Bachelorarbeit

Gestaltung und Entwicklung einer stark parallelisierten dreidimensionalen Simulation von orbitalen Himmelskörpern.

Autor:	Dennis Goßler
Matrikel-Nr.:	11140150
Adresse:	Oswald-Greb-Str. 7
	42859 Remscheid
	dennisgossler98@gmail.com

Erstprüfer:	Prof. Dr. Christian Kohls
Zweitprüfer:	Alexander Dobrynin, M.Sc.

Remscheid, 30.06.2022

Inhaltsverzeichnis

Inhaltsverzeichnis ..... B
Abbildungsverzeichnis ..... F
Tabellenverzeichnis ..... G
1 Abstract 2
2 Einleitung 3
2.1 Relevanz 3
2.2 Zielsetzung 3
2.3 Recherchephase 4
2.4 Projektplan 4
2.4.1 Definitionsphase 4
2.4.2 Durchführung 5
2.4.3 Thesenüberprüfung 5
2.4.4 Auswertung 5
2.5 Testumgebung 5
2.6 Grundaufbau der Anwendung 6
2.6.1 Gravitation 6
2.6.2 Kollisionen 6
2.6.3 Kollisionsbearbeiter 7
2.7 Benutzeroberfläche 7
2.7.1 Elementauflistung 7
2.7.2 Schrift 8
2.8 Benutzeroberflächendesign 8
2.9 Parallelisierung 9
2.9.1 Sweep and Prune 9
2.9.2 Gravitationssystem 10
2.9.3 Kollisionsbearbeiter 11
2.10 Programmablauf 11
2.10.1 Aktualisierungsfunktionen 12

2.10.2 Renderfunktion 12
3 Hauptteil 13
3.1 Paralleles Iterieren auf einer Liste 13
3.1.1 Herangehensweise 13
3.1.2 Funktionsaufbau 14
3.2 Sweep and Prune Algorithmus 15
3.2.1 Datenstruktur 15
3.2.2 Sequenzielle Kollisionserkennung 16
3.2.3 Parallele Kollisionserkennung 18
3.2.4 Veränderung der Objektpositionen 18
3.2.5 Sequenzielle Teststruktur 18
3.2.6 Parallele Teststruktur 19
3.3 Das Gravitationssystem 20
3.3.1 Gravitationsobjekt 20
3.3.2 Gravitationsmanager 20
3.3.3 Umsetzung des Gravitationsalgorithmus 21
3.3.4 Parallelisierung 22
3.3.5 Teststruktur 22
3.4 Kollisionsbearbeiter 24
3.4.1 Kollisionsachse 24
3.4.2 Abprall von Objekten 25
3.4.3 Überschneidung der Objektboxen 25
3.4.4 Zersplitterung 26
3.4.5 Parallelisierung 26
3.4.6 Teststruktur 27
3.5 OpenGL Rendering 28
3.5.1 Simpler Renderprozess 28
3.5.2 Renderprozess am selben Modell 29
3.5.3 Instancing 30
3.5.4 Testdurchführung 30

3.6 Entwicklung der Benutzeroberfläche 32
3.6.1 Grundelement 32
3.6.2 Einschränkungen der Größe und Position 32
3.6.3 Schachtelung von Elementen 32
3.6.4 Schrift 33
3.6.5 Interaktion 34
3.7 Aufbau der Thesenüberprüfung 35
3.7.1 Einstellungen 35
3.7.2 Testkonfigurationsdatei 35
3.7.3 Speicherung der Testresultate 36
3.7.4 Testsysteme 36
4 Fazit 37
4.1 Ergebnisse 37
4.1.1 Zusammengetragene Ergebnisse 37
4.1.2 Vergleich der Testergebnisse 38
4.1.3 Overhead der Parallelisierung 40
4.2 Erkenntnisse 41
4.3 Ausblick 41
Quellenverzeichnis I
Q1 Projekt OuterSpace I
Q2 Demtröder2006_Book_Experimentalphysik1 I
Q3 Collisions in 1-dimension I
Q4 Veranschaulichung der Anwendung I
Q5 Mathematics of Satellite Motion I
Q6 JUnit I
Q7 Kotlinx serialization I
Q8 Kotlinx coroutines II
Q9 ADynamic Bounding Volume Hierarchy for Generalized Collision Detection II
Q10 Lightweight Java Game Library II
Q11 42 Years of Microprocessor Trend Data II

Q12 Sweep and prune II
Q13 Font Rendering II
Q14 Enhanced Sweep and Prune II
Anhang III
A1 Sweep-and-prune Testfälle III
A2 Gravitationssystem Testfälle IV
A3 Testsysteme VIII
A4 Diagrammresultate IX
A5 Testresultate X
A6 Overhead Testcode XIII

Abbildungsverzeichnis

Abbildung 1: [Virtueller Prototyp der Einstellungsoberfläche]..... 8
Abbildung 2: [Klassendiagramm SAP]..... 9
Abbildung 3: [Klassendiagramm Gravitationssystem]..... 10
Abbildung 4: [Klassendiagramm Kollisionsbearbeiter]..... 11
Abbildung 5: [Aktivitätsdiagramm Aktualisierungsfunktion]..... 12
Abbildung 6: [Listenzerteilung zur Parallelisierung]..... 14
Abbildung 7: [Erweiterungsfunktion foreachParallel]..... 14
Abbildung 8: [Klassendefinition SAP ohne Funktionen]..... 15
Abbildung 9: [Klassendefinition EndPoint]..... 16
Abbildung 10: [Schnittstelle IHitbox]..... 16
Abbildung 11: [SAP-Kollisionsüberprüfung der X-Achse]..... 17
Abbildung 12: [SAP-Kollisionsüberprüfung der Y /Z-Achse]..... 17
Abbildung 13: [Schnittstelle IGravity]..... 20
Abbildung 14: [Enum GravityProperties]..... 21
Abbildung 15: [Aufbau der Gravitationsfunktion]..... 22
Abbildung 16: [Schnittstelle IApplier]..... 24
Abbildung 17: [Testskript Renderverfahren]..... 31
Abbildung 18: [Schachtelung von UI-Elementen]..... 33
Abbildung 19: [Zugehöriger Code von Abbildung 11]..... 33
Abbildung 20: [Beispiel einer settings.json Datei]..... 35
Abbildung 21: [Beispiel der Test - Konfigurationsdatei]..... 36
Abbildung 22: [Test mit 500 Objekten]..... 38
Abbildung 23: [Test mit 1000 Objekten]..... 39
Abbildung 24: [Test mit 5000 Objekten]..... 39
Abbildung 25: [Overhead der Sortierfunktion zum prozentualen Gesamtaufwand)]..... 41

Tabellenverzeichnis

Tabelle 1: [Leistungstest 1: redundante Objekte].....29
Tabelle 2: [Leistungstest 2: gleiches Modell] 29
Tabelle 3: [Leistungstest 3: Instancing].....30
Tabelle 4: [Zusammenfassung: Aktualisierungszeiten der untersuchten Systeme].....37
Tabelle 5: [Zusammenfassung: FPS der untersuchten Systeme] 38
Tabelle 6: [Overhead-Berechnung].....40

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig und ohne fremde Hilfe angefertigt habe. Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Gummersbach, 30.06.2022

30.06.22, Gopf
Name (Unterschrift)

1 Abstract

Diese Bachelorarbeit beschäftigt sich mit der Entwicklung und der Erarbeitung einer Anwendung, bei der manche Prozesse sowohl parallel als auch sequenziell ausführt werden. Anhand eines ausgewählten Szenarios soll veranschaulicht werden, inwiefern sich gewisse Problemstellungen, wie zum Beispiel Kollisionserkennungsberechnungen, parallelisieren lassen. Den Problemstellungen liegen in der Regel immer eine größere Anzahl von Objekten zugrunde, auf die eine Operation mit geringem Berechnungsaufwand angewendet wird.

Unter Berücksichtigung spezieller Bedingungen wird die Anwendung auf mehrere Systeme appliziert und liefert jeweils Indikatoren über die sequenzielle und parallele Leistung der entwickelten Module. Außerdem wird untersucht, welcher Mehraufwand beim Auslagern und Synchronisieren von gewissen Problemstellungen entsteht.

Die Analyse liefert zudem Erkenntnisse, ab welcher Objektanzahl sich das parallele Ausführen dieser Anwendung lohnt.

2 Einleitung

Die Bachelorarbeit demonstriert den Prozess einer Programmentwicklung. Das Programm ist sequenziell sowie parallel ausführbar. Die geforderte Applikation ist so ausgelegt, dass sie mit einer großen Anzahl von Daten interagiert, um viele kleine Berechnungen in Gruppen zu parallelisieren. Das Szenario soll einen normalen Entwicklungsprozess abbilden und damit zeigen, welchen Mehraufwand und welche Vorteile eine Parallelisierung gewisser Prozesse mit sich bringt.

Die zu entwickelnde Anwendung zeigt eine dreidimensionale Abbildung unserer Welt, einschließlich ihrer orbitalen Himmelskörper.

Einstellungen über die Menge der Objekte und deren Verhalten kann individuell gesetzt werden. Diese Einstellungsmethode ermöglicht, bei potenziellen Kollisionen die Menge und Streuung der entstandenen Kind - Objekte zu bestimmen. Durch verschiedene Kameraperspektiven kann die Simulation aus unterschiedlichen Blickwinkeln betrachtet werden. Dadurch ist es möglich, sich frei im dreidimensionalen Raum zu bewegen.

2.1 Relevanz

Computeranwendungen verwenden oftmals Algorithmen, die nicht sehr rechenintensiv sind, aber auf viele Objekte angewendet werden. Eine Parallelisierung könnte dabei helfen, diese Berechnungszeiten zu verkürzen. Zudem setzen die größten CPU - Hersteller weitestgehend auf mehr Kerne in ihren CPU’s, anstatt schnellere Taktfrequenzen einzusetzen, die das parallele Bearbeiten von Aufgaben zusätzlich begünstigen. [Q11 42 Years of Microprocessor Trend Data S.II]

2.2 Zielsetzung

Das Ziel dieser Bachelorarbeit ist die Entwicklung einer Simulation, die die Kollisionsberechnungen und Bewegungen von Objekten stark parallelisiert und erkennbar macht. Sie überprüft außerdem, ob Leistungsverbesserungen durch die Parallelisierung erkennbar werden. Hierfür wird eine geeignete Projektarchitektur bestimmt. Daraus resultiert eine *Kotlinanwendung*, die mithilfe der *Kotlinx* Bibliothek [Q8 Kotlinx coroutines S.II] die vorgegebenen Prozesse parallelisiert.

2.3 Recherchephase

Zur Ermittlung von dreidimensionalen Kollisionen im Raum muss in der Recherchephase ein Kollisionsalgorithmus gefunden werden. Der Algorithmus soll eine Parallelisierung ermöglichen.

Außerdem gilt es, für die Himmelskörper eine geeignete Formel zur Berechnung einer Gravitation zu finden.

2.4 Projektplan

Die Planung vollzieht sich in mehreren Schritten. Mit Fortschreiten der einzelnen Projektphasen ist das weitere Vorgehen agil geplant. Die Applikation der Bachelorarbeit beruht auf der Entwicklung, Planung und den Tests einer einzigen Person. Das gesamte Projekt beinhaltet vier Hauptphasen. Die gesamte Anwendung basiert auf einem weiteren bereits entwickelten Projekt. Deswegen wird oftmals nur von einer Anpassung oder Ergänzung berichtet. Dieses Thema wird im Kapitel 2.6 [Grundaufbau der Anwendung S.6] noch ausführlicher behandelt.

2.4.1 Definitionsphase

Die erste Phase besteht größtenteils aus dem Planen der Architektur, dem Erstellen von Diagrammen und dem Gestalten der Benutzeroberfläche.

Die Architektur ist so gestaltet, dass einzelne Module austauschbar sind. Somit ist das schnelle Auswechseln eines parallel ausgeführten Systems mit einem sequenziellen System gewährleistet [2.9 Parallelisierung S.9].

Um gewisse Abläufe und Prozesse zu veranschaulichen, sind UML Diagramme mittels *Draw.io* zu erstellen [2.10 Programmablauf S.11].

Für das Gestalten der Benutzeroberfläche (*UI*) wird ein virtueller Prototyp erstellt. Dieser zeigt eine Vorabversion der UI und soll eine grobe Version des Layouts zeigen. Zur Vorlagenerstellung des Prototyps dient die Webapplikation *Figma* [2.8 Benutzeroberflächendesign S.8].

2.4.2 Durchführung

Im Verlauf dieser Phase werden die Applikation und die zugehörigen Algorithmen entwickelt. Durch *JUnit* werden die verwendeten Algorithmen in ihrer sequenziellen und parallelen Form getestet. [2.5 Testumgebung S.5]

Die Projektdurchführung ist nochmals detaillierter im Abschnitt 3 [Hauptteil S.13] beschrieben.

2.4.3 Thesenüberprüfung

Um zu überprüfen, ob und inwiefern sich das Parallelisieren einzelner Algorithmen eignet, wird es der herkömmlichen sequenziellen Gestaltungsweise gegenübergestellt. Durch unterschiedlich skalierte Testdurchläufe wird die Leistung des jeweiligen Systems hervorgebracht. Auf der Grundlage von unterschiedlich starken und schwachen CPU's werden die Systeme ausgetestet. Die verwendeten Systeme sind im Anhang A3 [Testsysteme S.VIII] aufgelistet.

2.4.4 Auswertung

Im Verlauf dieser Phase werden die gesammelten Daten zusammengetragen und evaluiert. Es ist an dieser Stelle zu überprüfen, inwiefern sich die Parallelisierung auf die unterschiedlichen Systeme auswirkt. Es ist ebenso zu überprüfen, ob diese Methode CPU's mit wenig Kernen dennoch einen signifikanten Vorteil bietet.

2.5 Testumgebung

Um die verschiedenen Anwendungsalgorithmen zu überprüfen, werden dementsprechend Tests entwickelt. Diese Tests verwenden die *JUnit* Bibliothek [Q6 JUnit S.I]. Auf Grund der mangelnden Eignung mancher Objekte für die dazugehörigen Tests wird es nötig, separate Testobjekte zu erstellen, die auf die zugehörige Schnittstelle zugreifen. Die Testung gewisser privater Methoden eines Algorithmus werden durch Reflexion überprüft.

2.6 Grundaufbau der Anwendung

Die zu entwickelnde Simulation basiert auf einer Projektarbeit, die im Zuge des Wahlpflichtfaches *Computergrafik und Animation* entstanden ist. In diesem WPF wurde eine dreidimensionale Weltraumsimulation geschaffen, welche es ermöglicht, verschiedene Sonnensysteme zu generieren und diese zu animieren. Die Applikation nutzt *Kotlin* als Programmiersprache und *OpenGL* zur dreidimensionalen Darstellung.

Das oben genannte Projekt wurde in Zusammenarbeit mit Frau Anastasia Chouliaras erstellt [Q1 Projekt OuterSpace S.I]. Die Anwendung dient als Grundstruktur und ist an die gegebene Problemstellung anzupassen.

Viele der entwickelten Projektmerkmale sind für die spezifische Problemstellung nicht geeignet und müssen modifiziert oder umgeschrieben werden. In den folgenden Abschnitten des Kapitels wird darauf näher eingegangen.

2.6.1 Gravitation

Das Projekt Outer Space [Q1 Projekt OuterSpace S.I] besitzt Planeten und Monde, die sich auf Umlaufbahnen um ein zentrales Objekt bewegen. Das alte System nutzt trigonometrische Funktionen zum Platzieren der orbitalen Himmelskörper an einem bestimmten Zeitpunkt. Dieses System eignet sich nur sehr bedingt für die zu entwickelnde Applikation, da die zu platzierenden Objekte sich nur sehr starr auf einer Umlaufbahn platzieren lassen. Des Weiteren ermöglicht das neue System eine Anpassung der Geschwindigkeit bei erfolgten Kollisionen.

Das alte System ist gegen einen neuen Algorithmus auszuwechseln, der es ermöglicht, dass alle Objekte miteinander interagieren. Hierfür ist das Newtonsche Gravitationsgesetz[1] zur Veränderung der Objektpositionen einzusetzen.

2.6.2 Kollisionen

Die Grundstruktur des zu entwickelnden Systems besitzt kein Kollisionsalgorithmus. Wie in Abschnitt 2.3 [Recherchephase S.4] beschrieben, gilt es einen geeigneten Algorithmus zu favorisieren, der die gegebenen Kriterien erfüllt.

---

[1] [Q2 Demtröder2006_Book_Experimentalphysik1 S.I] (S.80 2.47a)

2.6.3 Kollisionsbearbeiter

Nach einer erfolgten Kollision soll eine Interaktion der kollidierten Objekte stattfinden. Bei diesem Prozess entscheidet der Kollisionsbearbeiter, ob die beiden Objekte voneinander abprallen oder in kleinere Objekte zerlegt werden sollen.

2.7 Benutzeroberfläche

Das Projekt Outer Space [Q1 Projekt OuterSpace S.I] besitzt ein sehr rudimentäres Benutzeroberflächensystem. Entsprechend der Bildschirmbreite und Bildschirmhöhe lassen sich Elemente nur prozentual platzieren. Außerdem ist das System nur für die Ausgabe von Bilddateien geeignet und gewährleistet dementsprechend keine direkte Interaktion. Dieses System gilt es so anzupassen, dass auch sehr komplexe Benutzeroberflächen designt werden können.

2.7.1 Elementauflistung

Ein beliebiger Nutzer soll in der Lage sein, eigenständig Texte und Zahlen einzugeben, Knöpfe zu betätigen, Schieberegler zu verschieben als auch Optionen an- und auszuschalten. Folgende Aufzählung impliziert alle zu entwickelnden UI Elemente.

- **Eingabe**
  - Auswahlfeld
  - Knopf
  - Schieberegler
  - Textfeld
  - Umschaltknopf
  - Veränderbarer Text
  - Zahlenfeld
- **Ausgabe**
  - Bild
  - Text
- **Anordnung**
  - Anordnungsliste
  - Anordnungsrechteck
  - Kreis
  - Rechteck
  - Scrollleiste

2.7.2 Schrift

Die zu entwickelnde Anwendung benutzt OpenGL als Darstellungsbibliothek. Diese Bibliothek besitzt keine native Umsetzung zur Darstellung von Schrift.

Da für dieses Projekt oftmals eine direkte Ausgabe von Text von Nöten ist, gilt es diese Funktion zu implementieren. Ein Beispiel für eine solche Textausgabe sind zum Beispiel die *FPS*[2], welche dem Nutzer einen Leistungsindikator über ausgeführte Applikationen geben.

2.8 Benutzeroberflächendesign

Um Einstellungen für die Applikation vorzunehmen, ist eine Benutzeroberfläche zu implementieren. Diese Einstellungen sollen zum Beispiel entscheiden, ob die Applikation parallel oder sequenziell ausgeführt wird und wie viele Elemente generiert werden sollen. Die folgende Abbildung dient als Vorlage der Einstellungsoberfläche und beschreibt das Farbschema der Applikationselemente.

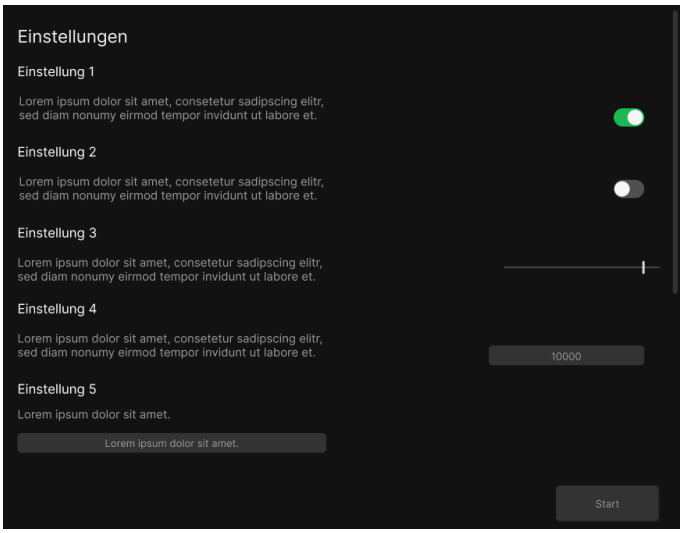


Abbildung 1: [Virtueller Prototyp der Einstellungsoberfläche]

[2] Ist die Abkürzung für *frames per second* (die Bildrate).

2.9 Parallelisierung

Um eine parallele als auch sequenzielle Ausführung der Applikation zu ermöglichen, wird an diesen Stellen eine abstrakte Klasse eingesetzt. Die jeweilige abstrakte Klasse definiert nur Methoden, die sequenziell ausgeführt werden. Wird eine Methode auch parallel ausgeführt, wird diese ebenfalls als abstrakt deklariert und in deren Kindklassen implementiert. Dabei ist zu berücksichtigen, dass die parallelen Kindklassen eine Variable namens *jobCount* enthalten. Diese Variable wird dann eingesetzt, wenn entschieden werden soll, wie viele Jobs eingesetzt werden.

2.9.1 Sweep and Prune

Das unten dargestellte UML - Klassendiagramm veranschaulicht das verwendete System anhand des Kollisionsalgorithmus. Hierbei werden die zwei Methoden *sort()* und *checkCollision()* in der *ParallelSAP* Klasse parallel sowie in der SAP Klasse sequenziell implementiert.

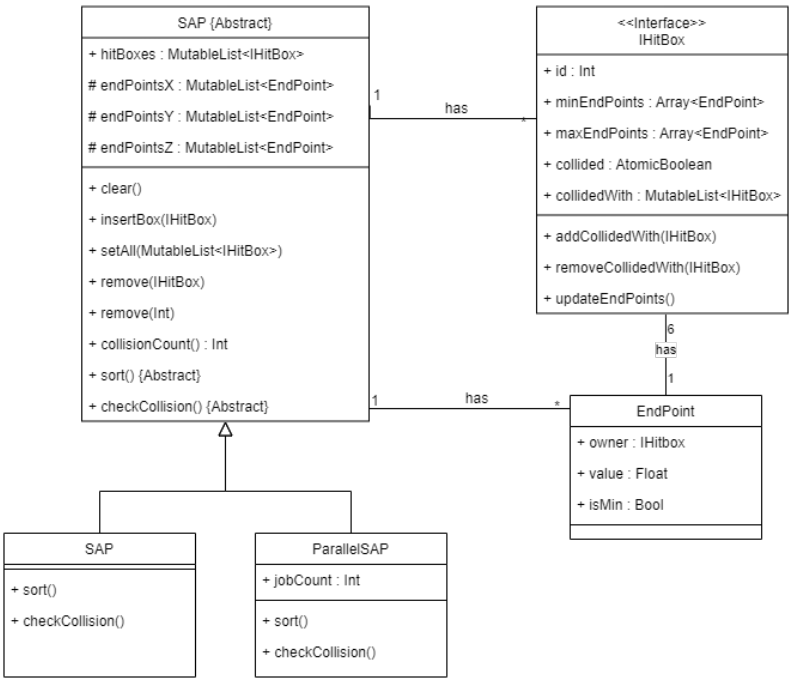


Abbildung 2: [Klassendiagramm SAP]

2.9.2 Gravitationssystem

Ähnlich wie im oberen Abschnitt 2.9.1 [Sweep and Prune S.9] soll diese Abbildung das genutzte System zum Austauschen der parallelen und sequenziellen Module veranschaulichen.

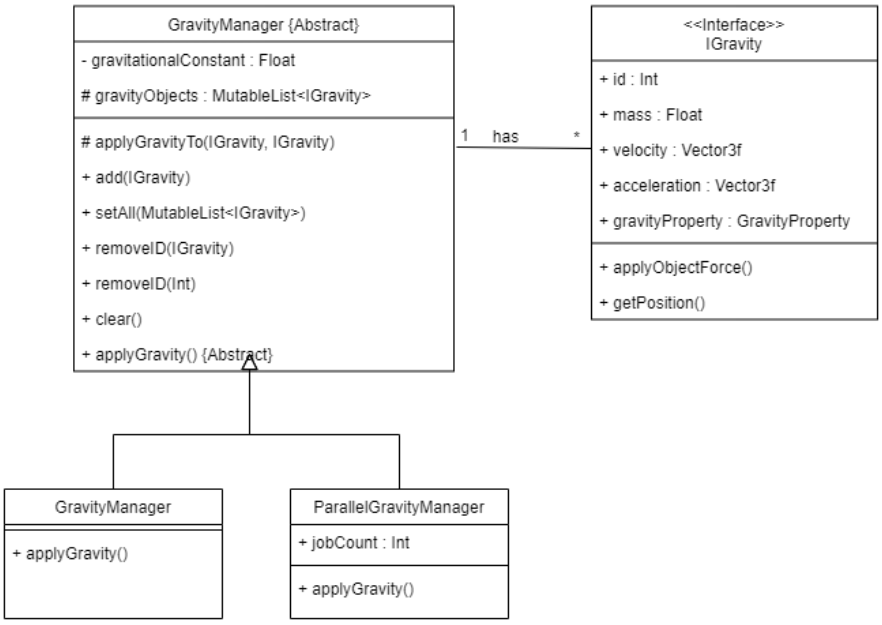


Abbildung 3: [Klassendiagramm Gravitationssystem]

2.9.3 Kollisionsbearbeiter

Folgendes Diagramm zeigt den Aufbau des Kollisionsbearbeiters.

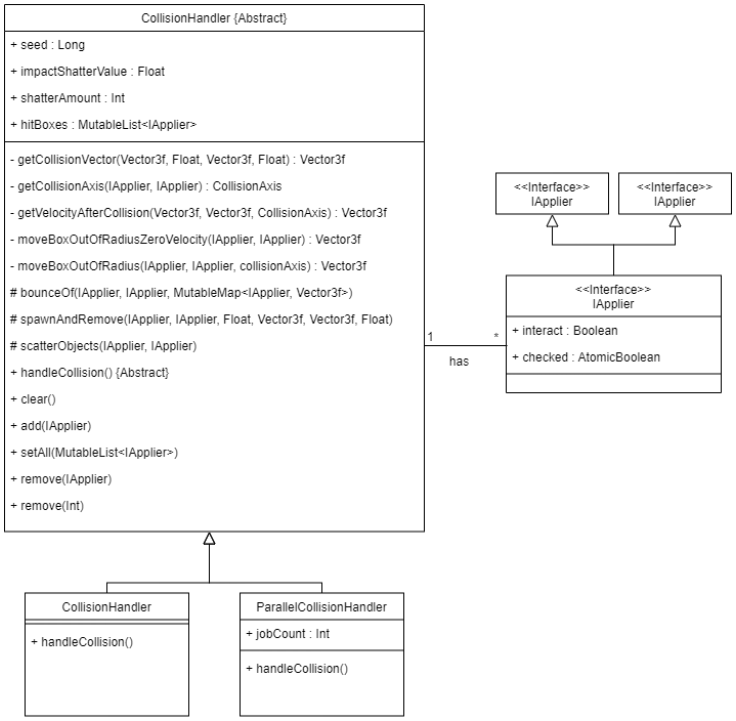


Abbildung 4: [Klassendiagramm Kollisionsbearbeiter]

2.10 Programmablauf

Zu Beginn der Applikation wird eine Schleife gestartet, die so lange von vorne beginnt, bis die Applikation durch den Nutzer beendet wird. Diese Schleife beinhaltet eine Funktion für das Aktualisieren der visuellen Objekte. Sie impliziert ebenfalls eine Funktion für die Aktualisierung der Benutzeroberfläche und eine Funktion für das Zeichnen der Objekte auf dem Bildschirm (*rendern*). Die beiden Aktualisierungsfunktionen sind auf eine bestimmte Anzahl von Aktualisierungen pro Sekunde (*UPS*) limitiert. Die UPS der Benutzeroberfläche sind auf 60 festgesetzt. Die maximalen UPS der Objekte können in der Benutzeroberfläche jedoch auf 1 bis 600 Aktualisierungen pro Sekunde festgelegt werden.

Bei dem wiederholten Ausführen der Schleife wird die *Renderfunktion* jedoch so oft wie möglich aufgerufen. Werden die festgelegten UPS nicht in einer Sekunde erreicht, werden sie zur Vermeidung eines Aufstaus von Aktualisierungen gedrosselt.

2.10.1 Aktualisierungsfunktionen

Die Aktualisierungsfunktion der Benutzeroberfläche (*UI*) aktualisiert die einzelnen UI - Elemente. Dabei wird beispielsweise überprüft, ob die Computermaus über einem Element positioniert ist.

In der Aktualisierungsfunktion der Objekte wird die aktuelle UPS - Anzahl berechnet und dem Nutzer durch die Benutzeroberfläche angezeigt. Zudem werden die Objekte durch Ihre Container [2.9 Parallelisierung S.9] aktualisiert. Bei der Aktualisierung kann ein Objekt verschoben, umgefärbt oder entfernt werden. Folgendes Diagramm zeigt den Ablauf der Objektveränderungen.

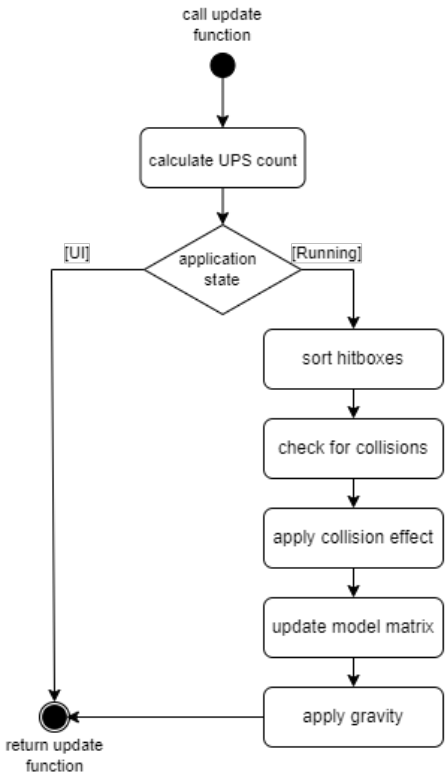


Abbildung 5: [Aktivitätsdiagramm Aktualisierungsfunktion]

2.10.2 Renderfunktion

In der *Renderfunktion* werden die Objektrohdaten an die Grafikkarte übermittelt, damit ein Bild ausgegeben werden kann. Außerdem wird bei der Renderfunktion die aktuelle Framerate (*FPS*) berechnet und angezeigt. Der genaue Prozess zum Anzeigen der Objekte wird im Kapitel 3.5 [OpenGL Rendering S.28] noch genauer beschrieben.

3 Hauptteil

Der Hauptteil beschäftigt sich mit der Entwicklung und den daraus folgenden Problemstellungen der einzelnen Implementierungsphasen. Zudem wird die Planung und Durchführung der Analysephase beschrieben.

3.1 Paralleles Iterieren auf einer Liste

In der Anwendung wird oftmals über eine Liste iteriert und auf das ausgewählte Objekt eine Operation angewendet. Benötigt eine Operation eine längere Zeitspanne, kann es von Vorteil sein, dass die Berechnungen auf den einzelnen Objekten parallel ausgeführt werden.

Dieser Abschnitt befasst sich mit der Entwicklung einer Funktion höherer Ordnung, die es ermöglicht, auf den gegebenen Objekten eine Operation anzuwenden.

3.1.1 Herangehensweise

Eine objektbezogene Aufgabenstellung kann sehr unterschiedliche Zeitspannen in Anspruch nehmen. Deswegen nimmt die *foreachParallel* Funktion als ersten Parameter eine Jobanzahl entgegen. Mit diesem Wert wird die Liste in verschiedene Abschnitte unterteilt. Beim Start der Funktion wird die Abschnittsgröße (*c*) und ein Restwert (*r*) bestimmt.

$$c = floor\left(\frac{list.size}{jobCount}\right)$$

$$r = list.size - (c * jobCount)$$

Eine *For - Schleife* zählt nun von 0 bis zur Jobanzahl und erstellt dementsprechend viele Jobs und fügt diese einer Job - Liste hinzu. Jeder Job bearbeitet somit einen gewissen unabhängigen Bereich.

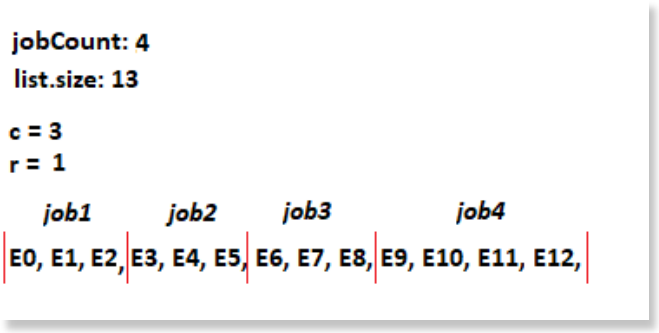


Abbildung 6: [Listenzerteilung zur Parallelisierung]

3.1.2 Funktionsaufbau

Die *foreachParallel* Funktion erweitert die Klasse *List* und nimmt ein *predicate* entgegen, dass die Operation auf das zugehörige Element vornimmt. Zusätzlich kann das *predicate* auch einen Integer aufnehmen, welches den aktuellen Index des Zugriffselements enthält. Die Funktion *foreachParallelIndexed* wird überladen und somit kann die Funktion auch ohne Zugriffsindex ausgeführt werden.

```
@OptIn(DelicateCoroutinesApi::class)
suspend fun <T> List<T>.foreachParallel(jobCount : Int, predicate : ((T)->Unit)) {
    this.foreachParallelIndexed(jobCount) { t: T, _: Int -> predicate(t) }
}

@OptIn(DelicateCoroutinesApi::class)
suspend fun <T> List<T>.foreachParallelIndexed(jobCount : Int, predicate : ((T, index: Int)->Unit)) {
    val items = this
    val jobs = mutableListOf<Job>()

    val chunkSize = items.size / jobCount
    val remains = items.size - (chunkSize * jobCount)

    for(jobIndex in 0 until jobCount){
        jobs.add(GlobalScope.launch(){
            for(index in jobIndex * chunkSize until (jobIndex + 1) * chunkSize + if(jobIndex != jobCount-1) 0 else remains){
                predicate(items[index], index)
            }
        })
    }
    jobs.joinAll()
}
```

Abbildung 7: [Erweiterungsfunktion foreachParallel]

3.2 Sweep and Prune Algorithmus

Der Sweep and Prune Algorithmus (*SAP*) [Q12 Sweep and prune S.II] ist ein Algorithmus zur effizienten Kollisionserkennung von Objekten im dreidimensionalen Raum. Ein Objekt (*Hitbox*) im *SAP* definiert sich durch seinen achsenorientierten Begrenzungskasten. Auf jeder Achse des dreidimensionalen Koordinatensystems besitzt jedes Objekt bei dieser Darstellungsform einen minimalen (*min*) und einem maximalen *(max*) Wert. Daher hat jedes Objekt in einer dreidimensionalen Umgebung sechs Werte. Die Gesamtheit aller Werte einer Achse werden in einer sortierten Liste gespeichert.

3.2.1 Datenstruktur

Um den Algorithmus darzustellen, werden gewisse Datenstrukturen benötigt. Hierbei orientiert man sich stark an den Strukturen aus dem Paper [Q12 Sweep and prune S.II].

```
abstract class AbstractSAP{

    var hitBoxes : MutableList<IHitBox> = mutableListOf()

    protected var endPointsX : MutableList<EndPoint> = mutableListOf()
    protected var endPointsY : MutableList<EndPoint> = mutableListOf()
    protected var endPointsZ : MutableList<EndPoint> = mutableListOf()

    fun clear(){...}

    fun insertBox(hitBox : IHitBox){...}

    fun setAll(hitBoxes: MutableList<IHitBox>){...}

    fun remove(hitBox : IHitBox){...}

    fun remove(id : Int){...}

    fun collisionCount() = hitBoxes.fold(0){acc, iHitBox -> acc + if (iHitBox.collided.get()) 1 else 0}

    abstract suspend fun sort()

    abstract suspend fun checkCollision()
}
```

Abbildung 8: [Klassendefinition SAP ohne Funktionen]

Das *SAP* enthält alle Objekte und drei sortierte Listen aus deren Endpunkten. Die Endpunktlisten sind immer doppelt so lang, wie die Liste der *Hitboxen*.

```
data class EndPoint(
    val owner : IHitBox,
    var value : Float,
    val isMin : Boolean
)
```

Abbildung 9: [Klassendefinition EndPoint]

Ein Endpunkt enthält immer die Referenz auf seinen Besitzer, den jeweiligen Koordinatenwert und enthält ebenfalls die Festlegung der Frage, ob es sich um den *min* Wert seines Besitzers handelt.

```
interface IHitBox {
    val id : Int

    val minEndPoints : Array<EndPoint>
    val maxEndPoints : Array<EndPoint>

    var collided : AtomicBoolean
    var collisionChecked : AtomicBoolean
    var collidedWith : MutableList<IHitBox>

    fun addCollidedWith(hitBox : IHitBox)
    fun removeCollidedWith(hitBox : IHitBox)

    fun updateEndPoints()
    fun translateLocal(vec : Vector3f)
}
```

Abbildung 10: [Schnittstelle IHitbox]

Jede *Hitbox* besitzt einen unikalen Identifizierer, der bei der Erstellung vom *SAP* zugewiesen wird. Außerdem enthält jede *Hitboxstruktur* diejenigen Kollisionspartner, die durch das *SAP* ermittelt werden.

3.2.2 Sequenzielle Kollisionserkennung

Um zu erkennen, ob ein Objekt mit den anderen *n* - Boxen im *Sap* kollidiert, wird über die *endPointXListe* iteriert. Sobald es sich um einen minimalen Endpunkt handelt, wird eine zweite Schleife gestartet, die ab dem Endpunktindex beginnt und so lange läuft, bis der zugehörige maximale Endpunktwert gefunden werden konnte. Alle minimalen Endpunkte zwischen den Objektendpunkten können als Kollisionen auf der X-Achse betrachtet werden.

```
endPointsX.forEachIndexed { index, endpoint ->
    if (endpoint.isMin) {

        var i = index + 1
        while ( i < endPointsX.size){
            val endpointNext = endPointsX[i]

            if (endpointNext.isMin){
                val collideWith = endpointNext.owner
                collideWith.collided.set(true)
                collideWith.collidedWith.add(endpoint.owner)
                endpoint.owner.collided.set(true)
                endpoint.owner.collidedWith.add(collideWith)

            }else
                if(endpoint.owner.id == endpointNext.owner.id)
                    break

            i++
        }
    }
}
```

Abbildung 11: [SAP-Kollisionsüberprüfung der X-Achse]

Um zu überprüfen, ob die Objekte auch auf den anderen Achsen kollidieren, wird danach über die Liste der Objekte iteriert. Wenn ein Objekt ein oder mehrere Kollisionen aufweist, werden diese sukzessiv überprüft. Die Überprüfung vergleicht die minimalen und maximalen Werte der potenziell kollidierenden Objekte und entscheidet, ob eine Kollision auf der Y - und Z - Achse stattfindet.

```
hitBoxes.forEach { hitBox ->
    if(hitBox.collided.get()){
        hitBox.collidedWith.toList().forEach { collideHitBox ->

            if(!collideHitBox.collisionChecked.get() &&(
                (hitBox.maxEndPoints[1].value < collideHitBox.minEndPoints[1].value
                || hitBox.minEndPoints[1].value > collideHitBox.maxEndPoints[1].value) ||
                (hitBox.maxEndPoints[2].value < collideHitBox.minEndPoints[2].value
                || hitBox.minEndPoints[2].value > collideHitBox.maxEndPoints[2].value)
            ))
            {
                if(hitBox.collidedWith.size < 2)
                    hitBox.collided.set(false)

                if(collideHitBox.collidedWith.size < 2)
                    collideHitBox.collided.set(false)

                collideHitBox.collidedWith.remove(hitBox)
                hitBox.collidedWith.remove(collideHitBox)
            }
        }
        hitBox.collisionChecked.set(true)
    }
}
```

Abbildung 12: [SAP-Kollisionsüberprüfung der Y /Z-Achse]

3.2.3 Parallele Kollisionserkennung

Beim Ausführen des Algorithmus wird statt der normalen *foreachIndexed Schleife* die neu entwickelte *foreachParallelIndex* Erweiterungsfunktion der Listklasse verwendet [3.1 Paralleles Iterieren auf einer Liste S.13].

Da beim Durchlaufen der Liste auf andere Listenobjekte zugegriffen werden muss, muss gewährleistet sein, dass ein simultanes Zugreifen von zwei unterschiedlichen Jobs verhindert wird. Dieses Problem wird gelöst, indem atomare Typen, wie zum Beispiel *AtomicBoolean,* verwendet werden. Außerdem werden Operationen, wie ein Kollisionspartner hinzufügen/ entfernen, über eine synchronisierte Funktion in der *Hitbox* Klasse ausgeführt.

Dieses System wird auch beim zweiten Teil der Kollisionserkennung verwendet.

3.2.4 Veränderung der Objektpositionen

Wie im Abschnitt 3.2 [Sweep and Prune Algorithmus S.15] beschrieben, müssen alle Endpunktlisten vor der Ausführung der Kollisionserkennung stets sortiert sein. Wenn ein oder mehrere Objekte verschoben oder skaliert werden, müssen seine Endpunktpositionen in den Endpunktlisten neu einsortiert werden. Da in dieser Applikation davon ausgegangen werden kann, dass fast alle Objekte in einem Updatezyklus die Position verändern, ist es effizienter, die drei Listen komplett durchzusortieren.

In der *sortParallel* Funktion des SAP-Objektes werden die drei Listen in drei verschiedenen Jobs parallel sortiert.

3.2.5 Sequenzielle Teststruktur

Um sicherzustellen, dass der SAP - Algorithmus sequenziell sowohl als auch parallel fehlerfrei funktioniert, sind verschiedene Tests entwickelt [A1 Sweep-and-prune Testfälle S.III]. Die einzelnen Tests benutzen *mocking*, um ein schnelleres Ausführen der Tests zu ermöglichen. Es werden statt den Hitboxobjekten eigene Testobjekte verwendet, welche auch die Schnittstelle *IHitbox* implementieren.

Die sequenzielle Kollisionserkennung wird getestet, indem drei zuvor bestimmte Anordnungen von Objekten auf Kollisionen hin überprüft werden. Diese Anordnungen

wurden mit der Software *Blender*[3] erstellt. Die Anzahl der stattgefundenen Kollisionen entscheidet über den Erfolg oder Misserfolg eines Tests.

3.2.6 Parallele Teststruktur

Die sequenzielle als auch die parallele Teststruktur greifen auf ähnliche Testmethoden zurück. Bei der parallelen Teststruktur muss allerding streng darauf geachtet werden, dass die Variablen bei laufendem Algorithmus threadsicher gesetzt werden. Wird eine Variable nicht threadsicher gesetzt, kann es passieren, dass gewisse Kollisionen nicht erfasst werden.

Um diese Problemstellung zu testen, werden in einem 500 x 500 x 500 großen Bereich 5000 zufällig platzierte Testobjekte erschaffen. Vorab werden diese mit dem sequenziellen Algorithmus auf Kollisionen hin überprüft. Die sequenziellen Testergebnisse ermöglichen einen Vergleich der Ergebnisse mit den parallelen Ausführungen. Der Algorithmus wird mit einer Jobanzahl von 1 bis 100 getestet.

[3] *Blender* ist eine Software, um dreidimensionale Szenen/ Modelle zu modellieren und zu gestalten.

3.3 Das Gravitationssystem

Wie in Abschnitt 2.6.1 [Gravitation S.6] beschrieben, sollen Himmelskörper einer Gravitation ausgesetzt werden. Hierbei wird das Newtonsche Gravitationsgesetz[4] auf die einzelnen Objekte angewendet. Der folgende Absatz beschäftigt sich mit der Implementierung und dem Test des Algorithmus.

3.3.1 Gravitationsobjekt

Damit ein Objekt der Gravitation ausgesetzt werden kann, muss es die *IGravity* Schnittstelle implementieren. Implementiert ein Objekt diese Schnittstelle, kann es einem *GravityObjectContainer* zugeordnet werden. Jedes *IGravity* Objekt besitzt eine Masse, eine Geschwindigkeit, eine Beschleunigung und eine *GravityProperty*. Außerdem muss ein Objekt, das die Schnittstelle implementiert, in der Lage sein, seine aktuelle Position wiederzugeben.

```
interface IGravity {
    val id : Int

    var mass : Float
    var velocity : Vector3f
    var acceleration : Vector3f
    var gravityProperty : GravityProperties

    fun applyObjectForce()
    fun getPosition() : Vector3f
}
```

Abbildung 13: [Schnittstelle IGravity]

3.3.2 Gravitationsmanager

Der *GravityObjectManager* dient als Container zum Verwalten von *IGravity* Objekten. Es können dem Container *IGravity* Objekte per *add* Funktion hinzugefügt werden. Durch das Aufrufen der *update* Methode werden alle zugehörigen Objekte des Managers versucht, zu verändern. Durch die *GravityProperty* eines *IGravity* Objektes wird entschieden, wie sich die Objekte aufeinander auswirken.

---

[4] Q2 Demtröder2006_Book_Experimentalphysik1 [S.68 (2.57b)]

```
enum class GravityProperties {
    source, // Other IGravity are attracted to it
    adopter, // This IGravity object is moved by other IGravity objects
    sourceAndAdopter, // source and adopter combined
    nothing // Doesn't interact with other objects (only the own velocity will be applied)
}
```

Abbildung 14: [Enum GravityProperties]

Ein *IGravity* Objekt kann zum Beispiel als Quelle dienen, so dass andere Objekte angezogen werden können. Durch die *adopter* Eigenschaft kann das Objekt von anderen Objekten beeinflusst werden. Wenn ein Objekt die *GravityProperty nothing* besitzt, wird dieses Objekt nicht behandelt. Dennoch kann es durch eine Startgeschwindigkeit in eine bestimme Richtung fliegen.

3.3.3 Umsetzung des Gravitationsalgorithmus

Wie schon im Abschnitt 3.3 [Das Gravitationssystem S.20] erwähnt, wendet der Algorithmus das Newtonsche Gravitationsgesetz an. Um die Kraft *F* zwischen zwei Objekten zu ermitteln, wird nachfolgende Formel verwendet. In dieser Formel ist *G* als Gravitationskonstante, *m* als Masse eines Objektes und *r* als Distanz zwischen den beiden Objekten definiert.

$$F_{obj1} = F_{obj2} = \mathrm{G} \cdot \frac{m_{obj1} \cdot m_{obj2}}{r^2}$$

Um die Geschwindigkeit an Zeitpunkt t+1 zu berechnen, wird die berechnete Kraft (*F*), die Richtung ($\vec{r}$) von *obj1* zu *obj2* und die Beschleunigung ($\vec{a}$) verwendet.

$$\vec{r} = normalsize(\ obj1.pos\ -\ obj2.pos\ )$$

$$\vec{a}_{obj1} = min\left(\frac{\vec{r} \cdot F_{obj1}}{m_{obj1}},\ \begin{pmatrix}1.0\\1.0\\1.0\end{pmatrix}\right)$$

$$\vec{v}_{obj1} = \vec{v}_{obj1} + \vec{a}_{obj1}$$

Diese Berechnungen werden für jedes Objekt appliziert, welches die *GravityProperty adopter* besitzt.

3.3.4 Parallelisierung

Um das Modul zu parallelisieren, wird die f*oreachParallel* - *Erweiterungsfunktion* [3.1 Paralleles Iterieren auf einer Liste S.13] auf die Liste mit den Objekten, die von der Gravitation beeinflusst werden, angewendet.

```
//sequential
override suspend fun applyGravity() {

    adopterObjects.forEach{ ob1 ->
        for (ob2 in sourceObjects) {
            applyGravityTo(ob1, ob2)
        }
    }

    gravityObjects.forEach { ob1 ->
        ob1.applyObjectForce()
    }
}

//parallel
@OptIn(DelicateCoroutinesApi::class)
override suspend fun applyGravity() {

    adopterObjects.foreachParallel(jobCount){ ob1 ->
        for (ob2 in sourceObjects) {
            applyGravityTo(ob1, ob2)
        }
    }

    gravityObjects.foreachParallel(jobCount) { ob1 ->
        ob1.applyObjectForce()
    }
}
```

Abbildung 15: [Aufbau der Gravitationsfunktion]

Die obere Abbildung demonstriert sowohl den sequenziellen als auch den parallelen Aufbau der Gravitationsfunktionen.

3.3.5 Teststruktur

Für den Gravitationsalgorithmus sind insgesamt fünf Tests entstanden. Diese Tests benutzen spezielle *TestGravityObjekte*, die auch die *IGravity* Schnittstelle implementieren.

Der erste Test benutzt zwei Testobjekte, welche linear auf der X - , Y - und Z - Achse mit einem Abstand von -100 und 100 vom Ursprung (0,0,0) positioniert sind. Im Ursprung dieses dreidimensionalen Koordinatensystems befindet sich ein weiteres Objekt, das gleichzeitig von beiden Objekten angezogen wird. Da beide äußeren Objekte nicht dem

Einfluss der Gravitation unterliegen und beide Objekte die gleiche Masse besitzen, sollte das Objekt im Ursprung sich nicht bewegen.

Bei der Verifizierung des zweiten, dritten und vierten Testfalls wird ein Objekt beobachtet, das auf einer bestimmten Achse ein weiteres Objekt im Ursprung umkreist und sich nach einer bestimmten Zeit (t) wieder an seinem Ausgangspunkt ($\vec{p}_{t_0}$) befindet. Hierfür werden die jeweiligen Variablen, d.h. der Radius (*r*), die Masse des Testobjektes im Ursprung ($m_{Central}$) und die Gravitationskonstante (*G*) festgelegt. Die initiale Geschwindigkeit ($\vec{v}$) und die orbitale Periode (*T*) sind mittels der folgenden Formeln bestimmt.

$$\vec{v} = \sqrt{\frac{G * m_{Central}}{r}}$$

$$T = \sqrt{r^3 * \frac{4 * \pi^2}{G * m_{Central}}}$$

Somit kann beim zweiten Testfall davon ausgegangen werden, dass das zentrale Objekt eine Masse von 2 Masseeinheiten besitzt. Das umkreisende Objekt hat einen Abstand von 20 Längeneinheiten und eine Periode von 154 Zeiteinheiten. Die Gravitationskonstante *G* beträgt 6,674.

Der fünfte Test überprüft die parallele Variante des Gravitationsalgorithmus. Bei diesem Versuchsaufbau werden 200 Testobjekte mit einer zufälligen Masse und Position erschaffen. Um diesen Testaufbau überprüfen zu können, wird der Algorithmus sowohl parallel als auch sequenziell ausgeführt. Die Testobjektkoordinaten des sequenziellen Algorithmus werden anschließend mit den Koordinaten des parallelen Algorithmus verglichen.

Im Anhang sind die Abbildungen und genutzten Werte der Testfälle hinterlegt, siehe A2 [Gravitationssystem Testfälle S.IV].

3.4 Kollisionsbearbeiter

Kollidieren zwei Objekte miteinander, verändert der Kollisionsbearbeiter die Objekte, die die *IApplier* Schnittstelle nach den vorgegebenen Parametern implementieren. Hierbei wird entschieden, ob die beiden Objekte entweder voneinander abprallen [3.4.2 Abprall von Objekten S.25] oder in viele Splitter zerlegt werden [3.4.4 Zersplitterung S.26]. Zudem müssen die Positionen so verändert werden, dass die Boxen sich nicht mehr überschneiden [3.4.3 Überschneidung der Objektboxen S.25].

```
interface IApplier : IGravity, IHitBox {
    var interact : Boolean
    var checked : AtomicBoolean
}
```

Abbildung 16: [Schnittstelle IApplier]

3.4.1 Kollisionsachse

Fliegen zwei Objekte aufeinander zu und kollidieren danach, existieren immer eine oder mehrere Kollisionsachsen. Diese Achsen werden verwendet, um zum Beispiel im Abschnitt 3.4.2 [Abprall von Objekten S.25] die Geschwindigkeit auf einer oder mehreren Achsen anzupassen.

Um die Kollisionsachse zu bestimmen, werden die *Hitboxpositionen* auf den Zeitpunkt t_{-1} umgerechnet. Hierfür wird die jeweilige Objektgeschwindigkeit ($\vec{v}$) von den minimalen und maximalen Endpunktpositionen abgezogen. Die Positionen am Zeitpunkt t_{-1} werden dann miteinander verglichen.

Findet auf einer Achse keine Kollision am Zeitpunkt t_{-1} statt, ist diese Achse eine Kollisionsachse. Somit gibt es die möglichen Kombinationen einer Kollision auf der X - , Y - , Z - , XY - , XZ - , YZ - und der XYZ - Achse. Außerdem kann der Sonderfall eintreten, dass beide Objekte mit einer Initialgeschwindigkeit erschaffen werden, die sie am Zeitpunkt t_{-1} auf allen Achsen kollidieren lässt. Demnach kann die Kollisionsachse nicht bestimmt werden.

3.4.2 Abprall von Objekten

Bei der Kollision von zwei Objekten prallen diese voneinander ab. Um die daraus resultierenden Geschwindigkeiten ($\vec{u}$) zu berechnen, wird folgende Formel[5] verwendet. Dabei ist m die Masse eines Objektes und $\vec{v}$ die Geschwindigkeit vor der Kollision.

$$\vec{u}_1 = \frac{(m_1 - m_2)}{m_1 + m_2}\,\vec{v}_1 + \frac{2\,m_2}{m_1 + m_2}\,\vec{v}_2 = \frac{(m_1 - m_2)\cdot\vec{v}_1 + 2\,m_2\cdot\vec{v}_2}{m_1 + m_2}$$

Da nur die Geschwindigkeit auf den Kollisionsachsen variiert, werden nur die Werte auf den Kollisionsachsen ausgetauscht.

3.4.3 Überschneidung der Objektboxen

Bei einer Kollision überschneiden sich die beiden achsenorientierten Hitboxen der Objekte. Da es in der realen Welt bei einer Kollision nicht zu einer Überschneidung kommt, müssen die beiden Objekte auf ihren Flugbahnen so weit zurückgesetzt werden, bis sie sich nur noch berühren. Dieses Verfahren benutzt ebenfalls die ermittelte Kollisionsachse [3.4.1 Kollisionsachse S.24].

Wenn die Kollisionsachse der beiden Objekte ermittelt werden kann, wird die Verschiebung auf den Kollisionsachsen bestimmt. Sollte eines der beiden Objekte die Eigenschaft besitzen, nicht mit anderen Objekten zu interagieren, wird nur ein Objekt bewegt.

Ist es nicht möglich, eine Bestimmung der Kollisionsachse zwischen den beiden Objekten vorzunehmen, wird eine andere Funktion benutzt, um die Objekte zu translatieren. Die Berechnung der Funktion zielt auf die kleinstmögliche Translation auf einer bestimmten Achse, um die Objekte aus dem jeweiligen Begrenzungsrahmen zu verschieben.

[5] Quelle der Formel: [Q3 Collisions in 1-dimension S.I] (229)

3.4.4 Zersplitterung

Bei einer Kollision von zwei Objekten besteht die Möglichkeit, dass diese nicht voneinander abprallen, sondern in viele kleine Objekte zerlegt werden. Überschreitet die Aufprallgeschwindigkeit ($\vec{v}_c$) einen gewissen Wert, kommt es zu einer Zersplitterung der Objekte.

$$\vec{v}_c = \max(\vec{v}_{obj1}, \vec{v}_{obj2}) - \min(\vec{v}_{obj1}, \vec{v}_{obj2})$$

Dieser Wert und die Anzahl der resultierenden Objekte (c) können über die Einstellungen festgelegt werden. Zur Berechnung der jeweiligen Objektskalierungen wird das Gesamtvolumen (V) der beiden kollidierenden Objekte ermittelt. Die neu erschaffenen Objekte sind Würfel, die alle die gleiche Länge (l) und Masse (m) besitzen.

$$l = \sqrt[3]{\frac{V}{c}}$$

$$m = \frac{(Obj1.m + Obj2.m)}{c}$$

Die beiden kollidierenden Objekte werden nach den Berechnungen entfernt.

3.4.5 Parallelisierung

Damit alle Funktionen des Kollionsbearbeiters parallel sowie sequenziell das gleiche Resultat liefern, muss insbesondere darauf geachtet werden, dass keine *race conditions* auftreten. Hierfür wird erstens ein *AtomicBoolean* verwendet, um zu überprüfen, ob ein Objekt abgearbeitet wurde. Außerdem werden alle resultierenden Geschwindigkeiten nicht während des parallelen Iterierens gesetzt, sondern in einer *ConcurrentHashMap* zwischengespeichert und am Ende sequenziell gesetzt.

Der Programmcode der *Zersplitterfunktion* wird außerdem in einen parallelen und einen synchronisierten Abschnitt aufgeteilt. Der synchronisierte Abschnitt plaziert und löscht die neuen und alten Objekte.

Als weitere Problematik stellt sich das zufällige Verteilen der Objektpositionen nach der Zersplitterung heraus. Um das gleiche Resultat bei beiden Ausführungsarten zu garantieren, kann ein *seed* festgelegt werden, der die zufällige Streuung der Geschwindigkeiten bestimmt. Dieser *seed* wird für jeden neu generierten Würfel mit der

kombinierten Objektposition addiert. (Die Parallelisierung setzt eine zeitliche Unabhängigkeit voraus, so dass nicht festgelegt ist, wann welcher Würfel erschaffen wird. Somit kann nicht bei unterschiedlichen Durchläufen der Anwendung garantiert werden, dass die Identifikationsnummer eines Würfels identisch bleibt.)

3.4.6 Teststruktur

Um zu überprüfen, ob der Algorithmus fehlerfrei funktioniert, sind einige Tests entwickelt worden. Diese Tests verwenden bekannte Eingangsgeschwindigkeiten und Positionen, um nach der Kollision diese mit festgeschriebenen Werten zu vergleichen. So kann garantiert werden, dass die Simulation gewisse Szenarien möglichst naturgetreu abbildet.

Die Resultate beim parallelen und sequenziellen Ausführen des Algorithmus sind in zwei verschiedene Tests aufgeteilt. Dabei muss im Einzelnen sichergestellt werden, dass die sequenziellen Resultate akkurat sind. Sie dienen als Vergleichswerte für die parallelen Resultate.

Der erste Test überprüft das Abprallen von 50000 Objekten in einem Berechnungszyklus. Durch das Vergleichen der Positionen und der Beschleunigung auf den jeweiligen Achsen wird sichergestellt, dass beide Verfahren dasselbe Resultat liefern.

Der zweite Test lässt 1000 Objekte zersplittern. Bei der Überprüfung werden die Positionen und Geschwindigkeiten von sequenziellen und parallel berechneten Objekten verglichen. Hierbei ist davon auszugehen, dass die Position der Objekte in den jeweiligen Listen und die Identifikationsnummer nicht immer identisch sind.

3.5 OpenGL Rendering

Beim Rendering werden innerhalb eines *frames* alle Objekte neu dargestellt. Im Rahmen des OpenGL Renderverfahrens gibt es verschiedene Möglichkeiten des Transfers von Objektdaten an die Grafikkarte. Unter den Daten befinden sich unter anderem die Transformationsmatrix[6], die Farbe und weitere Eigenschaften, die die Grafikkarte für den Zeichnungsprozess benötigt. Das aktuelle Kapitel beschäftigt sich mit den einzelnen Möglichkeiten, ein Objekt möglichst effizient an die Grafikkarte zu transferieren und darzustellen. Dabei werden drei verschiedene Methoden entwickelt und mittels eines Testdurchlaufs miteinander verglichen.

Außerdem ist die Leistung der verschiedenen Renderverfahren mittels Testversuchen dargestellt.

3.5.1 Simpler Renderprozess

Die separate Behandlung mehrerer Würfel ermöglicht eine vereinfachte Form der Darstellung. Bei dieser Verfahrensweise wird jedes einzelne Würfelmodell, jede Transformationsmatrix und jede weitere Eigenschaft für jeden einzelnen Würfel separat hochgeladen und vom zuständigen *Shader* bearbeitet. Dieser Prozess bietet die einfache Bearbeitung und das Hinzufügen einzelner Eigenschaften des Würfels. Aufgrund des hohen Anteils an redundanten Daten ist dieser Prozess zwar ohne großen Aufwand zu implementieren, aber sehr schlecht skalierbar.

[6] Die Transformationsmatrix beinhaltet die Skalierung, Verschiebung und die Rotation.

IDs	Object Count	Average FPS	Program RAM Usage	Program VRAM Usage
0	0	2970.09	371 kB	481 kB
1	10	2784.03	371 kB	481 kB
2	20	2784.06	373 kB	481 kB
3	50	2396.77	372 kB	481 kB
4	100	1992.03	373 kB	481 kB
5	200	1443.73	376 kB	481 kB
6	500	699.23	380 kB	481 kB
7	1000	321.70	393 kB	481 kB
8	2000	177.07	412 kB	481 kB
9	5000	76.22	465 kB	483 kB
10	10000	39.08	556 kB	485 kB
11	20000	19.85	734 kB	491 kB
12	50000	7.95	1256 kB	505 kB
13	100000	3.89	2114 kB	530 kB
14	200000	1.88	3825 kB	582 kB
15	500000	0.80	9004 kB	732 kB
16	1000000	0.39	13451 kB	983 kB

Tabelle 1: [Leistungstest 1: redundante Objekte]

3.5.2 Renderprozess am selben Modell

Zur Vermeidung von redundanten Daten und Hochladeprozessen kann es von Vorteil sein, das Modell lediglich einmal hochzuladen und vor jedem Rendern einmal zu aktivieren. Um Verzögerungen auf der Grafikkarte zu vermeiden, sollten Objekte, die ein gleiches Modell verwenden, hintereinander dargestellt werden. Diese Methode ist immer noch sehr praktikabel, erfordert aber zusätzlichen Programmcode. Sie bietet aber immer noch die Möglichkeit, vor jedem Renderprozess spezifische Eigenschaften eines Objektes hochzuladen. Diese Methode wird für die Benutzeroberfläche genutzt, da sehr oft ein spezifisches Rechteck mit verschiedenen Eigenschaften Verwendung findet [3.6 Entwicklung der Benutzeroberfläche S.32].

IDs	Object Count	Average FPS	Program RAM Usage	Program VRAM Usage
0	0	3134.45	370 kB	481 kB
1	10	2976.30	373 kB	481 kB
2	20	2911.94	373 kB	481 kB
3	50	2642.30	372 kB	481 kB
4	100	2290.59	372 kB	481 kB
5	200	1807.54	372 kB	481 kB
6	500	1094.92	372 kB	481 kB
7	1000	674.03	373 kB	481 kB
8	2000	378.08	374 kB	481 kB
9	5000	163.43	375 kB	481 kB
10	10000	83.40	379 kB	481 kB
11	20000	42.11	383 kB	481 kB
12	50000	16.68	415 kB	481 kB
13	100000	8.56	432 kB	481 kB
14	200000	4.24	468 kB	481 kB
15	500000	1.77	880 kB	481 kB
16	1000000	0.84	1021 kB	481 kB

Tabelle 2: [Leistungstest 2: gleiches Modell]

3.5.3 Instancing

Für die Darstellung einer Vielzahl von ähnlichen Objekten eignet sich das *Instancing*. Dieses Verfahren benutzt ähnlich wie Abschnitt 3.5.2 [Renderprozess am selben Modell S.29] nur ein Modell. Das Charakteristische an diesem Verfahren ist, dass nur eine Übermittlung pro Frame von der CPU aus zur Grafikkarte durch nur einen Renderaufruf erfolgt. Dieser Aufruf enthält zusätzlich die Anzahl der zu rendernden Objekte. Pausen, die während des Instancing – Verfahrens ausbleiben, verhindern eine Bearbeitung von Eigenschaften einzelner Objekte. Vor dem Renderprozess stellt ein *FloatArray* der Grafikkarte die dazugehörigen Daten zur Verfügung. Durch diese Daten erhalten die Objekte dennoch eine eigene Transformationsmatrix. Dabei muss außerdem die Informationsstruktur des *FloatArray* einmalig der Grafikkarte mitgeteilt werden. Dieses Prozedere eignet sich insbesondere für besonders große Darstellungsmengen desselben Objektmodells.

IDs	Object Count	Average FPS	Program RAM Usage	Program VRAM Usage
0	0	3089.02	370 kB	481 kB
1	10	3088.12	371 kB	481 kB
2	20	3109.05	371 kB	481 kB
3	50	3111.75	371 kB	481 kB
4	100	3104.71	372 kB	481 kB
5	200	3104.62	373 kB	481 kB
6	500	2983.02	372 kB	481 kB
7	1000	2984.14	372 kB	481 kB
8	2000	3053.60	373 kB	481 kB
9	5000	3067.02	374 kB	481 kB
10	10000	3074.15	379 kB	483 kB
11	20000	3096.33	384 kB	485 kB
12	50000	2413.01	410 kB	487 kB
13	100000	1467.71	432 kB	494 kB
14	200000	821.16	471 kB	507 kB
15	500000	355.43	513 kB	548 kB
16	1000000	183.39	944 kB	614 kB

Tabelle 3: [Leistungstest 3: Instancing]

3.5.4 Testdurchführung

Die Auswertung jedes der drei Renderverfahren geschieht mittels eines Testskripts [Abbildung 17 S.31]. Das Testskript beschreibt die insgesamt 17 Testdurchläufe. Bei jedem der 17 Testdurchläufe wird die Objektanzahl jeweils um die festgeschriebene Anzahl erhöht. Vor jedem Wechsel zu einem anderen Verfahren muss der genutzte Rechner neugestartet werden. Außerdem muss der Rechner von jeglichen nicht genutzten IO - Geräten getrennt werden. Alle Leistungstests werden mittels desselben Rechners (1) [A3 Testsysteme S.VIII] evaluiert.

```
{"cycleCount":600, "cycleSettings":[
 {"updateFrequency":1,"useSampleData":false},
 {"updateFrequency":1,"objectCount":10,"useSampleData":false},
 {"updateFrequency":1,"objectCount":20,"useSampleData":false},
 {"updateFrequency":1,"objectCount":50,"useSampleData":false},
 {"updateFrequency":1,"objectCount":100,"useSampleData":false},
 {"updateFrequency":1,"objectCount":200,"useSampleData":false},
 {"updateFrequency":1,"objectCount":500,"useSampleData":false},
 {"updateFrequency":1,"objectCount":1000,"useSampleData":false},
 {"updateFrequency":1,"objectCount":2000,"useSampleData":false},
 {"updateFrequency":1,"objectCount":5000,"useSampleData":false},
 {"updateFrequency":1,"objectCount":10000,"useSampleData":false},
 {"updateFrequency":1,"objectCount":20000,"useSampleData":false},
 {"updateFrequency":1,"objectCount":50000,"useSampleData":false},
 {"updateFrequency":1,"objectCount":100000,"useSampleData":false},
 {"updateFrequency":1,"objectCount":200000,"useSampleData":false},
 {"updateFrequency":1,"objectCount":500000,"useSampleData":false},
 {"updateFrequency":1,"objectCount":1000000,"useSampleData":false}]
}
```


Abbildung 17: [Testskript Renderverfahren]

Der *cycleCount* beschreibt, wie viele Zeitintervalle (0.05s) für jeden Testdurchlauf benötigt werden.

3.6 Entwicklung der Benutzeroberfläche

Um die Applikation auf mehreren Systemen zu testen, ist ein Benutzeroberflächensystem entwickelt. Dieses System verwendet verschiedenste Benutzeroberflächenelemente, um eine einfache Schnittstelle mit den Benutzern zu ermöglichen. Dieses Kapitel geht auf die Entwicklung eines solchen Systems ein.

3.6.1 Grundelement

Jedes Element des Benutzeroberflächensystems erbt seine grundlegenden Eigenschaften von einem abstrakten Grundelement namens *GUIElement*. Dieses Grundelement basiert auf Einschränkungen wie Größe und Position. Das Grundelement besitzt zudem eine Farbe, eine Liste von inneren Elementen und von verschiedensten Methoden.

3.6.2 Einschränkungen der Größe und Position

Ein *Constraintsystem* ermöglicht die Determination von relativen Größen und Positionen der einzelnen Elemente. Diese Einschränkungen aus dem *Constraintsystem* liefern entweder eine relative Größe oder eine relative Position. Jedes Element besitzt jeweils ein *widthConstraint*, *heightConstraint, positionXConstraint* und ein *positionYConstraint*.

Die Einschränkungen ermöglichen den einfachen Transfer in andere Messgrößen wie zum Beispiel Pixelangaben. Im Gegensatz dazu arbeitet das OpenGL System mit relativen Angaben in Abhängigkeit zur Fenstergröße.

3.6.3 Schachtelung von Elementen

Wie im Anschnitt 3.6.1 [Grundelement S.32] beschrieben, besitzt jedes Benutzeroberflächenelement eine Liste von inneren Grundelementen. Dieses Schachtelungssystem wird inspiriert von *HTML*[7]*.* Es erlaubt eine Schachtelung von Elementen innerhalb anderer Elemente.

[7] Hypertext Markup Language

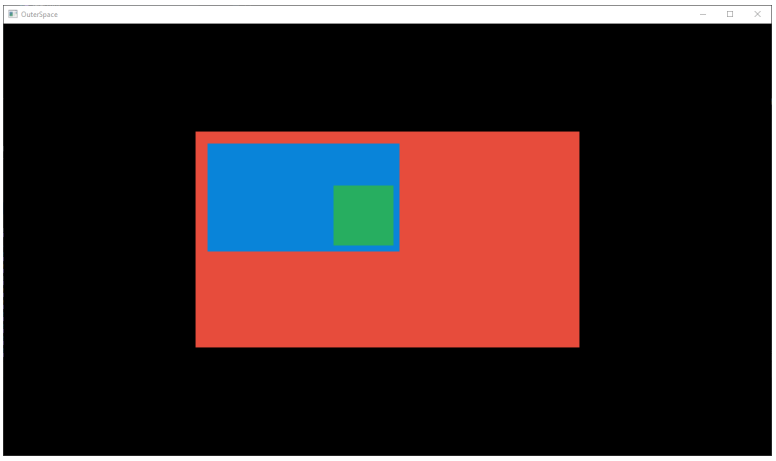


Abbildung 18: [Schachtelung von UI-Elementen]

```
Box(Relative(0.5f),Relative(0.5f), Center(), Center(), Color.red, children = listOf(
        Box(Relative(0.5f),Relative(0.5f), PixelLeft(20), PixelTop(20), Color.blue, children = listOf(
            Box(AspectRatio(), PixelHeight(100), PixelRight(10), PixelBottom(10), Color.green)
    ))
))
```

Abbildung 19: [Zugehöriger Code von Abbildung 11]

Die Abbildung 18 [Schachtelung von UI-Elementen] S.33 zeigt die Visualisierung des Codes aus Abbildung 19 [Zugehöriger Code von Abbildung 11] S.33. Diese exemplarische Darstellung verdeutlicht den Aufbau eines UI - Systems und veranschaulicht die Darstellung von komplexen Szenarien und deren Positionierung durch das *Constraintsystem*.

3.6.4 Schrift

Um eine komplexe Interaktion von Benutzern und dem jeweiligen Programm zu gewährleisten, ist die Ausgabe von Text oftmals erforderlich. OpenGL bietet keine native Unterstützung zur Ausgabe von Text. Ein Implementierungsbeispiel siehe Q13 [Font Rendering S.II] dient hierbei als Vorlage.

Um eine spezifische Schrift darzustellen, wird zunächst eine Einstellungsdatei (*.fnt*) und eine dazugehörige Bilddatei geladen. Die Einstellungsdatei liefert Daten zur Zusammensetzung des Bildes. Diese Daten implizieren die Pixelpositionen einzelner Buchstaben und seiner dazugehörigen Eigenschaften.

Um einen Text dazustellen, wird jeder Charakter eines Strings einzeln behandelt. Durch jeden Charakter wird einem *Mesh* ein passendes Rechteck hinzugefügt. Dieses

Rechteck enthält Texturkoordinaten, welche die zugehörigen Pixelpositionen der Schriftbilddatei enthalten. Bezogen auf diese Rechtecke wird im Renderprozess anschließend der zugehörige Auszug der Bilddatei gelegt und angezeigt.

Durch die Skalierung und die Translation dieser Rechtecke kann die Größe und Position festgelegt werden.

3.6.5 Interaktion

Das UI - System benutzt die herkömmlichen Ein - und Ausgabegräte eines Computers. Ein UI - Element kann sowohl auf einen Rechts – als auch auf einen Linksklick reagieren. Durch das Klicken mit der Maus in dem Hauptbenutzeroberflächenelement wird eine Klicküberprüfungsfunktion ausgelöst. In der Klicküberprüfungsfunktion eines *GUIElements* wird die aktuelle Mausposition mit den Eckpunkten des Elements verglichen. Ist die Maus im Element positioniert, werden alle inneren Elemente des aktuell untersuchten Elements auf die gleiche Weise überprüft. Anschließend wird die *OnClick* Funktion von dem Element ausgelöst, welches in der Schachtelung am tiefsten positioniert ist und den Mauszeiger in seinem Aktionsradius hat. Durch diese Verfahrensweise wird der Fokus zusätzlich auf dieses Element gelegt.

Das UI - System kann zusätzlich Tastatureingaben verarbeiten. Die Tastatureingaben werden anschließend an das Element weitergegeben, das im Fokus der Anwendung steht. Die Eingaben werden ignoriert und nicht weiter behandelt, wenn kein Element im Fokus steht.

Zusätzlich lassen sich Interaktionen wie *OnHover* und *OnPress* nutzen, um Eingaben oder Veränderungen auszulösen.

3.7 Aufbau der Thesenüberprüfung

Dieser Abschnitt beschreibt den Aufbau und die Durchführung des jeweiligen Testsystems. Zudem wird darauf eingegangen, wie Einstellungen gespeichert und geladen werden.

3.7.1 Einstellungen

In der Anwendung können verschiedenste Einstellungen vorgenommen werden, um das System ausgiebig testen zu können. Diese Einstellungen werden in einer Datenklasse zwischengespeichert. Beim Terminieren der Applikation wird die Datenklasse serialisiert und neben der Applikation im Dateisystem gespeichert. Durch das erneute Starten wird die gespeicherte Datei in ein Objekt der Datenklasse umgewandelt und kann erneut modifiziert werden. Dieser Prozesse basieren auf der *Kotlinx* Bibliothek [Q7 Kotlinx serialization S.I].


```
{
    "applyGravityEffect":false,
    "updateFrequency":600,
    "objectCount":10000,
    "seed":2753164209591102606,
    "shatterAmount":5,
    "impactVelocity":3.7183075
}
```


Abbildung 20: [Beispiel einer settings.json Datei]

3.7.2 Testkonfigurationsdatei

Um die verschiedenen Systeme [A3 Testsysteme VIII] möglichst ohne großen manuellen Aufwand testen zu können, findet eine Testkonfigurationsdatei Anwendung. Die Datei beinhaltet eine Liste von Einstellungsdaten und eine Anzahl von Aktualisierungszyklen. Diese verschiedenen Einstellungen werden sukzessiv eingespielt, so dass das System eigenständig die verschiedenen Tests automatisch abarbeitet. Nach jedem Testdurchlauf werden die Testresultate einer Ausgabedatei angehangen.

```
{
    "cycleSettings":[
    {
        "first":7500, "second":{"updateFrequency":999,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":5000, "second":{"updateFrequency":999,"objectCount":10,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":5000, "second":{"updateFrequency":999,"objectCount":50,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },
        "first":2000, "second":{"updateFrequency":999,"objectCount":100,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":2000, "second":{"updateFrequency":999,"objectCount":500,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":1500, "second":{"updateFrequency":999,"objectCount":1000,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":1500, "second":{"updateFrequency":999,"objectCount":5000,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":500, "second":{"updateFrequency":999,"objectCount":10000,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":7500,"second":{"executeParallel":false,"updateFrequency":999,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":5000,"second":{"executeParallel":false,"updateFrequency":999,"objectCount":10,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":5000,"second":{"executeParallel":false,"updateFrequency":999,"objectCount":50,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":2000,"second":{"executeParallel":false,"updateFrequency":999,"objectCount":100,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":2000,"second":{"executeParallel":false,"updateFrequency":999,"objectCount":500,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":1500,"second":{"executeParallel":false,"updateFrequency":999,"objectCount":1000,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":1500,"second":{"executeParallel":false,"updateFrequency":999,"objectCount":5000,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    },{
        "first":500,"second":{"executeParallel":false,"updateFrequency":999,"objectCount":10000,"seed":1,"shatterAmount":5,"impactVelocity":3.0}
    }]
}
```


Abbildung 21: [Beispiel der Test - Konfigurationsdatei]

3.7.3 Speicherung der Testresultate

Wie im vorigen Abschnitt beschrieben, werden die Testresultate in einer Datei neben der Applikation im Dateisystem abgelegt. Die Resultate pro Test enthalten jeweils die Test-ID, die Anzahl der erschaffenen Objekte, die durchschnittlichen *FPS* sowie die durchschnittlichen Bearbeitungszeiten des Kollisions - , Kollisionsbearbeiter - und des Gravitationssystems gemessen in Nanosekunden. Die Testergebnisse werden zeilenweise an die Resultatdatei angehangen.

3.7.4 Testsysteme

Um die aufgestellte These zu überprüfen, werden sieben verschiedene Systeme verwendet. Alle Systeme besitzen unterschiedlich starke Prozessoren mit einer Anzahl von 1 - 6 Kernen (2 - 12 virtuelle Kernen). Beim Ausführen der Tests wird vorausgesetzt, dass alle Computer auf der neuesten Version ihres Betriebssystems laufen. Voraussetzung ist außerdem, dass die Systeme keine Aktualisierungen oder andere Änderungen während der Testdurchläufe vornehmen und dass sie vor dem Starten des Tests neugestartet werden. Eine Liste aller Testsysteme ist im Anhang A3 [Testsysteme S.VIII] zu finden.

4 Fazit

Aus den verschiedenen Testdurchläufen resultiert eine Analyse, die verschiedene Testergebnisse hervorbringt. Zudem wird beschrieben, inwiefern die gewonnenen Informationen mit der aufgestellten These in Einklang zu bringen sind.

4.1 Ergebnisse

Auf den sieben getesteten Systemen sind die Tests wie im Abschnitt 3.7.2 [Testkonfigurationsdatei S.35] ausgeführt. Eine Minimierung der Fehlerquelle durch im Hintergrund aktiv laufende Programme kann dadurch erfolgen, dass die Tests bei einer Vielzahl von Systemen jeweils dreimalig Mal ausgeführt werden. Die Durchschnittsberechnung beruht im Einzelnen auf einer Zusammenfassung der Ergebnisse aus den jeweiligen drei Testdurchläufen.

4.1.1 Zusammengetragene Ergebnisse

Um die Resultate in den sieben Dateien miteinander zu vergleichen, sind die durchschnittlichen Aktualisierungszeiten und die durchschnittlichen FPS in den folgenden Tabellen zusammengefasst.

			Testsystem [1-7] Average Updatetime						
Object Count	**Ex. UPS**	**Run Parallel**	**7 [1|2]**	**6 [2|4]**	**1 [4|4]**	**5 [4|8]**	**2 [4|8]**	**3 [6|12]**	**4 [6|12]**
0	7500	true	5,02	4,67	1,05	1,73	1,50	1,96	1,66
10	5000	true	3,44	4,70	1,11	1,65	1,43	1,98	1,73
50	5000	true	4,35	5,55	1,36	1,79	1,68	2,29	1,93
100	2000	true	5,38	6,49	1,71	2,07	2,16	2,72	2,27
500	2000	true	16,92	12,77	3,61	3,99	6,24	3,95	4,75
1000	1500	true	29,72	21,18	6,34	5,72	7,53	5,79	8,07
5000	1500	true	305,44	76,29	43,36	25,99	30,98	27,48	41,13
10000	500	true	839,60	142,80	125,68	66,40	73,08	63,74	92,83
0	7500	false	0,44	0,89	0,13	0,05	0,04	0,12	0,14
10	5000	false	0,73	1,01	0,17	0,08	0,06	0,14	0,14
50	5000	false	1,08	1,58	0,30	0,18	0,17	0,26	0,24
100	2000	false	1,82	2,31	0,57	0,36	0,36	0,45	0,45
500	2000	false	13,36	9,92	3,70	2,49	6,17	2,76	2,77
1000	1500	false	26,02	20,15	7,79	5,58	7,45	5,82	5,81
5000	1500	false	272,42	105,35	99,72	60,13	54,34	58,67	63,14
10000	500	false	821,28	251,57	321,21	180,44	152,80	168,27	187,23

Tabelle 4: [Zusammenfassung: Aktualisierungszeiten der untersuchten Systeme]

			Testsystem [1-7] Average FPS						
Object Count	**Ex. UPS**	**Run Parallel**	**7 [1|2]**	**6 [2|4]**	**1 [4|4]**	**5 [4|8]**	**2 [4|8]**	**3 [6|12]**	**4 [6|12]**
0	7500	true	550	180	2147	2518	1165	612	2353
10	5000	true	685	185	2148	2641	1113	637	2367
50	5000	true	682	184	2143	2744	1074	625	2300
100	2000	true	633	183	2066	2709	1070	620	2185
500	2000	true	345	177	1534	2296	964	666	1389
1000	1500	true	240	168	894	1814	877	664	781
5000	1500	true	31	116	199	353	292	291	211
10000	500	true	12	66	75	146	131	140	99
0	7500	false	720	186	2505	2827	1142	544	2677
10	5000	false	707	186	2472	2816	1137	534	2875
50	5000	false	750	186	2436	2831	1135	546	2824
100	2000	false	763	186	2354	2828	1119	579	2801
500	2000	false	405	181	1524	2694	1042	599	2107
1000	1500	false	260	173	799	1896	973	629	1084
5000	1500	false	35	88	94	166	173	155	146
10000	500	false	12	38	31	55	64	57	51

Tabelle 5: [Zusammenfassung: FPS der untersuchten Systeme]

4.1.2 Vergleich der Testergebnisse

Es sind acht Diagramme erstellt worden, die die durchschnittlichen Aktualisierungszeiten der sequenziellen Ausführung einer parallelen Ausführung gegenüberstellen. Im nachfolgenden Text sind drei dieser Diagramme dargestellt, die den Wendepunkt symbolisieren. Die restlichen fünf Diagramme siehe A4 [Diagrammresultate S.IX] sind im Anhang zu finden. Die X - Achsen der Diagramme bilden die IDs, die Kerne und die virtuellen Kerne der Testsysteme ab.

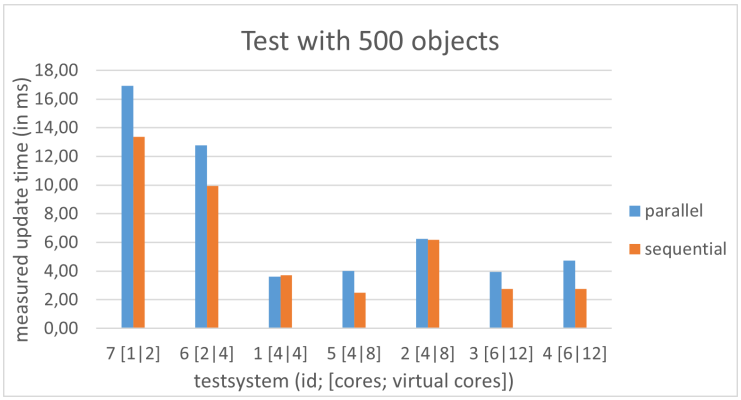


Abbildung 22: [Test mit 500 Objekten]

Die Abbildung 22: [Test mit 500 Objekten] zeigt, dass alle Systeme, ausgenommen System 4, bei einer Anzahl von 500 Objekten nicht von der Parallelisierung profitieren. Dies ist dadurch begründet, dass die gemessenen Aktualisierungszeiten beim parallelen Ausführen größer sind als bei einer sequenziellen Ausführung.

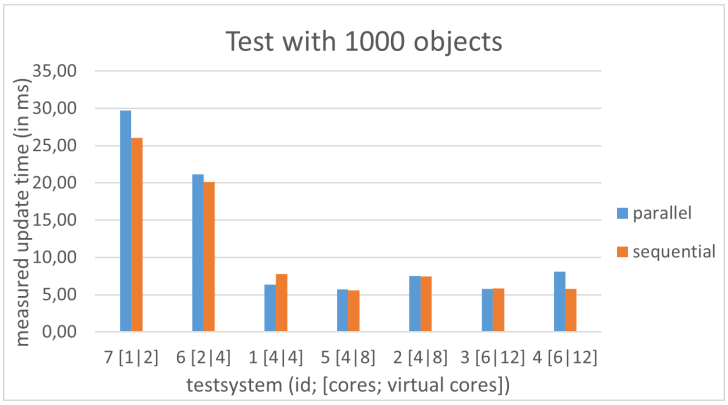


Abbildung 23: [Test mit 1000 Objekten]

Bei einem Anstieg der Objekte auf 1000 zeigen Tests bei einer parallelen Ausführung, dass bei vielen Systemen ein leichter Zeitvorteil zu verzeichnen ist.

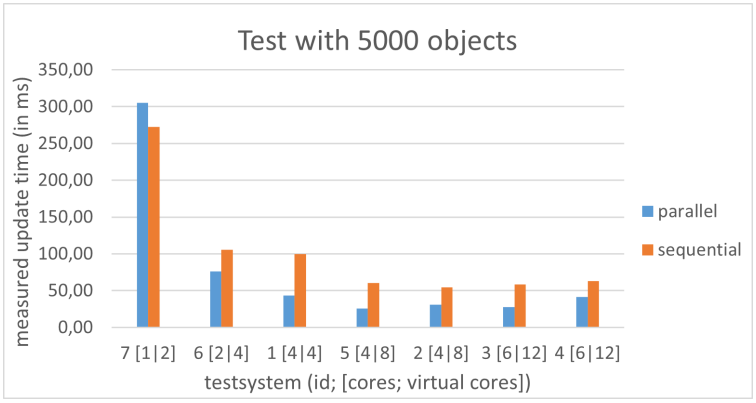


Abbildung 24: [Test mit 5000 Objekten]

Die Abbildung 24: [Test mit 5000 Objekten] veranschaulicht, dass bei einem Anstieg der Menge auf 5000 Objekte alle Systeme, außer System 7, eine deutlich minimierte durchschnittliche Aktualisierungszeit durch das Parallelisieren benötigen.

4.1.3 Overhead der Parallelisierung

Um den Overhead zu ermitteln, der zusätzlich erforderlich ist, um eine Aufgabenstellung zu parallelisieren, wird die SAP - Sortierfunktion des Systems umgeschrieben [A6 Overhead Testcode S.XIII]. Die veränderte Version synchronisiert direkt nach jedem Sortieraufruf seinen Job. Dadurch wird die Funktion ähnlich der sequenziellen Variante ausgeführt, sie benötigt aber weiterhin den Overhead der Joberstellung.

Zur Berechnung des Overheads werden zuvor dieselben Endpunkte einmal sequenziell sortiert und die verbrauchte Zeit gemessen. Die dabei ermittelte Zeit wird von der Zeit abgezogen, die den Overhead inkludiert und liefert die ungefähre Zeit, die benötigt wird, um eine Aufgabe auszulagern. Um einen Fehler zu minimieren oder gar auszuschließen, wird das oben genannte Verfahren 100 - mal wiederholt und es wird daraus ein Mittelwert errechnet. Dieser Test wird nur auf dem ersten Testsystem ausgeführt und analysiert.

object count	sort with overhead (in ms)	sort (in ms)	overhead (in ms)
0	0,086	0,004	0,081
10	0,100	0,010	0,089
50	0,127	0,037	0,090
100	0,145	0,059	0,086
500	0,342	0,244	0,098
1000	0,582	0,483	0,099
5000	2,607	2,503	0,104
10000	5,935	5,800	0,135

Tabelle 6: [Overhead-Berechnung]

Auf dem ersten Testsystem werden insgesamt vier Aufgabenstellungen durch jeweils vier Jobs parallelisiert. Daher ist davon auszugehen, dass das Auslagern in diesem Szenario[8] ungefähr $1{,}2ms$ bis $2{,}1ms$ pro Aktualisierung dauert. Diese Annahme deckt sich zudem mit den Resultaten aus Tabelle 4 [Zusammenfassung: Aktualisierungszeiten der untersuchten Systeme] S.37.

[8] Erstes Testsystem + diese spezifische Applikation

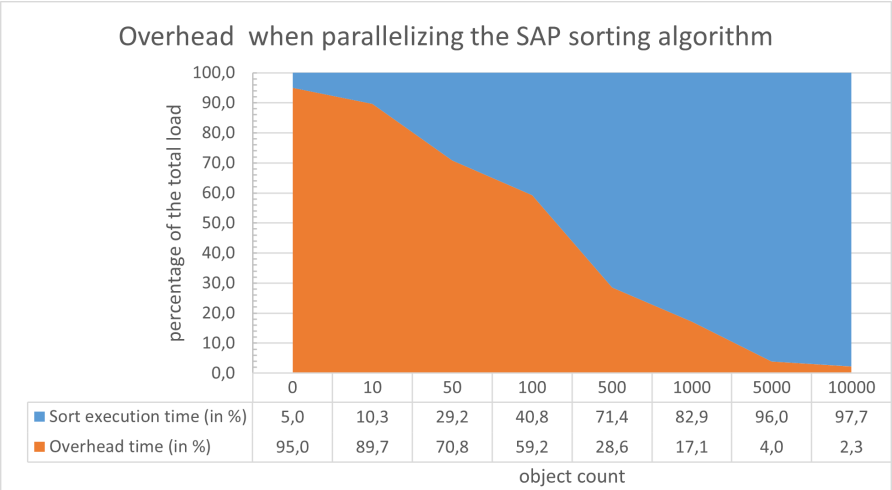


Abbildung 25: [Overhead der Sortierfunktion zum prozentualen Gesamtaufwand)]

Die Abbildung 25 [Overhead der Sortierfunktion zum prozentualen Gesamtaufwand)] S.41 demonstriert mittels prozentualer Angaben, wie sich die Gesamtzeit des Aufrufens der parallelen Sortierfunktionsvariante zusammensetzt.

4.2 Erkenntnisse

Kapitel 4.1.2 [Vergleich der Testergebnisse S.38] und Kapitel 4.1.3 [Overhead der Parallelisierung S.40] verdeutlichen, dass das parallelisierte Ausführen von kleineren Berechnungen auch zu einer signifikanten Leistungsverbesserung bei größeren Datenmengen führen kann. Das Auslagern und das Synchronisieren von einem Job bringt einen gewissen Overhead mit sich und kann zur Folge haben, dass sich bei gewissen Problemstellungen nur mühsam große Leistungsverbesserungen erreichen lassen.

4.3 Ausblick

Die Applikation verdeutlicht, dass auf Grundlage der *KotlinX* Bibliothek sich auch sehr verstrickte Aufgaben leicht parallelisieren lassen. Durch einen gewissen Mehraufwand bei der Programmentwicklung können diese verstrickten Problemstellungen, beispielsweise durch die Verwendung von atomaren Typen, parallel ausführt werden. Das Kapitel 4.1.3 [Overhead der Parallelisierung S.40] hat zudem gezeigt, dass ein nicht zu vernachlässigender Overhead beim Auslagern und Synchronisieren der Jobs entstehen kann. Dieser Overhead kann noch genauer untersucht werden. Dies kann zur Folge haben, dass mit kleineren Berechnungen und einer geringeren Anzahl von

Objekten die parallele Ausführung weiter begünstigt wird. Zudem kann überprüft werden, ob sich andere Sprachen in diesem Kontext ähnlich verhalten. Natürlich lässt sich der Kollisionsalgorithmus oder die anderen eingesetzten Algorithmen auf andere und bessere Art implementieren, so dass sie sich eher für die parallelisierte Implementierung eignen. Die beiden Paper Q9 [ADynamic Bounding Volume Hierarchy for Generalized Collision Detection S.II] und Paper Q14 [Enhanced Sweep and Prune S.II] können dabei als Vorlage für einen sequenziellen Ansatz dienen.

Quellenverzeichnis

Q1 Projekt OuterSpace

Chouliaras, A. & Gossler, D. (2021, 22. August). *GitHub - DennisGoss99/Prj_OuterSpace: 3D Space game*. Projekt OuterSpace. Abgerufen am 19. Mai 2022, von https://github.com/DennisGoss99/Prj_OuterSpace

Q2 Demtröder2006_Book_Experimentalphysik1

Demtröder, W. (2006). *Mechanik und Wärme* (4. Aufl., Bd. 1). Springer.

Q3 Collisions in 1-dimension

Fitzpatrick, R. (2006, 2. Februar). *Collisions in 1-dimension*. farside.ph.utexas.edu. Abgerufen am 10. Juni 2022, von https://farside.ph.utexas.edu/teaching/301/lectures/node76.html

Q4 Veranschaulichung der Anwendung

Goßler, D. G. [DennisGoss99]. (2022, 29. Juni). *Bachelorarbeit Video* [Video]. YouTube. https://youtu.be/4EW30FbgFxs

Q5 Mathematics of Satellite Motion

Henderson, T. (o. D.). *Mathematics of Satellite Motion*. The Physics Classroom. Abgerufen am 17. Mai 2022, von https://www.physicsclassroom.com/class/circles/Lesson-4/Mathematics-of-Satellite-Motion

Q6 JUnit

JUnit. (o. D.). *JUnit – About*. Abgerufen am 19. Mai 2022, von https://junit.org/junit4/

Q7 Kotlinx serialization

Kotlin. (2022a, Mai 26). *GitHub - Kotlin/kotlinx.serialization: Kotlin multiplatform / multi-format serialization*. GitHub. Abgerufen am 20. Juni 2022, von https://github.com/Kotlin/kotlinx.serialization

Q8 Kotlinx coroutines

Kotlin. (2022b, Juni 10). *GitHub - Kotlin/kotlinx.coroutines: Library support for Kotlin coroutines*. GitHub. Abgerufen am 20. Juni 2022, von https://github.com/Kotlin/kotlinx.coroutines

Q9 ADynamic Bounding Volume Hierarchy for Generalized Collision Detection

Larsson, T. & Akenine-Möller, T. (2005). *ADynamic Bounding Volume Hierarchy for Generalized Collision Detection*. Vriphys. Abgerufen am 10. Juni 2022, von http://vcg.isti.cnr.it/vriphys05/material/paper55.pdf

Q10 Lightweight Java Game Library

LWJGL - Lightweight Java Game Library. (o. D.). LWJGL. Abgerufen am 19. Mai 2022, von https://www.lwjgl.org/

Q11 42 Years of Microprocessor Trend Data

Rupp, K. (2018, Februar). *42 Years of Microprocessor Trend Data*. GPGPU/MIC Computing. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Q12 Sweep and prune

Terdiman, P. (2017, September). *Sweep-and-prune* (Version 0.2). http://www.codercorner.com/SAP.pdf

Q13 Font Rendering

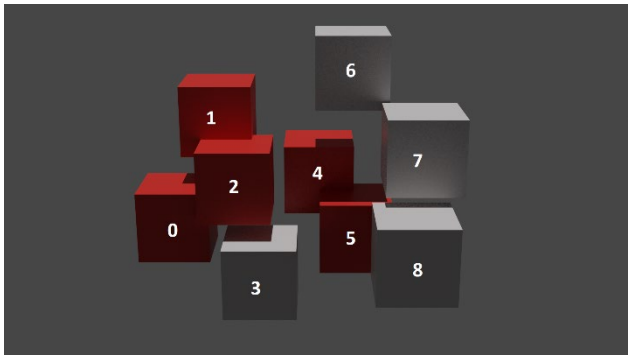
T.M. [ThinMatrix]. (2015, 31. Oktober). *OpenGL 3D Game Tutorial 32: Font Rendering* [Video]. YouTube. https://www.youtube.com/watch?v=mnIQEQoHHCU&feature=youtu.be

Q14 Enhanced Sweep and Prune

Tracy, D. J., Buss, S. R. & Woods, B. M. (2009). *Article: Enhanced Sweep and Prune*. mathweb.ucsd.edu. Abgerufen am 24. Mai 2022, von https://mathweb.ucsd.edu/%7Esbuss/ResearchWeb/EnhancedSweepPrune/

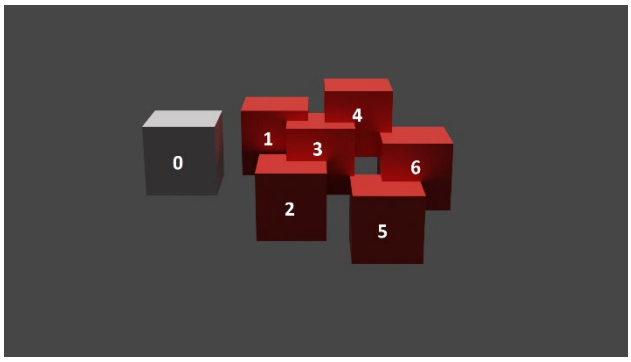
Anhang

A1 Sweep-and-prune Testfälle



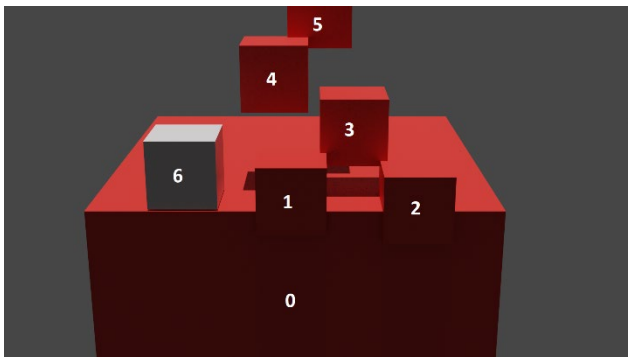
1. Testfall
Würfelanzahl: 9
Würfel kollidieren: 5

Kollisionen:
0: {2}
1: {2}
2: {0,1}
3: {}
4: {5}
5: {4}
6: {}
7: {}
8: {}



2. Testfall
Würfelanzahl: 7
Würfel kollidieren: 6

Kollisionen:
0: {}
1: {3}
2: {3,4}
3: {1,2,4,5}
4: {2,3,6}
5: {3,6}
6: {4,5}



3. Testfall
Würfelanzahl: 7
Würfel kollidieren: 6

Kollisionen:
0: {1,2}
1: {0,3}
2: {0,3}
3: {1,2}
4: {5}
5: {4}
6: {}

A2 Gravitationssystem Testfälle

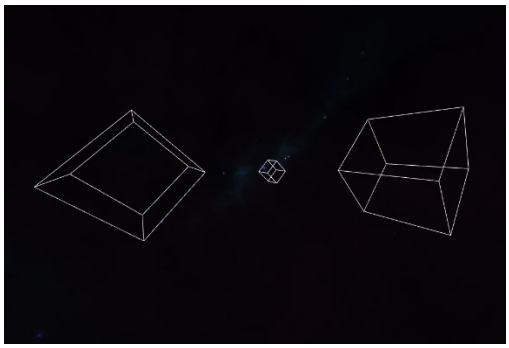


Bild zum 1 Testfall

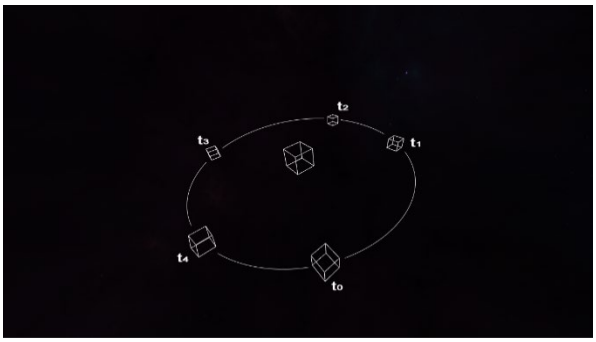


Bild zum 2,3 und 4 Testfall

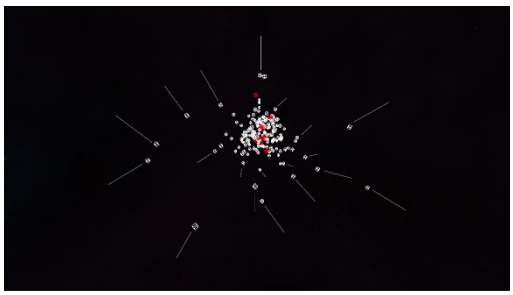


Bild zum 5 Testfall

$$v = \sqrt{\frac{G * m_{Central}}{r}}$$

$$T = \sqrt{r^3 * \frac{4 * \pi^2}{G * m_{Central}}}$$

($\vec{v}$) = Initiale Geschwindigkeit (G) = Gravitationskonstante = 6.674

($m_{Central}$) = Masse des Testobjektes im Zentrum (r) = Radius der Umlaufbahn

(T) = Periodenzeit

Quelle der Formeln zur Berechnung:

[Q5 Mathematics of Satellite Motion S.I]

1. Testfall [zwei stationäre beeinflussen ein Testobjekt]

Objekt 1: $\vec{p} = \begin{pmatrix}100\\100\\100\end{pmatrix}$; *wird nicht von anderen Objekten beeinflusst*

Objekt 2: $\vec{p} = \begin{pmatrix}0\\0\\0\end{pmatrix}$; *wird von anderen Objekten beeinflusst*

Objekt 3: $\vec{p} = \begin{pmatrix}-100\\-100\\-100\end{pmatrix}$; *wird nicht von anderen Objekten beeinflusst*

2. Testfall [Testobjekt umkreist Objekt im Zentrum auf der X-Achse]

Objekt 1:

$$\vec{p} = \begin{pmatrix}0\\0\\0\end{pmatrix}$$

$$m_{Central} = 2$$

wird nicht von anderen Objekten beeinflusst

Objekt 2:

$$\vec{p}_{t_0} = \begin{pmatrix}20\\0\\0\end{pmatrix} \Rightarrow r = 20$$

$$v_y = \sqrt{\frac{G * m_{Central}}{r}} = \sqrt{\frac{6.674 * 2}{20}} = 0.81694555 \Rightarrow \vec{v} = \begin{pmatrix}0\\0.81694555\\0\end{pmatrix}$$

$$T = \sqrt{r^3 * \frac{4 * \pi^2}{G * m_{Central}}} = \sqrt{20^3 * \frac{4 * \pi^2}{6.674 * 2}} = 153.8214 \approx 154$$

wird von anderen Objekten beeinflusst

3. Testfall [Testobjekt umkreist Objekt im Zentrum auf der Y-Achse]

Objekt 1:

$$\vec{p} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$m_{Central} = 6$$

wird nicht von anderen Objekten beeinflusst

Objekt 2:

$$\vec{p}_{t_0} = \begin{pmatrix} 0 \\ 36 \\ 0 \end{pmatrix} \Rightarrow r = 36$$

$$v_z = \sqrt{\frac{G * m_{Central}}{r}} = \sqrt{\frac{6.674 * 6}{36}} = 1.054672 \Rightarrow \vec{v} = \begin{pmatrix} 0 \\ 0 \\ 1.054672 \end{pmatrix}$$

$$T = \sqrt{r^3 * \frac{4 * \pi^2}{G * m_{Central}}} = \sqrt{36^3 * \frac{4 * \pi^2}{6.674 * 6}} = 214.4692 \approx 214$$

wird von anderen Objekten beeinflusst

4. Testfall [Testobjekt umkreist Objekt im Zentrum auf der Z-Achse]

Objekt 1:

$$\vec{p} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$m_{Central} = 80$$

wird nicht von anderen Objekten beeinflusst

Objekt 2:

$$\vec{p}_{t_0} = \begin{pmatrix} 0 \\ 0 \\ 600 \end{pmatrix} \Rightarrow r = 600$$

$$v_x = \sqrt{\frac{G * m_{Central}}{r}} = \sqrt{\frac{6.674 * 80}{600}} = 0.9433274 \Rightarrow \vec{v} = \begin{pmatrix} 0.9433274 \\ 0 \\ 0 \end{pmatrix}$$

$$T = \sqrt{r^3 * \frac{4 * \pi^2}{G * m_{Central}}} = \sqrt{600^3 * \frac{4 * \pi^2}{6.674 * 80}} = 3996.3972 \approx 3996$$

wird von anderen Objekten beeinflusst

5. Testfall [testen des parallelen Systems]

Objekt [1 …200]:

$$\vec{p}_i = \begin{pmatrix} Math.Random.NextInt(0,101) \\ Math.Random.NextInt(0,101) \\ Math.Random.NextInt(0,101) \end{pmatrix}$$

$$m_{Central} = Math.Random.NextInt(0,101)$$

$$\vec{v} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

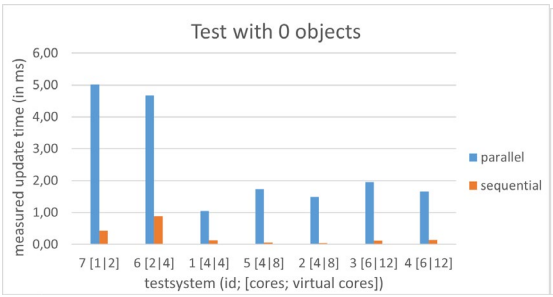
Alle Objekte werden und können andere Objekte beeinflussen

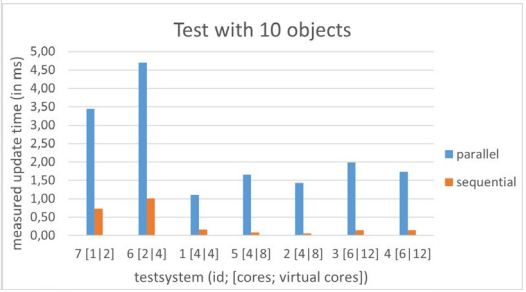
A3 Testsysteme

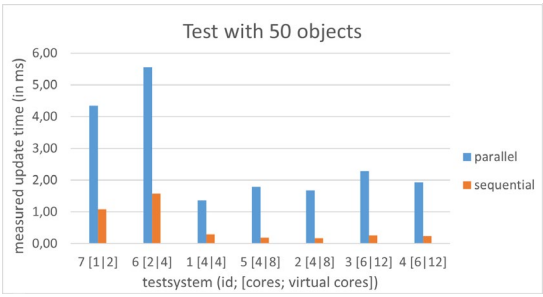
ID	Prozessor	K	V-K	Grafikkarte	Betriebssystem	RAM	VRAM
1	Intel Core i5-4460 @ 3.20 GHz	4	4	AMD RX 5700	WIN-H 10 x64	16 GB 1600MHz	8 GB
2	Intel Core i7-1165G7 @ 2.80 GHz	4	8	Intel(R) Iris(R) XE Graphics	WIN-H 10 x64	32 GB 3200 MHz	15,8 GB
3	Intel Core i7-9750H @2.60 GHz	6	12	NVIDIA GTX 1660 Ti	WIN-H 10 x64	16 GB	14 GB
4	AMD Ryzen 5 2600 @ 3.7 GHz	6	12	Radeon RX 590	WIN-H 10 x64	16 GB 1067 MHz	8 GB
5	Intel Core i7 6700K @ 4.2 GHz	4	8	NVIDIA GTX 1080	WIN-P 10 x64	32 GB 4100 MHz	8 GB
6	Intel Core i5 7267U @ 3.10 GHz	2	4	Intel(R) Iris Plus Graphics 650	macOS Monterey 12.3.1	8 GB 2133 MHz	1536 MB
7	Intel Pentium 4 630 @ 2796.84 MHz	1	2	NVIDIA GTX 460	WIN-H 7 x64	3 GB	1 GB

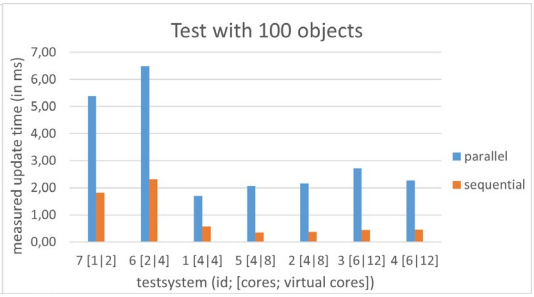
A4 Diagrammresultate

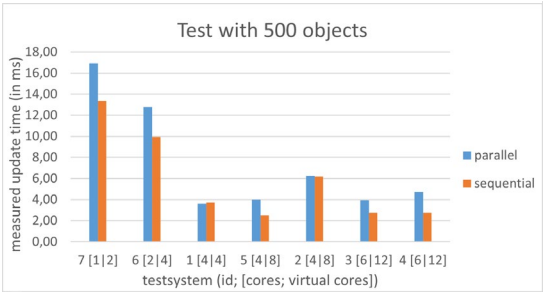
Folgende Diagrammresultate zeigen die durchschnittlichen Aktualisierungszeiten der einzelnen Systeme. (bezogen auf die Objektanzahl)

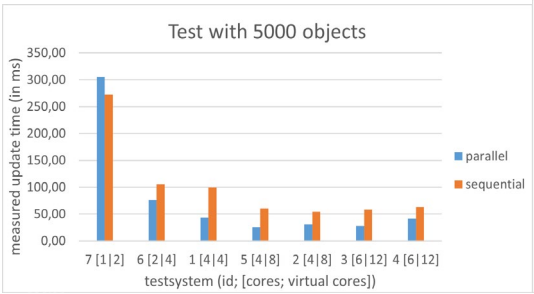


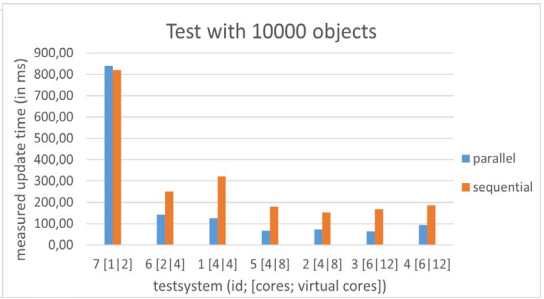












A5 Testresultate

Testsystem: [1]

object count	execute UPS	run parallel	avg FPS	avg update time (in ns)	avg collion time	avg collion handler time	avg gravity calc time
0	7500	true	2147,22333	104790,333	50123	13652,6667	29145,3333
10	5000	true	2147,95	110611,667	51437,3333	13779,6667	32691,3333
50	5000	true	2142,81333	136307	64074,3333	16189	38467,6667
100	2000	true	2065,5	170618,667	79886,3333	22944	42334
500	2000	true	1533,76333	360668,333	220468,333	39753	53544,3333
1000	1500	true	893,693333	634084	419331	65670,3333	72232,3333
5000	1500	true	198,64	4335622	3422412,67	232900	280371,333
10000	500	true	74,8633333	12568190,3	10010032,7	658455,333	861920,667
0	7500	false	2504,54333	12968,3333	2655	1076,33333	594,333333
10	5000	false	2471,61	16867,3333	4281,66667	802,333333	1214,66667
50	5000	false	2436,14	29869,6667	13575,3333	1160	3112,66667
100	2000	false	2354,05333	57292,6667	29599,6667	2072	6266,33333
500	2000	false	1524,34667	370479,667	265242,333	22443,6667	37065,6667
1000	1500	false	798,67	778881,667	615718,333	22402,6667	68197,3333
5000	1500	false	94,0066667	9971990,33	8940837	175543,667	474135,667
10000	500	false	30,5166667	32120724	28775468	666660,333	1698893,67

Testsystem: [2]

object count	execute UPS	run parallel	avg FPS	avg update time (in ns)	avg collion time	avg collion handler time	avg gravity calc time
0	7500	true	1165,456667	149544,3333	81473	21638	36001,33333
10	5000	true	1113,246667	143158,3333	81858	18660,66667	33715
50	5000	true	1073,51	168236	96116,33333	19700,33333	38635,33333
100	2000	true	1070,37	215935,6667	124622,6667	22519,66667	48241,33333
500	2000	true	964,3066667	623948	259488,3333	41553	91793,66667
1000	1500	true	877,2966667	752764,6667	423159,3333	55471,33333	100845
5000	1500	true	292	3098033	2293300	197738,6667	262301,6667
10000	500	true	130,59	7308223	5650282,333	458314,6667	410878
0	7500	false	1141,92	3708,333333	1225,666667	523,6666667	306
10	5000	false	1137,21	6210,666667	2573,666667	401	1005
50	5000	false	1135,176667	17044,33333	8807,333333	827,6666667	3236,666667
100	2000	false	1118,5	36239,33333	21500	1525,666667	6505,333333
500	2000	false	1042,193333	617175,6667	204662,6667	24575	49055
1000	1500	false	972,6233333	745295,3333	502305	19639	89638
5000	1500	false	173,23	5434113	4725617	91501,33333	297718,6667
10000	500	false	63,88666667	15279773,33	13629209,67	291257,3333	749688,6667

Testsystem: [3]

object count	execute UPS	run parallel	avg FPS	avg update time (in ns)	avg collion time	avg collion handler time	avg gravity calc time
0	7500	true	611,68	195695,5	102060	28423	56324
10	5000	true	636,875	198365	100431	27947,5	60997,5
50	5000	true	624,65	228529	127633	29494,5	57799,5
100	2000	true	619,67	272090,5	159254	37164,5	58968
500	2000	true	666,035	394583,5	232637	55965,5	72212,5
1000	1500	true	663,715	579325,5	378060,5	59057	84538,5
5000	1500	true	291,17	2747891	1963005	220839	294400
10000	500	true	139,96	6373788	4746333	510233	419487,5
0	7500	false	544,285	11570	2654	1139	569
10	5000	false	534,155	14390,5	4177,5	765	1085
50	5000	false	545,59	26156	12699	925	2639,5
100	2000	false	579,085	45030	26652,5	1825	5153,5
500	2000	false	599,475	275509,5	200150,5	12724,5	28309
1000	1500	false	629,12	582317,5	453792,5	15909,5	56325
5000	1500	false	155,02	5866632	5221182	105305,5	285911
10000	500	false	57,41	16827297	15207682	359233,5	695583

Testsystem: [4]

object count	execute UPS	run parallel	avg FPS	avg update time (in ns)	avg collion time	avg collion handler time	avg gravity calc time
0	7500	true	2352,76	166334	70516	29719	56726
10	5000	true	2367,34	173155	72684,5	29292	60385
50	5000	true	2300,12	193025,5	89915	31808	56410,5
100	2000	true	2184,93	226908	107142	36644	62299
500	2000	true	1389,385	474905,5	269056	79111,5	82171
1000	1500	true	781,09	806960	494644,5	129387	104615,5
5000	1500	true	211,025	4113152,5	2870730	582509	293224,5
10000	500	true	99,19	9283205,5	6624646,5	1162731	616411,5
0	7500	false	2676,945	14154,5	3186	1483	662
10	5000	false	2874,945	14259	3477	580,5	987,5
50	5000	false	2823,93	23909,5	10350,5	823,5	2704,5
100	2000	false	2801,495	45071,5	23165	1359,5	5323
500	2000	false	2106,88	276769,5	196859	18250	27353,5
1000	1500	false	1083,805	580829,5	458040,5	12593,5	56136
5000	1500	false	146,045	6314364	5647454	94567,5	309549,5
10000	500	false	51,36	18723247	16830944,5	355301,5	960898

Testsystem: [5]

object count	execute UPS	run parallel	avg FPS	avg update time (in ns)	avg collion time	avg collion handler time	avg gravity calc time
0	7500	true	2518,25	173112	80569	28793	53746
10	5000	true	2640,55	165392	80033	27058	51866
50	5000	true	2744,32	178640	94182	26456	50530
100	2000	true	2708,69	207251	108502	29405	58508
500	2000	true	2296,17	399348	226864	58033	88673
1000	1500	true	1813,6	572365	365237	67103	103964
5000	1500	true	353,04	2599236	1911866	226951	240894
10000	500	true	145,84	6640410	5042596	499256	548600
0	7500	false	2826,81	5268	1994	762	364
10	5000	false	2816,09	8386	3885	512	1529
50	5000	false	2830,94	18365	9494	840	4518
100	2000	false	2827,79	35593	21168	1473	8214
500	2000	false	2693,51	248901	167376	12541	45979
1000	1500	false	1895,89	557849	417642	20467	83764
5000	1500	false	166,19	6012875	5118971	142123	481861
10000	500	false	54,73	18044337	15721513	432074	1277383

Testsystem: [6]

object count	execute UPS	run parallel	avg FPS	avg update time (in ns)	avg collion time	avg collion handler time	avg gravity calc time
0	7500	true	179,73	467197,667	252635	63247	112454,333
10	5000	true	184,576667	470331,667	257189,667	60712	114650,333
50	5000	true	184,24	555246,333	310306	65707,3333	137164,333
100	2000	true	182,956667	648696,333	369245	76183,3333	153207,667
500	2000	true	176,76	1276942,67	751259,333	137023	264670,333
1000	1500	true	168,31	2118256	1295583	232508,667	364476
5000	1500	true	116,493333	7629026	6385202,33	366980	504678
10000	500	true	65,8933333	14279964,3	11733167	755794,667	1018662,67
0	7500	false	185,996667	88613	22793,6667	8794,66667	5827,33333
10	5000	false	186,226667	100603,667	27824,3333	4386,33333	8229
50	5000	false	186,116667	157517,333	81224,6667	6827,33333	15368
100	2000	false	185,97	231143	146418,333	7069,66667	24510
500	2000	false	181,27	992431,667	701909	49484,6667	120375,667
1000	1500	false	172,75	2014912,67	1490466	86833,6667	248483,667
5000	1500	false	88,0666667	10535019,7	9573556,33	166086	487236,667
10000	500	false	38,3833333	25157127,7	22637930	589297,667	1269174,33

Testsystem: [7]

object count	execute UPS	run parallel	avg FPS	avg update time (in ns)	avg collion time	avg collion handler time	avg gravity calc time
0	7500	true	550,03	501587	269015	66289,5	116992,5
10	5000	true	684,96	344181	176685	40626,5	89669
50	5000	true	682,085	434561	223885,5	41807,5	126279,5
100	2000	true	632,685	538192,5	275216	53269,5	141557,5
500	2000	true	344,81	1692017	1094399	154578,5	278797
1000	1500	true	239,63	2971817,5	2153772,5	144126,5	456250
5000	1500	true	31,485	30543701	24477935,5	1242660	3019987,5
10000	500	true	11,935	83959812	69670389,5	3841877,5	6345641
0	7500	false	719,52	43558	11758,5	5983,5	2486
10	5000	false	706,64	73025,5	28552,5	3934	8164,5
50	5000	false	749,855	107945,5	46990	3719,5	21467
100	2000	false	763,435	182403	95230,5	6204	38226
500	2000	false	404,825	1336216,5	915513	71550,5	207634
1000	1500	false	259,71	2602170	1956977,5	59165,5	369912
5000	1500	false	35,455	27241640,5	21226561,5	998428	3233838
10000	500	false	12,1	82128286	67528935	3514006,5	7181174

A6 Overhead Testcode

```
suspend fun runWithOverhead(v1 : MutableList<EndPoint>,v2 : MutableList<EndPoint>,v3 : MutableList<EndPoint>) :
Long {
    return measureNanoTime {
        GlobalScope.launch {
            v1.sortBy { it.value }
        }.join()

        GlobalScope.launch {
            v2.sortBy { it.value }
        }.join()

        GlobalScope.launch {
            v3.sortBy { it.value }
        }.join()
    }
}

suspend fun runWithoutOverhead(v1 : MutableList<EndPoint>,v2 : MutableList<EndPoint>,v3 :
MutableList<EndPoint>) : Long {
    return measureNanoTime {
        v1.sortBy { it.value }
        v2.sortBy { it.value }
        v3.sortBy { it.value }
    }
}

@OptIn(DelicateCoroutinesApi::class)
override suspend fun sort() {

    var r1 = 0L
    var r2 = 0L

    if(Random.nextInt(2) == 0) {
        r1 = x1(endPointsX.toMutableList(), endPointsY.toMutableList(), endPointsZ.toMutableList())
        r2 = x2(endPointsX.toMutableList(), endPointsY.toMutableList(), endPointsZ.toMutableList())
    }
    else {
        r2 = x2(endPointsX.toMutableList(), endPointsY.toMutableList(), endPointsZ.toMutableList())
        r1 = x1(endPointsX.toMutableList(), endPointsY.toMutableList(), endPointsZ.toMutableList())
    }

    //[...]
}
```