Möglichkeiten zur Performance-Optimierung und Vermeidung von Einbußen bei Spielen auf Android

Matthäus J. Baur

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2012

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 25. Juni 2012

Matthäus J. Baur

Inhaltsverzeichnis

Erklärung **iii**

Vorwort **vi**

Kurzfassung **vii**

Abstract **viii**

1 Einleitung **1**
1.1 Motivation 1
1.2 Problemstellung 2
1.3 Aufbau der Arbeit 2

2 Aktueller Stand **3**
2.1 Android 3
2.2 Dalvik Virtual Machine 4
2.3 JIT Compiler 5
2.4 Garbage Collection 7

3 Werkzeuge und Design Methoden **8**
3.1 Werkzeuge 8
3.1.1 TraceView 8
3.1.2 TimingLogger 10
3.1.3 Allocation Tracker 11
3.1.4 Memory Analyzing Tool 11
3.2 Design Methoden 12
3.2.1 Objekte 12
3.2.2 Static und Final 12
3.2.3 Fließkommazahlen 13
3.2.4 Schleifen 13
3.2.5 Container 13
3.2.6 Object Pooling 18

4 Performance Tests **19**

4.1 Testverlauf 19
4.2 Testergebnisse 20
4.2.1 Container 20
4.2.2 Listen 20
4.2.3 Maps 32
4.2.4 Sets 42
4.2.5 Typübergreifender Vergleich 48
4.2.6 Schleifen 50
4.2.7 Fließkommazahlen 54
4.2.8 Getter und Setter 55
4.2.9 Static und Final 57
4.2.10 Object Pool 58
4.3 Übertragung auf Spiel 58
4.3.1 Entitäten 58
4.3.2 Events 61
4.3.3 Services 62
4.4 Einfluss auf die Engine 63
4.4.1 Simulation ohne Optimierung 63
4.4.2 Simulation mit Optimierung 64
4.4.3 Ergebnis 77

5 Ausblick **79**

A Inhalt der CD-ROM **81**
A.1 Masterarbeit 81
A.2 Literatur 81
A.3 Performancetest-Daten 81
A.4 Source-Code 82

Quellenverzeichnis **83**
Literatur 83
Online-Quellen 83

Vorwort

Diese Masterarbeit entstand im Laufe des vierten Semesters im Masterstudiengang Interactive Media mit dem Schwerpunkt Game Development an der Fachhochschule Oberösterreich am Campus Hagenberg. Diese Arbeit gilt als Abschluss meines Masterstudiums, welches auf dem an der Hochschule Augsburg absolvierten Bachelor Abschluss aufbaut und ich in Hagenberg besuchen durfte.

Bedanken möchte ich mich an dieser Stelle an den Personen, die mich während meines Studiums und beim schreiben dieser Arbeit unterstützt haben. Dazu zählt unter anderem meine Familie mit meinen Eltern, als auch meine Schwester Manuela mit ihrem Freund Michael, die mich immer wieder motiviert und aufgebaut haben. Speziell danken möchte ich auch Markus, meinem Mitbewohner im Studentenheim, der mich im laufe des Studiums immer wieder zu neuen Ideen und Lösungsansätzen angeregt hat.

Und nicht zuletzt möchte ich auch meinem Betreuer Roman Divotkey danken, der mich bei der Umsetzung des Projektes und der Ausarbeitung dieser Arbeit hilfreich zur Seite gestanden hat.

Kurzfassung

Die vorliegende Arbeit setzt sich mit dem Thema der möglichen Performance-Optimierungen von Spielen für die Android Plattform mit Java auseinander.

Zunächst wird der aktuelle Stand von Android betrachtet, wobei hier speziell die technische Seite mit den Themen *Dalvik Virtual Machine* und *Garbage Collection*, sowie des *JIT-Compilers* genauer betrachtet werden. Darauf folgt die Beschreibung verschiedener Werkzeuge wie *TraceView* und *Allocation Tracker*, mit denen die Leistung einer Android Applikation und damit auch einem Spiel gemessen werden kann. Schließlich werden bestimmte *Design Methoden* und *Programmiertechniken* vorgestellt, die zur Performance-Optimierung genutzt werden können.

Im Hauptteil der Arbeit werden diese Methoden und Techniken schließlich mit verschiedenen Geräten getestet und auf ihre Wirksamkeit überprüft. Die Ergebnisse dieser Tests fließen schrittweise in den Versuch der Optimierung der bestehenden Game Engine Cogaen ein. Dabei werden bestimmte Funktionalitäten der Engine mithilfe einer Simulation getestet und ausgewertet. Zu diesen Funktionen gehören das *Entitäten-*, *Event-* und *Service-System* der Engine. Zur Auswertung gehört sowohl die Dokumentation der erreichten und nicht erreichten Erfolge der Optimierung, als auch ob die Nutzung in einer bestehenden Engine sinnvoll ist.

Abstract

The present work describes possible performance optimizations for games in java on android.

First a general explanation of the *state-of-the-art* of android especially the technical view will be shown. The topics are the *dalvik virtual machine*, *Garbage Collection* and the *JIT-Compiler*. Furthermore this work describes some different tools like *TraceView* and *AllocationTracker* which both help to measure the performance of an application. Furthermore an introduction of different *design methods* and *programming technics* follows.

The main parts of the thesis are tests of this methods and technics with different devices to show the effectivity of them. Results are used to optimize the game engine cogaen for its use on android devices. The tests include different functionalities of the engine such as *entity-*, *event-* and *service-system* in an game simulation. The results of this test shows the possible success of the optimization and if they are useful for combination to an existing system.

Kapitel 1

Einleitung

In diesem Kapitel folgt die Motivation zu der Arbeit, die eigentliche Problemstellung zu dem Thema „Möglichkeiten der Performance-Optimierung und Vermeidung von Einbußen bei Spielen auf Android“ und schließlich die Beschreibung des Aufbaus dieser Arbeit.

1.1 Motivation

Die Verbreitung von Geräten mit Android wächst und damit auch der Markt von Anwendungen die für die Android Plattform entwickelt werden (siehe Abschnitt 2.1). Durch die besonderen Interaktionsmöglichkeiten wie Touchscreen und verschiedener Sensoren, als auch der Mobilität des Geräts selbst, kann die Spielerfahrung durch *Smartphones* und *Tablets* auf eine neue Ebene erweitert werden.

Ein Grund für die hohe Beliebtheit von Android bei vielen Geräteherstellern ist vor allem die Kostengünstige Verwendung des Systems. Das Betriebssystem ist eine *freie Software* die unter der *Apache-Lizenz* steht [1]. Da bereits viele Hersteller von Mobilgeräten auf Android als Betriebssystem zurückgreifen und es seitens der Entwickler kaum Vorgaben für die Nutzung gibt, folgt daraus, dass der Anteil von unterschiedlichen Geräten mit deutlich unterschiedlichen Hardware Spezifikationen steigt. So gibt es Geräte mit Prozessoren von 528 Mhz und Arbeitsspeicher von 192 MB (HTC Dream (G1) [11]) bis hin zu Dualcore Prozessoren mit je 1,2 Ghz und einem Arbeitsspeicher von 1 GB (Samsung Galaxy Nexus [12]). Die Leistungsfähigkeit der Geräte kann sich also sehr stark voneinander unterscheiden. Dennoch sollten Anwendungen auf möglichst vielen, so wohl leistungsfähigen als auch leistungsschwachen Geräten, vergleichbar gut und flüssig laufen um dem Spieler eine bestmögliche Spielerfahrung zu liefern. Besonders für

[1] Ausnahme ist hier der Systemkern, der unter *GPL 2* steht und der Quelltext der *Tablet* Version 3.x (Honeycomb), welche erst im Zuge der Veröffentlichung der Version 4 (Ice Cream Sandwich) veröffentlicht wurde.

Spieleentwickler ist dies eine große Herausforderung, da Spiele besonders viel Leistung von einem Gerät abfordern können.

1.2 Problemstellung

Es stellt sich nun aber die Frage, welche Möglichkeiten bietet Android um die Herausforderung zu bewältigen, trotz leistungsschwacher Geräte annehmbare Performance zu erreichen und die begrenzten Ressourcen sparsam einzusetzen. *Google* bietet hier bereits eine Reihe von Design Methoden an, auf die Entwickler beim Erstellen einer Anwendung achten sollten [13]. Die Frage die sich dabei aber stellt ist, wie viel Laufzeit, Speicher oder Strom lässt sich durch diese Methoden einsparen, welche Kompromisse müssen eingegangen werden um eines dieser Ressourcen einsparen zu können und wie schwierig ist eine Umsetzung. Speziell in Bezug auf Umsetzbarkeit und Wartbarkeit der Anwendung bei bestehenden Projekten.

1.3 Aufbau der Arbeit

In der vorliegenden Arbeit wird zunächst der aktuelle Stand von Android betrachtet. Dabei wird allgemein das System in Augenschein genommen und erklärt was Android eigentlich ist und wie es sich bis heute entwickelt hat. Daraufhin folgt die Betrachtung der technischen Seite mit den Themen *Dalvik Virtual Machine*, *Garbage Collection* und *JIT-Compiler*.

Es folgt eine Vorstellung von verschiedenen Werkzeugen die das Android SDK mitliefert und beim Identifizieren von Performanceproblemen und sogenannten *Flaschenhälsen* hilfreiche Unterstützung liefern. Schließlich werden verschiedene *Design Methoden* und *Programmiertechniken* vorgestellt, auf die bei der Entwicklung von performancekritischen Anwendung geachtet werden sollte.

In wie weit diese Methoden und Techniken für Spiele geeignet sind und wie viel Leistungseinsparung erzielt werden kann, wird durch eine Reihe von isolierten Performancetest ermittelt. Nachdem diese Tests vorgenommen und ausgewertet wurden, fließen die Ergebnisse in dem Versuch die Game Engine *Cogaen* für Android Geräte zu optimieren. Dabei soll festgestellt werden, welche Optimierung sich lohnen und welche Auswirkungen sie auf die Engine haben.

Kapitel 2

Aktueller Stand

In diesem Kapitel wird zunächst die allgemeine Entwicklung von Android beschrieben. Danach folgt die Beschreibung des Systems aus technischer Sicht, speziell über die Dalvik Virtual Machine, dem JIT-Compiler der in Android genutzt wird und die Garbage Collection.

2.1 Android

Android ist eine Open Source Plattform für mobile Geräte und wird hauptsächlich für Mobiltelefone, *Smartphones* und *Tablets* genutzt. Entwickelt und gepflegt wird das System von der *Open Handset Alliance*, dessen Hauptmitglied das Unternehmen *Google* ist. Dieses kaufte im Sommer 2005 das von Andy Rubin gegründete Unternehmen *Android* auf, welches an einem freien mobilen Betriebssystem gearbeitet hat.

Im Lauf der letzten Jahre konnte sich Android auf dem Markt erfolgreich durchsetzen und andere Marktführende Systeme wie Symbian OS überholen. Das rasante Wachstum konnte besonders 2011 beobachtet werden. 2010 lag der Anteil des Android Betriebssystems noch bei etwa 22,7 % (im Vergleich zum Vorjahr bei 3,6 %) und damit noch hinter Symbian (37,6 %). Im vierten Quartal 2011 erreichte es einen Anteil von 50,9 %, gefolgt vom ebenfalls wachsenden Konkurrenten iOS mit 23,8 % (Angaben nach dem Marktforschungsunternehmen *Gartner* für 2010 [22] und das vierte Quartal 2011 [24]). Grund für diese Entwicklung ist die freie und kostengünstige Verwendbarkeit des Systems, weshalb Gerätehersteller gerne auf dieses System zurückgreifen.

Zudem ist der Anteil von Spielern mobiler Geräte ebenfalls stark gestiegen (Abbildung 2.1). Dies liegt mit Sicherheit an der einfachen Bedienung der Geräte, den verschiedenen Eingabemöglichkeiten, der hohen Mobilität und der steigenden Hardwareleistung.

Doch trotz immer besser werdender Hardware, müssen Entwickler für die Android Plattform dennoch für den schlechtesten Fall gerüstet sein. Schlech-

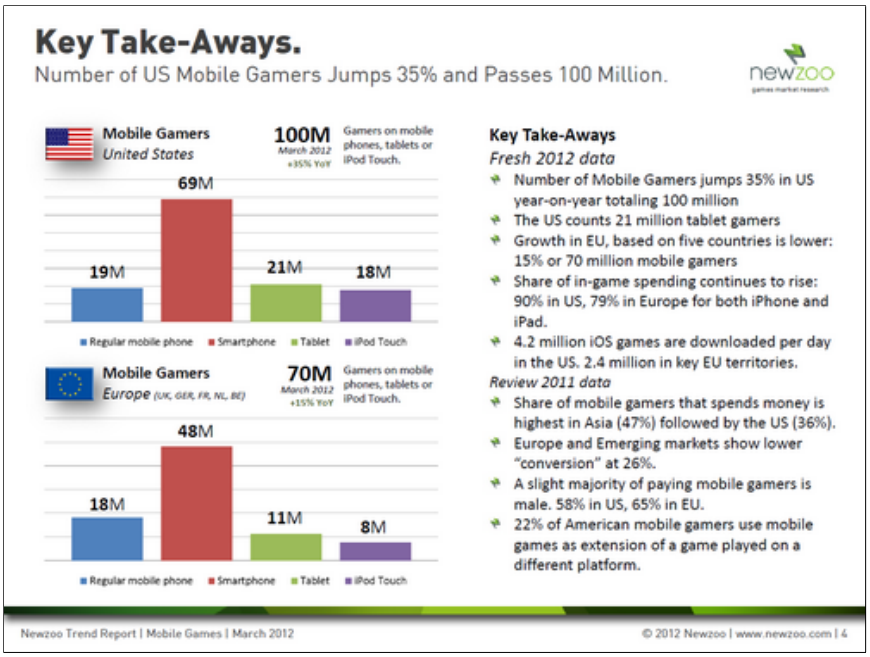

Abbildung 2.1: 2012 ist die Anzahl der Spieler mit Mobilgeräten in den USA um 35 % und in Europa um 15 % gestiegen (nach einem Bericht von *newzoo.com* [17].

tester Fall bedeutet hierbei eine geringe Prozessor Leistung, wenig Speicher und begrenzte Akkulaufzeit. Tatsächlich liegt die Mindestanforderung bei einem Prozessor von 250 MHz und einem Arbeitsspeicher von 64 MB.

2.2 Dalvik Virtual Machine

Die unter der Apache-Lizenz 2.0 vertriebene DVM (Dalvik Virtual Machine), benannt nach einer isländischen Stadt (Abbildung 2.2), ist eine virtuelle Maschine mit der für die JVM (Java Virtual Machine) übersetzte Programme ressourcensparend ausgeführt werden können. Es ist für anspruchsvolle Applikationen mit gewöhnlichem Java SDK 5.0 (bzw. 1.5) und für Hardware mit geringen Ressourcen ausgelegt. Die Mindestanforderung liegt hierbei bei einem Single-Core Prozessor mit 250 bis 500 MHz und einem Speicher von 64 MB, von dem 20 MB für die eigentliche Applikation zur Verfügung stehen. *Swap Space* steht nicht zur Verfügung und die Stromquelle ist eine Batterie und daher nur für eine begrenzte Zeit mobil nutzbar (nach [18, S. 1, Abschnitt 1]).

Bei der Java Virtual Machine, die als Stack-Maschine entworfen wurde, werden Instruktionen mithilfe von *.class*-Dateien dargestellt. Dieser Bytecode wird in einem *Dispatch*-Prozess ausgeführt, der aus drei Phasen besteht.

Die erste Phase (*fetch*) lädt den Bytecode vom Stack. In der zweiten Phase (*decode*) wird auf die notwendigen Attribute auf dem Stack zugegriffen. Und in der dritten und letzten Phase (*execute*) werden die im Byte-Code dargestellten Funktionen ausgeführt.

Im Gegensatz zur JVM wurde die DVM als Register-Maschine entworfen. Bei dieser besteht der Opcode[1] aus 2 Byte anstatt aus einem und umfasst 220 eigene Befehle. Das hat zum einen den Grund, um juristischen Streitigkeiten mit Sun bzw. Oracle zu vermeiden, aber hauptsächlich um die VM sinnvoll an die Prozessorarchitektur anzupassen. Operanden und Bytecode werden in Abhängigkeit des Opcodes in bestimmte (virtuelle) Register geschrieben. Wenn ein Stack und ein Register aber als lineares Array implementiert wurden, unterscheidet sich die Realisierung des Dispatch-Prozesses kaum. Ein Zeitgewinn wird aber möglich, wenn die Register in echten Prozessor Registern abgebildet werden, wie es bei der DVM der Fall ist. Somit ist die VM besser an die Hardware angepasst und kann besser mit Maschinensprache umgehen. Ausgerichtet ist die DVM primär auf die ARM-Architektur. Eine herkömmliche Stack-Maschine wäre zwar auf jeder Prozessorarchitektur möglich, durch die starke Hardware Ausrichtung ist eine Portabilität auf andere Architekturen aber nicht mehr gegeben.

Ein weiterer Unterschied zur JVM ist, dass der Java Bytecode nicht im *.class*-Format dargestellt wird, sondern in einem eigenen Format, dem *.dex*-Format (dalvik executable). Beim Erstellen der Anwendung werden durch das Programm *dx* mehrere .class-Dateien in dalvik-executable-Format übersetzt. Das heißt, aus mehreren Dateien entsteht eine einzige welche in mehrere Segmente mit vorgegebener Reihenfolge unterteilt ist. Dabei werden die Strings für die Bezeichnung von Klassen, Methoden usw. nur einmal in der Datei gespeichert und Wiederholungen entfernt. Die Dateilänge reduziert sich dabei im Vergleich zu einer unkomprimierten jar.-Datei auf durchschnittlich 35 %.

2.3 JIT Compiler

Da bei einem normalen Java Programm der Bytecode von einem Interpreter in Maschinencode übersetzt wird, ist die Ausführungsgeschwindigkeit oft langsamer als bei nativ kompilierten Programmen, welche direkt vom Prozessor ausgeführt werden und nicht von einer virtuellen Maschine. Mithilfe eines JIT-Compilers (Just-In-Time) wird versucht die Ausführungsgeschwindigkeit zu beschleunigen, indem zur Laufzeit leistungsintensive Programmteile identifiziert und optimiert werden und schließlich in Maschinencode übersetzt. Dieser wird meistens zwischengespeichert um später wiederverwendet zu werden, sodass eine wiederholte Übersetzung nicht nötig ist. Für

[1] Nummer eines Maschinenbefehls für bestimmte Prozessortypen (z.B. Addition, Multiplikation, kopieren von Registern usw.).

Abbildung 2.2: Die Dalvik Virtual Machine ist nach einem Ort in Island benannt, in dem Verwandte des Techlead Dan Bornstein leben (Bildquelle [10]).

die Optimierung besteht die Möglichkeit ganze Dateien, einzelne Funktionen bzw. Methoden oder bestimmte Code Abschnitte zu optimieren. Konkret kann eine Optimierung das Programm speziell an das Zielsystem anpassen (CPU, OS), beispielsweise durch Nutzung von speziellen Befehlen des Zielsystems. Abhängig durch gesammelte Statistiken über das Verhalten des Programms in der Umgebung, können Abschnitte neu arrangiert und compiliert werden. Weitere mögliche Optimierungen sind *Dead Code Elimintation*[2], *Loop-Invariant Code Motion*[3], *Loop Unrolling*[4] und *Konstantenfaltung*[5]. Da beim ersten Start des Programms durch das Laden und Kompilieren des Bytecodes eine gewisse Verzögerung entsteht muss ein Kompromiss geschlossen werden, wie viele Optimierungen eigentlich nötig sind. Im Grunde gilt, je mehr Optimierungen, desto schneller ist der Maschinencode, aber desto langsamer ist auch der erste Start des Programms.

Speziell für den JIT-Compiler des Android Systems, der in der Version 2.2 (Froyo) im Jahr 2010 veröffentlicht wurde, gab es bestimmte Anforderungen neben der Verbesserung der Leistung. So durfte der Zuwachs des zusätzlichen Speicheraufwands nur minimal sein, er musste zu dem Sicher-

[2]Dead Code Elimination bezeichnet das Entfernen von Anweisungen im Programmcode, der nicht verwendet wird.

[3]Loop-Invariant Code Motion bedeutet Anweisungen aus einer Schleife herauszunehmen, die durch die Schleife nicht verändert werden.

[4]Mit Loop Unrolling bzw. Unwinding wird das Auflösen einer Schleife bezeichnet, indem die Inhalte untereinander geschrieben werden.

[5]Bei der Konstantenfaltung können bereits zur Zeit der Kompilierung Berechnungen mit Konstanten vorgenommen werden, was zur Laufzeit dann nicht mehr nötig ist.

heitsmodell der DVM passen und durfte nur eine kure „warm up“ Phase besitzen, also nicht zu lange für den Start benötigen. Wie Bill Buzbee in seinem Vortrag auf der Google IO 2010 erklärte [9], wurden zwei verschiedene Typen von JIT-Compilern überprüft. Zum einen Methoden-basiert, bei dem häufig aufgerufene Methoden optimiert werden, und zum anderen Trace-basiert, bei dem häufig aufgerufene Ausführungspfade optimiert werden. Angewendet wurde schließlich ein Trace-basierter JIT-Compiler, da dieser bessere Ergebnisse bei der Geschwindigkeit und dem investierten Speicher lieferte. Im Vergleich zu der vorrangegangenen Version 2.1 (Eclair) konnte mit dem JIT-Compiler bei Benchmark Tests eine 2mal bis 5mal bessere Leistung festgestellt werden. Der benötigte Speicher für das Cachen der Informationen liegt in etwa bei 100 kBytes. Anfänglich gab es einige Schwierigkeiten mit den Oberflächen der verschiedenen Hersteller, beispielsweise Sense von HTC.

Laut Bill Buzbee ist es nicht auszuschließen, dass in Zukunft eine JIT-Compiler entworfen wird der sowohl Methoden-basiert als auch Trace-basiert Optimierungen vornimmt, da ein Mobilgerät das aufgeladen wird einem Server ähnelt, bei dem ein Methoden-basierte JIT-Compiler passender wäre.

2.4 Garbage Collection

Damit ein Spiel flüssig läuft und eine annehmbare FPS Rate (Frames Per Second) erzielt wird, sollten alle Berechnungen und Aktionen die für die Simulation und Darstellung des Spielgeschehens notwendig sind, innerhalb von etwa 16 Millisekunden erledigt sein. Hier kann aber der Garbage Collector, wie er in Java verwendet wird zu Problemen führen. Werden nämlich im laufenden Spiel Instanzen erzeugt und somit Speicher alloziert, muss dieser früher oder später wieder aufgeräumt werden. Bei den Android Versionen vor 2.3 (Gingerbread) wurde ein „stop-the-world“-Ansatz verfolgt. Dabei wird die ganze Applikation angehalten, während der Garbage Collector mit dem *mark-and-sweep*-Verfahren[6] den Heap von nicht referenzierten Objekten befreit. Abhängig von der Größe des Heaps und der Anzahl der Objekte die im Speicher liegen, kann dies bis zu 100 ms oder sogar mehr Zeit in Anspruch nehmen. Das führt in einer Echtzeitanwendung, wie einem Spiel, zu rucklern und beeinflusst das Spielgeschehen negativ. Ab Gingerbread wurde der *Concurrent* Garbage Collector eingeführt. Dieser stoppt nicht die gesamte Applikation, sondern räumt den Heap in einem eigenen Thread auf.

[6] Beim mark-and-sweep-Algorithmus werden ausgehen vom Wurzelknoten alle Referenzierungen im Heap ermittelt und markiert. Objekte, die nicht länger referenziert sind, also nicht mehr vom Wurzelknoten erreichbar sind, werden entfernt. Beim mark-sweep-compact-Algorithmus werden zusätzlich noch die Speicherblöcke der übrigen Objekte verschoben um eine Fragmentierung zu vermindern.

Kapitel 3

Werkzeuge und Design Methoden

In diesem Kapitel werden die verschiedenen Werkzeuge wie TraceView, TimingLogger und Allocation Tracker vorgestellt, welche von der Android SDK bereitgestellt werden. Schließlich werden Design Methoden und Programmiertechniken gezeigt, die zur Performanceoptimierung angewendet werden können.

3.1 Werkzeuge

Es gibt eine Reihe von Werkzeugen, mit deren Hilfe Anwendungen analysiert und optimiert werden können. Einige davon sind bereits im Android SDK enthalten. Mit diesen Werkzeugen ist es möglich festzustellen, welche Ursachen zu Performanceproblemen beitragen und ob *Flaschenhälsen*[1] im laufe der Entwicklung einer Anwendung entstanden sind. Der Vorgang der Analyse wird *profiling* bezeichnet und die Werkzeuge als *Profiler*.

3.1.1 TraceView

Mithilfe von TraceView können die Ausführungszeiten der einzelnen Komponenten einer Anwendung grafisch dargestellt und detaillierte Informationen abgerufen werden.

Für die Auswertung der Anwendung wird eine *.trace*-Datei erzeugt (profiling). Diese besteht aus einem *data-file*, welches die Trace-Daten enthält und einem *key-file*, welches eine Zuordnung von binären Kennungen zu Thread- und Methodennamen anbietet. Diese beiden Dateien werden automatisch zu einer *.trace*-Datei verbunden.

Die Darstellung der Informationen ist in ein *Timeline*-Panel (siehe Ab-

[1] Flaschenhälse, oder Englisch Bottlenecks, sind vereinzelte Engstellen im System die zu Leistungsbeeinträchtigung führen können.

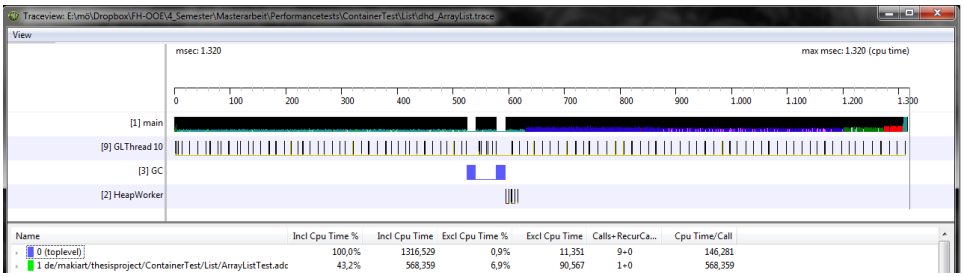

Abbildung 3.1: Das Timeline Panel stellt die zeitliche Abfolge der Methoden anhand eines Zeitstrahls dar und unterteilt diese in die einzelnen Threads.

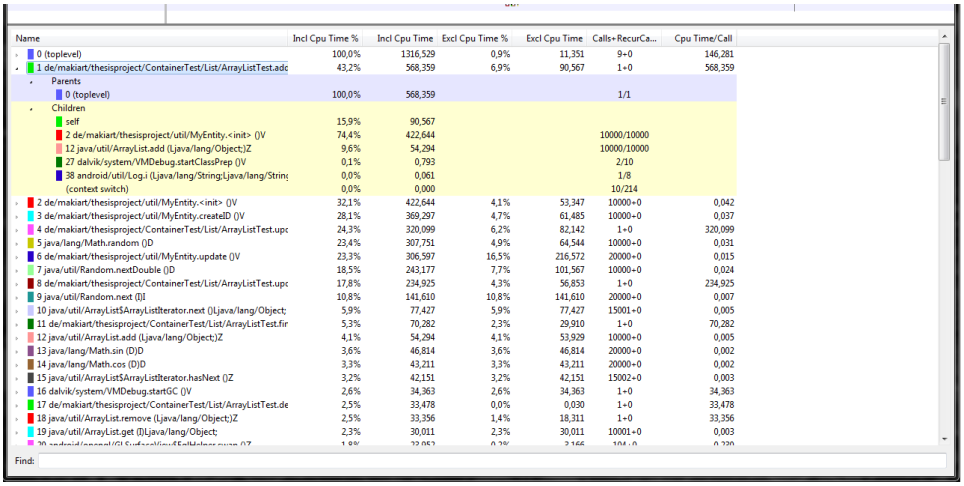

Abbildung 3.2: Im Profil Panel werden die eigentlich Informationen dargestellt.

bildung 3.1) und einem *Profile*-Panel (siehe Abbildung 3.2) aufgeteilt. Im Timeline-Panel wird anhand eines Zeitstrahls, der die Zeit des trace-Vorgangs repräsentiert, die zeitliche Abfolge und Dauer der einzelnen Methoden als auch die verschiedenen Threads der Anwendung dargestellt.

Das Profile-Panel zeigt genauere Informationen der einzelnen Methoden an. Dabei werden die inklusiven und exklusiven Zeiten in Form von Zeit und Prozentangaben ausgegeben. Die exklusive Zeit beschreibt wie viel Zeit die Anwendung in der Methode verbracht hat. Die inklusive Zeit hingegen zeigt, wie viel Zeit in der Methode (*Parent*) und jede aufgerufene Funktion (*Children*) verbracht wurde.

Um Traceview nutzen zu können gibt es zwei Ansätze. Beim ersten Ansatz wird die *Debug*-Klasse (android.os.Debug) verwendet, mit der an einem bestimmten Code-Abschnitt der Tracevorgang gestartet und wieder gestoppt

werden kann. Da durch die Befehle *Debug.startMethodTracint(„calc“)* und *Debug.stopMethodTracing()* genau angegeben werden kann, wann das Programm untersucht (traceing) werden soll, ist dieser Ansatz sehr präzise.

```
1 // start tracing
2 Debug.startMethodTracing("calc");
3 // ...
4 // stop tracing
5 Debug.stopMethodTracing();
```

Für das Speichern der Trace-Datei wird eine SD-Karte benötigt, außerdem muss im *AndroidManifest* die *Permission*[2] zum schreiben auf externe Speicher eingetragen werden. Die Trace-Datei kann nach Abschluss des Profiling Vorgangs von der SD-Karte herunter geladen werden. Dabei hilft DDMS[3], mit dessen Hilfe auf das Gerät zugegriffen und einfach per Knopfdruck die Trace-Datei heruntergeladen werden kann.

Der zweite Ansatz, wie Traceview genutzt werden kann, ist mithilfe von DDMS. Hierbei wird in der DDMS Perspektive von Eclipse der Button *start method profiling* bzw. *stop method profiling* gedrückt. Dieser Ansatz ist nicht besonders präzise, da nicht genau angegeben werden kann wann das Tracing vorgenommen wird. Dieser Ansatz lohnt sich daher nur, wenn kein Zugriff auf dem Code der Applikation möglich ist oder keine Präzision wie bei der vorherigen Methode notwendig ist. Ab Android Version 2.2 wird für dieses Vorgehen keine SD-Karte und keine *Permission* zum schreiben auf diese benötigt. Die Informationen werden stattdessen vom Gerät aus direkt auf die Entwicklungsumgebung gestreamed.

3.1.2 TimingLogger

Die Hilfsklasse *andoird.util.TimingLogger* wird verwendet um die benötigte Zeit an bestimmten Stellen des Programmcodes im Log auszugeben (beispielsweise unter LogCat). Dabei ist es möglich die Ausgabe so zu unterteilen, dass ein genaues Bild der benötigten Zeit ausgegeben wird.

Beim Erzeugen eines TimingLogger-Objektes wird ein *tag*-Name angegeben, mit dessen Hilfe die Informationen aus der Log-Datei gefiltert werden können.

```
1 TimingLogger timingLog = new TimingLogger(TAG, "myMethod");
2 // do work A
3 timingLog.addSplit("work A");
4 // do work B
```

[2]Permissions regeln die Zugriffsrechte unterschiedlicher Funktionen einer Android Anwendung (z.B. Internetzugriff und Schreibzugriff.

[3]Dalvik Debug Monitor Server ist ein umfangreiches Debugging Werkzeug, welches im Android SDK enthalten ist und auch in der IDE Eclipse integriert ist. Es unterstützt *port-forwarding* Dienste, *Screen Capture*, Thread und Heap Informationen, Logcat, Manipulation von SMS, Telefonanrufen und GPS-Position.

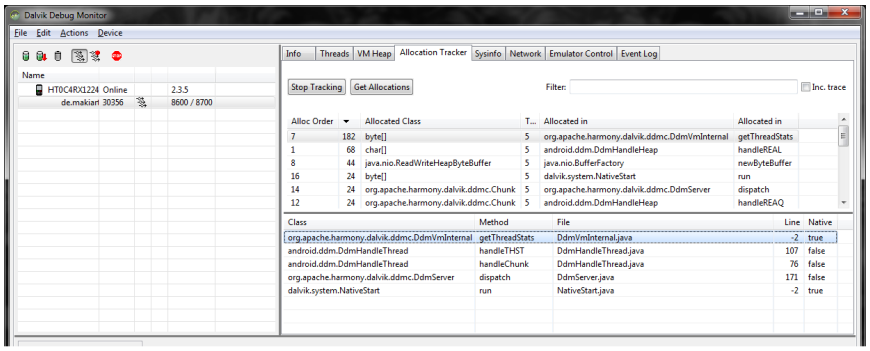

Abbildung 3.3: Der AllocationTracker listet alle Informationen auf, um die Herkunft und Größe der Objekte feststellen zu können.

```
5 timingLog.addSplit("work B");
6 // do work C
7 timingLog.addSplit("work C");
8 timingLog.dumpToLog();
```

3.1.3 Allocation Tracker

Der Allocation Tracker 3.3 ist eine weitere Funktion des DDMS, mit dem festgestellt werden kann, welche Klassen und Threads Objekte alloziert haben. Dabei wird die Reihenfolge der allozierten Objekte ausgegeben und zusätzliche Informationen wie Ursprung der Allokation (Datei, Klasse, Zeile) und Größe des Objekts.

Dabei ist es möglich, wie mit *TraceView* sehr schnell über die DDMS-Perspektive den Tracking-Vorgang zu starten und zu stoppen. Für genauere Kontrolle des Vorgangs, können aber auch im Programmcode die beiden Debug Methoden

```
startAllocCounting()
stopAllocCounting()
```

verwendet werden. So kann genau bestimmt werden, ab wann die Allokationen untersucht werden sollen.

3.1.4 Memory Analyzing Tool

Das Memory Analyzer Tool ist ein mächtiges Werkzeug, mit dem Speicherlecks identifiziert werden können. Mit dem Programm, welches als *Standalone*-Version und als Eclipse-Plugin verfügbar ist, werden *Heap Dumps* analysiert. Heap Dumps sind Abbildungen des Java Heaps in textueller oder binärer Form. Mit Hilfe dieser Abbildung und einem Werkzeug wie Allocation Tracker oder dem Memory Analyzing Tool können die verschiedenen

Objekte und Referenzen, die im Laufe des Lebenszyklus des Programms erzeugt und gelöscht worden sind, rekonstruiert werden.

Das Abbild kann mithilfe des DDMS erzeugt werden oder wenn genauer analysiert werden soll mit der Methode der Debug-Klasse

```
dumpHprofData(/sdcard/MyDump.hprof).
```

Dabei ist aber zu beachten, dass im *AndroidManifest* die *Permission android:name="android.permission.DUMP"* für das Erzeugen der Heap-Dumps eingeholt werden muss. Sonst können keine Informationen für den Dump abgerufen werden. Da Dalvik nur ein ähnliches aber nicht identisches Format für die Heap Dumps im Gegensatz zu Java verwendet, müssen diese zusätzlich noch konvertiert werden. Dazu wird das vom SDK mitgelieferte Werkzeug *hprof-conv* verwendet:

```
hprof-conv MyDump.hprof conv-MyDump.hprof.
```

3.2 Design Methoden

Es gibt verschiedene *Methoden*, mit deren Hilfe Performanceeinbußen verhindert werden können. In diesem Kapitel werden einige Methoden vorgestellt und später mit Performance Tests überprüft, wie viel Leistung oder auch Zeit mit diesen Methoden gewonnen werden kann (mehr zu den Tests in Abschnitt 4) und ob sich eine Integration dieser Methoden in ein bestehendes System lohnen würde.

3.2.1 Objekte

Grundsätzlich gilt bei der Performance-Optimierung, das Objekterzeugungen vermieden werden sollen. Denn jede Objektinstanz die erzeugt wurde, muss wieder entfernt werden, sobald sie nicht weiter verwendet wird und dadurch Speicher unnötig verbraucht. In Java wird der Speicher durch den *Garbage Collector* wieder frei gegeben (siehe Abschnitt 2.4). Abhängig von der Android Version kann das „aufräumen“ durch den Garbage Collector, während sich die Applikation in der *Game Loop* befindet, zum stocken des Spiels führen und den Spielfluss daher beeinträchtigen.

3.2.2 Static und Final

In der Regel wird beim kompilieren eine Klassen Initialisierungsmethode generiert (<clinit>), die beim erstmaligen ausführen der Klassen aufgerufen wird. Dabei werden die Attribute gespeichert um später, mithilfe von *field lookups*, darauf zugreifen zu können. Durch die *final*-Anweisung wird diese Methode nicht benötigt und die Konstanten werden direkt aus der *.dex*-Datei aufgerufen. So werden *field lookups* nicht benötigt. Dies funktioniert aber nur

mit primitiven Datentypen wie beispielsweise *int* und Konstanten des Typs String. Durch das Schlüsselwort *static* kann ebenfalls ein Performancegewinn erzielt werden, da hier kein Zugriff auf eine Objektinstanz nötig ist. Natürlich ist die Verwendung von *static* und *final* Schlüsselwörtern abhängig von der Situation in der sie verwendet oder eben nicht verwendet werden können.

3.2.3 Fließkommazahlen, Ganzzahlen und Festkommazahlen

Besitzt das Android-Gerät keine FPU (Floating Point Unit) und nutzt keinen JIT-Compiler, so kann ein int etwa doppelt so schnell bei Berechnungen als ein float sein (nach dem Abschnitt „Use Floating-Point Judiciously“ im Artikel „Designing for Performance“ des Android Developer Guides [13]). Bei moderner Hardware und eben der Unterstützung einer FPU besteht aber kaum ein Geschwindigkeitsunterschied. Durch die Verwendung einer FPU ist aber ein höherer Stromverbrauch möglich. Zwischen float und double besteht kaum ein Geschwindigkeitsunterschied, lediglich der benötigte Speicher ist bei float halb so groß wie bei double.

3.2.4 Schleifen

Java bietet einer Reihe von Schleifen an die für Spiele relevant sind. Das wären die for-, while- und for-each-Schleife (Enhanced For Loop). Bei den Schleifen sollte vermieden werden, in jedem Schleifendurchlauf Konstante Werte neu abzufragen. Beispiels Weise kann die Länge des zu iterierenden Containers als lokale Variable außerhalb der Schleife gespeichert werden, wodurch eine schnellere Iteration möglich ist. Nach den Angabe der Android Developer Webseite über Designing for Performance [13] ist die sogenannte for-each-Schleife am besten für das iterieren von Containern wie Array und Maps geeignet. Für die ArrayList ist jedoch eine einfache Schleife die hoch zählt für performancekritische Iterationen zu empfehlen. Selbst ohne JIT-Compiler kann eine solche Schleife drei mal schneller sein als eine for-each. Was auch von Dan Bornstein in seinem Vortrag bei der Google IO 2008 empfiehlt [8].

3.2.5 Container

Im *Package java.util* der Java Plattform befinden sich verschiedenen *Container* mit deren Hilfe Objekte gespeichert und wieder darauf zugegriffen werden können. Android unterstützt alle diese Container der Java Umgebung, aber abhängig von Funktionsweise und Leistung sind einige *Container* mehr und andere weniger geeignet um in einem Spiel verwendet werden zu können.

Als Basisschnittstellen für die Container dienen *Java.util.Collection* und *Java.util.Map*. Diese Liefern Basisoperationen, mit deren Hilfe Elemente hinzugefügt, gelöscht, selektiert, gefunden oder deren Anzahl ermittelt werden

Tabelle 3.1: Basistypen der Container unter *Java.util*.

	Basistyp	konkrete Implementierung
Collection	List	ArrayList
		LinkedList
		Stack
		Vector
	Set	EnumSet
		HashSet
		LinkedHashSet
		TreeSet
	Queue	LinkedList
		ArrayDeque
		PriorityQueue
Assoziativspeicher	Map	EnumMap
		HashMap
		Hashtable
		IdentityHashMap
		LinkedHashMap
		Properties
		TreeMap

können (entnommen aus dem Buch „Java ist auch eine Insel“ [5, Abschnitt 13.1.2]). Es können daher nicht nur konkrete Klassen aus *Java.util* als Container verwendet werden, sondern auf Basis von *Java.util.Collection* und *Java.util.Map* eigene Container implementiert werden. Folgende Tabelle 3.1 beschreibt die Zugehörigkeit der konkreten Klassen zu ihren Interfaces.

ArrayList

ArrayList ist eine Implementierung von *List* und unterstützt das Hinzufügen, Entfernen und Ersetzen von Elementen, darunter auch *NULL*-Elemente. Im Gegensatz zum Array, das eine feste Größe besitzt, ist die Länge einer ArrayList veränderbar. Die Operationen *size*, *isEmpty*, *get*, *set*, *iterator* und *listIterator* arbeiten in konstanter Zeit, während das Hinzufügen der Elemente n in linearer Zeit O(n) abläuft[4]. Eine ArrayList verwendet intern ein Array einer bestimmten Länge, welches bei Bedarf durch ein längeres Er-

[4] *“The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.”* [7].

setzt wird. Dabei müssen aber alle Elemente in das von dem ursprünglichen Array in das neue kopiert werden. Über die Position eines Elements kann sehr schnelle auf dieses zugegriffen werden. Sollen aber Objekte mitten in der Liste gelöscht oder eingefügt werden, müssen alle nachfolgenden Listenelemente verschoben werden. Diese Form von Container ist bei Spielen also für Aufgaben gut geeignet, bei denen nur am Ende der Liste Elemente manipuliert werden oder gezielt auf Elemente zugegriffen wird. In den Performancetests wird dieser Container berücksichtigt (siehe Abschnitt 4).

LinkedList

LinkedList implementiert sowohl *List* als auch *Deque*. Es handelt sich um eine verkette Liste, bei der die Listenelmente durch Hilfsobjekte miteinander verbunden sind. Soll auf ein Element an einer bestimmten Position zugegriffen werden, so wird im Gegensatz zur ArrayList die ganze Liste nach dem Element durchsucht[5]. Dafür ist das Einfügen und entfernen von Elementen wesentlich Zeitsparender, da lediglich die Verkettung der Hilfsobjekte verändert werden muss. Auch dieser Container wird bei den Performancetests berücksichtigt.

Stack

Stack ist eine Implementierung von *List* und dient als Stack (LIFO[6]). Die Elemente werden wie bei einem Stapel abgelegt, während der Zugriff nur auf das oberste Element möglich ist.

Vector

Vector besitzt als Implementierung von *List* dieselben Eigenschaften wie ArrayList. Der Unterschied zwischen den beiden ist, dass die Methoden von Vector *synchronisiert* sind. Sollten verschiedene *Threads* auf das gleiche Vector-Objekt zugreifen, ist sichergestellt, dass sich die Threads nicht gegenseitig in die Quere kommen. Wie stark diese Thread-Sicherheit die Leistung beeinflusst wird in den Performancetests überprüft.

EnumMap

EnumMap ist eine Map Implementierung mit Enum-Typen als Keys. Null-Keys werden nicht unterstützt. Die Reihenfolge der Elemente ist abhängig von der „natural order“ der Keys.

[5]Ist der Index des gesuchten Elements kleiner als die halbe Länge der Liste, wird vom Anfang aus, ansonsten vom Ende aus gesucht. So wird im schlechtesten Fall nur die halbe Liste durchsucht.

[6]Last-In-First-Out, das zuletzt eingefügte Element wird als erstes wieder verwendet.

HashMap

HashMap ist eine Implementierung des Map-Interfaces. Es erlaubt alle Objekt Typen, darunter sowohl Null-Werte als auch Null-Keys. Die HasMap ist annähernd identisch zur Hashtable, wobei die Hashtable synchronized ist. Es besteht keine Sicherheit, dass sich die Reihenfolge der Elemente in der Map im Laufe der Benutzung nicht ändern. Für die Erstellung der Map werden zwei optionale Parameter angeboten: *capacity* und *load factor*. *Capacity* beschreibt dabei die anfängliche Länge der Map, bzw. wie viele Elemente hinzugefügt werden. Wird dieser Parameter nicht genutzt, ist der Anfangswert standardmäßig 16. *Load factor* ist ein *float* der definiert, wie weit die Map gefüllt werden darf, bis sie sich automatisch verlängert wird. Der Standardwert ist 0,75.

Hashtable

Hashtable ist eine Implementierung einer *Map*, deren Einträge aus Schlüssel- und Datenobjekt bestehen und nicht *NULL* sein dürfen (im Gegensatz zur *HashMap* und *LinkedHashMap*). Sowohl der Wert als auch der Key kann mit einem beliebigen Objekt versehen werden. Der Zugriffsschlüssel der Tabelle ist eindeutig. Wird ein Element mit einem bereits vorhandenen Schlüssel eingefügt, wird der alte Wert mit diesem Schlüssel ersetzt.

IdentityHashMap

Die IdentityHashMap entspricht einer Hash Tabelle, bei der Werte und Keys auf Referenz-Gleichheit anstatt Objekt-Gleichheit verglichen werden. Daher ist dieser Container nur Sinnvoll, wenn Referenz-Gleichheit notwendig ist. Null-Werte und -Keys werden unterstützt. Die Reihenfolge ist nicht konstant.

LinkedHashMap

Eine Hash Tabelle als doppelt verkettete Liste. Die Reihenfolge in der die Elemente iteriert werden ist berechenbar, da sie durch das Einfügen der Keys definiert wird. Null-Elemente werden unterstützt.

Properties

Bei einem Properties-Objekt sind sowohl Key und Wert Strings. Es ist möglich Elemente von Dateien zu streamen. Properties können hierarchisch verbunden werden, indem dem Konstruktor ein weiteres Properties (Eltern-Objekt) übergeben wird. Schlägt nun eine Suche nach einem Element fehl, wird die Anfrage auf das übergeordnete Properties-Objekt weitergeleitet.

EnumSet

EnumSet ist eine Implementierung von *Set* für die Verwendung von Enum-Typen. Es sind nur Enum-Typen und keine Null-Elemente erlaubt. Das Set durchläuft alle Elemente in der Reihenfolge, in der sie als Enum deklariert wurden. Es gilt als leistungsfähigere alternative zu int-basierenden „bit flags“, was Zeit und Speicher angeht.

HashSet

HashSet implementiert eine Menge (*Set*) und verwaltet seine Elemente in einer *HashMap*. Es kann alle Objekt-Typen aufnehmen, auch Null-Elemente. Elemente können in der Menge nicht mehrfach enthalten sein und sind daher eindeutig. Wird versucht der Menge ein Element hinzuzufügen, dessen Wert bereits enthalten ist, wird dieses nicht ersetzt, sondern der Inhalt des Containers bleibt unverändert.

LinkedHashSet

LinkedHashSet ist eine Implementierung einer doppelt-verketteten Liste des Set-Interfaces. Die Elemente werden in der Reihenfolge durchlaufen, in der sie eingefügt worden sind.

TreeSet

Beim TreeSet werden die Elemente in einem balancierten Binärbaum gespeichert. Neue Elemente werden sortiert in die Datenstruktur eingepflegt, dieser Vorgang ist daher zeitaufwendig, aber mit dem Vorteil, dass eine geordnete Ausgabe schnell verläuft. Das Suchen eines Elements entspricht log(n) wobei n der Anzahl der Elemente entspricht. Bei Einfüge- und Löschvorgängen ist eine Umstrukturierung des Baumes möglich, weshalb diese Operationen Zeitaufwendig sind. Die Elemente müssen aber die Methode *compareTo* implementieren, um von TreeSet verglichen zu werden.

ArrayDeque

ArrayDeque ist ein größenveränderbares Array welches *Deque* implementiert. Es ist nicht Thread sicher und unterstützt keinen konkurrierenden Zugriff von verschiedenen Threads. Es kann als Stack oder Queue verwendet werden. Die meisten Operationen arbeiten in konstanter Zeit, während Operationen wie *remove, removeFirstOccurence, removeLastOccurence, contains, iterator.remove()* in linearer Zeit arbeiten.

PriorityQueue

Je nachdem welcher Konstruktor für die PriorityQueue gewählt wurde, werden die Elemente in „natürlicher Ordnung“[7] oder nach einem bestimmten *Comparator* definiert. Null-Elemente und nicht-Comparable Objekte werden von PriorityQueue nicht unterstützt. Das *head*-Element ist das kleinste Element in der Queue auf das zugegriffen wird.

3.2.6 Object Pooling

Wenn im Laufe des Spiels sehr viele Instanzen erstellt werden, diese nur kurz gebraucht und schließlich verworfen werden, kann es sein, dass der Garbage Collector sehr häufig anspringen muss um wieder Speicher frei zu geben. Passiert das zu häufig, kann der Spielfluss stark beeinflusst werden (z.B. durch Ruckler, Einfrieren des Spiels).

Um das zu verhindern kann das Object Pool Pattern verwendet werden. Dabei verwaltet ein Pool die Instanzen, die in einem privaten Array speichert werden. Durch eine *acquire*-Methode kann ein Client auf die Instanzen zugreifen. Zu diesem Zeitpunkt ist die abgerufene Instanz für andere Clients gesperrt. Die *release* Methode gibt die Instanz schließlich wieder frei, jedoch nicht für den Garbage Collector. Die Instanz wird stattdessen wieder für die nächste Verwendung in dem privaten Array wieder gespeichert. Dabei ist wichtig, dass der Zustand der Instanz zurückgesetzt wird, um Seiteneffekte bei der Wiederverwendung zu vermeiden. Sollte der Pool keine Instanzen mehr besitzen, kann er entweder einen Fehler ausgeben oder eine neue Instanz erstellen, die nach der Benutzung wieder gespeichert wird. Dabei kann das Erstellen neuer Instanzen begrenzt und nicht mehr verwendete Instanzen für den Garbage Collector freigegeben werden.

Nachteil des Patterns ist das Ansteigen des Speicherverbrauchst. Zum einen ist das auf Mobilgeräten, die nur auf begrenzten Speicher zugreifen können ärgerlich. Es ergibt sich aber auch das Problem, dass der Garbage Collector aufgrund der hohen Anzahl an Instanzen die im Speicher liegen, diesen länger durchlaufen muss um Speicher frei zu geben. Auch kann der zeitliche Aufwand durch die Nutzung zusätzlicher Methoden erhöht werden.

[7] durch die *compareTo*-Methode definierte Reihenfolge der Elemente.

Kapitel 4

Performance Tests

In diesem Kapitel werden Vorgehensweise und Ergebnisse der Performancetests für die Überprüfung der im vorangegangenen Kapitel beschriebenen Design Methoden und Programmiertechniken dargestellt.

4.1 Testverlauf

Für die Tests wurden drei physische Geräte aber kein Emulator verwendet, da bei diesem durch die Emulation des Systems sehr viel Leistung verloren gehen kann und die Ergebnisse verfälscht werden. Verwendet wurde ein HTC Desire HD (DHD) mit einem Singlecore Qualcomm MSM8255 Prozessor mit 1024 MHz, 1 GB Arbeitsspeicher, einem Kapazitiven Touchscreen mit einer Auflösung von 480x800 und der Android Version 2.3.5 (Gingerbread). Sowie ein HTC Dream (T-Mobile G1) mit einem Singlecore Qualcomm MSM7201A Prozessor mit 528 MHz 192 MB Arbeitsspeicher und einem Kapazitiven Touchscreen mit einer Auflösung von 320x480 und einer der Android Version 2.2 (Froyo). Zusätzlich wurde bei den Simulationen das Tablet Asus Transformer tf300t mit einem NVIDIA Tegra 3 Quadcore Prozessor mit 1100 MHz, 1 GB Arbeitsspeicher, einem kapazitiven Touchscreen mit einer Auflösung von 1280x752 und einer Android Version von 4.0.3 verwendet. In den Tests wurden isoliert voneinander die verschiedenen Design Methoden aus Kapitel 3.2 getestet und abschließend anhand einer Simulation mit der Game Engine Cogaen kombiniert. Um äußere Einflüsse zu vermeiden, wurden die Geräte bei den Tests auf *Flugmodus*[1] gestellt.

[1] Als Flugmodus wird eine Einstellung bei Mobilgeräten bezeichnet, bei der verschiedene Schnittstellen zur drahtlosen Übertragung am Gerät abgeschaltet werden (z.B. W-LAN und Bluetooth).

Tabelle 4.1: Benötigte Zeit in Millisekunden für verschiedene Funktionen die an Listen angewendet werde können beim Testgerät HTC Desire HD.

Funktion	ArrayList	LinkedList	Stack	Vector
addElement	568,359	674,317	570,831	597,663
updateElement by Index	234,925	31582,52	285,553	303,985
updateElement by Iterator	320,099	332,489	504,944	505,829
findElement by Index	0,092	4,517	0,152	0,061
findElement by Element	70,282	75,531	153,107	151,733
deleteElement by Index	0,183	4,364	0,183	0,152
deleteElement by Element	33,478	101,501	34,76	34,729
deleteAllElements	8,667	0,183	14,648	14,649

4.2 Testergebnisse

Im folgenden Abschnitt werden die Testergebnisse der Performance-Tests zu den Methoden der Performance-Optimierung 3.2 beschrieben und ausgewertet, sowie die Gründe für diese Ergebnisse erklärt. Am Ende der jeweiligen Abschnitte werden die Ergebnisse verglichen und eine Empfehlung für bestimmte Anwendungsfälle ausgesprochen.

4.2.1 Container

Die bereits im Abschnitt Design Methoden beschriebenen Container wurden getestet, indem eine festgelegte Anzahl von Entitäten mit verschiedenen Aktionen verarbeitet wurden. Diese Aktionen bestanden aus, Anlegen der Entitäten in einen Container, durch die Entitäten im Container iterieren, einzelne Entitäten finden, einzelne Entitäten entfernen und schließlich alle Entitäten aus dem Container entfernen. Dabei wurde berücksichtigt, dass verschiedene Container-Typen nicht direkt verglichen werden können, da unterschiedliche Konzepte die Bedienung beeinflussen. So kann in einer Liste ein Element nicht mithilfe eines Schlüssels gefunden werden, wohl aber mit einem Index, wohingegen Maps keine Indizes unterstützen. Daher wurden Listen mit Listen, Maps mit Maps und Sets mit Sets verglichen. Bei Überschneidungen von Funktionalitäten wurden diese auch Typübergreifend verglichen, wie beispielsweise update und clear.

4.2.2 Listen

Die von Java bereitgestellten Listen *ArrayList*, *LinkedList*, *Stack* und *Vector* werden in diesem Performancetest Funktionen getestet, die in den beiden Tabellen 4.1 und 4.2 zu finden sind.

Tabelle 4.2: Benötigte Zeit in Millisekunden für verschiedene Funktionen die an Listen angewendet werde können beim Testgerät HTC Dream (G1).

Funktion	ArrayList	LinkedList	Stack	Vector
addElement	2064,605	2288,299	1922,028	1904,083
updateElement by Index	1234,558	56642,608	1398,59	1419,403
updateElement by Iterator	1721,1	1523,59	2206,757	2246,43
findElement by Index	0,335	11,596	0,702	0,336
findElement by Element	273,774	271,912	555,054	597,2
deleteElement by Index	0,824	11,872	0,763	0,977
deleteElement by Element	103,881	112,579	112,61	116,211
deleteAllElements	22,858	0,397	49,744	49,774

Elemente hinzufügen

Für das Hinzufügen von Elementen (*addElement()*) wurde vor den Performance Tests angenommen, dass bei der *ArrayList* ein hoher Zeitaufwand entstehen könnte, da die Inhalte des internen Arrays in der Liste in ein neues Array kopiert werden müssen, sobald das alte Array voll ist bzw. eine bestimmte Menge an Elementen enthält. Die *LinkedList* müsste dagegen schneller Inhalte einfügen können, da zum Anlegen der Elemente nur eine Referenz zum nächsten bzw. zum vorherigen Element angelegt werden muss. Der *Vector* funktioniert ähnlich wie die *ArrayList*, da er aber threadsicher ist, kann davon ausgegangen werden, dass die Abläufe langsamer statt finden. Da der *Stack* von *Vector* erbt, werden sich wahrscheinlich ähnliche Werte ergeben.

Die *ArrayList* hat beim HTC Dream (G1) 2064,605 ms und beim HTC Desire HD 568,359 ms für das Einfügen der 10000 Elemente benötigt. Im Vergleich zu den anderen Listen war sie damit beim G1 die zweitlangsamste Liste (siehe Abbildung 4.2), beim Desire HD wurde jedoch mit dieser Liste am wenigsten Zeit benötigt (siehe Abbildung 4.1). Das lässt sich darauf zurückführen, dass beim Erstellen der Liste intern ein Array mit einer bestimmten Länge erstellt wird. Ist dieses Array voll bzw. ist eine bestimmte Menge an Elementen eingefügt worden (Standardmäßig 75 % der Länge des Arrays), wird eine größeres Array erstellt und alle Elemente des alten Arrays in dieses kopiert. Ab einer gewissen Anzahl an Elementen benötigt dieser

Vorgang viel Zeit.

Mit 2288,299 ms beim G1 und 674,317 ms beim Desire HD, war die *LinkedList* beim Einfügen der Elemente bei beiden Geräte im Vergleich die langsamste Liste.

Beim *Stack* wurden für das Einfügen 1922,028 ms beim G1 und 570,831 ms beim Desire HD benötigt. Der *Stack* war zweitschnellster „Kandidat" beim Desire HD. Beim G1 lag er in etwa, wie bereits erwartet, auf der selben Höhe wie Vector und war damit am schnellsten. Das liegt daran, dass dieselbe *add*-Methode für das Hinzufügen der Elemente verwendet wurde, die durch die Vererbung von *Vector* gegeben ist. Hier stellt sich natürlich auch die Frage, warum *Stack* die Methode anbietet, da eigentlich die Methode *push* für einen Stapel verwendet wird.

Der *Vector* war mit 1904,083 ms beim G1, zusammen mit dem *Stack*, die schnellste Liste beim Einfügen der Elemente. Beim Desire HD war er aber mit 597,663 ms die zweitlangsamste Liste. Aufgrund der gegebenen Thread Sicherheit ist *Vector* langsamer als eine ArrayList, trotz selber Funktionalität. Das *Vector* und *Stack* etwas schneller sind (ca. 20 ms) als ArrayList kann ein Messfehler gewesen sein.

Iteration mit Index

Es kann erwartet werden, dass die *ArrayList* bei der Iteration mithilfe des Index deutlich schneller ist als bei den anderen Listen, da im Grunde ein Array iteriert wird und kein größerer Aufwand aufgebracht werden muss. Die *LinkedList* hingegen wird wahrscheinlich nur langsam iteriert werden können, da es keinen Index wie bei einem Array gibt. Weil der *Stack* für so eine Iteration eigentlich nicht gedacht ist, kann auch hier der ein solcher Vorgang langsamer sein. Aufgrund der Thread-Sicherheit, die vor allem bei der Iteration wichtig ist, kann auch beim *Vector* eine langsamere Iteration angenommen werden.

Bei den Performancetests hat sich wie erwartet ergeben, dass das Iterieren mit Index bei der *ArrayList* am schnellsten im Vergleich zu den anderen Listen ist (siehe Abbildung 4.4 für das G1 und Abbildung 4.3 für das Desire HD). Die benötigten Zeiten lagen beim G1 bei 1234,558 ms und beim Desire HD bei 234,925 ms. Der Abstand zwischen der *ArrayList*, *Stack* und *Vector* ist etwas geringer ausgefallen als erwartet, was darauf zurück zu führen ist, dass im Grunde dieselbe Methodik in der *get(index)* steckt. Bei den drei Listen wird auf ein über den Index auf ein internes Array zugegriffen.

Die *LinkedList* war mit 56642,608 ms beim G1 und 31582,52 ms beim Desire HD mit Abstand die langsamste Liste bei der Iteration mit Index. Das liegt daran, dass nicht direkt mit dem Index ein Element aus der Liste ausgewählt werden kann, sondern die ganze Liste bis zu dem gegebenen Index durchlaufen werden muss.

Überraschend war das Ergebnis beim *Stack* (G1 1398,59 ms, Desire HD

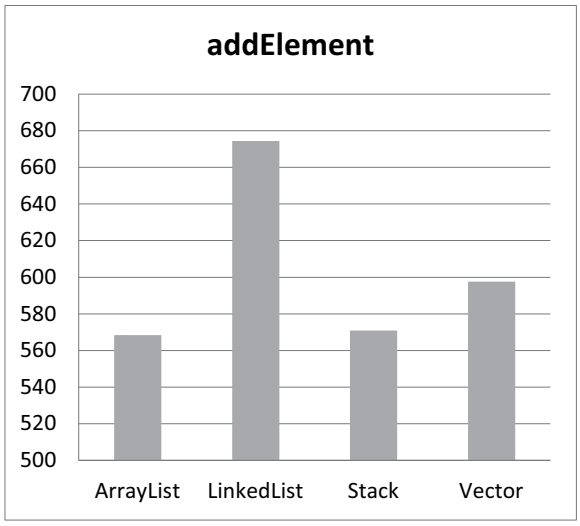

Abbildung 4.1: Benötigte Zeit in Millisekunden der getesteten Listen beim Einfügen von Elementen mit dem Testgerät Desire HD.

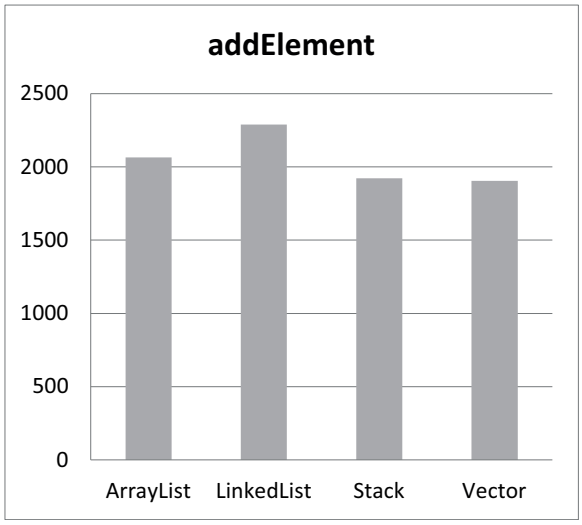

Abbildung 4.2: Benötigte Zeit in Millisekunden der getesteten Listen beim Einfügen von Elementen mit dem Testgerät HTC Dream (G1).

285,553 ms, der genau wie *Vector* (G1 1419,403 ms, Desire HD 303,985 ms), nur etwas langsamer als die *ArrayList* gewesen ist. Das, wie bereits beschrieben, an der gleichen Methodik der *get(index)*-Methode liegt.

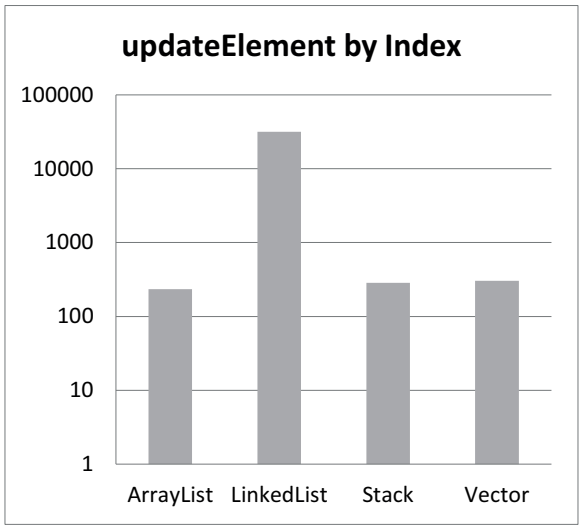

Abbildung 4.3: Benötigte Zeit in Millisekunden der getesteten Listen beim Iterieren mithilfe des Index am Testgerät Desire HD.

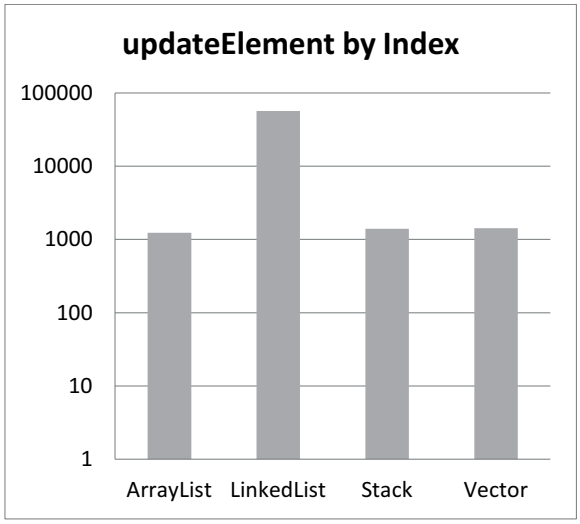

Abbildung 4.4: Benötigte Zeit in Millisekunden der getesteten Listen beim Iterieren mithilfe des Index am Testgerät HTC Dream (G1).

Iteration mit Iterator

Man kann sicherlich davon ausgehen, dass die *ArrayList* bei der Iteration der Elemente mithilfe eines Iterators langsamer als mit Index durchgegangen wird. Da zunächst ein Iterator erstellt wird, im Laufe der Iteration mit hasNext() überprüft werden muss ob es noch ein weiteres Element gibt und schließlich mit next() dieses Element geholt wird. *LinkedList* wird mit einem

Iterator sicherlich schneller sein als mit Index. Der *Stack* hingegen, könnte wie bei der *ArrayList* langsamer ausfallen, aber damit auf gleicher Höhe mit dem *Vector* sein.

ArrayList war wie erwartet mit Iterator langsamer als mit Index, gehört aber mit 1721,1 ms beim G1 und 320,099 ms beim Desire HD dennoch zu den schnelleren Listen und ist beim Desire HD auch die schnellste Liste (siehe Abbildung 4.6 und 4.5).

Die *LinkedList* hat wie erwartet deutlich bessere Ergebnisse als mit Index geliefert. Beim G1 hat sie mit 1523,59 ms den besten Wert erreicht. Beim Desire HD liegt die *LinkedList* mit 332,489 ms etwa mit der etwa *ArrayList* gleichauf. Anstatt eines *Iterators* wird bei der *LinkedList* ein *ListIterator* verwendet, der zusätzliche Funktionalität liefert, wie Beispielsweise Elemente an der aktuellen Stelle einfügen.

Der *Stack* (G1 2206,757 ms, Desie HD 504,944 ms), sowie der Vector (G1 2246,43 ms, Desire HD 505,829 ms) lieferten die schlechtesten Ergebnisse, wobei der *Stack* beim G1 etwas besser abschnitt als der *Vector*. Die ist womöglich wieder auf das *synchronized*-Schlüsselwort und damit auf die Thread-Sicherheit zurückzuführen.

Finden von Elementen mit Index

Da bei der *ArrayList* direkt auf die Position des Elements im internen Array zugegriffen wird, ist das Finden von Elementen anhand des Index sicherlich sehr schnell. Bei der *LinkedList* wird diese Aufgabe wahrscheinlich langsamer erfüllt werden, weil diese Liste eigentlich dem Konzept eines Index nicht entspricht. Auch der *Stack* ist im Grunde nicht auf einen Zugriff auf die Elemente mithilfe des Index ausgelegt. Der Vector könnte der *ArrayList* gegenüber vergleichbare Werte liefern, da lediglich ein Lesezugriff erfolgt und die Elemente nicht verändert werden.

Wie bereits erwartet, gehört die *ArrayList* mit 0,335 ms beim G1 und 0,092 ms beim Desire HD zu den schnellsten (siehe Abbildung 4.8 für das G1 und 4.7 für das Desire HD). Überraschenderweise ist sie beim Desire HD aber nicht die schnellste Liste, sondern knapp hinter dem *Vector*. Da der Abstand nur ca. 0,03 ms beträgt kann er aber vernachlässigt werden.

Die auffallend langsamste Liste mit 11,596 ms beim G1 und 4,517 ms beim Desire HD war die *LinkedList*. Wie bereits beim Abschnitt über die Iteration mit Index beschrieben 4.2.2, muss bei einer *LinkedList* beinahe die gesamte Liste durchsucht werden um das Element zu finden.

Eine schnellere Liste ist dagegen der *Stack* mit 0,702 ms beim G1 und 0,152 ms beim Desire HD. Ist aber etwas langsamer als *ArrayList* und *Vector*.

Der *Vector* ist die schnellste Liste beim Desire HD (0,061 ms) und beim G1 (0,336 ms)) knapp hinter der ArrayList.

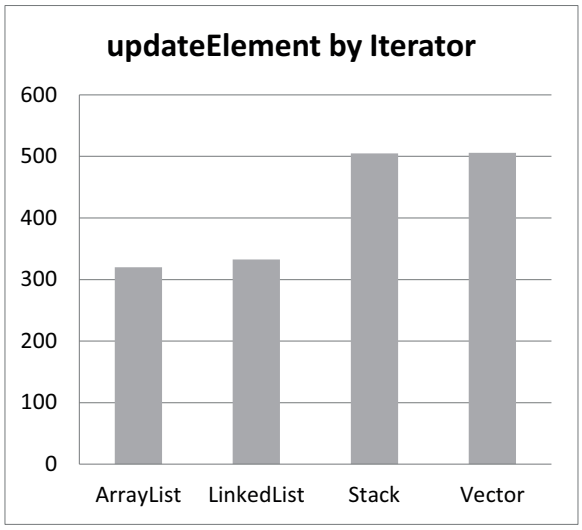

Abbildung 4.5: Benötigte Zeit in Millisekunden der getesteten Listen beim Iterieren mithilfe eines Iterators am Testgerät Desire HD.

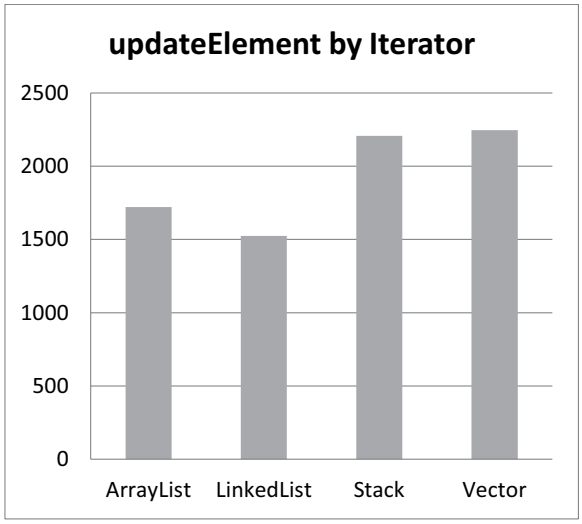

Abbildung 4.6: Benötigte Zeit in Millisekunden der getesteten Listen beim Iterieren mithilfe eines Iterators am Testgerät HTC Dream (G1).

Finden von Elementen mit Objektüberprüfung

Beim Finden eines Elements, indem das gesuchte Objekt verglichen wird, ist die *ArrayList* sicherlich langsamer als würde der Index des Objektes verwendet werden, da zunächst die Liste iteriert und jedes Objekt einzeln verglichen werden muss. Bei der *LinkedList* sind womöglich vergleichbare Werte zu erwarten wie der ArrayList, weil Iteration und Objektvergleich

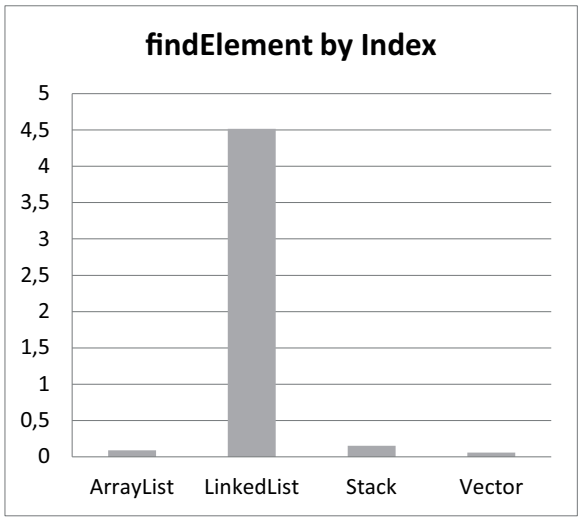

Abbildung 4.7: Benötigte Zeit in Millisekunden der getesteten Listen beim Finden von Elementen mit Angabe eines Index am Testgerät Desire HD.

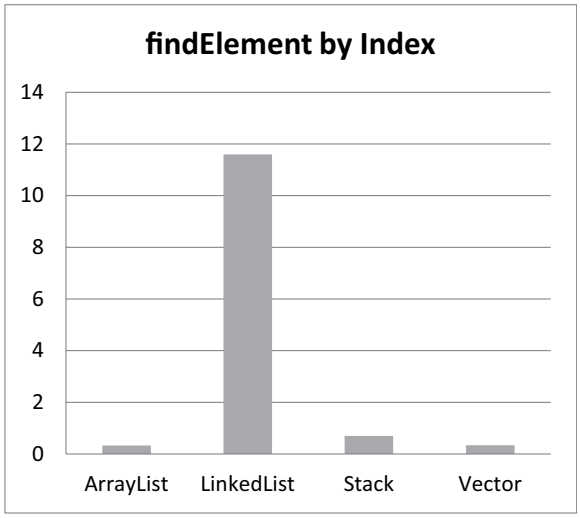

Abbildung 4.8: Benötigte Zeit in Millisekunden der getesteten Listen beim Finden von Elementen mit Angabe eines Index am Testgerät HTC Dream (G1).

sehr ähnlich sind. Auch beim *Stack* können sich zur *ArrayList* vergleichbare Werte ergeben, wobei Stack auch für diese Funktion nicht gedacht ist. Beim *Vector* sind wahrscheinlich ebenfalls ähnliche Werte wie bei der *ArrayList* zu erwarten, da *Vector* dieselbe Funktionalität wie *ArrayList* bietet.

Die schnellste Liste (siehe Abbildung 4.10 für das G1 und 4.9 für das Desire HD) beim Finden der Elemente durch Objektüberprüfung ist die

ArrayList mit 273,774 ms beim G1 und 70,282 ms beim Desire HD.

Auf etwa der gleichen Geschwindigkeit wie die *ArrayList* liegt die *LinkedList* (G11 271,912 ms, Desire HD 75,531 ms), wobei auf dem G1 etwas bessere Ergebnisse zu finden sind.

Stack (G1 555,054 ms, Desire HD 153,107 ms) und *Vector* (G1 597,2 ms, Desire HD 151,733 ms) liegen etwa auf gleicher Höhe und sind deutlich langsamer als *ArrayList* und *LinkedList*.

Element mit Index löschen

Wie bereits beim Finden und iterieren mithilfe eines Index, ist das Entfernen von Elementen bei der *ArrayList* ebenfalls sehr schnell, da auch hier über die direkte Position im internen Array zugegriffen wird. Der *Vector* wird langsam sein, wegen der Thread-Sicherheit. Die *LinkedList* wird wahrscheinlich langsamer sein, weil kein direkter Zugriff mit dem Index möglich ist. Der *Stack* wird Vergleichbar mit Vector sein.

Die schnellste Liste (G1 0,824 ms und Desire HD 0,183 ms) war die *ArrayList* (Abbildung 4.12 und 4.11). Wie in den Abschnitten „Iteration mit Index“ und „Finden von Elementen mit Index“ beschrieben, sind die guten Ergebnisse auf den direkten Zugriff des internen Arrays zurückzuführen.

LinkedList war mit 11,872 ms beim G1 und 4,364 ms beim Desire HD die langsamste Liste.

Stack lag in etwa auf gleicher Höhe wie die *ArrayList* (G1 0,763 ms und Desire HD 0,183 ms).

Der *Vector* war ebenfalls auf fast gleich schnell wie die *ArrayList* (G1 0,977 ms und Desire HD 0,152 ms).

Element mit Objektüberprüfung löschen

Die *ArrayList* wird sicherlich langsamer sein als beim Löschen mit Index, da wie beim Finden von Elementen zunächst eine Objektüberprüfung statt findet. Die *LinkedList* könnte ähnliche Resultate wie beim Finden des Objektes liefern, genau wie *Stack* und *Vector*.

Mit 103,881 ms beim G1 und 33,478 ms beim Desire HD war die *ArrayList* auch hier wieder die schnellste Liste (4.14, 4.13). Dabei werden nach dem durchsuchen der Liste nicht alle Elemente kopiert, sondern nur die welche nach der Index-Position des zu löschenden Elements stehen. Die Elemente werden ab der Position des gelöschten Elements kopiert.

Bei der *LinkedList*, welche die langsamste Liste war (G1 112,579 ms, Desire HD 101,501 ms) werden ebenfalls alle Elemente durchlaufen. Wird das Elemente gefunden werden die Verknüpfungen entfernt (*unlink(Node<E> x)*). Dabei wird das next-Element des zu löschenden Elements auf das des vorherigen gesetzt und das prev-Element des zu löschenden auf das des folgenden.

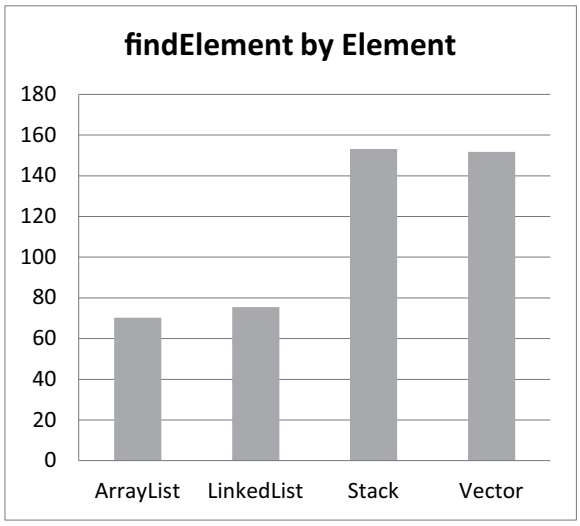

Abbildung 4.9: Benötigte Zeit in Millisekunden der getesteten Listen beim Finden von Elementen mithilfe von Objektüberprüfung am Testgerät Desire HD.

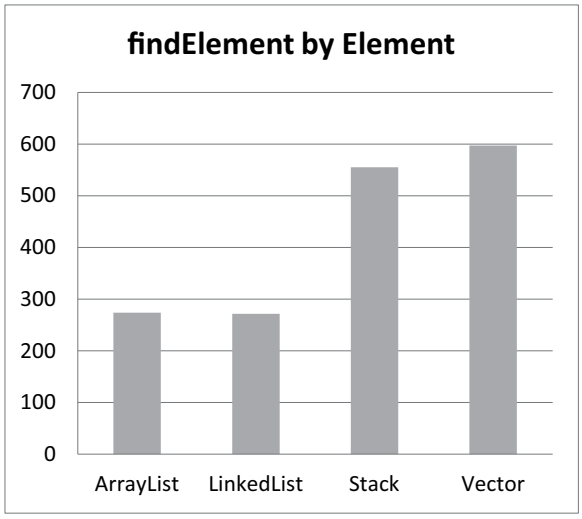

Abbildung 4.10: Benötigte Zeit in Millisekunden der getesteten Listen beim Finden von Elementen mithilfe von Objektüberprüfung am Testgerät HTC Dream (G1).

Beim Desire HD hatte der *Vector* etwa die gleichen Werte wie die *ArrayList* (34,729 ms), während er beim G1 (116,211 ms) deutlich langsamer war. Nach Betrachtung des Source-Codes konnte ich feststellen, dass auffällig viele Methodenaufrufe beim Entfernen eines Elements vorgenommen werden. Zusätzlich sind die Methoden *removeElement(Object obj)*, *indexOf(Object*

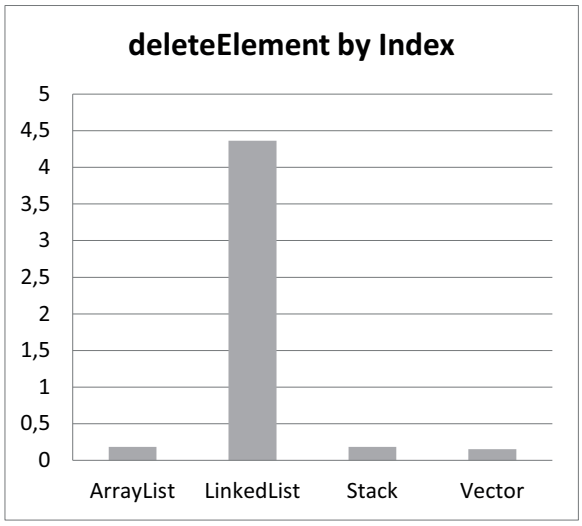

Abbildung 4.11: Benötigte Zeit in Millisekunden der getesteten Listen beim Entfernen von Elementen mit Index am Testgerät Desire HD.

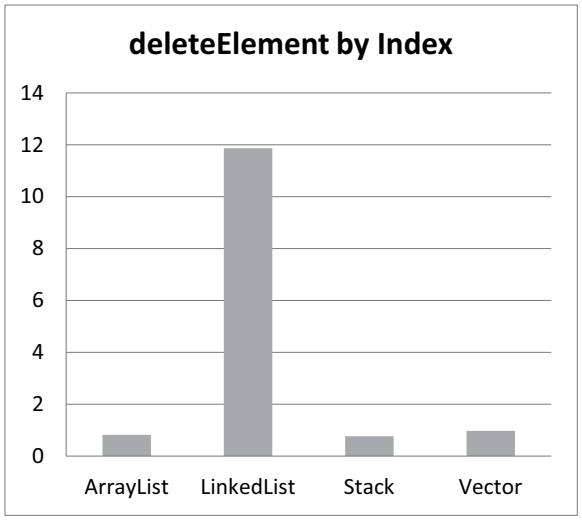

Abbildung 4.12: Benötigte Zeit in Millisekunden der getesteten Listen beim Entfernen von Elementen mit Index am Testgerät HTC Dream (G1).

o, int index) und *removeElementAt(int index)* alles *synchronized*, weshalb ein höherer Zeitaufwand möglich ist.

Das selbe Vorgehen ist beim *Stack* zu beobachten, nur das hier noch die Vererbung dazwischen liegt (G1 112,61 ms, Desire HD 34,76 ms).

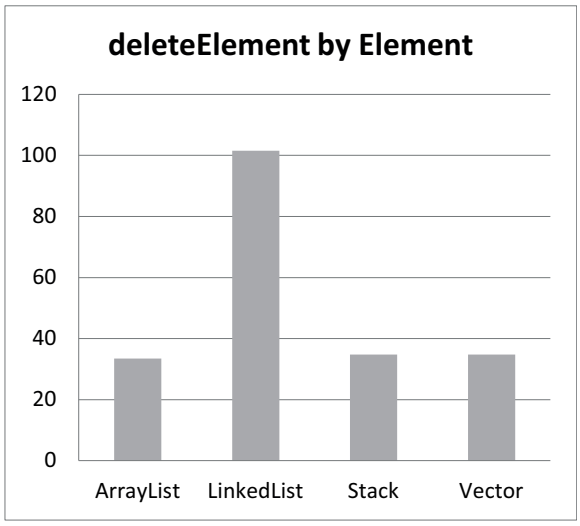

Abbildung 4.13: Benötigte Zeit in Millisekunden der getesteten Listen beim Entfernen von Elementen mit Objektüberprüfung am Testgerät Desire HD.

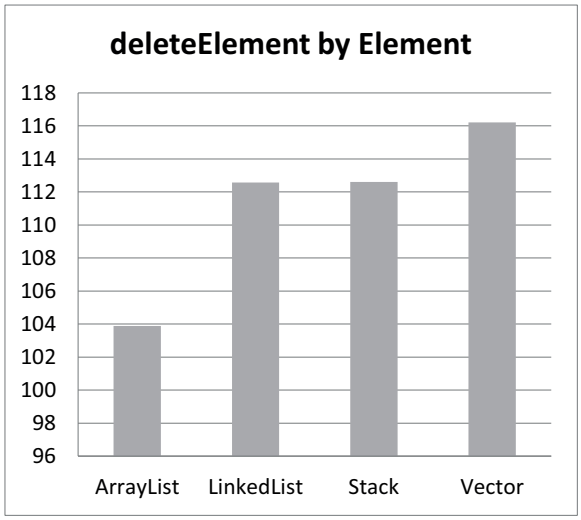

Abbildung 4.14: Benötigte Zeit in Millisekunden der getesteten Listen beim Entfernen von Elementen mit Objektüberprüfung am Testgerät HTC Dream (G1).

Löschen aller Elemente

Die Geschwindigkeit der *ArrayList* beim Entfernen aller Elemente ist sicherlich abhängig von der Anzahl der Elemente, je weniger Elemente desto schneller. Sollte aber im Vergleich zu den anderen Listen schnell sein, da lediglich ein Array iteriert wird und die Werte zurückgesetzt werden müssen. Die *LinkedList* sollte ebenfalls schnell sein, da die Verknüpfungen der Ele-

mente im Grunde egal sind und die Elemente der Liste nur auf *null* gesetzt werden müssen. Ähnliche Ergebnisse wie bei der *ArrayList* sollte der *Vector* und der *Stack* liefern.

ArrayList war bei den Tests die zweitschnellste Liste (G1 22,858 ms und Desire HD 8,667 ms, siehe Abbildung 4.16 und 4.15).

Die *LinkedList* war mit 0,397 ms beim G1 und 0,183 ms beim Desire HD mit Abstand die schnellste Liste beim Entfernen aller Einträge. Trotz der Vorgehensweise in der *clear()*-Methode, da dort sowohl der Wert des Elements auf *null* gesetzt wird, als auch die *next*- und *prev*-Verknüpfungen[2]

Der *Vector* hat ein ähnliches Vorgehen wie die *ArrayList*, nur das es einen zusätzlichen Methodenaufruf gibt (wenn *clear()* verwendet wird) und die Methode *removeAllElements() synchronized* ist (G1 49,774 ms, Desire HD 14,649 ms).

Der *Stack* hatte sehr ähnliche Werte wie der *Vector* aufgrund gleicher Vorgehensweise (G1 49,744 ms, Desire HD 14,648 ms).

Ergebnis

Die Arraylist lieferte die besten Ergebnisse beim Hinzufügen von Elementen, Iterieren mit Index und Iterator, beim Finden von Elementen mit Index und Objektvergleich als auch dem Löschen mit Index und Objektvergleich. Außer beim Entfernen von allen Elementen ist diese Liste daher für fast alle Funktionen am besten geeignet.

4.2.3 Maps

Bei den Folgenden Tests wurden die drei Map-Container *HashMap*, *HashTable*, *IdentityHashMap* und *Properties* getestet (siehe die Tabellen die Tabellen 4.5, 4.6, 4.3 und 4.4).

Elemente hinzufügen

Die *HashMap* lag mit 4072,602 ms beim G1 und 1034,76 ms beim Desire HD im Mittelfeld der getesteten Maps (siehe Abbildung 4.18 und 4.17). In der *put(K key, V value)*-Methode werden die bereits eingefügten Elemente überprüft ob der übergebene Schlüssel bereits enthalten ist und der Wert gegebenenfalls ersetzt werden muss. In dieser Schleife wird bei jedem Durchgang ein Objekt für die Schlüsselüberprüfung erstellt. Dieses Objekt wird sowohl mit *equals()* als auch mit „==“ verglichen. Diese Überprüfung kostet diesem Map etwas Zeit, im Gegensatz zur *HashTable*, bei der nur mit

[2]Nach dem Kommentar in Zeile 447 ist dies nicht erforderlich, bietet aber ein paar Vorteile:
„Clearing all of the links between nodes is “unnecessary”, but:
- helps a generational GC if the discarded nodes inhabit more than one generation
- is sure to free memory even if there is a reachable Iterator“ [26].

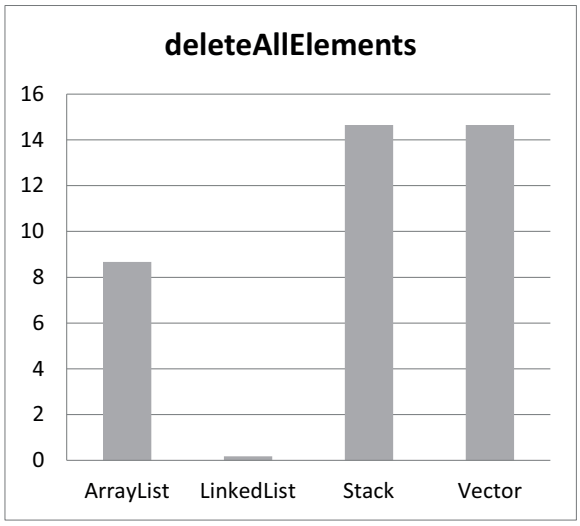

Abbildung 4.15: Benötigte Zeit in Millisekunden der getesteten Listen beim Entfernen aller Elemente am Testgerät Desire HD.

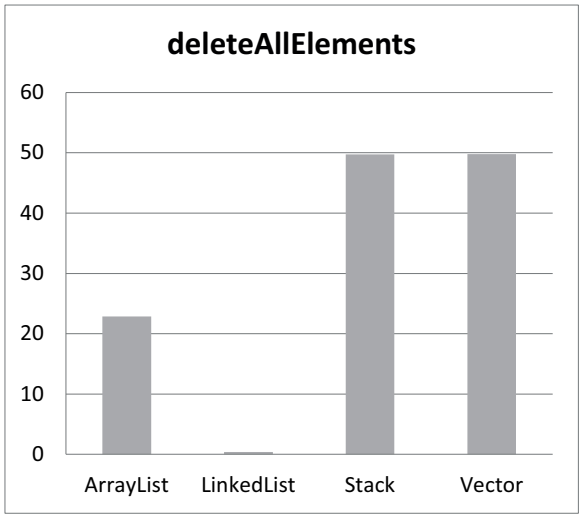

Abbildung 4.16: Benötigte Zeit in Millisekunden der getesteten Listen beim Entfernen aller Elemente am Testgerät HTC Dream (G1).

equals() verglichen wird. Für das eigentliche Einfügen wird die Methode *addEntry(int hash, K key, V value, int bucketIndex)* aufgerufen.

Mit 3945,709 ms beim G1 und 1000,823 ms beim Desire HD hatte die *HashTable* das beste Ergebnis beim Einfügen der Elemente, trotz *synchronized*. Im Gegensatz zur *HashMap* wird nicht in jedem Schleifendurchlauf ein neues Objekt erzeugt, dass für die Überprüfung notwendig ist, denn der *Key* wird nur mit *equals()* verglichen. Er muss daher mit *e.key* nur einmal

Tabelle 4.3: Benötigte Zeit in Millisekunden für verschiedene Funktionen die an Maps angewendet werden können beim Testgerät Desire HD.

Funktion	HashMap	HashTable	IdentityHashMap
addElement	1034,76	1000,823	1390,808
updateElement	569,366	673,218	1147,033
findElement by Key	0,153	0,152	0,183
findelement by Value	168,06	223,603	493,653
deleteElement by Key	0,122	0,092	0,122
deltetAllElements	17,579	17,486	47,455

Tabelle 4.4: Benötigte Zeit in Millisekunden für verschiedene Funktionen die an Maps angewendet werden können beim Testgerät Desire HD (2. Tabelle, Aufteilung auf zwei Tabellen Aufgrund von Problemen mit der Breite).

Funktion	LinkedHashMap	Properties
addElement	1169,953	1027,282
updateElement	547,516	661,988
findElement by Key	0,152	0,091
findelement by Value	169,22	221,832
deleteElement by Key	0,122	0,03
deltetAllElements	33,326	17,517

Tabelle 4.5: Benötigte Zeit in Millisekunden für verschiedene Funktionen die an Maps angewendet werden können beim Testgerät HTC Dream (G1).

Funktion	HashMap	HashTable	IdentityHashMap
addElement	4072,602	3945,709	5615,967
updateElement	2383,148	2844,177	4949,249
findElement by Key	0,366	0,671	0,428
findelement by Value	564,087	801,331	1741,242
deleteElement by Key	0,488	0,458	0,488
deltetAllElements	42,816	43,183	118,073

abgerufen werden. Das Einfügen eines Elements wird schließlich in der *put(K key, V value)*-Methode vorgenommen.

Das schlechteste Ergebnis beim Einfügen von Elementen lieferte die *IdentityHashMap* mit 5615,967 ms beim G1 und 1390,808 ms beim Desire HD.

Der Test ergab bei der *LinkedHashMap* das zweitschlechteste Ergebnis

Tabelle 4.6: Benötigte Zeit in Millisekunden für verschiedene Funktionen die an Maps angewendet werden können beim Testgerät HTC Dream (G1) (2. Tabelle, Aufteilung auf zwei Tabellen Aufgrund von Problemen mit der Breite).

Funktion	LinkedHashMap	Properties
addElement	4695,16	3847,046
updateElement	2347,32	2868,408
findElement by Key	0,428	0,397
findelement by Value	589,477	793,824
deleteElement by Key	0,458	0,397
deltetAllElements	78,797	42,908

(G1 4695,16 ms, Desire HD 1169,953 ms). Als *put(K key, V value)*-Methode wird die Vererbte von *HashMap* verwendet.

Das zweitbeste Ergebnis hingegen ergab sich bei *Properties* (G1 3847,046 ms, Desire HD 1027,282 ms). Verwendet wird hier die Vererbte *put(K key, V value)*-Methode von *HashTable*.

Elemente iterieren

Beim iterieren der Elemente und dem damit verbundenen Aufruf der *update()*-Methode der Testelemente, hatte die *HashMap* das zweitbeste Ergebnis mit 2383,148 ms beim G1 und 569,366 ms beim Desire HD (Abbildung 4.20 und 4.19).

Das zweitschlechteste Ergebnis, aber damit im Mittelfeld liegend, lieferte *HashTable* (G1 2844,177 ms, Desire HD 673,218 ms), was auf *synchronized* zurückzuführen ist.

IdentityHashMap lieferte bei den Tests das schlechteste Ergebnis mit 4949,249 ms beim G1 und 1147,033 ms beim Desire HD.

Im Gegensatz dazu gab die *LinkedHashMap* das beste Ergebnis (G1 2347,32 ms, Desire HD 547,516 ms). Im Gegensatz zu der *HashMap*, welche die *get()*-Methode von *AbstractMap* verwendet, besitzt die *LinkedHashMap* eine eigene Implementierung von *get()*.

Properties war mit 2868,408 ms beim G1 und 661,988 ms beim Desire HD in etwa auf der Höhe der *Hashtable*, nur etwas besser.

Element mit Key finden

Das Finden von Elementen durch den Key ist bei der *HashMap* sehr schnell. Beim G1 konnte mit 0,366 ms das Beste Ergebnis erzielt werden (siehe Abbildung 4.22 für das G1 und 4.21), während er beim Desire HD mit 0,153 ms etwa auf der gleichen Höhe wie *HashTable* und *LinkedHashMap* lag.

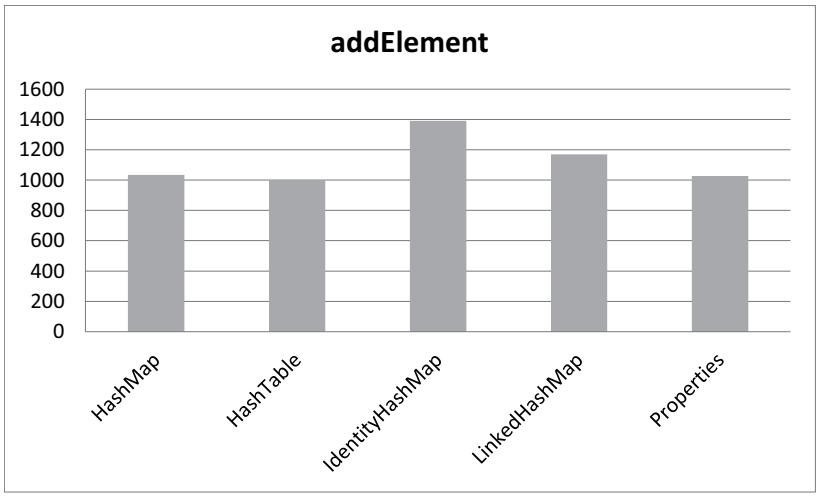

Abbildung 4.17: Benötigte Zeit in Millisekunden der getesteten Maps beim Einfügen von Elementen mit dem Testgerät Desire HD.

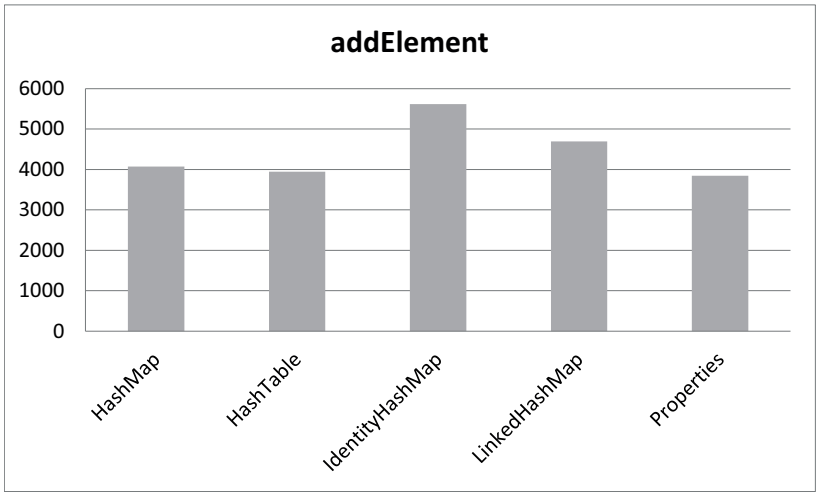

Abbildung 4.18: Benötigte Zeit in Millisekunden der getesteten Maps beim Einfügen von Elementen mit dem Testgerät HTC Dream (G1).

Die *HashTable* lieferte beim G1 mit 0,671 ms das schlechteste Ergebnis, was womöglich an dem *synchronized*-Befehl liegt, was bei einem leistungsschwachen Gerät stärker ins Gewicht fällt. Denn beim Desire HD lag die *HashTable* gleichauf mit der *HashMap* und der *LinkedHashMap*.

Beim G1, zusammen mit der *LinkedHashMap* (0,428 ms), lag die *IdentityHashMap* im Mittelfeld mit 0,428 ms. Etwas schlechter als die anderen Maps war die *IdentityHashMap* beim Desire HD (0,183 ms), was an die häufigere Objekt Zuordnung im Gegensatz zur *HashMap* und *HashTable* zu-

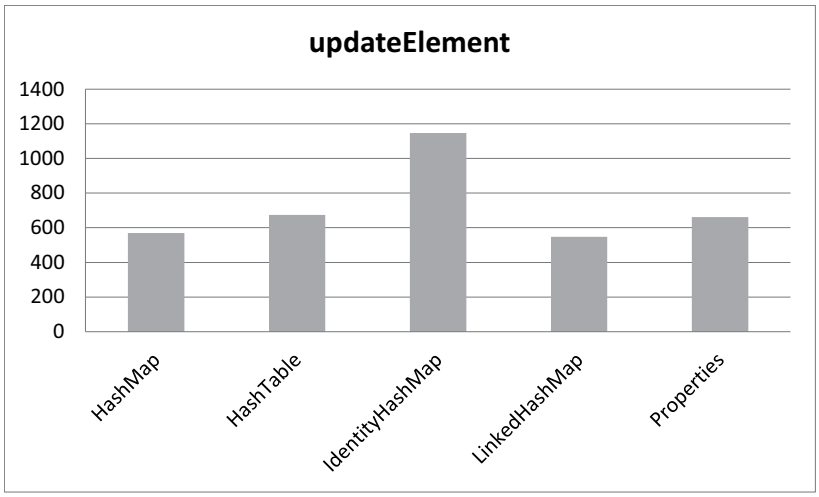

Abbildung 4.19: Benötigte Zeit in Millisekunden der getesteten Maps beim Iterieren der Elemente mit dem Testgerät Desire HD.

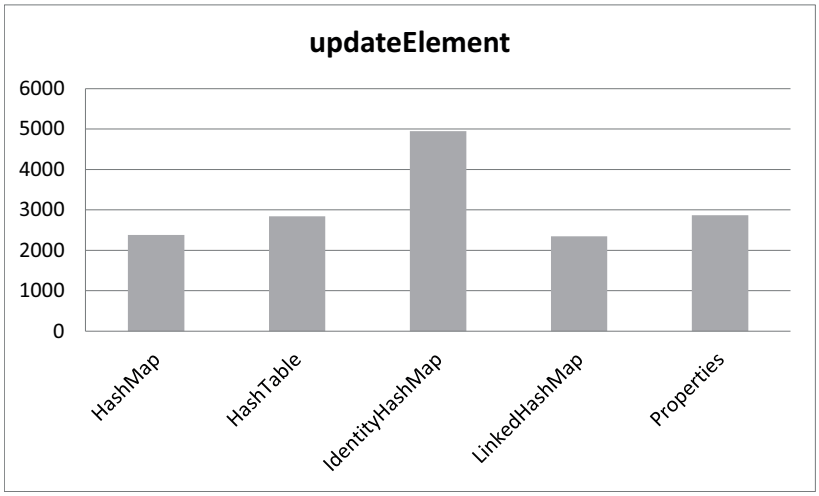

Abbildung 4.20: Benötigte Zeit in Millisekunden der getesteten Maps beim Iterieren der Elemente mit dem Testgerät HTC Dream (G1).

rückgeführt werden kann. Die *LinkedHashMap* lag beim Desire HD bei 0,152 ms zusammen mit *HashMap* und *HashTable* ebenfalls im Mittelfeld. Es verwendet im Grunde das gleiche Verfahren zum Finden der Elemente wie bei der *HashMap*, da aber noch ein Aufruf der vererbten *getEntry()*-Methode vorgenommen wird, gibt es keine genaue Übereinstimmung mit der *HashMap*

Das beste Ergebnis beim Desire HD liefert *Properties* mit 0,091 ms. Beim G1 war es die zweitbeste Map mit 0,397 ms.

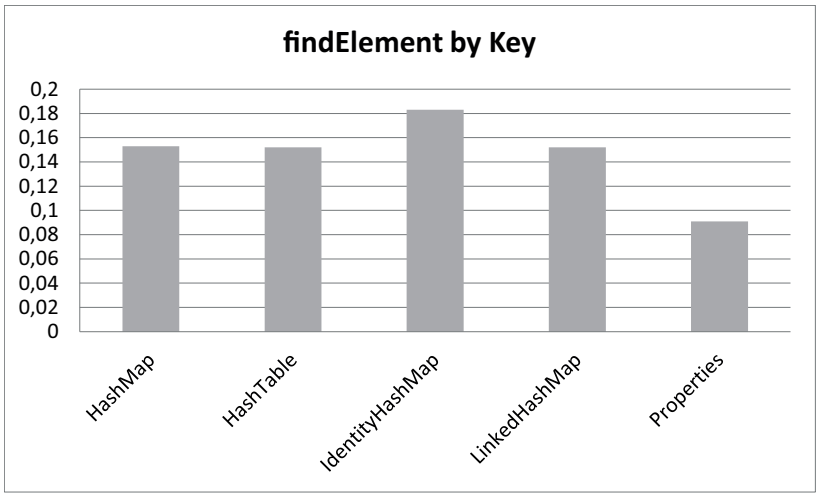

Abbildung 4.21: Benötigte Zeit in Millisekunden der getesteten Maps beim Finden von einem Element durch Verwendung eines Keys mit dem Testgerät Desire HD.

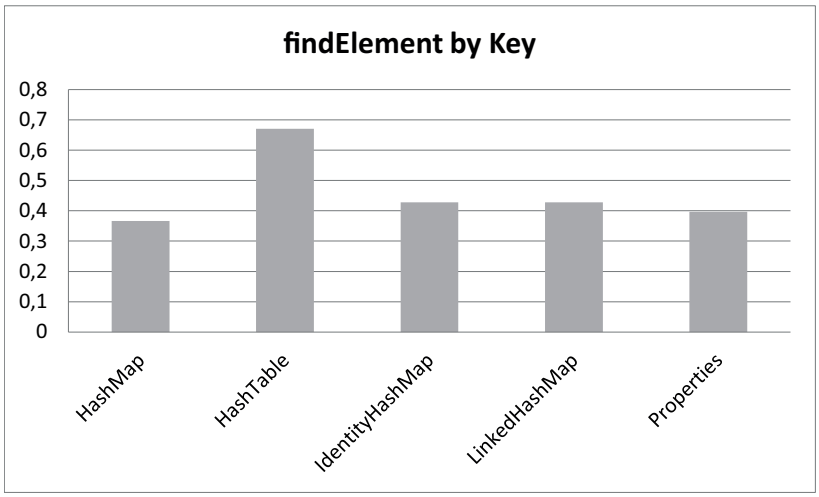

Abbildung 4.22: Benötigte Zeit in Millisekunden der getesteten Maps beim Finden von einem Element durch Verwendung eines Keys mit dem Testgerät HTC Dream (G1).

Element mit Value finden

Hier gab die *HashMap* das beste Ergebnis (4.24 und 4.23) beim G1 mit 564,087 ms und 168,06 ms beim Desire HD.

Ein schlechteres Ergebnis gab die *HashTable* mit 801,331 ms beim G1 und 223,603 ms beim Desire HD. War aber deutlich besser als *IdentityHashMap* und etwa gleichauf mit *Properties*.

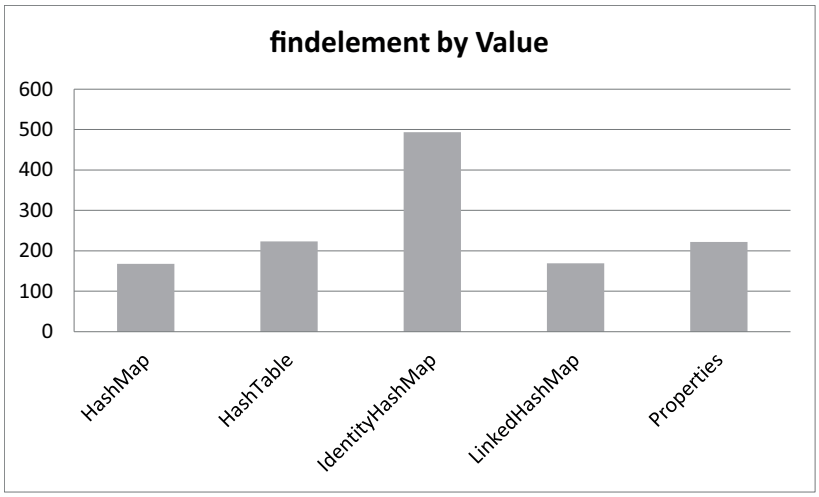

Abbildung 4.23: Benötigte Zeit in Millisekunden der getesteten Maps beim Finden von einem Element mithilfe des Wertes mit dem Testgerät Desire HD.

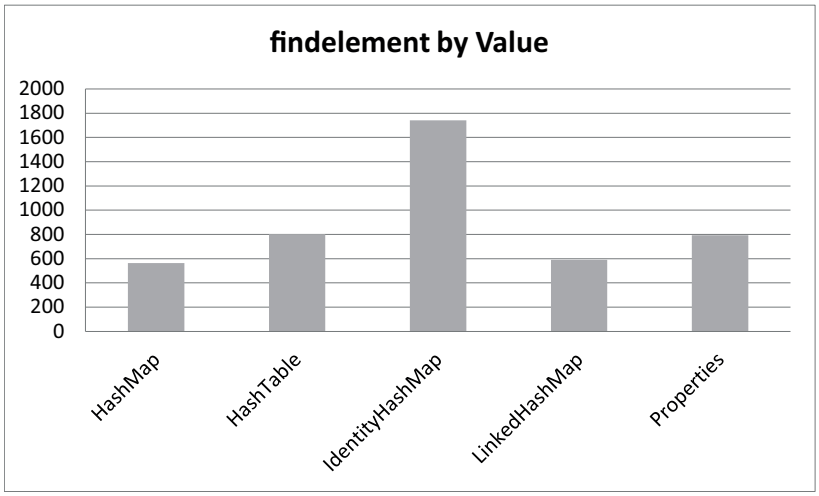

Abbildung 4.24: Benötigte Zeit in Millisekunden der getesteten Maps beim Finden von einem Element mithilfe des Wertes mit dem Testgerät HTC Dream (G1).

Das schlechteste Ergebnis liefert die *IdentityHashMap* mit 1741,242 ms beim G1 und 493,653 ms beim Desire HD.

Vergleichbare Ergebniss zur *HashMap* lieferte die *LinkedHashMap* mit 589,477 ms beim G1 und 169,22 ms beim Desire HD.

Etwa auf der Höhe des *HashTable* lag *Properties* mit 793,824 ms beim G1 und 221,832 ms beim Desire HD.

Element mit Key löschen

In Abbildung 4.26 für das G1 und 4.25 für das Desire HD lieferte die *HashMap* zusammen mit *LinkedHashMap* beim Desire HD das schlechteste Ergebnis mit 0,122 ms. Auch beim G1 lieferte es das schlechteste Ergebnis, war aber auf der Höhe der *IdentityHashMap*. Das Ergebnis kann darauf zugeführt werden, dass neben der Überprüfung mit „==“ und *equals* in jeder Abfrage noch eine Zuweisung vorgenommen wird.

Zweitbeste Ergebnis lieferte die *HashTable* beim G1 mit 0,458 ms und beim Desire HD mit 0,092 ms. Dies liegt daran, dass weniger Abfragen als bei der *HashMap* vorgenommen werden.

Zu den schlechteren Ergebnissen führte die *IdentityHashMap* (G1 0,488 ms und Desire HD 0,122 ms), da der Aufruf der Methode *closeDeletion()* noch zusätzliche Zeit benötigt.

Die *LinkedHashMap* lag auf der Höhe der *HashMap* (G1 0,458 ms und Desire HD 0,122 ms), da sie dieselbe *remove()*-Methode verwendet.

Das beste Ergebnis beim Desire HD (0,03 ms und 0,397 ms beim G1) kam bei *Properties* heraus, welches dieselbe *remove()*-Methode wie *HashTable* verwendet.

Alle Elemente löschen

HashMap hatte zusammen mit *HashTable* und *Properties* das beste Ergebnis (G1 42,816 ms und Desire HD 17,579 ms, siehe Abbildung 4.28 und 4.27). Das Verfahren beim Entfernen aller Elemente ist dem der *HashTable* sehr ähnlich, daher ergeben sich fast gleich Ergebnisse.

HashTable hatte mit 43,183 ms beim G1 und 17,486 ms beim Desire HD wie bereits genannt zusammen mit *HashMap* und *Properties* das beste Ergebnis.

IdentityHashMap lieferte das schlechteste Ergebnis beim G1 mit 118,073 ms und Desire HD 47,455 ms. Es wird zwar das gleich Verfahren wie bei der *HashMap* angewendet, anstatt eines Entry-Arrays wird aber ein Object-Array verwendet.

Zweitschlechtestes Ergebnis lieferte *LinkedHashMap* mit 78,797 ms beim G1 und 33,326 ms beim Desire HD. Diese Map verwendet die *clear()*-Methode der *HashMap* und setzt danach das *vorherige* und *nachfolgende* Element auf das Head-Element der verketten Liste.

Properties lieferte mit 42,908 ms beim G1 und 17,517 ms beim Desire HD vergleichbare Ergebnisse wie *HashMap* und *HashTable*. Es verwendet die *clear()*-Methode der *HashTable*, weshalb die Ergebnisse sehr ähnlich sind.

Ergebnis

Beim Hinzufügen von Elementen waren die Ergebnisse bei der HashMap, der HashTable und Properties sehr ähnlich. Abhängig davon welche Funkti-

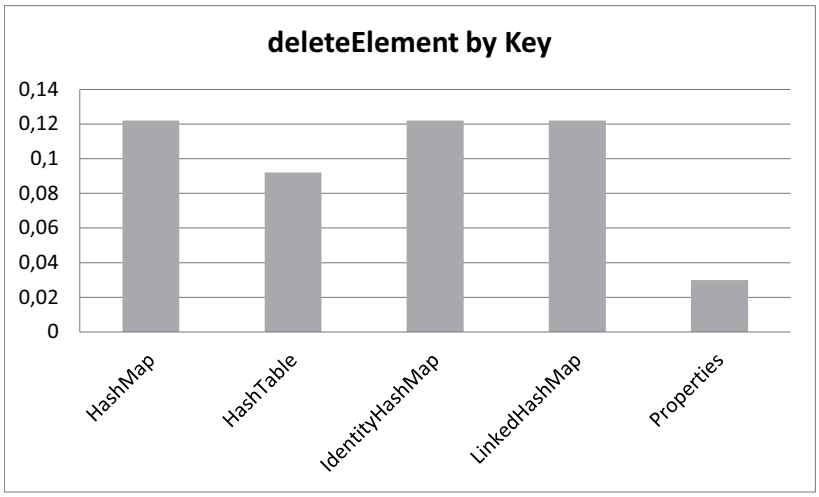

Abbildung 4.25: Benötigte Zeit in Millisekunden der getesteten Maps beim Löschen eines Elements mithilfe des Keys bei dem Testgerät Desire HD.

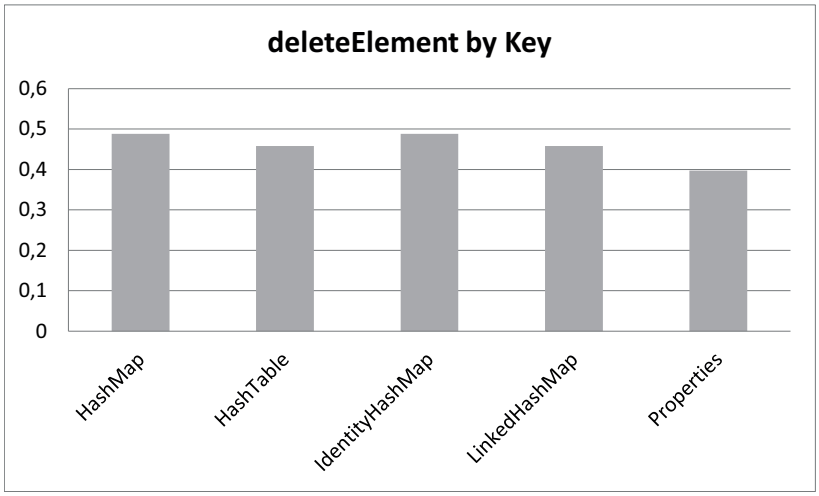

Abbildung 4.26: Benötigte Zeit in Millisekunden der getesteten Maps beim Löschen eines Elements mithilfe des Keys bei dem Testgerät HTC Dream (G1).

on der jeweiligen Map benötigt wird, sollte eine dieser drei bevorzugt werden. Auch beim Iterieren ergaben sich ähnliche Werte, wobei die HashMap besser abschnitt. Beim Finden von Elementen tat sich Properties hervor, wobei bei mehrmaliger Suche die HashMap bessere Ergebnisse lieferte. Sollen alle Elemente aus der Map möglichst schnell entfernt werden, so empfiehlt es sich die HashMap, HashTable oder Properties zu verwenden, selbstverständlich Abhängig davon, welche Funktionsweise notwendig ist. Die IdentityHashMap ist im Gegensatz zu den anderen Maps deutlich langsamer. Wenn auf

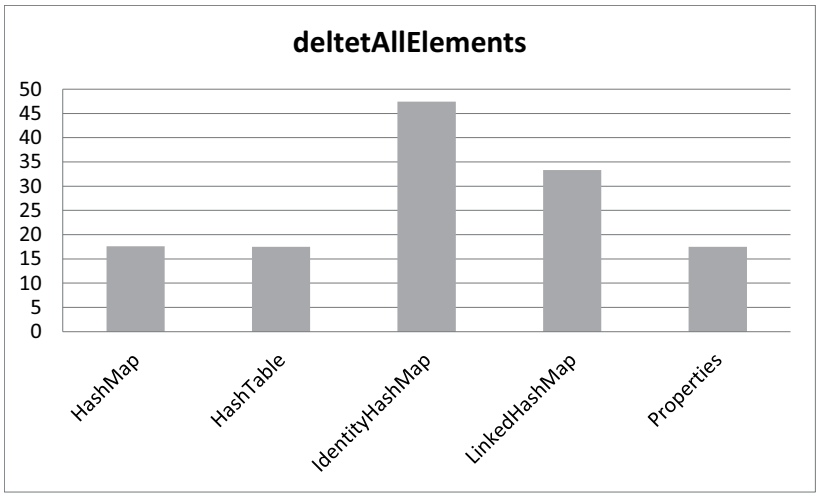

Abbildung 4.27: Benötigte Zeit in Millisekunden der getesteten Maps beim Löschen aller Elemente bei dem Testgerät Desire HD.

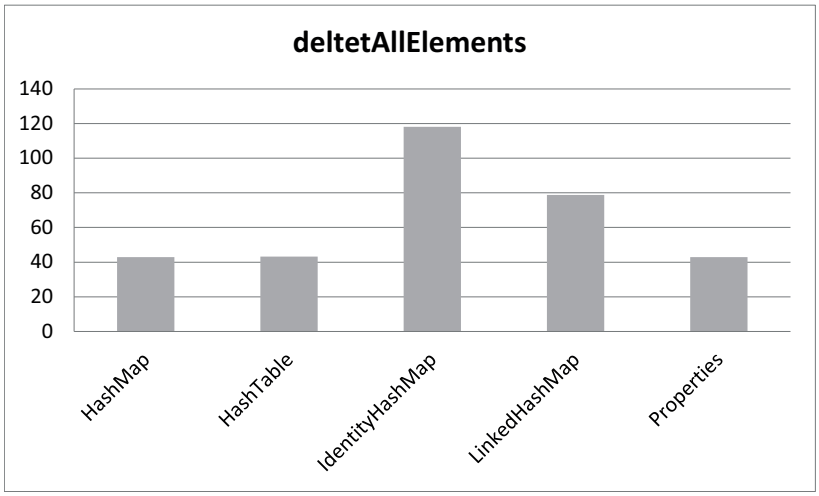

Abbildung 4.28: Benötigte Zeit in Millisekunden der getesteten Maps beim Löschen aller Elemente bei dem Testgerät HTC Dream (G1).

ein Identitätsvergleich der Elemente verzichtet werden kann, so sollte diese IdentityHahsMap auf jeden Fall vermieden werden.

4.2.4 Sets

Im folgenden Abschnitt finden sich die Auswertungen der Test mit den beiden Set-Typen *HashSet* und *LinkedHashSet* (siehe die Tabellen 4.8 und 4.7).

Tabelle 4.7: Benötigte Zeit in Millisekunden für verschiedene Funktionen die an Sets angewendet werden können beim Testgerät Desire HD.

Funktion	HashSet	LinkedHashSet
addElement	932,342	1040,771
updateElement	429,413	392,547
findElement	222,747	191,254
deleteElement	0,153	0,153
deltetAllElements	17,67	32,623

Tabelle 4.8: Benötigte Zeit in Millisekunden für verschiedene Funktionen die an Sets angewendet werden können beim Testgerät HTC Dream bzw. G1.

Funktion	HashSet	LinkedHashSet
addElement	3621,246	4177,368
updateElement	1782,776	1711,456
findElement	795,715	695,007
deleteElement	0,58	0,611
deltetAllElements	42,664	80,505

Elemente hinzufügen

Beim Einfügen der Elemente ist *HashSet* schneller (3621,246 ms beim G1 und 934,174 ms Desire Hd) als die *LinkedHashSet* (4177,368 ms beim G1 und 1054,535 ms beim Desire HD), da die *LinkedHashMap* zusätzlich noch eine Verknüpfung zu vorherigen und nächsten Element des eingefügten Elements erstellen muss. Damit kann später beim Aufruf der Inhalte auf die Reihenfolge der Erstellung zurückgegriffen werden (Abbildung 4.29 und 4.30).

Elemente iterieren

Da bei einer *HashSet* intern eine HashMap verwendet wird und bei einer *LinkedHashSet* eine LinkedHashMap, konnten vergleichbare Ergebnisse wie bei der Iteration bei den jeweiligen Maps beobachtet werden. Für die *LinkedHashSet* konnten bessere Werte erzielt werden (G1 1711,456 ms, Desire HD 392,426 ms) als bei der *HashSet* (G1 1782,776 ms, Desire HD 420,105 ms) wie in Abbildung 4.31 und 4.32 zu sehen ist.

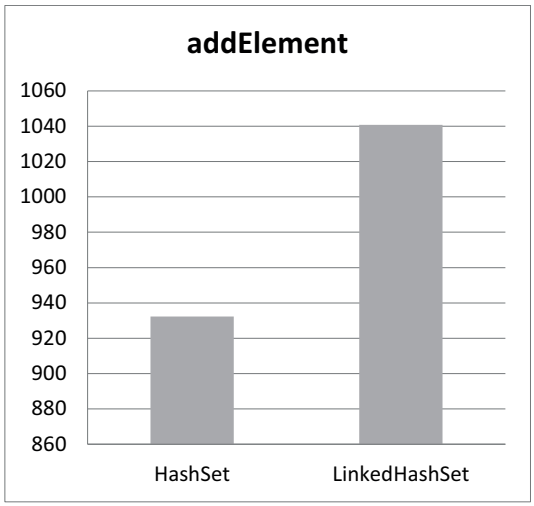

Abbildung 4.29: Benötigte Zeit in Millisekunden der getesteten Sets beim Hinzufügen von Elementen bei dem Testgerät Desire HD.

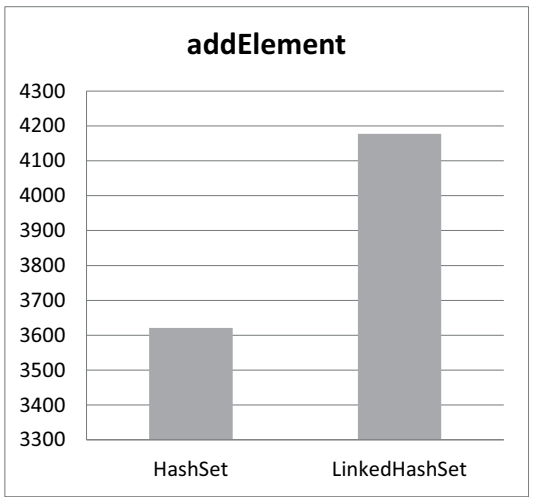

Abbildung 4.30: Benötigte Zeit in Millisekunden der getesteten Sets beim Hinzufügen von Elementen bei dem Testgerät HTC Dream (G1).

Element finden

Auch beim Finden von Elementen wird letztlich nur der Container iteriert, was zu demselben Ergebnis wie beim vorherigen Punkt führt. Beim G1 mit 695,007 ms und 191,864 ms beim Desire HD ist die *LinkedHashSet* schneller als die *HashSet* mit 795,715 ms beim G1 und 221,71 ms beim Desire HD (siehe Abbildung 4.33 und 4.34.

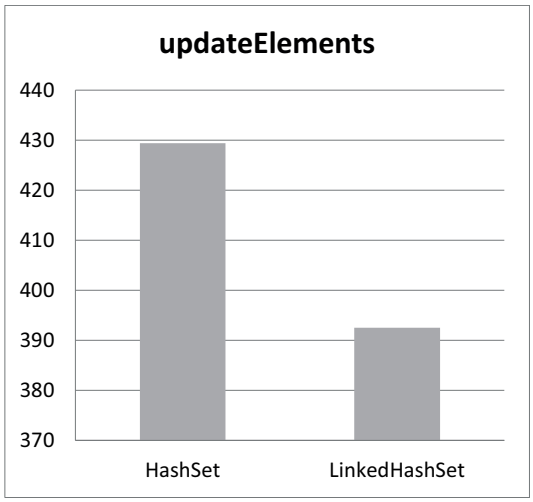

Abbildung 4.31: Benötigte Zeit in Millisekunden der getesteten Sets beim Iterieren der Elemente mit dem Testgerät Desire HD.

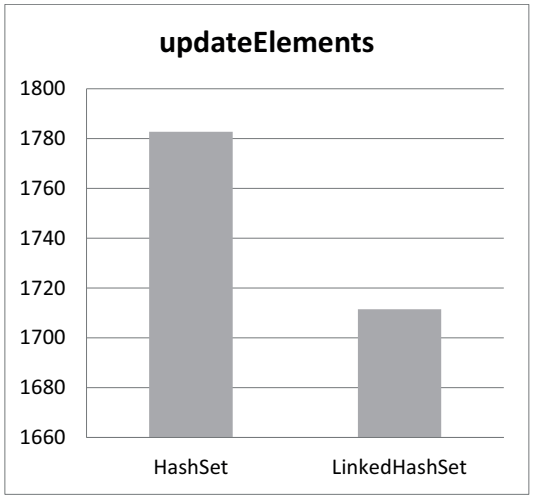

Abbildung 4.32: Benötigte Zeit in Millisekunden der getesteten Sets beim Iterieren der Elemente mit dem Testgerät HTC Dream (G1).

Element löschen

Bei Desire HD waren die Zeiten für das Löschen eines Elements bei *HashSet* und *LinkedHashSet* gleich (jeweils 0,153 ms). HashSet war beim G1 (0,58 ms) etwas besser (0.611 ms) wie in Abbildung 4.35 und 4.36 zu sehen ist.

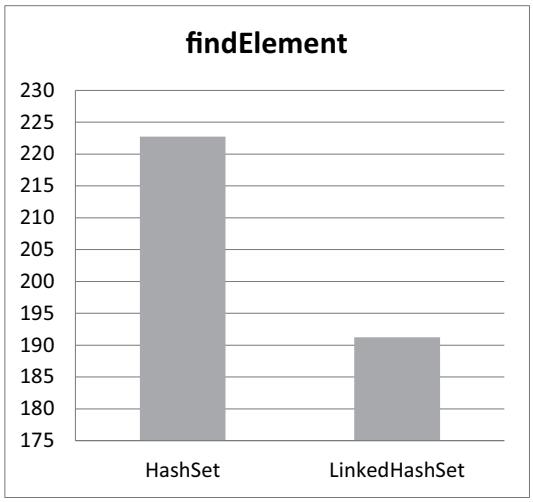

Abbildung 4.33: Benötigte Zeit in Millisekunden der getesteten Sets beim Finden eines Elements mit dem Testgerät Desire HD.

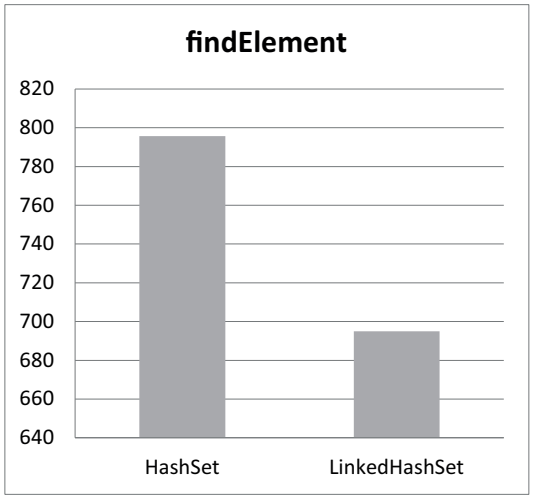

Abbildung 4.34: Benötigte Zeit in Millisekunden der getesteten Sets beim Finden eines Elements mit dem Testgerät HTC Dream (G1).

Alle Elemente löschen

Beim entfernen alle Elemente aus dem Container ist die *HashSet* (G1 42,664 ms, Desire HD 17,334 ms) schneller als die *LinkedHashSet* (G1 80,505 ms, Desire HD 42,664 ms), da bei der *LinkedHashSet* zusätzlich zum entfernen des Elements noch die Verknüpfungen zurückgesetzt werden (Abbildung 4.37 und 4.38).

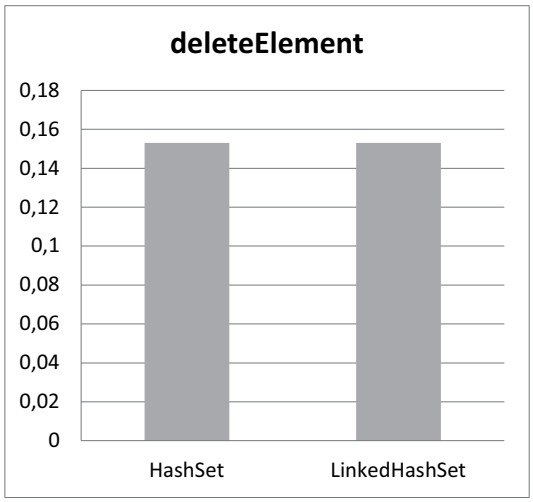

Abbildung 4.35: Benötigte Zeit in Millisekunden der getesteten Sets beim Löschen eines Elements mit dem Testgerät Desire HD.

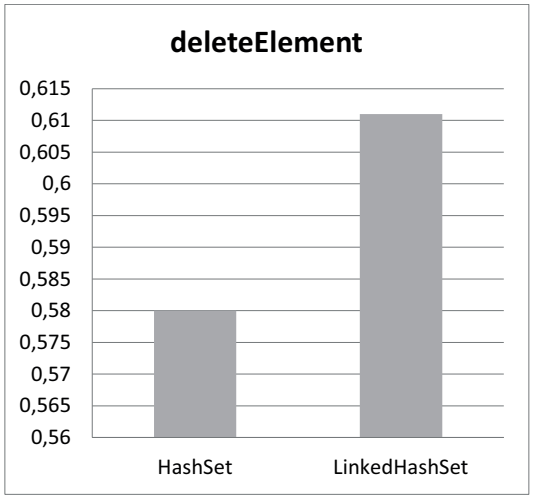

Abbildung 4.36: Benötigte Zeit in Millisekunden der getesteten Sets beim Löschen eines Elements mit dem Testgerät HTC Dream (G1).

Ergebnis

Da bei einer *LinkedHashSet* eine *LinkedHashMap* verwendet wird, ist das iterieren des Containers im Vergleich schneller als die *HashMap*, ähnlich wie bei den Map Ergebnissen. Dafür ist das Entfernen von Elementen schneller.

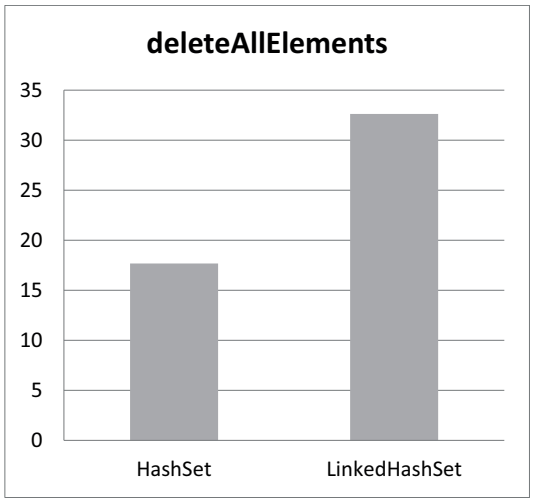

Abbildung 4.37: Benötigte Zeit in Millisekunden der getesteten Sets beim Löschen aller Elemente mit dem Testgerät Desire HD.

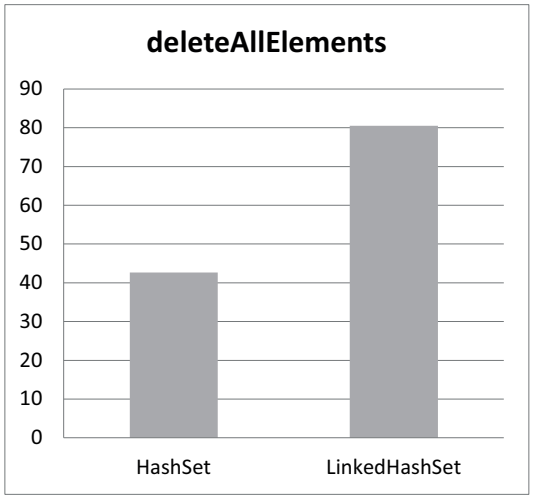

Abbildung 4.38: Benötigte Zeit in Millisekunden der getesteten Sets beim Löschen aller Elemente mit dem Testgerät HTC Dream (G1).

4.2.5 Typübergreifender Vergleich

Nun werden verschiedene Aufgaben wie das Hinzufügen, Iterieren, Finden und Löschen von Elementen bei den vorher untersuchten Container Typen miteinander verglichen. Die Ergebnisse können in den Tabellen 4.10 für das G1 und 4.9 für das Desire HD detailliert betrachtet werden.

Tabelle 4.9: Benötigte Zeit in Millisekunden für verschiedene Funktionen bei der Bag und dem Array bei dem Testgerät Desire HD.

Funktion	Bag	Array
addElement	64548,493	489,35
updateElement	326,11	180,472
findElement	9,338	0,061
deleteElement	42,175	0,122
deltetAllElements	1847,29	15,197

Tabelle 4.10: Benötigte Zeit in Millisekunden für verschiedene Funktionen bei der Bag und dem Array bei dem Testgerät G1.

Funktion	Bag	Array
addElement	150845,855	1650,482
updateElement	1380,402	882,263
findElement	21,545	0,305
deleteElement	128,845	0,366
deltetAllElements	34,546	4962,371

Hinzufügen von Elementen

Die besten Zeiten beim Hinzufügen von Elementen lieferte das Array mit mit 1650,482 beim G1 und 489,35 ms beim Desire HD. Danach folgt die ArrayList mit 568,359 beim Desire HD und 2064,605 beim G1.

Iterieren

Beim Iterieren durch die Elemente im Container ist das Array ebenfalls am schnellsten (G1 882,263 ms , Desire HD 180,572 ms), wiederum gefolgt von ArrayList (G1 1234,558 ms, Desrie HD 234,925 ms).

Finden eines Elements

Soll ein Element im Container gefunden werden, ist ebenfalls das Array mit 0,305 beim G1 und 0,061 beim Desire HD zu empfehlen. Diesmal dicht gefolgt von Vector (G1 0,336 ms, Desire HD 0,061 ms) und, etwas langsamer als Vector, dennoch schneller als die restlichen getesteten Container, die ArrayList (G1 0,335 ms, Desire HD 0,092 ms).

Tabelle 4.11: Durchschnittlich aufgebrachte Zeit für Garbage Collection mit verschiedenen Schleifen bei 10 000 Aufrufen am Testgerät Desire HD.

Container	for	for lokal var	foreach	while	while lokal var
Array	0	0	0	0	0
ArrayList	0	0	1	7	-
HashMap	-	-	2	7	-

Löschen eines Elements

Um ein Element innerhalb des Containers zu löschen, ist beim Desire HD Properties (Desire HD 0,03 ms) am schnellsten gewesen, gefolgt von Hash-Table (Desire HD 0,092 ms), HashMap, IdentityHashMap und dem Array (alle drei beim Desire HD 0,122 ms). Beim G1 war jedoch das Array mit 0,366 ms am schnellsten und wurde von Properties (G1 0,397 ms) gefolgt.

Löschen aller Elemente

Beim Entfernen aller Elemente aus dem Container, war die LinkedList (G1 0,397 ms, Desire HD 0,183 ms) bei weitem am schnellsten. Gefolgt von der ArrayList und dem Array mit 34,546 ms beim G1 und 15,197 ms beim Desire HD.

Ergebnis

Sollten keine besonderen Kriterien für Elemente eines Containes gelten, wie beispielsweise dass ein Element nur einmal in der Liste vorkommen darf, dann ist ein einfaches Array zu empfehlen. Auch die ArrayList bietet eine schnelle alternative, sollte sich die Menge des Inhalts ändern.

4.2.6 Schleifen

Bei den Performancetests der verschiedenen Schleifen, wurde die verbrauchte Zeit in der eine Garbage Collection vorgenommen wurde gemessen. Dadurch konnte bei unterschiedlichen Iterationen beobachtet werden, welche Schleife durch eben diese Garbage Collection den Spielfluss beeinflussen kann. Dabei wurden die einzelnen Schleifen jeweils 10 000 (4.11 und 4.12), 100 000 (4.13 und 4.14) und 1 000 000 mal (4.15 und 4.16) aufgerufen und 5 Elemente eines Containers damit iteriert. Als Container wurden Array, ArrayList und HashMap verwendet.

Tabelle 4.12: Durchschnittlich aufgebrachte Zeit für Garbage Collection mit verschiedenen Schleifen bei 10 000 Aufrufen am Testgerät G1.

Container	for	for lokal var	foreach	while	while lokal var
Array	0	0	0	0	0
ArrayList	0	0	0	0	-
HashMap	-	-	0	0	-

Tabelle 4.13: Durchschnittlich aufgebrachte Zeit für Garbage Collection mit verschiedenen Schleifen bei 100 000 Aufrufen am Testgerät Desire HD.

Container	for	for lokal var	foreach	while	while lokal var
Array	0	0	0	0	0
ArrayList	0	0	31	25	-
HashMap	-	-	39	34	-

Tabelle 4.14: Durchschnittlich aufgebrachte Zeit für Garbage Collection mit verschiedenen Schleifen bei 100 000 Aufrufen am Testgerät G1.

Container	for	for lokal var	foreach	while	while lokal var
Array	0	0	0	0	0
ArrayList	0	0	719	701	-
HashMap	-	-	689	667	-

Tabelle 4.15: Durchschnittlich aufgebrachte Zeit für Garbage Collection mit verschiedenen Schleifen bei 1 000 000 Aufrufen am Testgerät Desire HD.

Container	for	for lokal var	foreach	while	while lokal var
Array	0	0	0	0	0
ArrayList	0	0	310	281	-
HashMap	-	-	349	378	-

Tabelle 4.16: Durchschnittlich aufgebrachte Zeit für Garbage Collection mit verschiedenen Schleifen bei 1 000 000 Aufrufen am Testgerät G1.

Container	for	for lokal var	foreach	while	while lokal var
Array	0	0	0	0	0
ArrayList	0	0	7050	7092	-
HashMap	-	-	7304	7279	-

for-Schleife

Wie bei der klassischen for-Schleife üblich wurde eine Variable i für den beginn der Iteration auf 0 gesetzt und bei jeder Iteration um eins erhöht. Bei jedem Durchlauf wird die Länge des Containers abgefragt und schließlich mit i verglichen. Was hier die Geschwindigkeit zum Teil beeinträchtigt ist die wiederholte Abfrage der Länge des Containers.

```
1  for(int i = 0; i < mContainer.length; i++) {
2      mContainer[i].update();
3  }
```

Bei den Tests konnte festgestellt werden, dass durch die for-Schleife keine Garbage Collection hervorgerufen wurde. Diese Schleife ist daher für Performancekritische Situationen zu empfehlen. Nachteil ist aber, dass diese Art der Anwendung der for-Schleife nicht bei einer Map verwendet werden kann.

for-Schleife mit lokalen Variablen

Die for-Schleife ist bereits gut, sie kann aber noch optimiert werden. So kann die Länge des Containers in einer Variable außerhalb der Schleife gespeichert werden, um ein wiederholtes Aufrufen des Wertes zu vermeiden. Zusätzlich kann der zu iterierende Container als lokale Variable gespeichert werden, um den Zugriff auf eine Membervariable zu ebenfalls zu vermeiden. Werden Änderungen an dem lokalen Container vorgenommen, wird die Membervariable schließlich von der lokalen Version ersetzt.

```
1  int containerSize = mContainer.length;
2  TestElement[] localContainer = mContainer;
3
4  for(int i = 0; i < containerSize; i++) {
5    localContainer[i].update();
6  }
7
8  mContainer = localContainer;
```

Trotz der zusätzlichen Variablen sind bei den Tests keine nennenswerten Garbage Collections aufgetreten. Aufgrund der reduzierten Zugriffe, konnte aber Zeit eingespart werden.

foreach-Schleife

Mit einer foreach-Schleife können die Inhalte eines Containers sehr einfach und übersichtlich durchlaufen werden.

```
1  for(TestElement element : mContainer) {
2    element.update();
3  }
```

Jedoch wurden bei den Tests mit der ArrayList und besonders bei der HashMap des öfteren der Garbage Collector aufgerufen. Bei häufigeren Aufrufen kamen doch sehr hohe Zeiten für die Garbage Collection heraus. Diese Schleife sollte also bei Performance-kritischen Anwendungen vermieden werden.

while-Schleife

Die while-Schleife überprüft, im Gegensatz zu der for-Schleife, nur ob eine Iteration fortgeführt werden soll. Die Bedingung, um den Schleifendurchlauf zu unterbrechen muss unabhängig von der Schleife geschehen. Als Bedingung dient in diesem Test die *hasNext()*-Methode eines Iterators.

```
1  Iterator<TestElement> it = mContainer.values().iterator();
2  while(it.hasNext()) {
3    it.next().update();
4  }
```

Durch die Verwendung eines Iterators bei der ArrayList und der HashMap konnte ein Anstieg der aufgebrachten Zeit für die Garbage Collection beobachtet werden. Iteratoren sollten daher vermieden werden.

while-Schleife mit lokalen Variablen

Beim Array und der ArrayList kann die while-Schleife auch als zählende Schleife, ähnlich wie die for-Schleife verwendet werden. Dabei können die Werte wie die Länge des Containers und der Container selbst lokal gespeichert werden, um einen schnelleren Zugriff zu gewährleisten.

```
1  int count = 0;
2  TestElement[] localContainer = mContainer;
3  int len = localContainer.length;
4  while(count < len) {
5    localContainer[count].update();
6    count++;
7  }
8  mContainer = localContainer;
```

Ergebnis

Die besten Ergebnisse konnten mit einer einfachen for-Schleife erzielt werden. Foreach-Schleifen sollten vermieden werden, da sie intern auf Iteratoren zugreifen. Auch eigene Iteratoren sollten bei häufig aufgerufenen Schleifen nur mit Vorsicht verwendet werden. Bei Maps sind diese notwendig. Bei den Tests konnte aber nicht festgestellt werden, ob eine foreach oder eine while besser geeignet wäre.

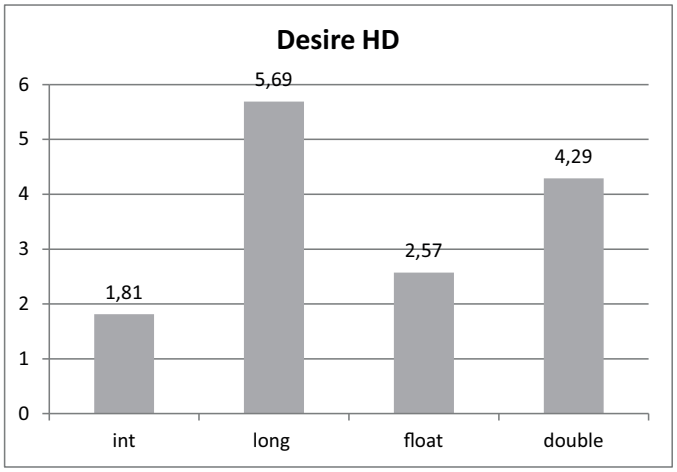

Abbildung 4.39: Benötigte Zeit in Millisekunden der getesteten Datentypen mit verschiedenen Grundoperationen mit FPU (Desire HD).

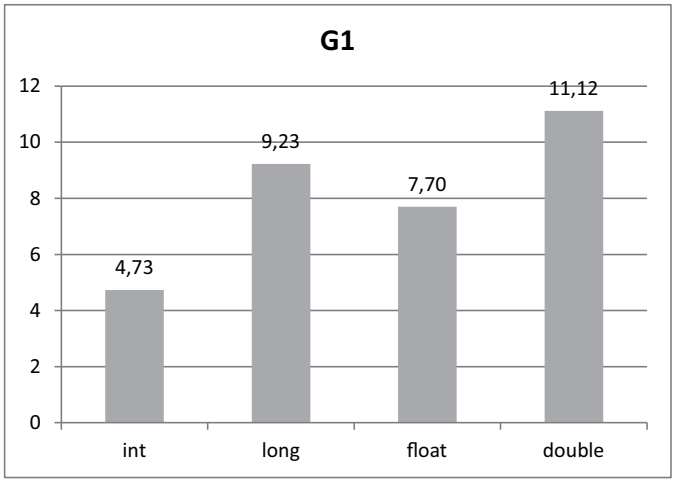

Abbildung 4.40: Benötigte Zeit in Millisekunden der getesteten Datentypen mit verschiedenen Grundoperationen ohne FPU (HTC Dream (G1)).

4.2.7 Fließkommazahlen

Bei diesem Test wurde überprüft, welche Datentypen (int, long, float, double) für Berechnungen mit den Grundoperationen (+, -, *, /) besser geeignet sind. Dabei wurde ein Testgerät mit einer FPU (Desire HD) und einem ohne (Dream bzw. G1) verwendet. Dabei wurde festgestellt, wie in den Abbildungen 4.39 und 4.40 zu sehen ist, dass Berechnungen mit int deutlich schneller verlaufen als mit den restlichen getesteten Typen. Die Leistung bei der Berechnung mit Fließkomma Werten wird mit einer FPU wie erwartet deutlich gesteigert. Long-Werte sollten wenn möglich vermieden werden.

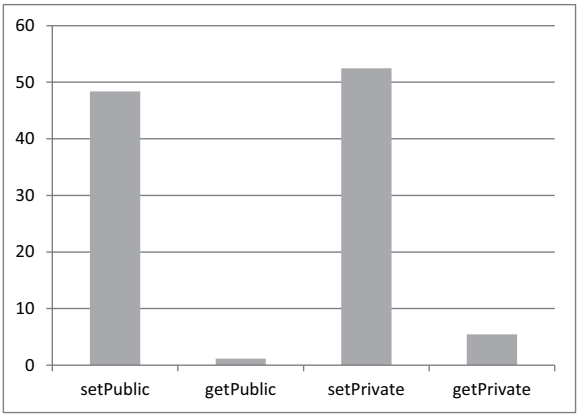

Abbildung 4.41: Benötigte Zeit in Millisekunden für den Zugriff mit getter- und setter-Methoden zum Vergleich mit Zugriff auf public-Variablen bei dem Testgerät Desire HD.

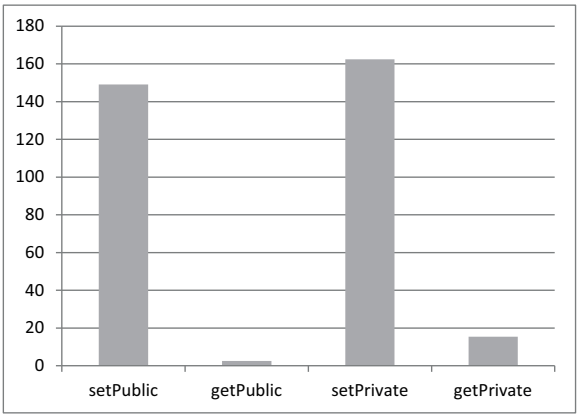

Abbildung 4.42: Benötigte Zeit in Millisekunden für den Zugriff mit getter- und setter-Methoden zum Vergleich mit Zugriff auf public-Variablen bei dem Testgerät HTC Dream (G1).

4.2.8 Getter und Setter

Bei diesem Test wurde überprüft, wie groß der Unterschied zwischen dem Zugriff auf public Variablen und über getter- und setter-Methoden ist (siehe Abbildung 4.41 und 4.42, sowie im Detail 4.43, 4.44, 4.45 und 4.46). Dabei konnte beobachtet werden, dass der Zugriff auf public Variablen deutlich schneller abgewickelt wird. Man kann durch dieses Ergebnis aber nicht einfach Rückschließen, dass alle Variablen als public deklariert werden sollten. Man muss hierbei einen Kompromiss zwischen Kapselung und Optimierung bei Zugriffen auf Variablen finden.

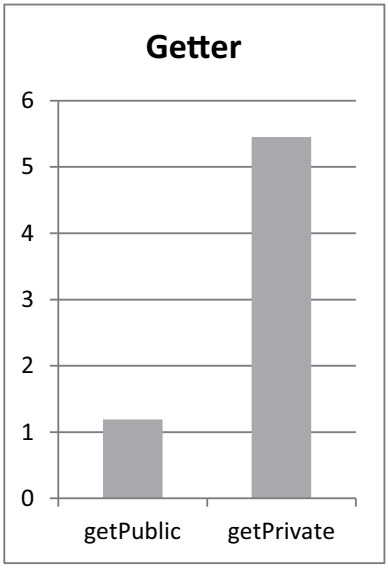

Abbildung 4.43: Benötigte Zeit in Millisekunden für den Zugriff mit getter- und setter-Methoden zum Vergleich mit Zugriff auf public-Variablen Testgerät Desire HD.

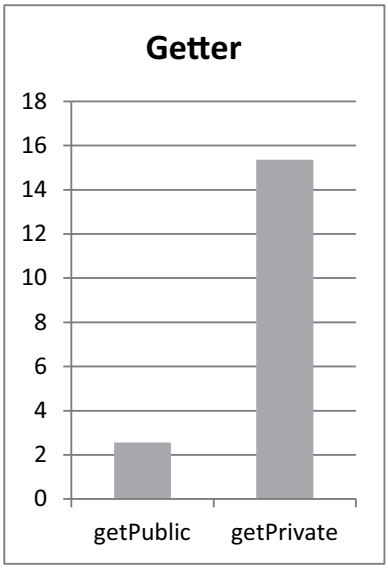

Abbildung 4.44: Benötigte Zeit in Millisekunden im Detail für den Zugriff mit getter-Methoden zum Vergleich mit Zugriff auf public-Variablen bei dem Testgerät HTC Dream (G1).

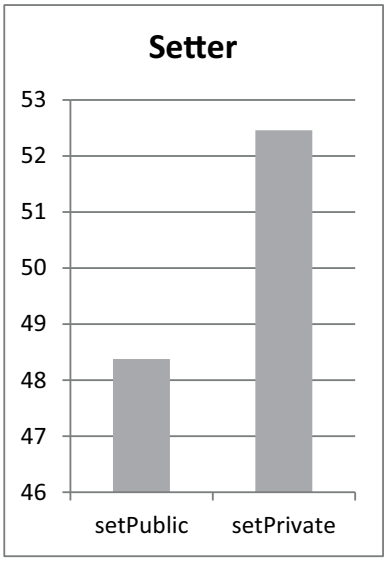

Abbildung 4.45: Benötigte Zeit in Millisekunden im Detail für den Zugriff mit setter-Methoden zum Vergleich mit Zugriff auf public-Variablen bei dem Testgerät Desire HD.

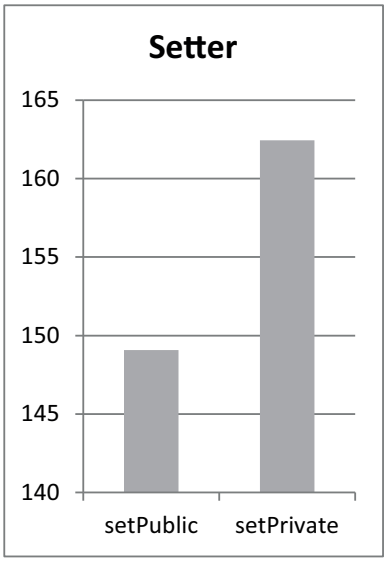

Abbildung 4.46: Benötigte Zeit in Millisekunden im Detail für den Zugriff mit setter-Methoden zum Vergleich mit Zugriff auf public-Variablen bei dem Testgerät HTC Dream (G1).

4.2.9 Static und Final

Um zu überprüfen welchen Einfluss „static“ sowie „static final“ hat, wurden in diesem Test Primitive Datentypen, Objekte und Methoden als solche

deklariert und aufgerufen um die Zeit für den Aufruf mit TraceView zu messen. Dabei konnte festgestellt werden, dass bei Primitiven Datentypen wie int durch das Hinzufügen des Schlüsselwortes static ein gewisser Geschwindigkeitsvorteil gegenüber normaler Variablen entsteht. Wird die Variable als static und final deklariert, so kann sogar ein deutlich schnellerer Zugriff beobachtet werden (siehe Abbildung 4.47 und 4.48).

Im Gegensatz dazu konnten bei einem komplexeren Objekt Geschwindigkeitseinbußen registriert werden. Diese Einbußen sind zwar im Vergleich zu den anderen Testergebnissen gering, es sollte aber, wenn nicht unbedingt nötig, auf static und static final verzichtet werden (siehe Abbildung 4.49 und 4.50).

Bei Methoden konnte eine Performanceoptimierung durch das Schlüsselwort static erreicht werden. Durch final wurde hingegen eine leichte Verlangsamung im Gegensatz zu rein statischen Methoden beobachtet (siehe Abbildung 4.51 und 4.52).

4.2.10 Object Pool

Durch das Wiederverwenden von Objekten können häufige Garbage Collections vermieden werden, da der Speicher nicht von unreferenzierten Objekten befreit werden muss. Das etwas komplexere Vorgehen beim Pooling führt aber dazu, dass mehr Zeit für das „erzeugen“ der Instanzen benötigt wird. Zudem steigt der Speicherverbrauch, je mehr Objektinstanzen im Pool liegen und auf ein Recycling warten. Daher ist es wichtig beim poolen einen Kompromiss zwischen Zugriffszeit, Speicher und Garbage Collection zu finden. Genauere Ergebnisse werden im Abschnitt 4.4.2 im Zuge der Simulation beschrieben.

4.3 Übertragung auf Spiel

Im folgenden Abschnitt werden die Testergebnisse in einer simulierten (4.53) aber üblichen Spielsituation übertragen, in der die bisher besprochenen Methoden von Nutzen sein können. Bei der Simulation wird die Position einer Entiät kontinuierlich verändert (Schiff) und bewegt sich dabei von einem Bildschirmende zum Anderen. Zur gleichen Zeit werden Entitäten an der Position des „Schiffs“ erzeugt (Geschosse) und deren Positionen ebenfalls verändert. Dabei verlassen sie nach gewisser Zeit den Bildschirm und werden wieder entfernt.

4.3.1 Entitäten

Innerhalb eines Spiels gibt es, natürlich abhängig vom Typ des Spiels, viele verschiedene Entitäten. Diese stellen die Spielwelt mithilfe ihres Zustandes

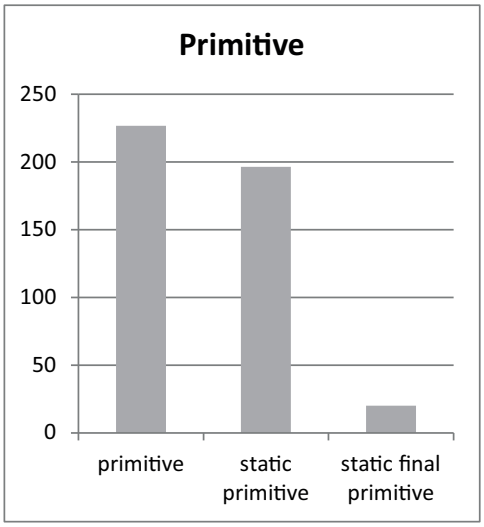

Abbildung 4.47: Zugriffszeit in Millisekunden bei primitiven Datentypen die als nicht static, static und static final deklariert wurden bei dem Testgerät Desire HD.

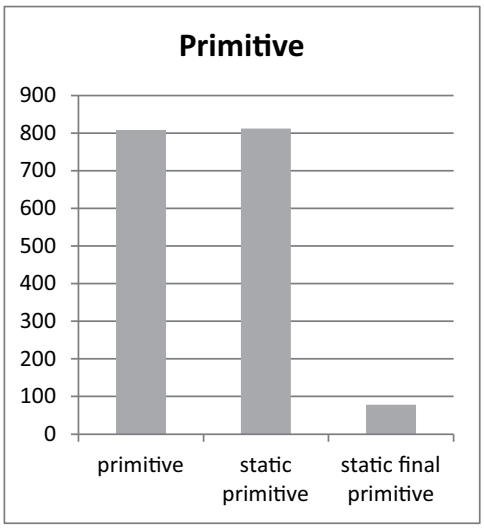

Abbildung 4.48: Zugriffszeit in Millisekunden bei primitiven Datentypen die als nicht static, static und static final deklariert wurden bei dem Testgerät HTC Dream (G1).

dar. Diese Zustände der Entitäten wiederum, werden durch Benutzereingaben und Spielzuständen geändert und verändern so den Verlauf des Spiels.

Abhängig von dem Spiel selbst kann es vorkommen, dass bestimmte Entitätstypen sehr häufig erstellt und auch wieder entfernt werden. Hier wäre die Nutzung eines Object Pools für diesen Entitätyp möglich. So können bereits

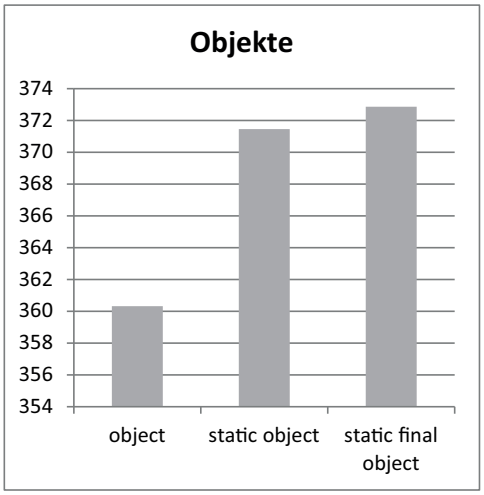

Abbildung 4.49: Zugriffszeit in Millisekunden bei Objekten die als nicht static, static und static final deklariert wurden bei dem Testgerät Desire HD.

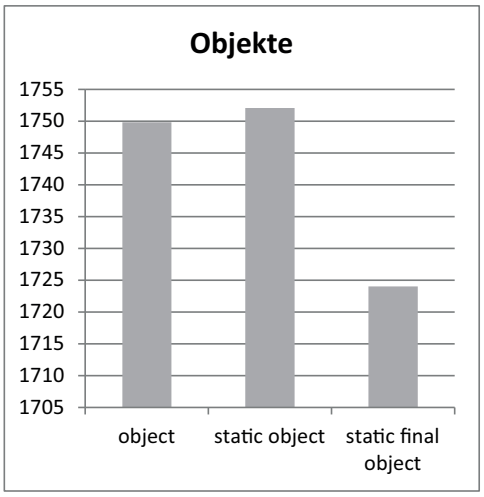

Abbildung 4.50: Zugriffszeit in Millisekunden bei Objekten die als nicht static, static und static final deklariert wurden bei dem Testgerät HTC Dream (G1).

verwendete Objektinstanzen wiederverwendet und eine Garbage Collection verhindert werden.

Auch die Wahl des Containers, mit dem die Entitäten im Spiel oder eben auch im Pool verwaltet werden ist wichtig. Dabei zählt vor allem das schnelle Hinzufügen neuer Entitäten und auch der schnelle Zugriff auf bestimmte Entitäten innerhalb des Containers. Weniger wichtig ist das Hinzufügen zu beginn des Spiels und das Entfernen am Ende. Daher bietet sich die Verwendung eines einfachen Arrays an. Da aber nie sicher ist, ob die Länge des

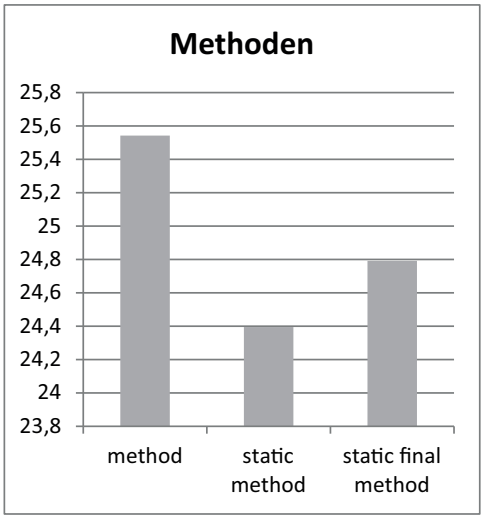

Abbildung 4.51: Zugriffszeit in Millisekunden bei Methoden die als nicht static, static und static final deklariert wurden bei dem Testgerät Desire HD.

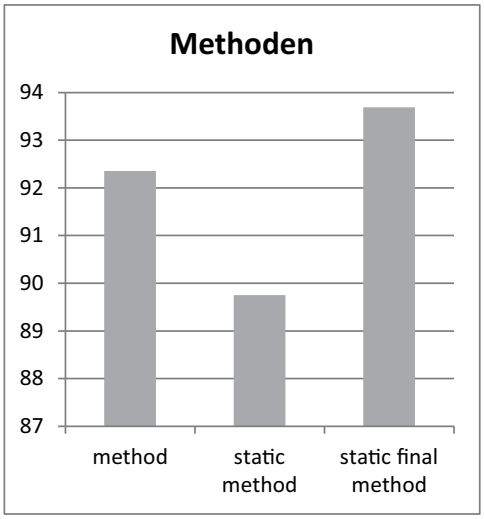

Abbildung 4.52: Zugriffszeit in Millisekunden bei Methoden die als nicht static, static und static final deklariert wurden bei dem Testgerät HTC Dream (G1).

Arrays im Laufe des Spiels ausreichend ist, wäre auch die ArrayList eine gute Alternative.

4.3.2 Events

Mithilfe von Events ist es möglich eine entkoppelte Kommunikation zwischen verschiedenen Komponenten eines Spiels aufzubauen. Dabei werden

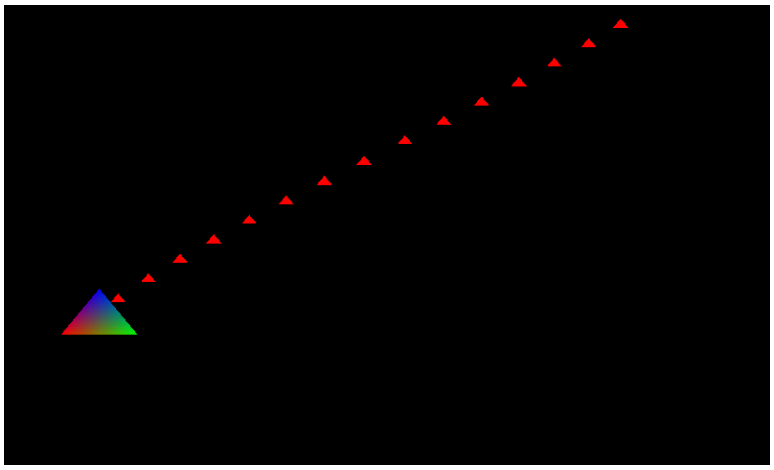

Abbildung 4.53: Die Testsimulation des Spiels. Das farbige Dreieck repräsentiert das „Schiff“ und bewegt sich von einem Bildschirmende zum Anderen. Die roten Dreiecke stellen die „Geschosse“ dar, welche sich von der Position des „Schiffs“ weg bewegen und beim Verlassen des Bildschirms entfernt werden.

sehr häufig sehr viele „Nachrichten“ verschickt, auf die bestimmte Listener horchen und um Gegebenenfalls auf diese zu reagieren.

Es kann besonders wichtig sein Events wieder zu verwenden um eine häufige Garbage Collection durch verbrauchte und nicht länger verwendete Events zu verhindern. Wichtig ist dabei, die Zustände der Events nach dem Gebrauch zurückzusetzen, da sonst ungewollte Seiteneffekte auftreten können.

Die Aufbewahrung der Events bzw. ihre Listener in einem Container, ob mit oder ohne Pool ist ebenfalls wichtig. Besonders die schnelle Iteration ist entscheidend, nicht nur Aufgrund des Spielgefühls durch einen flüssigen Ablauf, sondern auch um die Funktion der Events aufrecht zu erhalten. Daher ist auch hier die Verwendung eines einfachen Arrays oder einer ArrayList empfehlenswert.

4.3.3 Services

Unterschiedliche Services bieten verschiedene Funktionen an. Um auf diese verwenden zu können, muss der Container, in dem die Services eines Spiels gelistet sind, einen schnellen Zugriff unterstützen. Auch sind sie zum Teil in die regelmäßigen Update-Zyklen der Gameloop integriert, was auch eine schnelle Iteration nötig macht. Weniger wichtig ist aber das Einfügen neuer Services, da dies am Anfang eines Spiels geschieht. Daher ist der Gebrauch eines einfachen Arrays sicherlich am sinnvollsten, da die Länge zu beginn bereits bekannt ist, und keine neuen Services hinzugefügt werden müssen.

4.4 Einfluss auf die Engine

Die Erkenntnisse der durchgeführten Tests werden nun im Folgenden Abschnitt auf die bestehende Game Engine (Cogaen) übertragn. Dafür wurde eine Anwendung geschrieben, die ein typisches Spielverhalten auf einem Android Gerät simuliert. Die verschiedenen Ansätze wurden schließlich in die Engine beziehungsweise auch der Simulation übertragen und durch Tests überprüft. Bei den Tests wurde in einer Zeit von 60 Sekunden die resultierenden Frames Per Second, wie sie sich im Laufe der Zeit verändern, sowie die Anzahl und Zeit der Garbage Collections festgehalten und ausgewertet.

4.4.1 Simulation ohne Optimierung

Die Simulation wurde zunächst ohne Optimierungen durchgeführt. Dabei wurde, unter Zuhilfenahme der Game Engine, ein Spieltypisches Verhalten simuliert. Im folgenden die Auswertung der Logs der Frames Per Second (FPS) und der Traceview-Daten.

Frames Per Second

Bei der Simulation ohne Optimierungen konnten beim G1 in einer Minute durchschnittlich 38 FPS bei 2044 Updatezyklen erreicht werden. Dabei wurden 3841 ms nur für die 64 Garbage Collections verwendet. In dieser Zeit wurde die Simulation vom System angehalten um den Speicher aufzuräumen. Das Desire HD erreichte durchschnittlich 61 FPS bei 3493 Updatezyklen und benötigte 907 ms für die 172 Garbage Collections, wobei hier nicht die Simulation angehalten wurde, sondern parallel zur Simulation ablief. Als zusätzliches Testgerät wurde das Asus Transformer tf300t verwendet, welches durchschnittlich 63 FPS bei 3598 Updatezyklen in einer Minute erreicht hat. Die 200 Garbage Collections benötigten 820 ms für das Aufräumen, was wie beim Desire HD ebenfalls parallel zur Simulation ablief (siehe Abbildung 4.54 und 4.55).

Traceview Analyse

Zusätzlich zu der Aufzeichnung der FPS wurde mithilfe von Traceview versucht die „Hotspots“ der Game Engine zu finden (siehe Abbildung 4.56 und 4.57). Dabei konnte festgestellt werden, dass ein Großteil der Zeit in der update()-Methode des Cores verbracht wird. Hier werden die update()-Methoden der „updatebaren“ Komponenten des Spiels aufgerufen, wie beispielsweise Services. Beim G1 wurden 10,1 % der Zeit, also 1175,6 ms, in dieser Methode verbracht. Beim Desire HD waren es ebenfalls 10,1 %, bei 315,55 ms. Das tf300t verbrachte 15,5 % der Zeit mit 246,422 ms in dieser Methode.

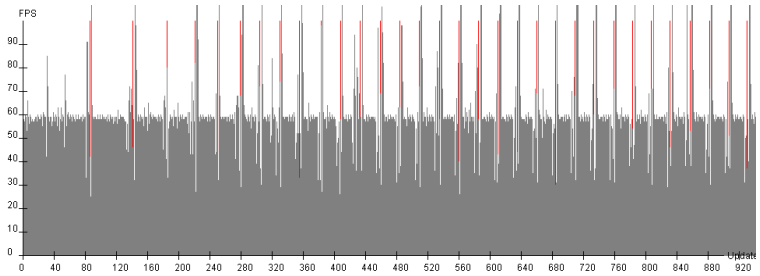

Abbildung 4.54: Verlauf der Frames Per Second in Abhängigkeit der Zeit (Updatezyklen) bei dem Testgerät Desire HD. Die roten Linien markieren die Garbage Collections.

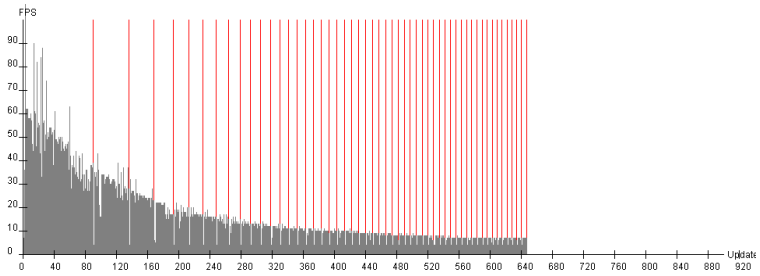

Abbildung 4.55: Verlauf der Frames Per Second in Abhängigkeit der Zeit (Updatezyklen) bei dem Testgerät HTC Dream (G1). Die roten Linien markieren die Garbage Collections.

Nach dem Core.update() folgt die update()-Methode im EventService. 6,5 % (762,178 ms) der getesteten Zeit verbrachte das G1 in dieser Methode. Das Desire HD war mit 6,4 % prozentual etwa gleichauf, wobei die absolute Zeit mit 199,73 ms kürzer war. Mit 9,7 % verbrachte das Transformer Tablet prozentual länger als die beiden anderen Geräte, war aber mit 154,906 ms auch schneller.

Die dritte auffälligste Methode bei diesem Test war die fireEvent(Event) vom EventService. In dieser Methode werden Events an die richtigen Listener weitergegeben. Das G1 verbrachte 5,2 % (606,34 ms) in dieser Methode, während das Desire HD 5 % (156,51 ms) benötigte. Das tf300t benötigte 7,5 % (119,733 ms).

4.4.2 Simulation mit Optimierung

Schließlich wurden schrittweise verschiedene Optimierungensversuche unternommen. Zunächst wurde Object Pooling angewendet. Dabei wurden Events und Entitäten für die Wiederverwendung aufbereitet. Als nächstes wurden Schleifen in der Game Engine ersetzt und auch Container ausgetauscht, um sie mit einer anderen Schleife testen zu können. Zu letzt wurden die einzelnen

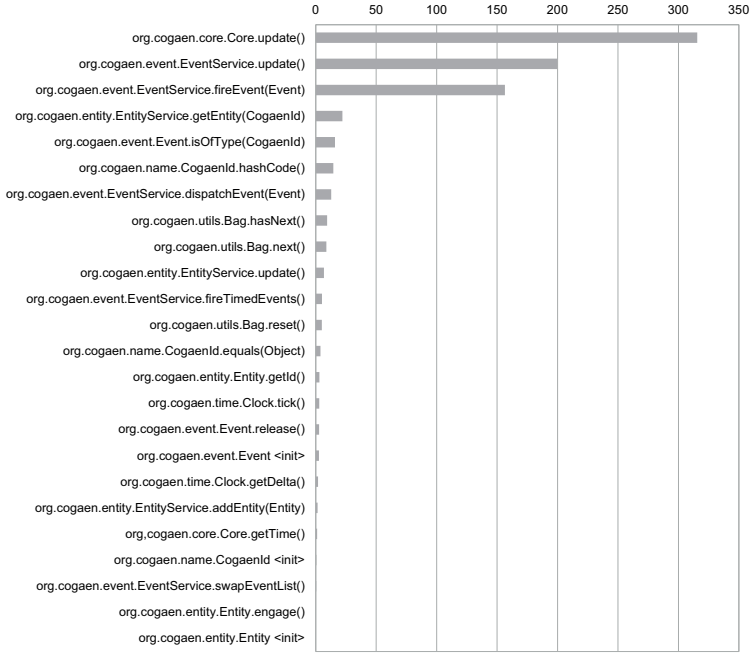

Abbildung 4.56: Hotspots der Engine in einem Simulationstest bei dem Testgerät Desire HD.

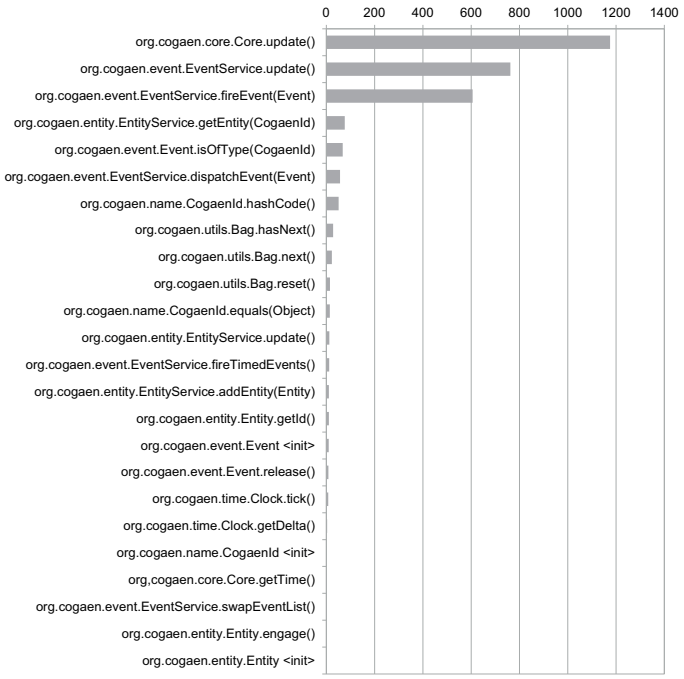

Abbildung 4.57: Hotspots der Engine in einem Simulationstest bei dem Testgerät HTC Dream (G1).

Tabelle 4.17: Ergebnis bei dem Test mit Object Pooling des Events zum Erstellen einer Entität (HTC Desire HD).

	ohne Optimierung	Create Entity Event
FPS	61	61
Updatezyklen	3493	3483
GC Anzahl	172	171
GC Zeit	907,67 ms	933

Optimierungsversuche kombiniert.

Event Pooling

Bei den Events wurde ein durch das Event selbst gehaltener Pool verwendet. Dadurch organisiert diese selbst die Wiederverwendung von bereits gebrauchten Events. Ein klassischer Object Pool wurde bei den Entitäten eingesetzt. Dabei ist eine eigene Instanz für das Abrufen und Wiederverwenden der Entitäten zuständig.

Bei dieser Optimierung wurde das Event für die Erstellung von Entitäten gepoolt. Dabei konnte festgestellt werden, dass beim Desire HD keine Verbesserungen entstanden (siehe Tabellen 4.17, 4.18 und 4.19). Die FPS blieben bei 61, die Anzahl der Updatezyklen verschlechterte sich sogar auf durchschnittlich 3483. Die Anzahl der Garbage Collection ging kaum herunter (172 Garbage Collections) während sich die Zeit für die Garbage Collection auf durchschnittlich 933 ms erhöht hat. Beim HTC Dream konnte ebenfalls kaum eine Verbesserung beobachtet werden. Die Updatezyklen waren zwar durchschnittlich höher, was aber auf einen Ausreiserwert zurückzuführen ist (Test 1 4098, Test 2 2033 und Test 3 2043 im Vergleich zur nicht optimierten Simulation 2044). Die Anzahl der Garbage Collection stieg auf 96 und damit auch die Zeit für die Garbage Collection auf 4990 ms. Auch bei dem Transformer Tablet konnte bei den Updatezyklen eine leichte Verschlechterung mit 3573 Zyklen beobachtet werden. Trotz der gesunkenen Anzahl der Garbage Collections auf 96, war die Aufgebrachte Zeit für die Garbage Collection mit 858 ms etwas höher als ohne Optimierung.

Das Poolen des Events für das Entfernen einer Entität ergab, wie beim vorherigen Event, ebenfalls keine Verbesserung beim Desire HD (siehe Tabellen 4.20, 4.21 und 4.22). Die Updatezyklen waren mit 3486 etwas besser als beim Erstellen und waren unter den nicht optimierten Wert. Die Anzahl der Garbage Collection hat sich mit 172 nicht verändern, wohingegen sich die Aufgewendete Zeit auf 935 ms erhöht hat. Beim G1 konnte bei den Updatezyklen mit 2057 eine leichte Verbesserung beobachtet werden, während sich die Anzahl der Garbage Collection mit 65 kaum verändert hat wurde

Tabelle 4.18: Ergebnis bei dem Test mit Object Pooling des Events zum Erstellen einer Entität (HTC Dream (G1)).

	ohne Optimierung	Create Entity Event
FPS	38	38
Updatezyklen	2044	2725
GC Anzahl	64	96
GC Zeit	3841,33 ms	4990

Tabelle 4.19: Ergebnis bei dem Test mit Object Pooling des Events zum Erstellen einer Entität (Asus Transformer tf300t).

	ohne Optimierung	Create Entity Event
FPS	63	63
Updatezyklen	3598	3573
GC Anzahl	200	197
GC Zeit	820,67 ms	858

Tabelle 4.20: Ergebnis bei dem Test mit Object Pooling des Events zum Entfernen einer Entität (HTC Desire HD).

	ohne Optimierung	Remove Entity Event
FPS	61	61
Updatezyklen	3493	3486
GC Anzahl	172	172
GC Zeit	907,67 ms	935

die Zeit für diese auf 935 ms erhöht. Beim tf300t konnte keine Veränderung beobachtet werden, lediglich die Zeit für die Garbage Collection hatte sich auf 854 ms erhöht.

Bei dem sehr häufig auftretenden Event für die Positionsänderung der Entitäten in der Simulation konnten wie bei den vorangegangenen Events beim Desire HD fast keine Verbesserungen festgestellt werden (siehe Tabellen 4.23, 4.24 und 4.25). Die Updatezyklen lagen etwa auf der Höhe der anderen Events (3481) und die Anzahl der Garbage Collections ging leicht runter auf 167 wobei sich die Zeit für diese auf 932 ms erhöht hatte. Die Tests mit dem G1 ergaben kaum Veränderungen zu denen ohne Optimierung bei den Updatezyklen (2044) und der Anzahl der Garbage Collections (63), wobei die Zeit für diese mit 3655 ms besser war. Die Werte für die

Tabelle 4.21: Ergebnis bei dem Test mit Object Pooling des Events zum Entfernen einer Entität (HTC Dream (G1)).

	ohne Optimierung	Remove Entity Event
FPS	38	38
Updatezyklen	2044	2057
GC Anzahl	64	65
GC Zeit	3841,33 ms	3750

Tabelle 4.22: Ergebnis bei dem Test mit Object Pooling des Events zum Entfernen einer Entität (Asus Transformer tf300t).

	ohne Optimierung	Remove Entity Event
FPS	63	63
Updatezyklen	3598	3598
GC Anzahl	200	199
GC Zeit	820,67 ms	854

Tabelle 4.23: Ergebnis bei dem Test mit Object Pooling des Events zur Positionsveränderung einer Entität (HTC Desire HD).

	ohne Optimierung	Update Entity Event
FPS	61	61
Updatezyklen	3493	3486
GC Anzahl	172	172
GC Zeit	907,67 ms	935

Updatezyklen beim tf300t waren mit 3597 etwa auf der Höhe der Tests ohne Optimierung, während die Anzahl der Garbage Collections mit 191 zurück ging aber bei 879 ms mehr Zeit beanspruchten.

Mit dem Shoot-Event, dass bei der Simulation die das „abfeuern“ der Dreiecke verwendet wird, konnte mit dem Desire HD keine besseren Werte für die Updatezyklen im Vergleich zur nicht optimierten Version erzielt werden (siehe Tabellen 4.26, 4.27 und 4.28), waren aber mit 3488 in einer Minute besser als bei den anderen Events. Die Anzahl der Garbage Collections blieb mit 173 etwa gleich, aber mit einer längeren Zeit für das Aufräumen (928 ms). Beim G1 waren die Updatezyklen mit 2053 etwas mehr. Die Anzahl der Garbage Collection blieb mit 65 in etwa gleich, wobei die benötigte Zeit etwas zurück ging (3832 ms). Die Updatezyklen blieben beim tf300t etwa

Tabelle 4.24: Ergebnis bei dem Test mit Object Pooling des Events zur Positionsveränderung einer Entität (HTC Dream (G1)).

	ohne Optimierung	Update Entity Event
FPS	38	38
Updatezyklen	2044	2044
GC Anzahl	64	63
GC Zeit	3841,33 ms	3655

Tabelle 4.25: Ergebnis bei dem Test mit Object Pooling des Events zur Positionsveränderung einer Entität (Asus Transformer tf300t).

	ohne Optimierung	Update Entity Event
FPS	63	63
Updatezyklen	3598	3597
GC Anzahl	200	191
GC Zeit	820,67 ms	879

Tabelle 4.26: Ergebnis bei dem Test mit Object Pooling des Events zum Abfeuern einer Entität (HTC Desire HD).

	ohne Optimierung	Shoot Entity Event
FPS	61	61
Updatezyklen	3493	3488
GC Anzahl	172	173
GC Zeit	907,67 ms	928

gleich (3600), ebenso die Anzahl der Garbage Collections (199) bei 845 ms.

Tabelle 4.27: Ergebnis bei dem Test mit Object Pooling des Events zum Abfeuern einer Entität (HTC Dream (G1)).

	ohne Optimierung	Shoot Entity Event
FPS	38	38
Updatezyklen	2044	2053
GC Anzahl	64	65
GC Zeit	3841,33 ms	3832

Tabelle 4.28: Ergebnis bei dem Test mit Object Pooling des Events zum Abfeuern einer Entität (Asus Transformer tf300t).

	ohne Optimierung	Shoot Entity Event
FPS	63	63
Updatezyklen	3598	3600
GC Anzahl	200	199
GC Zeit	820,67 ms	845

Tabelle 4.29: Ergebnis bei dem Test mit Object Pooling aller Events (HTC Desire HD).

	ohne Optimierung	alle Events
FPS	61	61
Updatezyklen	3493	3500
GC Anzahl	172	277
GC Zeit	907,67 ms	899

Tabelle 4.30: Ergebnis bei dem Test mit Object Pooling aller Events (HTC Dream (G1)).

	ohne Optimierung	alle Events
FPS	38	38
Updatezyklen	2044	2040
GC Anzahl	64	72
GC Zeit	3841,33 ms	3612

Trotz der bisher vergleichsweise schlechten Ergebnissen, konnte beim poolen aller Events mit dem Desire HD als Testgeräte eine Verbesserung beobachtet werden (siehe Tabellen 4.29, 4.30 und 4.31). Mit 3500 Updatezyklen in einer Minute wurde ein besseres Ergebnis erreicht als ohne Optimierung. Die Anzahl der Garbage Collections ging zwar auf 277 hoch, benötigte aber mit 899 ms weniger Zeit. Das G1 hingegen zeigte kaum Veränderung bei den Updatezyklen (2040). Die Anzahl der Garbage Collections war mit 72 höher, benötigte hier aber insgesamt ebenfalls weniger Zeit (3612). Mit 3590 Updatezyklen, 190 Garbage Collections in 907 ms konnte bei dem tf300t keine Verbesserung erreicht werden.

Mit dem nächsten Test wurden anstatt den Events die Entitäten wieder verwendet. Dabei ergaben sich wie bei den vorangegangenen Tests beim

Tabelle 4.31: Ergebnis bei dem Test mit Object Pooling aller Events (Asus Transformer tf300t).

	ohne Optimierung	alle Events
FPS	63	63
Updatezyklen	3598	3590
GC Anzahl	200	190
GC Zeit	820,67 ms	907

Tabelle 4.32: Ergebnis bei dem Test mit Object Pooling der Entitäten (HTC Desire HD).

	ohne Optimierung	Entitäten
FPS	61	61
Updatezyklen	3493	3500
GC Anzahl	172	277
GC Zeit	907,67 ms	899

Tabelle 4.33: Ergebnis bei dem Test mit Object Pooling der Entitäten (HTC Dream (G1)).

	ohne Optimierung	Entitäten
FPS	38	38
Updatezyklen	2044	2052
GC Anzahl	64	65
GC Zeit	3841,33 ms	3746

Desire HD kaum Verbesserungen (siehe Tabellen 4.32, 4.33 und 4.34). Es gab 3499 Zyklen und 173 Garbage Collections die zusammen 900 ms benötigten. Die Ergebnisse beim G1 waren etwas besser bei 2052 Updatezyklen und 65 Garbage Collections die 3746 ms benötigten. Aber beim tf300t konnten mit 3600 Zyklen und 858 ms der 232 Garbage Collections keine Verbesserungen beobachtet werden.

Desire HD ergab beim poolen der Repräsentationen der Entitäten ebenfalls keine Verbesserungen bei 3481 Updatezyklen und 172 Garbage Collections die 963 ms benötigten (siehe Tabellen 4.35, 4.36 und 4.37). Beim G1 wurden leichte Verbesserungen verbucht, auch wenn mit 2030 Updatezyklen weniger erreicht wurde, ist die benötigte Zeit für die 65 Garbage Collections auf 3738 ms runter gegangen. Das tf300t ergab fast gleiche Werte bei den

Tabelle 4.34: Ergebnis bei dem Test mit Object Pooling der Entitäten (Asus Transformer tf300t).

	ohne Optimierung	Entitäten
FPS	63	63
Updatezyklen	3598	3600
GC Anzahl	200	232
GC Zeit	820,67 ms	858

Tabelle 4.35: Ergebnis bei dem Test mit Object Pooling der Repräsentationen der Entitäten (HTC Desire HD).

	ohne Optimierung	Repräsentationen
FPS	61	61
Updatezyklen	3493	3481
GC Anzahl	172	172
GC Zeit	907,67 ms	963

Tabelle 4.36: Ergebnis bei dem Test mit Object Pooling der Repräsentationen der Entitäten (HTC Dream (G1)).

	ohne Optimierung	Repräsentationen
FPS	38	38
Updatezyklen	2044	2030
GC Anzahl	64	65
GC Zeit	3841,33 ms	3738

Updatezyklen (3599) und der benötigten Garbage Collections (199) die aber mit 884 ms etwas mehr Zeit beansprucht haben.

Beim Poolen der Entitäten und Repräsentationen hatte das Desire HD kaum Änderungen bei den Updatezyklen (3491) und der Anzahl der Garbage Collections (173) die mehr Zeit benötigten (874 ms) (siehe Tabellen 4.38, 4.39 und 4.40). Beim G1 etwas weniger Updatezyklen (2034) und etwa gleich viele Garbage Collections (63), die aber weniger Zeit benötigt haben (3607 ms). Die Ergebnisse beim tf300t waren fast identisch zu denen ohne Pooling (3599 Updatezyklen und 819 ms für 201 Garbage Collections mit).

Wenn sowohl Events als auch die Entitäten und Repräsentationen gepoolt werden, wurde bei den Tests festgestellt, dass beim Desire HD die gleiche Anzahl an Updatezyklen erreicht wurde wie ohne dem poolen (3493),

Tabelle 4.37: Ergebnis bei dem Test mit Object Pooling der Repräsentationen der Entitäten (Asus Transformer tf300t).

	ohne Optimierung	Repräsentationen
FPS	63	63
Updatezyklen	3598	3599
GC Anzahl	200	201
GC Zeit	820,67 ms	819

Tabelle 4.38: Ergebnis bei dem Test mit Object Pooling der Repräsentationen und Entitäten (HTC Desire HD).

	ohne Optimierung	Repräsentationen und Entitäten
FPS	61	61
Updatezyklen	3493	3491
GC Anzahl	172	173
GC Zeit	907,67 ms	948

Tabelle 4.39: Ergebnis bei dem Test mit Object Pooling der Repräsentationen und Entitäten (HTC Dream (G1)).

	ohne Optimierung	Repräsentationen und Entitäten
FPS	38	38
Updatezyklen	2044	2034
GC Anzahl	64	63
GC Zeit	3841,33 ms	3607

während die Anzahl der Garbage Collections auf 169 herunter gegangen ist

Tabelle 4.40: Ergebnis bei dem Test mit Object Pooling der Repräsentationen und Entitäten (Asus Transformer tf300t).

	ohne Optimierung	Repräsentationen und Entitäten
FPS	63	63
Updatezyklen	3598	3599
GC Anzahl	200	201
GC Zeit	820,67 ms	819

Tabelle 4.41: Ergebnis bei dem Test mit allen Object Pools (HTC Desire HD).

	ohne Optimierung	alle Pools
FPS	61	61
Updatezyklen	3493	3493
GC Anzahl	172	169
GC Zeit	907,67 ms	874

Tabelle 4.42: Ergebnis bei dem Test mit allen Object Pools (HTC Dream (G1)).

	ohne Optimierung	alle Pools
FPS	38	38
Updatezyklen	2044	2036
GC Anzahl	64	63
GC Zeit	3841,33 ms	3601

Tabelle 4.43: Ergebnis bei dem Test mit allen Object Pools (Asus Transformer tf300t).

	ohne Optimierung	alle Pools
FPS	63	63
Updatezyklen	3598	3598
GC Anzahl	200	191
GC Zeit	820,67 ms	561

und auch mit 874 ms weniger Zeit beansprucht haben (siehe Tabellen 4.41, 4.42 und 4.43. Das G1 hatte weniger Zyklen (2036), dafür aber etwas weniger Garbage Collections (63) die aber weniger Zeit benötigt haben (3601 ms). Beim tf300t blieben die Updatezyklen wie beim Desire HD auf der selben Höhe wie ohne dem Poolen (3598). Die Anzahl der Garbage Collections sank auf 191 herunter und beanspruchten deutlich weniger Zeit (561 ms).

Schleifen

In der update()-Methode im EventService wurde die foreach-Schleife durch eine einfache for-Schleife ersetzt (siehe Tabellen 4.44, 4.45 und 4.46. Dabei konnte festgestellt werden, dass beim Desire HD die Updatezyklen etwas weniger wurden (3482), während die Anzahl der Garbage Collections gleich

Tabelle 4.44: Ergebnis bei dem Test mit Verwendung einer einfachen for-Schleife in der update()-Methode des EventServices (HTC Desire HD).

	ohne Optimierung	EventService.update()
FPS	61	61
Updatezyklen	3493	3482
GC Anzahl	172	172
GC Zeit	907,67 ms	922

Tabelle 4.45: Ergebnis bei dem Test mit Verwendung einer einfachen for-Schleife in der update()-Methode des EventServices (HTC Dream (G1)).

	ohne Optimierung	EventService.update()
FPS	38	37
Updatezyklen	2044	1999
GC Anzahl	64	83
GC Zeit	3841,33 ms	3627

Tabelle 4.46: Ergebnis bei dem Test mit Verwendung einer einfachen for-Schleife in der update()-Methode des EventServices (Asus Transformer tf300t).

	ohne Optimierung	EventService.update()
FPS	63	63
Updatezyklen	3598	3600
GC Anzahl	200	200
GC Zeit	820,67 ms	793

blieb (172) aber mehr Zeit beim Aufräumen aufgebracht wurde (922 ms). Das G1 hatte ebenfalls weniger Updatezyklen (1999). Die Anzahl der Garbage Collections nahm hingegen zu (83), benötigten aber zusammen weniger Zeit (3627 ms). Die Ergebnisse beim tf300t waren bei der Anzahl der Updates etwa gleich (3600), genau wie die der Garbage Collections (200), dafür benötigten diese nur noch 793 ms.

Bei folgendem Test wurde der Bag-Container für Updatables mit einer ArrayList ersetzt und diese mit einer einfach for-Schleife in der update()-Methode iteriert (siehe Tabellen 4.47, 4.48 und 4.49). Die Updatezyklen beim Desire HD haben sich dabei kaum geändert (3492). Auch die der Garbage Collections blieb gleich (172), dafür benötigten diese mehr Zeit (934 ms). Beim G1 wurden vier Updatezyklen weniger erreicht (2040), die Anzahl der

Tabelle 4.47: Ergebnis beim Ersetzen der Bag durch eine ArrayList und der Verwendung einer for-Schleife in der update()-Methode des Cores (HTC Desire HD).

	ohne Optimierung	Core.update()
FPS	61	61
Updatezyklen	3493	3482
GC Anzahl	172	172
GC Zeit	907,67 ms	922

Tabelle 4.48: Ergebnis beim Ersetzen der Bag durch eine ArrayList und der Verwendung einer for-Schleife in der update()-Methode des Cores (HTC Dream (G1)).

	ohne Optimierung	Core.update()
FPS	38	38
Updatezyklen	2044	2040
GC Anzahl	64	63
GC Zeit	3841,33 ms	3716

Tabelle 4.49: Ergebnis beim Ersetzen der Bag durch eine ArrayList und der Verwendung einer for-Schleife in der update()-Methode des Cores (Asus Transformer tf300t).

	ohne Optimierung	Core.update()
FPS	63	63
Updatezyklen	3598	3600
GC Anzahl	200	201
GC Zeit	820,67 ms	817

Garbage Collections ging auf 63 zurück und benötigten mit 3716 ms weniger Zeit als ohne diesen Optimierungsschritt. Kaum Veränderungen der Wert gabe es beim tf300t mit 3600 Updatezyklen und 200 Garbage Collections die 817 ms benötigten.

Schleifen und Pooling

Schließlich wurden getestet wie die Simulation beeinflusst wird, wenn die Objekt Pools und die Schleifenoptimierungen kombiniert werden (siehe Tabellen 4.50, 4.51 und 4.52). Das Desire HD konnte kaum bessere Ergebnisse

Tabelle 4.50: Ergebnis des Tests mit allen Optimierungsversuchen (HTC Desire HD).

	ohne Optimierung	alle Optimierungen
FPS	61	61
Updatezyklen	3493	3492
GC Anzahl	172	167
GC Zeit	907,67 ms	930

Tabelle 4.51: Ergebnis des Tests mit allen Optimierungsversuchen (HTC Dream (G1)).

	ohne Optimierung	alle Optimierungen
FPS	38	38
Updatezyklen	2044	2042
GC Anzahl	64	63
GC Zeit	3841,33 ms	3564

Tabelle 4.52: Ergebnis des Tests mit allen Optimierungsversuchen (Asus Transformer tf300t).

	ohne Optimierung	alle Optimierungen
FPS	63	63
Updatezyklen	3598	3598
GC Anzahl	200	191
GC Zeit	820,67 ms	860

liefern. Die Updatezyklen blieben etwa gleich (3492), aber die Anzahl der Garbage Collections nahm ab (167) benötigten aber auch mehr Zeit (930 ms). Beim G1 blieben die Ergebnis auch fast gleich, so konnten nur 2042 Updatezyklen und 63 Garbage Collections erreicht werden. Dafür benötigten diese nur 3564 ms. Auch beim tf300t waren die Resultate sehr gering. Die Updatezyklen waren identisch zu den Ergebnissen ohne Optimierung (3598). Die Anzahl der Garbage Collections ging aber auf 191 zurück, benötigten aber mit 860 ms mehr Zeit.

4.4.3 Ergebnis

Eine Verbesserung der Werte konnte mit diesen Optimierungen kaum erzielt werden. Die Anzahl der Frames Per Second haben sich bei den einzelnen

Optimierungsmethoden nicht gändert. Lediglich beim HTC Dream bzw. G1 konnten leichte Verbesserungen beobachtet werden, die sich aber bei der optischen Betrachtung der Simulation als nicht sehr Wirksam herausstellten.

Im Laufe der Entwicklung der Simulation konnte aber anhand eines Fehlers festgestellt werden, dass kleine Veränderung an der Spiellogik viel bewirken können. So wurde beispielsweise ein Fehler gemacht, bei dem der Wert für die delta Time (zuletzt benötigte Zeit für das durchlaufen der Gameloop) für jede Entität einzeln abgefragt wurde. Dieser Wert wurde schließlich als lokale Variable vor der Schleife für die Entitäten gespeichert. Dies führte beim G1 einer Verbesserung der FPS von 17 auf 38.

Kapitel 5

Ausblick

Durch die vorangegangenen Tests konnte festgestellt werden, dass Optimierungen selbstverständlich möglich sind. Aber die Resultate nicht unbedingt die Größten Erfolge versprechen können. Dennoch kann durch das Vermeiden versteckter Instanziierungen, vor allem an performancekritischen Stellen wie Schleifen, eine Verbesserung der Leistung erzielt werden.

Die Optimierung einer bestehenden Engine erwies sich als schwierig. Hier mussten zunächst die Hotspots ausfindig gemacht werden, was aber nicht unbedingt direkt zu versteckten allokierungen führen muss. Bereits bei der Entwicklung eines Spiels oder einer Engine können bereits verschiedene Methoden angewandt werden, die Performanceeinbußen vermeiden können. So sollten häufig verwendete globale Werte lokal in einer Variable gespeichert werden. Auch sollte hauptsächlich als numerische Datentyp ein int verwendet werden. Es sollten nur einfache Containertypen verwendet werden, wenn keine speziellen Aufgaben mit diesen erzielt werden müssen. Um diese zu iterieren wäre die foreach-Schleife trotz ihres übersichtlichen und ordentlichen Aufbaus ungeeignet, da sie einen versteckten Iterator erzeugt, der zu Einbußen führt [2]. Vorsichtig sollte man mit dem Verzicht von Kapselung zugunsten der Zugriffsgeschwindigkeit sein. Es ist zwar weniger aufwendig auf ein öffentliches Attribut zuzugreifen, kann aber zu ungewollten Seiteneffekten führen. Wenn es möglich ist, können auch primitive Datentypen wie int und boolean und auch Methoden statisch gesetzt werden, je nach Aufgabe. Bei komplexeren Objekten funktioniert dies aber nicht und sollte vermieden werden. Im Test konnten zwar keine größeren Ergebnisse mit dem poolen von Objekten erreicht werden, aber ist eine Überlegung wert Objektinstanzen wiederzuverwenden, wenn sie in großer Zahl im Spiel auftreten und nach kurzer Zeit wieder entfernt werden. Dabei muss aber immer ein Kompromiss zwischen der benötigten Zeit für den Zugriff auf die Instanzen, der Garbage Collection und des benötigten Speicherverbrauchs eingegangen werden.

Letztendlich ist eine Optimierung aber nur notwendig, wenn auch der

Bedarf durch auffallend schlechter Performance besteht. Und dies kann bereits vermieden werden, wenn schon zu Beginn der Entwicklung einige der genannten Methoden angewendet werden.

Die Zeit spielt für die Entwickler von Anwendungen, speziell von Spielen auf der Android Plattform. Die Hardware der Geräte wird immer besser. So haben aktuelle Smartphones und Tablets bereits dual- und sogar quad-core Prozessoren verbaut. Auch die Größe des Arbeitsspeichers wird in jeder neuen Produktgeneration erhöht. Auch wird die Dalvik Virtual Machine immer weiter entwickelt, was am Beispiel des Sprungs von Froyo (2.2) und Gingerbread (2.3) schön zu beobachten war, da der hinzugekommene Concurrent Garbage Collector zu deutlichen Performanceverbesserungen führte. Zudem ist die populärste Android Version bisher Gingerbread (2.3) und die nachfolgenden Generationen[1].

Was in dieser Arbeit aber noch nicht berücksichtigt wurde, ist unter anderem die Verwendung und Verwaltung von Bildern. Diese haben selbstverständlich eine wichtige Rolle bei Spielen und benötigen viel und bei mobilen Geräten wertvollen Speicher. Auch wurde noch nicht auf die durch OpenGL genutzte 3d Darstellung eingegangen und was in diesem Bereich an Optimierungen möglich sind.

[1] Aktueller Stand Ice Cream Sandwich mit der Versionsnummer 4.0.3, da die Honeycomb bzw. 3.x Versionen nur für Tablets ausgelegt war, wurden sie hierbei nicht berücksichtigt.

Anhang A

Inhalt der CD-ROM

A.1 Masterarbeit

Pfad: /

da_im1010629002.pdf .	Masterarbeit (Gesamtdokument)

A.2 Literatur

Pfad: /

ehringer_d-dvm.pdf . .	Dokument zu [3]
nicola_c_u-dvm.pdf . .	Dokument zu [18]
bornstein_d-dvm.pdf .	Dokument zu [8]
buzbee_cheng-jit.pdf .	Dokument zu [9]
dubroy_p-memory.pdf .	Dokument zu [14]
pruett_c-games.pdf . .	Dokument zu [21]

A.3 Performancetest-Daten

Pfad: /messergebnisse/ergebnisse/

containertests.pdf . . .	Messergebniss der Containertests
floatingpoint.pdf	Messergebniss der Floatingpointtests
getter_setter.pdf	Messergebniss mit public und private
schleifentests.pdf	Messergebniss der Schleifentests
static_final.pdf	Messergebniss der Tests mit static und final
simulation.pdf	Messergebniss der Test mit der Simulation

Pfad: /messergebnisse/

daten/	Trace- und Log-Dateien der Performancetests

A.4 Source-Code

Pfad: /quellcode/loganalyzer/

bin/	Klassendateien
src/	Quellcode-Dateien
.classpath	eclipse Klassenpfad-Datei
.project	eclipse Projekt-Datei

Pfad: /quellcode/thesisprojekt/

assets/	Grafik-Dateien
bin/	Klassendateien
gen/	Generierte Dateien (Android)
res	Ressourcen-Dateien (Android)
src/	Quellcode-Dateien
.classpath	eclipse Klassenpfad-Datei
.project	eclipse Projekt-Datei
AndroidManifest.xml	Android Manifest

Quellenverzeichnis

Literatur

[1] W. Frank Ablesen, Charlie Collins und Robi Sen. *Unlocking Android - A Developer's Guide*. Manning, 2009.

[2] Kai Altstaedt. „Performancekritische Applikationenen in Android - Droidenlauf". In: *iX Developer* 3 (Mai 2012), S. 70–75.

[3] David Ehringer. „The Dalvik Virtual Machine Architecture". In: *Mindful Mischief - Adventures in Software Development* (2010). URL: http://www.kiddai.com/NCTU/ebl/dex/The_Dalvik_Virtual_Machine.pdf.

[4] Sven Haiges. *Android - Schnelleinstieg*. 1. Aufl. entwickler.press, 2011.

[5] Christian Ullenboom. *Java ist auch eine Insel*. 9. Aufl. Galileo Computing, 2011. URL: http://openbook.galileocomputing.de/javainsel9/.

[6] Mario Zechner. *Beginning Android Games*. 1. Aufl. Apress, 2011.

Online-Quellen

[7] *ArrayList*. URL: http://docs.oracle.com/javase/6/docs/api/index.html?java/util/ArrayList.html (besucht am 18.03.2012).

[8] Dan Bornstein. *Google IO 2008 - Dalvik VM Internals*. März 2008. URL: https://sites.google.com/site/io/dalvik-vm-internals (besucht am 08.06.2012).

[9] Ben Cheng und Bill Buzbee. *Google IO 2010 - A JIT Compiler for Android's Dalvik VM*. Mai 2010. URL: http://www.google.com/events/io/2010/sessions/jit-compiler-androids-dalvik-vm.html (besucht am 08.06.2012).

[10] *Dalvik JIT*. URL: http://android-developers.blogspot.com/2010/05/dalvik-jit.html (besucht am 12.04.2012).

[11] *Datenblatt des HTC Dreams*. URL: http://www.areamobile.de/handys/1612-htc-dream/datenblatt (besucht am 28.03.2012).

[12] *Datenblatt des Samsung Galaxy Nexus*. URL: http://www.areamobile.de/handys/3014-samsung-galaxy-nexus/datenblatt (besucht am 28.03.2012).

[13] *Designing for Performance*. URL: http://developer.android.com/guide/practices/design/performance.html (besucht am 20.03.2012).

[14] Patrick Dubroy. *Google IO 2011 - Memory Managmement for Android Apps*. Mai 2011. URL: http://www.google.com/events/io/2011/sessions/memory-management-for-android-apps.html (besucht am 08.06.2012).

[15] *Java OutOfMemoryError - Eine Tragödie in sieben Akten*. URL: http://blog.codecentric.de/2010/01/java-outofmemoryerror-eine-tragodie-in-sieben-akten/ (besucht am 04.04.2012).

[16] *Memory Analyzer (MAT)*. URL: http://www.eclipse.org/mat/ (besucht am 03.04.2012).

[17] *Mobile Games Trend Report*. URL: http://newzoo.com/ENG/1632-Mobile_Games_Trend_Report.html (besucht am 28.03.2012).

[18] Carlo U. Nicola. *Einblick in die Dalvik Virtual Machine*. 2009. URL: http://www.fhnw.ch/technik/imvs/publikationen/artikel-2009/einblick-in-die-dalvik-virtual-machine (besucht am 09.07.2012).

[19] *ObjectPool Design Pattern*. URL: http://sourcemaking.com/design_patterns/object_pool (besucht am 19.04.2012).

[20] *Profiling with Traceview and dmtracedump*. URL: http://developer.android.com/guide/developing/debugging/debugging-tracing.html (besucht am 22.03.2012).

[21] Chris Pruett. *Google IO 2009 - Writing Real-Time Games for Android*. März 2009. URL: http://www.google.com/events/io/2009/sessions/WritingRealTimeGamesAndroid.html (besucht am 20.06.2012).

[22] *Sales of Mobile Devices*. URL: http://www.gartner.com/it/page.jsp?id=1543014 (besucht am 16.04.2012).

[23] *Sales of Mobile Devices Grew*. URL: http://www.gartner.com/it/page.jsp?id=1848514 (besucht am 28.03.2012).

[24] *Smartphone Sales Soared*. URL: http://www.gartner.com/it/page.jsp?id=1924314 (besucht am 16.04.2012).

[25] *Sourcecode ArrayList*. URL: http://www.docjar.com/html/api/java/util/ArrayList.java.html (besucht am 21.05.2012).

[26] *Sourcecode LinkedList*. URL: http://www.docjar.com/html/api/java/util/LinkedList.java.html (besucht am 21.05.2012).

[27] *Sourcecode Stack*. URL: http://www.docjar.com/html/api/java/util/Stack.java.html (besucht am 21.05.2012).

[28] *Sourcecode Vector*. URL: http://www.docjar.com/html/api/java/util/Vector.java.html (besucht am 21.05.2012).

[29] *Spieleentwicklung mit Android: Sorgenkind Performance*. URL: http://www.androidig.de/index.php/2009/08/31/spieleentwicklung-mit-android-sorgenkind-performance/ (besucht am 24.04.2012).