicher

Wie in Abbildung 2.2 ersichtlich, werden vier Bytes weniger eingesetzt als im Szenario zuvor.

2.4.1. Custom Memory Alignment

Wie die Analyse zum Thema *Memory Alignment* ergeben hat, ist eine Anpassung der Speicherposition von essentieller Wichtigkeit. Dies betrifft insbesondere solche Prozessoren, welche unausgerichtete Speicherblöcke nicht verwalten können. Wie Jason Gregory in *Game Engine Architecture* erwähnt, ist in der Regel der Compiler für das Memory Alignment zuständig. Dennoch ergibt sich in bestimmten Szenarien die Notwendigkeit diese Anpassung manuell vorzunehmen. Dies ist zum Beispiel auf der PlayStation 3 der Fall, bei der die Vorgabe lautet, dass die Speicherblöcke nach einem 128-Bit-Schema angeordnet sind. Dadurch wird der maximale Speicherdurchsatz gewährleistet.

Daher liefern die verschiedenen Speicherallokatoren, auf die im nächsten Kapitel näher eingegangen wird, sogleich korrekt angeordnete Speicheradressen zurück. Für die Implementierung ist es notwendig, dass ein bestimmter Anteil an Speicher extra allokiert wird, um die Speicherblöcke korrekt anzuordnen. Dieser zusätzliche Platz wird außerdem dazu verwendet, um die abweichende Position festzuhalten und so im Falle einer Deallokation den Speicher korrekt freizugeben.

Praktisch wird deshalb je nach gewünschter Anpassung zusätzlicher Speicher in dieser Größe allokiert. Folglich werden bei einem 128-Bit-Schema zusätzlich 16 Bytes angefordert. Für eine korrekte Anordnung und Verwendung des Custom Memory Alignment beträgt die Anpassungsgröße den Wert einer Zweierpotenz. Unter Zuhilfenahme einer Bitmaske bei der dieser Wert um eins verringert wird und einer anschließend AND-Operation mit der unangepassten Speicheradresse lässt sich die vorzunehmende Anpassung errechnen. Der Speicherblock wird um das Ergebnis verschoben und im direkt vorhergehenden Byte dieser Wert gespeichert. Bei der Deallokation des Speichers wird der gesamte Block mithilfe des gespeicherten Wertes korrekt freigegeben.

3. Vergleich von Speicherallokatoren in der Spieleentwicklung

Aufgrund von modernen CPUs, zählen nicht mehr die Rechenoperationen an sich, sondern vor allem die Speicherzugriffsmuster zu den leistungsbeeinflussenden Faktoren bei der Spieleentwicklung. Dieser von Jason Gregory in *Game Engine Architecture* erwähnte Umstand, führt dazu, dass Speicher, der in kleinen und direkt aneinander liegenden Speicherblöcken allokiert wurde, wesentlich schneller vom Prozessor verwaltet wird.

Aufgrund dessen werden je nach Operation viele verschiedene Typen von dynamischen Speicherallokatoren verwendet. Nachfolgend werden deshalb einige dieser Allokatoren, welche innerhalb des Speichermanagements in aktuellen Spiele-Engines zum Einsatz kommen, näher erläutert sowie auf ihre Einsatzszenarien, Vor- und Nachteile eingegangen.

3.1. Heap Allokator

Beim Heap Allokator handelt es sich um Allokationen die standardmäßig mittels der Operatoren *new* und *delete* erfolgen. Da dieser ohne spezielle Aufgabe alle möglichen Arten von Speicherallokation verwalten muss, egal ob es sich dabei um Bytes, Kilobytes, Megabytes oder gar Gigabytes handelt, entsteht auf Dauer ein gewisser Overhead. Innerhalb der Operatoren werden die C-Befehle *malloc* und *free* verwendet. Diese müssen auf den meisten Betriebssystemen zunächst einen Kontext-Wechsel durchführen. Dabei wird in den Kernel-Modus gewechselt, die Operation durchgeführt und anschließend wieder in den Benutzer-Modus und somit das Programm wieder aktiviert. Dieser Wechsel zwischen den Modi ist während der Laufzeit relativ teuer und wird daher in der Spieleentwicklung möglichst vermieden.

Genau diesen Nachteil behandelt das Custom Memory Management mit diversen eigenen Speicherallokatoren, indem im Vorhinein eine bestimmte Größe an Speicher allokiert und dieser dann im Benutzer-Modus verwaltet wird. Das kann auf verschiedene Arten geschehen und je nach Szenario nochmals eine Verbesserung der Leistung herbeiführen.

Ein weiterer Nachteil ist die Positionierung der Speicherblöcke innerhalb des Heaps. Dieser entsteht dadurch, dass der Speicher willkürlich während der Laufzeit allokiert und freigegeben wird. Einerseits muss das Programm dabei möglicherweise zwischen komplett verschiedene Positionen im Heap springen. Andererseits entsteht durch diesen Vorgang auf Dauer auch etwas, das als *Speicherfragmentierung* bekannt ist. Die folgende Grafik soll dies verdeutlichen.

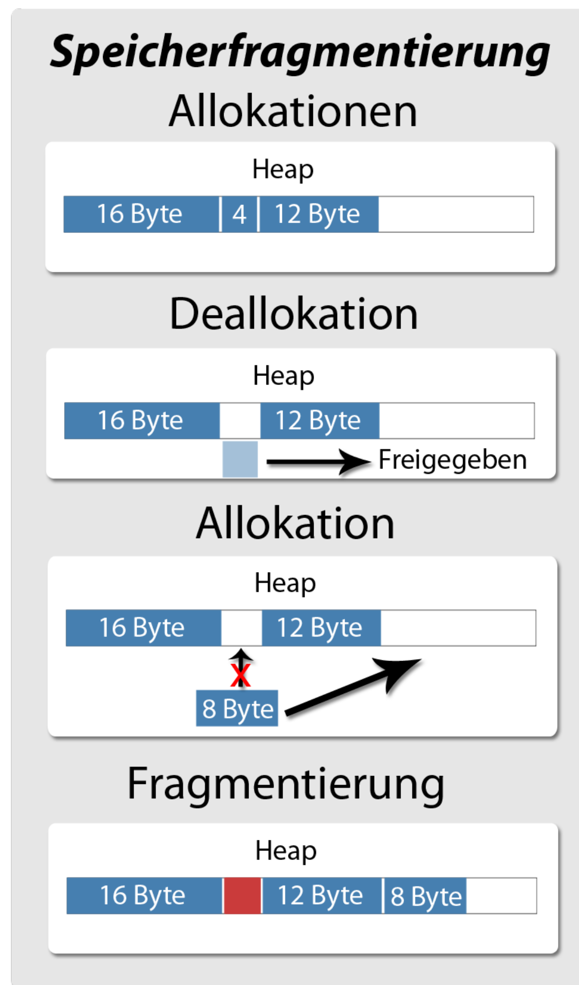

Abbildung 3.1.: Speicherfragmentierung im Heap

Durch diese Durchlöcherung des Speichers entsteht auf Dauer vor allem ein höherer Speicherverbrauch. Dies kann gerade bei Spielkonsolen wie die XBox360, Nintendo Wii oder die PS3, welche im Gegensatz zum PC über sehr begrenzte Speicherressourcen verfügen, früher oder später zu einer *Out of Memory Exception* führen. Sprich, es ist kein Platz mehr für neue Speicherallokationen verfügbar.

Ähnlich wie bei Festplatten bietet es sich nun als erste Überlegung an, dieses Problem mittels einer regelmäßigen Defragmentierung des verwendeten Speichers zu umgehen. Dadurch ist allerdings eine Aktualisierung sämtlicher Pointer in der Applikation notwendig bzw. kostet dies zusätzliche Leistung, die in anspruchsvollen Spieletiteln ohnehin sehr kostbar ist. Deshalb bietet sich hier ebenfalls die Verwendung von eigenen Speicherallokatoren an, welche den Heap bei Allokationen so organisieren, dass es erst gar nicht zur Fragmentierung kommt.

3.2. Stack Allokator

Bei einem Stack Allokator werden alle Speicherallokationen, wie der Name schon besagt, wie nach einem Stack-Muster durchgeführt. Dies bedeutet, dass zunächst ein größerer Speicherblock allokiert wird, in welchem die darauf erfolgenden Allokationen genügend Platz vorfinden. Auf die Anfrage nach neuem Speicher hin, wird nun jedes Mal der Pointer für das obere Ende des Stack Allokators zurückgeliefert. Dadurch entsteht eine enge Aneinanderreihung der allokierten Speicherobjekte, sodass der Speicher iterativ und ohne Sprünge ausgelesen werden kann. Zudem wird dadurch der zuvor angesprochene Kontext-Wechsel ausgeschaltet, da dieser lediglich einmal für den gesamten Allokator vorgenommen werden muss. Alle nachfolgenden Allokationen finden im Benutzer-Modus statt.

Diese Vorgangsweise hat jedoch einen entscheidenden Nachteil. Der allokierte Speicher kann in keiner beliebigen Reihenfolge freigegeben werden. Stattdessen wird immer zunächst der oberste Speicherblock deallokiert. Möchte das Programm nun also Speicher inmitten des Stack Allokators freigeben, so muss dies zunächst für alle davorliegenden Speicherobjekte getan werden. Dieser Nachteil ist aber zugleich ein großer Vorteil. Denn hierdurch wird eine Fragmentierung des Speichers verhindert. Der Stack Allokator wird deshalb vorzugsweise für die statischen Daten eines Levels verwendet. Da hier lediglich eine einzelne Heap-Allokation durchgeführt wird, wird die Ladezeit je nach Anzahl der Level-Objekte erheblich verkürzt. Sobald der Spieler den Level zu Ende gespielt hat, wird der allokierte Speicher des Stack Allokators komplett entfernt. Damit diese Vorgangsweise funktioniert, muss das Spiel allerdings auch tatsächlich in Level unterteilt und linear sein. Das trifft zum Beispiel auf Arcade Shooter oder Jump'n'Runs zu.

Um stets die Position zwischen zwei Speicherblöcken bzw. dem oberen Ende des Stacks zu ermitteln, werden sogenannte *Marker* verwendet. Bei jeder Allokation wird der Marker des Stack Allokators mit der Größe des angeforderten Speichers in Bytes nach hinten verschoben. Der Programmierer kann zu jeder Zeit die Position des Markers erfragen und sich dadurch die Position im Speicher des Stack Allokators außerhalb an einer anderen Stelle im Programm merken. Wenn dieser nun den Stack bis zu einem bestimmten Objekt freigeben möchte, muss er lediglich dessen zugehörigen Marker übergeben und der Allokator deallokiert den kompletten Speicher bis zu dieser Position.

3.2.1. Double-Ended Stack Allokator

Der Double-Ended Stack Allokator ist eine Abwandlung bzw. Erweiterung des zuvor besprochenen Stack Allokators. Anstatt die Allokationen innerhalb des reservierten Speichers lediglich von unten nach oben durchzuführen, erfolgen diese zugleich auch vom anderen Ende, also von oben nach unten. Der Vorteil dieser Vorgangsweise ist, dass der Speicher effizienter genutzt werden kann. Die beiden Enden treffen auch niemals aufeinander, sofern die Allokationen nicht größer als der reservierte Speicher des Stack Allokators sind. Daher ist es auch irrelevant, ob die beiden Speicherenden gleich groß sind.

Wie Jason Gregory in seinem Buch *Game Engine Architecture* erwähnt, wird der Double-Ended Stack Allokator unter anderem in Midways Hydro Thunder verwendet. Bei dem Spiel handelt es sich um ein Rennspiel mit Raketenbooten, welches auf diversen Plattformen erschienen ist. Mit seinem festen Levelschema eignet sich Hydro Thunder optimal für die Benutzung eines Stack Allokators. Midway setzt dabei auf einen Double-Ended Stack Allokator und lädt sämtliche Level-Ressourcen am unteren Ende des Allokators. Im Gegensatz dazu werden die temporären Speicherblöcke an das obere Ende gesetzt und in jedem Frame[3] erneuert.

3.2.2. Stack-Based Resource Allocation

Der Stack Allokator kommt zum Beispiel bei der sogenannten *Stack-Based Resource Allocation* zum Einsatz. Dabei werden die Ressourcen für ein Spiel mithilfe eines solchen Allokators geladen. Auf diese Weise werden beim Spielstart als aller erstes die LSR-Daten[4] geladen, welche das gesamte Spielgeschehen über im Speicher verbleiben. An dieser Stelle wird nun ein Marker gesetzt bzw. außerhalb des Allokators gespeichert. Anschließend lädt das Programm den ersten Level des Spiels, der direkt an das Ende der LSR-Daten gesetzt wird. Wenn der Spieler nun den zweiten Level erreicht, gibt der Allokator sämtliche Ressourcen bis zum vorhin gespeicherten Marker frei und lädt an der gleichen Anfangsposition die Ressourcen für den nächsten Level.

Ein Double-Ended Stack Allokator ermöglicht es außerdem Ressourcen getrennt voneinander an die beiden Enden des reservierten Speichers zu laden. Ein wesentlicher Vorteil dabei ist, dass die Speicherblöcke beliebig zwischen den beiden Enden hin und her getauscht werden können. Diese zusätzliche Dynamik ermöglicht es zum Beispiel an das untere Ende den aktuellen Level und auf der anderen Seite bereits den komprimierten nächsten Level zu laden. Sobald der Wechsel des Levels erfolgt, werden die Ressourcen des aktuellen Levels freigegeben und der nächste Level an dessen Stelle im Allokator dekomprimiert. Dieses Szenario wird ebenfalls von Jason Gregory in *Game Engine Architecture* beschrieben und ist bei Bionic Games Inc. in dieser Weise vorgekommen. Da die Dekomprimierung in der Regel wesentlich schneller ist als das Laden von einem optischen Medium, wie es bei der aktuellen Konsolengeneration üblich ist, wird die Ladezeit zwischen den Leveln auf diese Weise umgangen. Dies ist allerdings nur bei einem streng linearen Spielverlauf möglich.

3.3. Pool Allokator

In der Spieleentwicklung, aber auch der normalen Softwareentwicklung, ist es oftmals notwendig eine beträchtliche Menge an gleichgroßen Speicherblöcken zu reservieren. Darunter fallen zum Beispiel Tiles[5], Verknüpfungen in Listen oder Matrizen. Anstatt nun für jedes Objekt eine Heap-Allokation durchzuführen, wird alternativ ein sogenannter *Pool Allokator* eingesetzt. Dieser reserviert und allokiert Speicher für eine bestimmte Anzahl von gleichgroßen Objekten. Folglich

[3] Frame = Einzelbild
[4] LSR = Load-and-stay-resident
[5] Tile = Ein einzelner Grafikblock in einem Level einer 2D-Spielwelt die nur aus gleichgroßen Blöcken besteht.

wird nur einmal Speicher vom Heap angefordert, wodurch zahlreiche Kontext-Wechsel vermieden werden. Allerdings lässt sich der Pool Allokator, wie bereits erwähnt, nur bei stets gleichgroßen Speicherallokation einsetzen.

Um die freien Speicherplätze innerhalb des Pools zu verwalten, wird eine einfach verlinkte Liste angelegt. Über diese wird Speicher mit einem Aufwand von O(1) angefordert und freigegeben. Es ist lediglich ein Zurückliefern des nächsten freien Elements notwendig bzw. wird bei der Deallokation eines Speicherblocks, dieser wieder an die Liste angehängt. Theoretisch entsteht hier ein zusätzlicher Speicherverbrauch, da die Pointer für das nächste Element in der einfach verlinkten Liste ebenfalls Speicherplatz benötigen. Dies wird jedoch dadurch verhindert, indem die freien Speicherblöcke für die Pointer verwendet werden. Das funktioniert allerdings nur solange die Speicherblöcke die Größe $>=$ *sizeof(void*)* besitzen. Sprich, sobald die einzelnen Elemente im Pool Allokator von der Speichergröße her kleiner als Pointer sind, wird ein anderer Ansatz gewählt. So werden anstelle der Pointer Indices verwendet. Aufgrund dessen wird aber die Größe des Pools eingeschränkt, da eine Indizierung nur bis zum maximal möglichen Wert des Speicherblocks möglich ist. Damit ist zum Beispiel bei einem 16-bit Integer nur eine Anzahl von 2^{16} Elementen möglich.

3.4. Single-Frame Allokator

Die Berechnung der einzelnen Frames in der Spieleentwicklung erfolgt über die sogenannte *Game Loop bzw. Main Loop*. Innerhalb dieser Schleife werden Steuerungseingaben, KI, Grafiken, Physik und vieles mehr für das nächste am Bildschirm anzuzeigende Bild und seine Objekte errechnet. Oftmals kommt es darin zu temporären Speicherallokationen bzw. -deallokationen. Da diese in so einem Fall relativ teuren Operationen für jeden Frame durchgeführt werden, wird in der Praxis an deren Stelle ein *Single-Frame Allokator* verwendet. Es handelt sich diesbezüglich um einen Stack Allokator, welcher einen bestimmten Block im Speicher reserviert. Dieser setzt sich zu Beginn der Main Loop jedes Mal zurück und kann erneut verwendet werden. Dadurch werden die Objekte im Speicher niemals freigegeben und sind zudem wesentlich perfomanter. Der Nachteil eines Single-Frame Allokators betrifft insbesondere die Disziplin des Programmierers. Er hat darauf zu achten, dass der Speicher lediglich für den aktuellen Frame verfügbar ist und der hinzugehörige Pointer nirgendwo auf Dauer referenziert wird.

3.4.1. Double-Buffered Memory Allokator

Um letzteres Problem gewissermaßen aufzugreifen, wird ein Double-Buffered Memory Allokator eingesetzt. Dieser ermöglicht es dem Programm auch noch im direkt nachfolgenden Frame auf den Speicher zuzugreifen. Dabei werden zwei unterschiedliche Singe-Frame Allokatoren abwechselnd voneinander verwendet. Dies ist insbesondere für asynchrone Operationen von großem Nutzen, da ihr Ergebnis auch im nächsten Frame noch verfügbar ist.

4. Memory Tracking

Der *Memory Tracker* ist ein Tool, welches es dem Entwicklungsteam ermöglicht, bestimmte Debug-Informationen bezüglich der Verwendung des Speichers zu verfolgen. Memory Tracking ist daher ein wesentlicher Bestandteil des Custom Memory Managements in der Spieleentwicklung. Es liefert zahlreiche Informationen wie etwa den aktuellen Speicherverbrauch, Speicherleaks oder beschädigte Speicherblöcke zurück. Dieser Schritt ist unter anderem deshalb notwendig, da es sich bei Speicher um eine begrenzte Ressource handelt.

4.1. Marktanalyse zur Größe des Speichers auf aktuellen Spieleplattformen

Insbesondere auf dem Konsolenmarkt ist die durchschnittliche Größe des Arbeitsspeichers erheblich niedriger als bei PCs. Dies belegen die folgenden Zahlen zu den Konsolen wie sie auf www.amazon.de vertrieben werden:

- Microsoft XBox360: 512 MB für Arbeits- und Grafikkartenspeicher
- Nintendo Wii: 88 MB Arbeitsspeicher
- Sony Playstation 3: 256 MB Arbeitsspeicher

Nintendo hat mit seiner Ende 2012 erschienenen WiiU-Konsole auf diesen Umstand reagiert und zwei Gigabyte Speicher verbaut, wobei Spiele lediglich die Hälfte davon für Haupt- und Grafikkartenspeicher verwenden. Der restliche Speicher wird für das Betriebssystem benötigt, welches auch während eines Spiels im Hintergrund agiert. Unter Heranziehung der Steam-Hardware-Umfrage für Windows-Anwender in Abbildung 4.1, welche im Dezember 2012 durchgeführt wurde, zeigt sich diesbezüglich eine andere Situation.

System RAM		
Less than 512 MB	0.06%	0.00%
512 Mb to 999 MB	0.68%	0.00%
1 GB	3.59%	+0.27%
2 GB	13.95%	+0.29%
3 GB	20.00%	+0.15%
4 GB	**22.78%**	-0.65%
5 GB	1.18%	+0.10%
6 GB	6.95%	-0.35%
7 GB	1.05%	+0.20%
8 GB	21.25%	-0.14%
9 GB	0.12%	0.00%
10 GB	0.25%	+0.02%
11 GB	0.04%	0.00%
12 GB and higher	8.08%	+0.12%

Abbildung 4.1.: Steam-Hardware-Statistik im Dezember 2012

In dieser Statistik von http://store.steampowered.com/ zeigt sich, dass zirka 97% der Steam-Nutzer ein PC-System mit zwei Gigabyte Arbeitsspeicher oder mehr nutzen.

Diese Analyse der Marktsituation verdeutlicht, dass gerade auf Multiplattform- und Konsolen-Titeln der sparsame und gewissenhafte Umgang mit Speicher und seine genaue Betrachtung von äußerster Notwendigkeit geprägt ist.

4.2. Das Konzept eines Memory Trackers

Der Memory Tracker ist für verschiedene Aufgabenbereiche verantwortlich.

Speicherverbrauch: Zunächst ist er für die Messung des aktuellen Speicherverbrauchs zuständig. Dies kann zum Beispiel dadurch erreicht werden, indem ein detaillierter *Memory Dump*[6] analysiert wird. Dabei wird dieser in verschiedene Kategorien unterteilt und in einem Debug-Fenster oder einer Textdatei ausgegeben. Diese Unterteilungen entsprechen zum Beispiel den verschiedenen Speicherallokatoren, den vom Betriebssystem benötigten Speicher oder den geladene Daten von der SSD, Festplatte und anderen optischen Medien. Die Messung betrifft unter Umständen nicht nur den Arbeitsspeicher, sondern auch den Grafikkartenspeicher. Der sogenannte *VRAM* oder *Video RAM*[7] ist unter anderem für die Verwaltung der Texturen und 3D-Modelle zuständig, was bei nicht ausreichendem Speicher zu fehlenden Grafiken oder schweren Grafikfehlern führt. Um diese Fehler

[6] Memory Dump = Ein zu einem bestimmten Zeitpunkt aufgezeichneter Zustand des Speichers
[7] Video RAM = Grafikspeicher

oder andere Speicherfehler nicht nur auf den Systemen der Entwickler zu erkennen, werden dem Memory Tracker bestimmte Maximalgrenzen mitgeteilt, welche den Minimalanforderungen des Spiels entsprechen. Folglich werden unter anderem Out-of-Memory-Fehler bereits bekannt, bevor eine Person das Spielprojekt auf der Originalkonsole oder einem schwächeren Rechner testet.

Speicherleaks: In diesem Zusammenhang ist das Aufspüren von Speicherleaks eine weitere Aufgabe des Memory Trackers. Dabei handelt es sich um Speicher, welcher zwar allokiert, aber vom Programm nicht wieder freigegeben wird. Auf diese Weise hat das System nach einer unbestimmten Zeit keinen Platz mehr für neue Speicheranforderungen. Dies wird dadurch erreicht, indem zu jeder Allokation zusätzlicher Speicher reserviert wird. Hier werden zusätzliche Informationen wie etwa die Codezeile in welcher der Speicher des Leaks allokiert wird, die angeforderte Speichergröße oder der dazugehörige *Stacktrace*[8] gespeichert. Unter anderem wird darin eine doppelt verkettete Liste implementiert, bei der keine zusätzlichen Allokationen notwendig sind. Auf diese Weise sind dem Memory Tracker am Ende des Programms alle nicht freigegebenen Speicherblöcke bekannt und werden mit den gesammelten Debug-Informationen ausgegeben.

Bounds Checking: Bounds Checking ist eine zusätzliche Sicherheit, mit welcher der Programmierer feststellt, ob eine Beschädigung des Speichers vorliegt. Dadurch wird unter anderem gewährleistet, dass die Grenzen eines Arrays eingehalten werden und nicht an Position [-x] oder [(n-1)+x] geschrieben wird. An den beiden Enden des allokierten Speicherblocks wird eine gewisse Anzahl an zusätzlichen Bits angefordert und deren Speicherinhalt mit einem Token, wie zum Beispiel 0xAE oder 0xAB, gefüllt. An bestimmten Stellen des Programms oder bei der Deallokation des Blocks werden diese Bits auf ihre Intaktheit geprüft. Ist dies nicht der Fall, so wurde an eine falsche Speicheradresse geschrieben und dem Programmierer wird eine entsprechende Debug-Meldung ausgegeben.

Die Erfüllung dieser Aufgaben bringt eine gewisse Komplexität mit sich, die in bestimmten Szenarien zu Problemen führt. So implementiert jeder eigene Speicherallokator die Tracking-Funktionen entsprechend seiner Aufgabe und gibt Informationen wie etwa das Maximum sowie den derzeit tatsächlich benötigten Speicher bekannt. Sofern diese Implementierung nicht von jedem Programmierer und Allokator vorgenommen wird, gehen wichtige Debug-Daten verloren, weshalb hier eine genaue Vorgehensweise vereinbart werden muss. Außerdem sind Informationen bezüglich des Grafikkartenspeichers je nach Schnittstelle schwierig zu verfolgen, indem zum Beispiel Allokationen und Deallokationen des VRAMs versteckt sind.

4.3. Betrachtung anhand eines Programmbeispiels

In diesem Code-Beispiel wird ein simpler Memory Tracker in C++ betrachtet, welcher alle dynamischen Allokationen eines Programms auffängt und entsprechend verwaltet. Folgende Features sind darin inkludiert:

[8] Stacktrace = Der aktuelle Stapel am Stack

- Die Berechnung des aktuellen Speicherverbrauchs von dynamischen Allokationen. Auf Wunsch werden alle für die Debug-Informationen notwendigen Speicherblöcke hinzugerechnet.

- Die Ausgabe aller Speicherleaks inklusive der Position im Code, der Größe und einer Identifikationsnummer.

- Bounds Checking, welches am Ende des Programms die beschädigten Grenzen meldet.

- Verwendung von Tokens wie etwa 0xAF oder 0xAD, um freigegebene oder allokierte Blöcke im Speicher zu kennzeichnen.

Das folgende Beispiel setzt sich aus zwei Bausteinen zusammen. Zum einen besteht es aus den Dateien MemoryManagement.h und MemoryManagement.cpp, welche für das allgemeine Custom Memory Management zuständig sind. Darin wird eine MemoryManager-Klasse zur Verfügung gestellt, welche sich theoretisch um die Allokationen via der eigenen Speicherallokatoren kümmert. In diesem einfach Beispiel dient sie jedoch lediglich zum Aufruf der Methode für die Berechnung des verbrauchten Speicherplatzes. Zudem befinden sich hier sämtliche Überladungen von *global new* und *delete* bzw. das sogenannte *Placement new*, das die Funktionalität von*new* erweitert. Allgemein ist *Global new* ist nicht zwangsweise zu überladen. Stattdessen werden im Rahmen des Memory Managements alle entsprechenden Operator-Aufrufe durch eigene Lösungen ersetzt.

Zum anderen aus dem Memory Tracker, welcher in MemoryTracker.h deklariert und in MemoryTracker.cpp definiert wird. Zudem wird in der Header-Datei eine MemoryObject-Struktur deklariert, welche sämtliche Debug-Informationen zu den allokierten Speicherblöcken enthält. Der Memory Tracker selbst verwaltet die doppelt verkettete Liste der MemoryObject-Objekte und bietet noch weitere Funktionalitäten die in den folgenden Unterkapiteln erläutert werden.

4.3.1. MemoryManagement.h/.cpp und die MemoryManager-Klasse

In der Header-Datei des Memory Managements wird eine Präprozessor-Variable deklariert, die das Memory Tracking aktiviert, sofern es sich um ein Debug-Build handelt. Mithilfe dieses Schemas können einzelne Features des Memory Managements aktiviert bzw. deaktiviert werden.

```
#ifdef _DEBUG
#define MEMORY_TRACKING
#endif
```

Nachfolgend werden sämtliche Überladungen für die Operatoren *Placement new*, *global new* und *delete* deklariert.

```
void *__CRTDECL operator new(size_t size) _THROW1(_STD bad_alloc);

void *__CRTDECL operator new[](size_t size) _THROW1(_STD bad_alloc);

void * operator new(size_t size, const char *file, int line);

void * operator new[](size_t size, const char *file, int line);

void operator delete(void *ptr);

void operator delete[](void *ptr);

void operator delete(void *ptr, const char *file, int line);

void operator delete(void *ptr, const char *file, int line);
```

Die anschließende MemoryManager-Klasse dient in diesem Beispiel lediglich der Aufgabe, den aktuellen Speicherverbrauch der dynamischen Allokationen zu erfassen. Folglich wird exklusiv die zugehörige Methode des Memory Trackers aufgerufen und der Rückgabewert zurückgeliefert. Zudem wird eine globale Instanz der Klasse als extern verfügbar deklariert.

```
class MemoryManager
{
public:
        long GetCurrentMemoryUsage(bool includeDebugMemory = false);
};
extern MemoryManager g_memManager;
```

Abschließend wird der Operator *new* mithilfe von *Placement new* überschrieben. Außerdem wird eine globale Integer-Variable extern deklariert, welche für das Hochzählen einer Identifikationsnummer für die MemoryObjects-Objekte zuständig ist.

```
#ifdef MEMORY_TRACKING
        #undef DEBUG_NEW
        #define DEBUG_NEW new(__FILE__, __LINE__)
        #define new DEBUG_NEW
        extern unsigned int g_allocCounter;
#endif
```

In der MemoryManagement.cpp werden die Operator-Überladungen und die MemoryManager-Klasse implementiert. Von äußerster Notwendigkeit ist hier die Auflösung der Definition für den Operator *new*, da es ansonsten zu einer mehrfachen Neudefinition kommt. Zudem werden die im Header deklarierten, globalen Variablen definiert.

```
#undef new

unsigned int g_allocCounter = 0;
MemoryManager g_memManager;
```

Die verschiedenen Operator-Überladungen sind stets sehr ähnlich und sind wie folgt implementiert worden:

```
void * operator new(size_t size, const char *file, int line)
{
#ifdef MEMORY_TRACKING
        size_t allocSize = sizeof(MemoryObject) + BOUNDS_SIZE + size +
            BOUNDS_SIZE;
        void *ptr = malloc(allocSize);
        memset(ptr, 0xAF, allocSize);
        MemoryObject* memObj = static_cast<MemoryObject*>(ptr);
        memoryTracker.NewMemory(memObj, file, line, size);
        char* boundsObj = reinterpret_cast<char*>(memObj+1);
        //Setze Grenzen für Anfang.
        memset(boundsObj, 0xAB, BOUNDS_SIZE);
        //Setze Grenzen für Ende.
        memset(boundsObj+BOUNDS_SIZE+size, 0xAE, BOUNDS_SIZE);
        //Returniere Anfangsposition des freie Speicherblocks.
        return boundsObj+BOUNDS_SIZE;
#else
        void* ptr = malloc(size);
        return ptr;
#endif
}
```

Die Überladung eines *new*-Operators allokiert zusätzlich zum angeforderten Speicher Platz für ein MemoryObject-Objekt sowie zwei 32-Bit große Blöcke für das Bounds Checking. Die BOUNDS_SIZE-Konstante enthält die zu verwendende Größe für diese Grenzblöcke und ist in diesem Beispiel auf die besagten vier Bytes gesetzt. Weiters wird der gesamte Speicherblock mit dem 0xAF-Token gefüllt, um den allokierten Speicher zu markieren. Der Pointer wird dem Memory-Tracker als neues MemoryObject-Objekt zur Verwaltung übergeben. Mithilfe der entsprechenden Typenumwandlung lässt es sich durch den Speicher iterieren und folglich die Grenzen für das Bounds Checking mit den entsprechenden Tokens füllen. Abschließend wird ein Pointer für den freien Speicher zurückgeliefert. Dies alles kommt allerdings nur zustande, sofern die entsprechende Präprozessor-Variable definiert ist. Ansonsten wird lediglich Speicher allokiert und returniert.

```
void operator delete(void *ptr)
{
#ifdef MEMORY_TRACKING
        char* bytes = reinterpret_cast<char*>(ptr);
        char* boundsObj = bytes - BOUNDS_SIZE;
        MemoryObject* memObj = reinterpret_cast<MemoryObject*>(boundsObj)
             - 1;
        memoryTracker.FreeMemory(memObj);
        memset(memObj->m_ptr, 0xAD, memObj->m_size);
        free(memObj);
#else
        free(ptr);
#endif
}
```

Der überladene *delete*-Operator geht mithilfe der Typenumwandlung ganz ähnlich vor und liefert auf diese Weise das MemoryObject-Objekt an die Freigabe-Methode des Memory Trackers. Zudem wird der Speicher als freigegeben markiert und schließlich deallokiert. Abermals trifft dieser Vorgang nur zu, falls MEMORY_TRACKING aktiv ist.

4.3.2. MemoryTracker.h/.cpp und die MemoryTracker-Klasse

Die Header-Datei des Memory Trackers beinhaltet unter anderem eine konstante Variable für die zu verwendende Größe der Grenzen beim Bounds Checking. Das ermöglicht es dem Programmierer jederzeit eine Anpassung diesbezüglich vorzunehmen.

```
//Konstante für die Größe der Bounds für Bounds Checking in Bits.
const size_t BOUNDS_SIZE = 4;

//Beinhaltet die Daten über allokierten Speicher.
struct MemoryObject
{
        void* m_ptr;
        unsigned int m_allocId;
        const char* m_file;
        unsigned int m_line;
        unsigned int m_size;
        MemoryObject* mObjPrev;
        MemoryObject* mObjNext;
};
```

Weiters wird die Struktur für ein MemoryObjekt deklariert, welches sämtliche Debug-Informationen bei der Allokation des Speicherblocks sammelt.

In der MemoryTracker-Klasse befinden sich alle notwendigen Member für die Verwaltung der doppelt verketteten Liste für MemoryObject-Objekte, das Bounds Checking, die Ermittlung des Speicherverbrauchs sowie der Anzahl der gefunden Speicherleaks. Wie nachfolgend in der Deklaration der Klasse zu sehen ist, wird bei der doppelt verketteten Liste mittels Head und Tail auf sogenannte *Sentinels* zurückgegriffen. Diese ermöglichen das Hinzufügen und Entfernen von Elementen mit einem Aufwand von O(1).

```
//Memory Tracker
class MemoryTracker
{
public:
        MemoryTracker();

        ~MemoryTracker();

        void NewMemory(MemoryObject* memObj, const char* file, unsigned
            int line, unsigned int size);
```

```
        void FreeMemory(MemoryObject* memObj);

        double GetCurrentMemoryUsage(bool includeDebugMemory = false);
private:
        int m_leaks;

        MemoryObject* m_memObjRoot;

        MemoryObject m_memObjHead;

        MemoryObject m_memObjTail;
};
extern MemoryTracker memoryTracker;
```

Am Ende der Datei wird abschließend ein externer Verweis auf eine globale MemoryTracker-Instanz gesetzt.

Wie es die MemoryTracker.cpp zeigt, wird im Konstruktor des Memory Trackers die MemoryObject-Liste initialisiert. Im Konstruktor werden hingegen alle verbleibenden MemoryObject-Objekte mitsamt all ihren Informationen in der Konsole ausgegeben.

```
MemoryTracker::MemoryTracker()
{
        //Initialisierung der MemoryObject-Liste.
        m_memObjHead.mObjNext = &m_memObjTail;
        m_memObjHead.mObjPrev = &m_memObjTail;
        m_memObjTail.mObjNext = &m_memObjHead;
        m_memObjTail.mObjPrev = &m_memObjHead;
        m_memObjRoot = &m_memObjHead;
}

MemoryTracker::~MemoryTracker()
{
        cout << "Memory leaks: " << m_leaks << endl;
        for(MemoryObject* thisMemObj = m_memObjHead.mObjNext; thisMemObj
            != &m_memObjTail; thisMemObj = thisMemObj->mObjNext)
        {
                //Gibt Leaks mit Debug-Informationen aus
        }
}
```

Besonderes Augenmerk ist auf die beiden Methoden *NewMemory* und *FreeMemory* zu legen. Es handelt sich hierbei, um die eigentliche Verwaltung des Memory Trackers, welche bei allen Allokationen und Deallokationen im Einsatz ist.

```
void MemoryTracker::NewMemory(MemoryObject* memObj, const char* file,
    unsigned int line, unsigned int size)
{
        //Legt das MemoryObject in der Liste ab
        memObj->m_ptr = memObj+1;
        memObj->m_file = file;
        memObj->m_line = line;
        memObj->m_size = size;
        memObj->m_allocId = g_allocCounter;
        g_allocCounter++;

        MemoryObject* temp = m_memObjRoot->mObjNext;
        m_memObjRoot->mObjNext = memObj;
        memObj->mObjPrev = m_memObjRoot;
        memObj->mObjNext = temp;
        temp->mObjPrev = memObj;

        //Erhöht den Speicherleak-Zähler um eins.
        m_leaks++;
}
```

Der anschließende Code-Ausschnitt ist für die Entfernung des MemoryObject-Objekts aus der entsprechenden Liste zuständig. Zudem werden die Grenzen des Speicherblocks auf ihre Intaktheit geprüft.

```
void MemoryTracker::FreeMemory(MemoryObject* memObj)
{
        //Entfernt den Eintrag aus der Liste
        memObj->mObjNext->mObjPrev = memObj->mObjPrev;
        memObj->mObjPrev->mObjNext = memObj->mObjNext;
        m_leaks--;

        //Prüft die Grenze am Anfang des Speicherblocks.
        char* boundingBytes = reinterpret_cast<char*>(memObj+1);
        for(unsigned int i = 0; i < BOUNDS_SIZE; ++i)
        {
                if(boundingBytes[i] != static_cast<char>(0xAB))
                        //Error
        }

        //Prüft die Grenze am Ende des Speicherblocks.
        boundingBytes += BOUNDS_SIZE + memObj->m_size;
        for(unsigned int i = 0; i < BOUNDS_SIZE; ++i)
        {
                if(boundingBytes[i] != static_cast<char>(0xAE))
                        //Error
        }
}
```

Bei der letzten Methoden-Definition handelt es sich um die Berechnung des verwendeten Speicherplatzes. In diesem Schritt wird die aktuelle MemoryObject-Liste durchgegangen und die Speichergröße der einzelnen Speicherblöcke addiert und zurückgeliefert. Mithilfe des Boolean-Arguments wird festgelegt, ob der zusätzliche Speicherplatz für das Memory Tracking miteinbezogen werden.

```
double MemoryTracker::GetCurrentMemoryUsage(bool includeDebugMemory)
{
        double usage = 0;
        for(MemoryObject* thisMemObj = m_memObjHead.mObjNext; thisMemObj
            != &m_memObjTail; thisMemObj = thisMemObj->mObjNext)
        {
                if(includeDebugMemory)
                {
                        usage += thisMemObj->m_size + BOUNDS_SIZE*2 +
                            sizeof(MemoryObject);
                }
                else
                {
                        usage += thisMemObj->m_size;
                }
        }
        return usage;
}
```

Abschließend wird am Ende der Datei eine Instanz der MemoryTracker-Klasse angelegt. Um den Präprozessor anzuweisen, dass eine globale Instanz als erstes auf dem Stack des Programms initialisiert wird, wird auf Präprozessor-Anweisungen zurückgegriffen. Zu diesen zählt zum Beispiel *#pragma init_seg(compiler)* bei Microsoft. Dadurch werden sämtliche Allokationen und Deallokationen dem Memory Tracking mitgeteilt.

```
//Ein globales MemoryTracker-Objekt.
#pragma init_seg(compiler)
MemoryTracker memoryTracker;
```

5. Zusammenfassung

Mit Custom Memory Management werden dem Entwickler zahlreiche Methoden zur Verfügung gestellt, um für einen sparsamen und nachvollziehbaren Umgang im Bezug auf die Speicherverwaltung zu sorgen.

Das korrekte Verständnis über Stack, Heap und *Global Memory* ist besonders beim begrenzten Speichervorkommen auf Konsolen notwendig. Memory Alignment ist im Bezug auf die Spielkonsolen, aber auch PCs für den effizienten Einsatz von Speicherallokatoren von essentieller Wichtigkeit. Zudem ermöglicht eine Analyse des Stacks dem Entwickler die Applikation auf Schritt und Tritt zu verfolgen.

Eigene Speicherallokatoren sorgen für eine effektive und effiziente Speicherverwaltung in der Spieleentwicklung. Der Heap Allokator ist zwar vielseitig einsetzbar, bringt jedoch den Nachteil der Speicherfragmentierung mit sich. Aus diesem Grund wird unter anderem auf Stack und Pool Allokatoren zurückgegriffen, welche für die Verwaltung von Levelressourcen bzw. für eine große Anzahl von gleichen Objekten eingesetzt werden. Durch den Single-Frame und Double-Buffered Memory Allokator lassen sich Ressourcen innerhalb von einem oder zwei Frames effizienter verwalten, ohne auf dynamische Allokationen mittels *new, delete, malloc* oder *free* angewiesen zu sein.

Ein Memory Tracker sorgt für die Kontrolle der Speicherabläufe und unterstützt den Entwickler bei der Suche nach Speicherleaks oder beschädigtem Speicher. Deshalb wird hierfür zusätzlicher Speicherplatz für diverse Debug-Informationen angefordert und in einer doppelt verketteten Liste verwaltet. Dies ist besonders aufgrund des äußerst begrenzten Arbeitsspeichers auf Konsolen notwendig, wie es in der Marktanalyse in Kapitel 4.1 erläutert wird.

Literaturverzeichnis

[1] Jason Gregory, *Game Engine Architecture*, Taylor & Francis Ltd., Wellesley, Massachusetts, 2009.

[2] Mike McShaffry, *Game Coding Complete*, 3rd ed. Course Technology, Boston, 2009.

[3] David H. Eberly, *3D Game Engine Design. A Practical Approach to Real-Time Computer Graphics*, 2nd ed. Morgan Kaufmann, San Francisco, 2007.

[4] Alan Thorn, *Game Engine Design and Implementation: Foundations of Game Development*, Jones & Bartlett Pub, Massachusetts, 2011.

Abbildungsverzeichnis

2.1. Memory Alignment 4
2.2. Sparsames Memory Alignment 5

3.1. Speicherfragmentierung 7

4.1. Steam-Hardware-Statistik 12-2012 12

A. Komplettes Codebeispiel aus dem Kapitel “Memory Tracking”

```
/* MemoryManagement.h */
#pragma once

#ifdef _DEBUG
#define MEMORY_TRACKING
#endif

#include <new>
#include <crtdbg.h>
//Überschreibt global new
void *__CRTDECL operator new(size_t size) _THROW1(_STD bad_alloc);

//Überschreibt global new[]
void *__CRTDECL operator new[](size_t size) _THROW1(_STD bad_alloc);

//Überladung des Placement-news. Allokiert angeforderten Speicher
//plus MemoryObject.
//Gibt dem MemoryTracker Bescheid, dass neuer Speicher angefordert wurde.
void * operator new(size_t size, const char *file, int line);

//Überladung des Placement-new[]s für ein Array. Allokiert angeforderten
//Speicher plus MemoryObject.
//Gibt dem MemoryTracker Bescheid, dass neuer Speicher angefordert wurde.
void * operator new[](size_t size, const char *file, int line);

//Überladung des deletes. Gibt Speicher des übergegebenen Pointers und
//sein MemoryObject frei.
//Gibt dem MemoryTracker Bescheid, dass Speicher freigegeben wurde.
//Setzt den allokierten Speicher auf 0xAF Token.
void operator delete(void *ptr);

//Überladung des delete[]s für ein Array. Gibt Speicher des
//übergegebenen Pointers und sein MemoryObject frei.
//Gibt dem MemoryTracker Bescheid, dass Speicher freigegeben wurde.
//Setzt den allokierten Speicher auf 0xAF Token.
void operator delete[](void *ptr);

//Überladung des deletes für placement new. Gibt Speicher des
//übergegebenen Pointers und sein MemoryObject frei.
//Gibt dem MemoryTracker Bescheid, dass Speicher freigegeben wurde.
//Setzt den allokierten Speicher auf 0xAF Token.
```

```
void operator delete(void *ptr, const char *file, int line);

//Überladung des delete[]s für placement new. Gibt Speicher des
//übergegebenen Pointers und sein MemoryObject frei.
//Gibt dem MemoryTracker Bescheid, dass Speicher freigegeben wurde.
//Setzt den allokierten Speicher auf 0xAF Token.
void operator delete(void *ptr, const char *file, int line);

class MemoryManager
{
public:
        //HeapAlloc(), StackAlloc(), PoolAlloc()...

        //Liefert den Wert des derzeitigen Speicherverbrauchs
        //mitsamt allen Allokatoren zurück.
        //includeDebugMemory: Gibt an, ob der Speicherplatz der
        //Debug-Informationen wie etwa Bounds Checking und
        //Memory Objects des Memory Trackers miteinberechnet werden soll.
        double GetCurrentMemoryUsage(bool includeDebugMemory = false);
};
extern MemoryManager g_memManager;
```

```
/* MemoryManagement.cpp */
#include "MemoryManagement.h"
#include "MemoryTracker.h"
#include <Windows.h>

#undef new

unsigned int g_allocCounter = 0;
MemoryManager g_memManager;

//Überschreibt global new
void *__CRTDECL operator new(size_t size) _THROW1(_STD bad_alloc)
{       // try to allocate size bytes
#ifdef MEMORY_TRACKING
        size_t allocSize = sizeof(MemoryObject) + BOUNDS_SIZE + size +
            BOUNDS_SIZE;
        void *p = malloc(allocSize);
        memset(p, 0xAF, allocSize);
        if(p == 0)
        {
                cerr << "Out␣of␣Memory" << endl;
                static const std::bad_alloc nomem;
                _RAISE(nomem);
                exit(0);
        }
        MemoryObject* memObj = static_cast<MemoryObject*>(p);
        memoryTracker.NewMemory(memObj, "unknown␣file", 0, size);
        char* boundsObj = reinterpret_cast<char*>(memObj+1);
        memset(boundsObj, 0xAB, BOUNDS_SIZE);
        memset(boundsObj+BOUNDS_SIZE+size, 0xAE, BOUNDS_SIZE);
```

```
        return boundsObj+BOUNDS_SIZE;
#else
        void *p = malloc(size);
        if(p == 0)
        {
                cerr << "Out␣of␣Memory" << endl;
                static const std::bad_alloc nomem;
                _RAISE(nomem);
                exit(0);
        }
        return p;
#endif
}

//Überschreibt global new[]
void *__CRTDECL operator new[](size_t size) _THROW1(_STD bad_alloc)
{       // try to allocate size bytes
#ifdef MEMORY_TRACKING
        size_t allocSize = sizeof(MemoryObject) + BOUNDS_SIZE + size +
            BOUNDS_SIZE;
        void *p = malloc(allocSize);
        memset(p, 0xAF, allocSize);
        if(p == 0)
        {
                cerr << "Out␣of␣Memory" << endl;
                static const std::bad_alloc nomem;
                _RAISE(nomem);
                exit(0);
        }
        MemoryObject* memObj = static_cast<MemoryObject*>(p);
        memoryTracker.NewMemory(memObj, "unknown␣file", 0, size);
        char* boundsObj = reinterpret_cast<char*>(memObj+1);
        memset(boundsObj, 0xAB, BOUNDS_SIZE);
        memset(boundsObj+BOUNDS_SIZE+size, 0xAE, BOUNDS_SIZE);
        return boundsObj+BOUNDS_SIZE;
#else
        void *p = malloc(size);
        if(p == 0)
        {
                cerr << "Out␣of␣Memory" << endl;
                static const std::bad_alloc nomem;
                _RAISE(nomem);
                exit(0);
        }
        return p;
#endif
}

//Überladung des Placement-news. Allokiert angeforderten Speicher plus
    MemoryObject.
//Gibts dem MemoryTracker Bescheid, dass neuer Speicher angefordert wurde
    .
void * operator new(size_t size, const char *file, int line)
```

```
{
#ifdef MEMORY_TRACKING
        size_t allocSize = sizeof(MemoryObject) + BOUNDS_SIZE + size +
            BOUNDS_SIZE;
        void *ptr = malloc(allocSize);
        memset(ptr, 0xAF, allocSize);
        memset(ptr, 0xAF, sizeof(MemoryObject) + size);
        MemoryObject* memObj = static_cast<MemoryObject*>(ptr);
        memoryTracker.NewMemory(memObj, file, line, size);
        char* boundsObj = reinterpret_cast<char*>(memObj+1);
        memset(boundsObj, 0xAB, BOUNDS_SIZE);
        memset(boundsObj+BOUNDS_SIZE+size, 0xAE, BOUNDS_SIZE);
        return boundsObj+BOUNDS_SIZE;
#else
        void* ptr = malloc(size);
        return ptr;
#endif
}

//Überladung des Placement-new[]s für ein Array. Allokiert angeforderten
    Speicher plus MemoryObject.
//Gibt dem MemoryTracker Bescheid, dass neuer Speicher angefordert wurde.
void * operator new[](size_t size, const char *file, int line)
{
#ifdef MEMORY_TRACKING
        size_t allocSize = sizeof(MemoryObject) + BOUNDS_SIZE + size +
            BOUNDS_SIZE;
        void *ptr = malloc(allocSize);
        memset(ptr, 0xAF, allocSize);
        MemoryObject* memObj = static_cast<MemoryObject*>(ptr);
        memoryTracker.NewMemory(memObj, file, line, size);
        char* boundsObj = reinterpret_cast<char*>(memObj+1);
        memset(boundsObj, 0xAB, BOUNDS_SIZE);
        memset(boundsObj+BOUNDS_SIZE+size, 0xAE, BOUNDS_SIZE);
        return boundsObj+BOUNDS_SIZE;
#else
        void* ptr = malloc(size);
        return ptr;
#endif
}

//Überladung des deletes. Gibt Speicher an des übergegebenen Pointers und
     sein MemoryObject frei.
//Gibt dem MemoryTracker Bescheid, dass Speicher freigegeben wurde.
void operator delete(void *ptr)
{
#ifdef MEMORY_TRACKING
        char* bytes = reinterpret_cast<char*>(ptr);
        char* boundsObj = bytes - BOUNDS_SIZE;
        MemoryObject* memObj = reinterpret_cast<MemoryObject*>(boundsObj)
             - 1;
        memoryTracker.FreeMemory(memObj);
        memset(memObj->m_ptr, 0xAD, memObj->m_size);
```

```
        free(memObj);
#else
        free(ptr);
#endif
}

//Überladung des delete[]s für ein Array. Gibt Speicher an des ü
    bergegebenen Pointers und sein MemoryObject frei.
//Gibt dem MemoryTracker Bescheid, dass Speicher freigegeben wurde.
void operator delete[](void *ptr)
{
#ifdef MEMORY_TRACKING
        char* bytes = reinterpret_cast<char*>(ptr);
        char* boundsObj = bytes - BOUNDS_SIZE;
        MemoryObject* memObj = reinterpret_cast<MemoryObject*>(boundsObj)
            - 1;
        memoryTracker.FreeMemory(memObj);
        memset(memObj->m_ptr, 0xAD, memObj->m_size);
        free(memObj);
#else
        free(ptr);
#endif
}

//Überladung des deletes für Placement new, um im bei Fehlern Speicher
    fregeben zu können.
//Gibt Speicher an des übergegebenen Pointers und sein MemoryObject frei.
//Gibt dem MemoryTracker Bescheid, dass Speicher freigegeben wurde.
void operator delete(void *ptr, const char *file, int line)
{
#ifdef MEMORY_TRACKING
        char* bytes = reinterpret_cast<char*>(ptr);
        char* boundsObj = bytes - BOUNDS_SIZE;
        MemoryObject* memObj = reinterpret_cast<MemoryObject*>(boundsObj)
            - 1;
        memoryTracker.FreeMemory(memObj);
        memset(memObj->m_ptr, 0xAD, memObj->m_size);
        free(memObj);
#else
        free(ptr);
#endif
}

//Überladung des delete[]s für Placement new, um im bei Fehlern Speicher
    fregeben zu können.
//Gibt Speicher an des übergegebenen Pointers und sein MemoryObject frei.
//Gibt dem MemoryTracker Bescheid, dass Speicher freigegeben wurde.
void operator delete[](void *ptr, const char *file, int line)
{
#ifdef MEMORY_TRACKING
        char* bytes = reinterpret_cast<char*>(ptr);
        char* boundsObj = bytes - BOUNDS_SIZE;
        MemoryObject* memObj = reinterpret_cast<MemoryObject*>(boundsObj)
```

```
            - 1;
        memoryTracker.FreeMemory(memObj);
        memset(memObj->m_ptr, 0xAD, memObj->m_size);
        free(memObj);
#else
        free(ptr);
#endif
}

//Liefert den Wert des derzeitigen Speicherverbrauchs mitsamt allen
    Allokatoren zurück.
//includeDebugMemory: Gibt an, ob der Speicherplatz der Debug-
    Informationen
//wie etwa Bounds Checking und Memory Objects des Memory Trackers
    miteinberechnet werden soll.
double MemoryManager::GetCurrentMemoryUsage(bool includeDebugMemory)
{
        double usage = 0;
        //GetHeapAllocUsage(), GetStackAllocUsage(), GetPoolAllocUsage()
            ...

#ifdef MEMORY_TRACKING
        usage += memoryTracker.GetCurrentMemoryUsage(includeDebugMemory);
#endif

        return usage;
}
```

```
/* MemoryTracker.h */
#pragma once
#include <iostream>

using namespace std;

//Konstante für die Größe der Bounds für Bounds Checking in Bits.
const size_t BOUNDS_SIZE = 4;

//Beinhaltet Daten über allokierten Speicher.
struct MemoryObject
{
        void* m_ptr;
        unsigned int m_allocId;
        const char* m_file;
        unsigned int m_line;
        unsigned int m_size;
        MemoryObject* mObjPrev;
        MemoryObject* mObjNext;
};

//Memory Tracker
class MemoryTracker
{
```

```
public:
        //Konstruktor. Initialisiert MemoryObject-Liste.
        MemoryTracker();
        //Destruktor. Deaktiviert den MemoryTracker und
        //gibt gegebenenfalls Speicherleaks aus.
        ~MemoryTracker();

        //Signalisiert dem Memory Tracker, dass neuer Speicher inklusive
        //MemoryObject allokiert wurde.
        //Legt das MemoryObject in der Liste ab und erhöht den
        //Speicherleak-Zähler um eins.
        void NewMemory(MemoryObject* memObj, const char* file, unsigned
            int line, unsigned int size);
        //Signalisiert dem Memory Tracker, dass Speicher inklusive
        //MemoryObject freigegeben wurde.
        //Entfernt das zugehörige MemoryObject aus der Liste und senkt
        //den Speicherleak-Zähler um eins.
        //Checkt ob die Grenzen (Bounds) noch intakt sind.
        void FreeMemory(MemoryObject* memObj);
        //Liefert den Wert des derzeitigen Speicherverbrauchs zurück.
        //includeDebugMemory: Gibt an, ob der Speicherplatz der
        // Debug-Informationen wie etwa Bounds Checking und
        //Memory Objects miteinberechnet werden soll.
        double GetCurrentMemoryUsage(bool includeDebugMemory = false);
private:
        //Die Anzahl der aktuellen Leaks
        int m_leaks;
        //Root-Element einer doppelt verketteten Liste von MemoryObjects.
        MemoryObject* m_memObjRoot;
        //Sentinels für MemoryObject-Liste
        MemoryObject m_memObjHead;
        MemoryObject m_memObjTail;
};
extern MemoryTracker memoryTracker;
```

```
/* MemoryTracker.cpp */
#include "MemoryTracker.h"
#include <new>
#include <Windows.h>
#include "MemoryManagement.h"

//Konstruktor. Initialisiert MemoryObject-Liste.
MemoryTracker::MemoryTracker()
{
        m_memObjHead.mObjNext = &m_memObjTail;
        m_memObjHead.mObjPrev = &m_memObjTail;
        m_memObjTail.mObjNext = &m_memObjHead;
        m_memObjTail.mObjPrev = &m_memObjHead;
        m_memObjRoot = &m_memObjHead;
}

//Destruktor. Deaktiviert den MemoryTracker und
```

```
//gibt gegebenenfalls Speicherleaks aus.
MemoryTracker::~MemoryTracker()
{
        cout << "Memory␣leaks:␣" << m_leaks << endl;
        for(MemoryObject* thisMemObj = m_memObjHead.mObjNext; thisMemObj
            != &m_memObjTail; thisMemObj = thisMemObj->mObjNext)
        {
                cout << "Leak␣at␣" << thisMemObj->m_ptr << "␣with␣id␣" <<
                    thisMemObj->m_allocId << "␣in␣" << thisMemObj->
                    m_file << "␣at␣line␣" << thisMemObj->m_line << ",␣
                    size:␣" << thisMemObj->m_size << "␣byte(s)" << endl;
        }
}

//Signalisiert dem Memory Tracker, dass neuer Speicher inklusive
    MemoryObject allokiert wurde.
//Legt das MemoryObject in der Liste ab und erhöht den Speicherleak-Zä
    hler um eins.
void MemoryTracker::NewMemory(MemoryObject* memObj, const char* file,
    unsigned int line, unsigned int size)
{
        memObj->m_ptr = memObj+1;
        memObj->m_file = file;
        memObj->m_line = line;
        memObj->m_size = size;
        memObj->m_allocId = g_allocCounter;
        g_allocCounter++;

        MemoryObject* temp = m_memObjRoot->mObjNext;
        m_memObjRoot->mObjNext = memObj;
        memObj->mObjPrev = m_memObjRoot;
        memObj->mObjNext = temp;
        temp->mObjPrev = memObj;

        //cout << "Memory allocated at " << memObj->m_ptr << endl;
        m_leaks++;
}

//Signalisiert dem Memory Tracker, dass Speicher inklusive MemoryObject
    freigegeben wurde.
//Entfernt das zugehörige MemoryObject aus der Liste und senkt den
    Speicherleak-Zähler um eins.
//Checkt ob die Bounds noch intakt sind.
void MemoryTracker::FreeMemory(MemoryObject* memObj)
{
        memObj->mObjNext->mObjPrev = memObj->mObjPrev;
        memObj->mObjPrev->mObjNext = memObj->mObjNext;

        m_leaks--;

        char* boundingBytes = reinterpret_cast<char*>(memObj+1);
        for(unsigned int i = 0; i < BOUNDS_SIZE; ++i)
        {
```

```
                if(boundingBytes[i] != static_cast<char>(0xAB))
                {
                        cerr << "Starting␣bounds␣damaged␣from␣ptr␣" <<
                            memObj->m_ptr << "␣in␣" << memObj->m_file <<
                            "␣at␣line␣" << memObj->m_line << ",␣size:␣"
                            << memObj->m_size << "␣byte(s)" << endl;
                        //exit(ERROR);
                }
        }
        boundingBytes += BOUNDS_SIZE + memObj->m_size;
        for(unsigned int i = 0; i < BOUNDS_SIZE; ++i)
        {
                if(boundingBytes[i] != static_cast<char>(0xAE))
                {
                        cerr << "Ending␣bounds␣damaged␣from␣ptr␣" <<
                            memObj->m_ptr << "␣in␣" << memObj->m_file <<
                            "␣at␣line␣" << memObj->m_line << ",␣size:␣"
                            << memObj->m_size << "␣byte(s)" << endl;
                        //exit(ERROR);
                }
        }
}

//Liefert den Wert des derzeitigen Speicherverbrauchs zurück.
//includeDebugMemory: Gibt an, ob der Speicherplatz der Debug-
    Informationen
//wie etwa Bounds Checking und Memory Objects miteinberechnet werden soll
    .
double MemoryTracker::GetCurrentMemoryUsage(bool includeDebugMemory)
{
        double usage = 0;
        for(MemoryObject* thisMemObj = m_memObjHead.mObjNext; thisMemObj
            != &m_memObjTail; thisMemObj = thisMemObj->mObjNext)
        {
                if(includeDebugMemory)
                {
                        usage += thisMemObj->m_size + BOUNDS_SIZE*2 +
                            sizeof(MemoryObject);
                }
                else
                {
                        usage += thisMemObj->m_size;
                }
        }
        return usage;
}

//Ein globales MemoryTracker-Objekt.
#pragma init_seg(compiler)
MemoryTracker memoryTracker;
```