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

[dennis.gossler@smail.th-koeln.de](mailto:dennis.gossler@smail.th-koeln.de)

Erstprüfer: Prof. Dr. Christian Kohls

Zweitprüfer: Msc. Alexander Dobrynin

Remscheid, XX.XX.XXXX

Inhaltsverzeichnis

[Inhaltsverzeichnis B](#_Toc106664722)

[Abbildungsverzeichnis F](#_Toc106664723)

[Tabellenverzeichnis G](#_Toc106664724)

[1 Abstract 1](#_Toc106664725)

[2 Einleitung 2](#_Toc106664726)

[2.1 Relevanz 2](#_Toc106664727)

[2.2 Zielsetzung 2](#_Toc106664728)

[2.3 Recherchephase 3](#_Toc106664729)

[2.4 Projektplan 3](#_Toc106664730)

[2.4.1 Definitionsphase 3](#_Toc106664731)

[2.4.2 Durchführung 4](#_Toc106664732)

[2.4.3 Thesenüberprüfung 4](#_Toc106664733)

[2.4.4 Auswertung 4](#_Toc106664734)

[2.5 Testumgebung 4](#_Toc106664735)

[2.6 Grundaufbau der Anwendung 5](#_Toc106664736)

[2.6.1 Gravitation 5](#_Toc106664737)

[2.6.2 Kollisionen 5](#_Toc106664738)

[2.6.3 Kollisionsbearbeiter 6](#_Toc106664739)

[2.7 Benutzeroberfläche 6](#_Toc106664740)

[2.7.1 Elementauflistung 6](#_Toc106664741)

[2.7.2 Schrift 7](#_Toc106664742)

[2.8 Benutzeroberflächendesign 7](#_Toc106664743)

[2.9 Parallelisierung 8](#_Toc106664744)

[2.9.1 Sweep and Prune 8](#_Toc106664745)

[2.9.2 Gravitationssystem 9](#_Toc106664746)

[2.9.3 Kollisionsbearbeiter 10](#_Toc106664747)

[2.10 Programmablauf 10](#_Toc106664748)

[2.10.1 Aktualisierungsfunktionen 11](#_Toc106664749)

[2.10.2 Renderfunktion 11](#_Toc106664750)

[3 Hauptteil 12](#_Toc106664751)

[3.1 Paralleles iterieren auf einer Liste 12](#_Toc106664752)

[3.1.1 Herangehensweise 12](#_Toc106664753)

[3.1.2 Funktionsaufbau 13](#_Toc106664754)

[3.2 Sweep and Prune Algorithmus 13](#_Toc106664755)

[3.2.1 Datenstruktur 14](#_Toc106664756)

[3.2.2 Sequenzielle Kollisionserkennung 15](#_Toc106664757)

[3.2.3 Parallele Kollisionserkennung 16](#_Toc106664758)

[3.2.4 Veränderung der Objektpositionen 17](#_Toc106664759)

[3.2.5 Sequenzielle Teststruktur 17](#_Toc106664760)

[3.2.6 Parallele Teststruktur 17](#_Toc106664761)

[3.3 Das Gravitationssystem 18](#_Toc106664762)

[3.3.1 Gravitationsobjekt 18](#_Toc106664763)

[3.3.2 Gravitationsmanager 18](#_Toc106664764)

[3.3.3 Umsetzung des Gravitationsalgorithmus 19](#_Toc106664765)

[3.3.4 Parallelisierung 20](#_Toc106664766)

[3.3.5 Teststruktur 20](#_Toc106664767)

[3.4 Kollisionsbearbeiter 22](#_Toc106664768)

[3.4.1 Kollisionsachse 22](#_Toc106664769)

[3.4.2 Abprall von Objekten 23](#_Toc106664770)

[3.4.3 Überschneidung der Objektboxen 23](#_Toc106664771)

[3.4.4 Zersplitterung 24](#_Toc106664772)

[3.4.5 Parallelisierung 24](#_Toc106664773)

[3.4.6 Teststruktur 25](#_Toc106664774)

[3.5 OpenGL Rendering [Rohform] 26](#_Toc106664775)

[3.5.1 Simpler Renderprozess 26](#_Toc106664776)

[3.5.2 Renderprozess selbes Modell 27](#_Toc106664777)

[3.5.3 Instancing 27](#_Toc106664778)

[3.5.4 Testdurchführung 28](#_Toc106664779)

[3.6 Entwicklung der Benutzeroberfläche [Rohform] 29](#_Toc106664780)

[3.6.1 Grundelement 29](#_Toc106664781)

[3.6.2 Einschränkungen der Größe und Position 29](#_Toc106664782)

[3.6.3 Schachtelung von Elementen 29](#_Toc106664783)

[3.6.4 Schrift 30](#_Toc106664784)

[3.6.5 Interaktion 31](#_Toc106664785)

[3.7 Aufbau der Thesenüberprüfung [Rohform] 32](#_Toc106664786)

[3.7.1 Einstellungen 32](#_Toc106664787)

[3.7.2 Test - Konfigurationsdatei 32](#_Toc106664788)

[3.7.3 Speicherung der Testresultate 33](#_Toc106664789)

[3.7.4 Testsysteme 33](#_Toc106664790)

[4 Fazit 34](#_Toc106664791)

[4.1 Ergebnisse 34](#_Toc106664792)

[4.2 Erkenntnisse 34](#_Toc106664793)

[4.2.1 Parallelisierung minimaler aufwand 34](#_Toc106664794)

[4.3 Ausblick 34](#_Toc106664795)

[Quellenverzeichnis I](#_Toc106664796)

[Q1 Projekt OuterSpace I](#_Toc106664797)

[Q2 Demtröder2006\_Book\_Experimentalphysik1 I](#_Toc106664798)

[Q3 Collisions in 1-dimension I](#_Toc106664799)

[Q4 Mathematics of Satellite Motion I](#_Toc106664800)

[Q5 JUnit I](#_Toc106664801)

[Q6 Kotlinx serialization I](#_Toc106664802)

[Q7 Kotlinx coroutines II](#_Toc106664803)

[Q8 Lightweight Java Game Library II](#_Toc106664804)

[Q9 42 Years of Microprocessor Trend Data II](#_Toc106664805)

[Q10 Sweep and prune II](#_Toc106664806)

[Q11 Font Rendering II](#_Toc106664807)

[Anhang III](#_Toc106664808)

[A1 Sweep-and-prune Testfälle III](#_Toc106664809)

[A2 Gravitationssystem Testfälle IV](#_Toc106664810)

[A3 Testsysteme VIII](#_Toc106664811)

Abbildungsverzeichnis

[Abbildung 1: [Virtueller Prototyp der Einstellungsoberfläche] 7](#_Toc106664812)

[Abbildung 2: [Klassendiagramm SAP] 8](#_Toc106664813)

[Abbildung 3: [Klassendiagramm Gravitationssystem] 9](#_Toc106664814)

[Abbildung 4: [Klassendiagramm Kollisionsbearbeiter] 10](#_Toc106664815)

[Abbildung 5: [Aktivitätsdiagramm Aktualisierungsfunktion] 11](#_Toc106664816)

[Abbildung 6: [Listenzerteilung zur Parallelisierung] 13](#_Toc106664817)

[Abbildung 7: [Erweiterungsfunktion foreachParallel] 13](#_Toc106664818)

[Abbildung 8: [Klassendefinition SAP ohne Funktionen] 14](#_Toc106664819)

[Abbildung 9: [Klassendefinition EndPoint] 14](#_Toc106664820)

[Abbildung 10: [Schnittstelle IHitbox] 15](#_Toc106664821)

[Abbildung 11: [SAP-Kollisionsüberprüfung der X-Achse] 15](#_Toc106664822)

[Abbildung 12: [SAP-Kollisionsüberprüfung der Y /Z-Achse] 16](#_Toc106664823)

[Abbildung 13: [Schnittstelle IGravity] 18](#_Toc106664824)

[Abbildung 14: [Enum GravityProperties] 19](#_Toc106664825)

[Abbildung 15: [Aufbau der Gravitationsfunktion] 20](#_Toc106664826)

[Abbildung 16: [Schnittstelle IApplier] 22](#_Toc106664827)

[Abbildung 17: [Testskript Renderverfahren] 28](#_Toc106664828)

[Abbildung 18: [Schachtelung von UI-Elementen] 30](#_Toc106664829)

[Abbildung 19: [Zugehöriger Code von Abbildung 11] 30](#_Toc106664830)

[Abbildung 20: [Beispiel einer settings.json Datei] 32](#_Toc106664831)

[Abbildung 21: [Beispiel der Test - Konfigurationsdatei] 33](#_Toc106664832)

Tabellenverzeichnis

[Tabelle 1: [Leistungstest 1: redundante Objekte] 26](#_Toc106664833)

[Tabelle 2: [Leistungstest 2: gleiches Modell] 27](#_Toc106664834)

[Tabelle 3: [Leistungstest 3: Instancing] 28](#_Toc106664835)

# Abstract

Diese Bachelorarbeit beschäftigt sich mit der Entwicklung und der Erarbeitung einer Anwendung, bei der manche Prozesse 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, auf die eine Operation angewendet wird, zugrunde.

Unter Berücksichtigung spezieller Bedingungen wird die Anwendung auf mehrere Systeme appliziert und liefert jeweils Indikatoren über die sequenzielle und parallele Leistung der entwickelten Module.

[…]

# 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.

## Relevanz

Viele unserer genutzten Anwendungen werden auch heute noch weitestgehend sequenziell ausgeführt. Eine verlängerte Bearbeitungszeit ist oftmals die Folge. Eine Parallelisierung könnte dabei helfen, diese Wartezeiten 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. [Q9 42 Years of Microprocessor Trend Data S.II]

## 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 [Q7 Kotlinx coroutines S.II] die vorgegebenen Prozesse parallelisiert.

## 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.

## 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.5] noch ausführlicher behandelt.

### 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.8].

Um gewisse Abläufe und Prozesse zu veranschaulichen, sind UML Diagramme mittels *Draw.io* zu erstellen [2.10 Programmablauf S.10].

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.7]

### 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.4]

Die Projektdurchführung ist nochmals detaillierter im Abschnitt [3 Hauptteil S.12] beschrieben.

### 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.

### 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.

## Testumgebung

Um die verschiedenen Anwendungsalgorithmen zu überprüfen, werden dementsprechend Tests entwickelt. Diese Tests verwenden die *JUnit* Bibliothek [Q5 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.

## 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.

### 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.

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

### Kollisionen

Die Grundstruktur des zu entwickelnden Systems besitzt kein Kollisionsalgorithmus. Wie in Abschnitt [2.3 Recherchephase S.3] beschrieben, gilt es einen geeigneten Algorithmus zu favorisieren, der die gegebenen Kriterien erfüllt.

### 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.

## 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.

### 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

### 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]](#footnote-2)*, welche dem Nutzer einen Leistungsindikator über ausgeführte Applikationen geben.

## 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 : [Virtueller Prototyp der Einstellungsoberfläche]

## 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.

### 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.

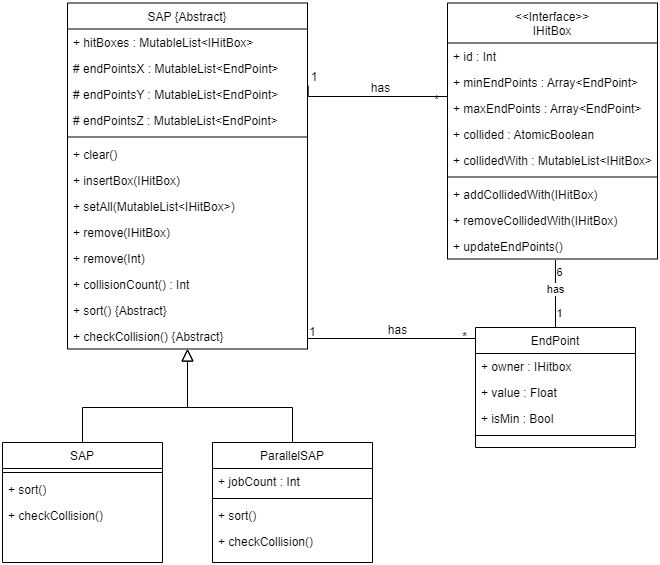


Abbildung : [Klassendiagramm SAP]

### Gravitationssystem

Ähnlich wie im oberen Abschnitt [2.9.1 Sweep and Prune S.8] soll diese Abbildung das genutzte System zum Austauschen der parallelen und sequenziellen Module veranschaulichen.

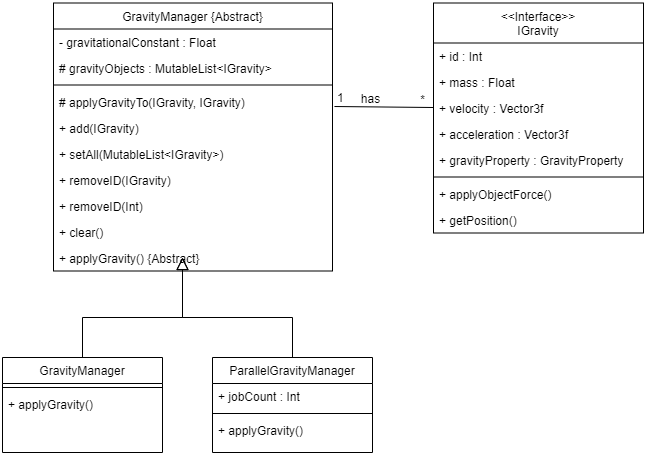


Abbildung : [Klassendiagramm Gravitationssystem]

### Kollisionsbearbeiter

Folgendes Diagramm zeigt den Aufbau des Kollisionsbearbeiters.

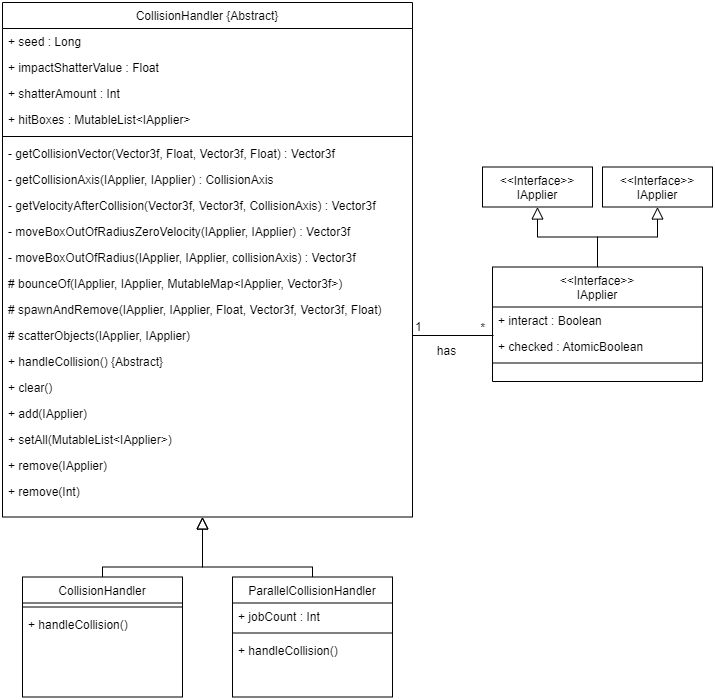


Abbildung : [Klassendiagramm Kollisionsbearbeiter]

## 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.

### 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.8] aktualisiert. Bei der Aktualisierung kann ein Objekt verschoben, umgefärbt oder entfernt werden. Folgendes Diagramm zeigt den Ablauf der Objektveränderungen.

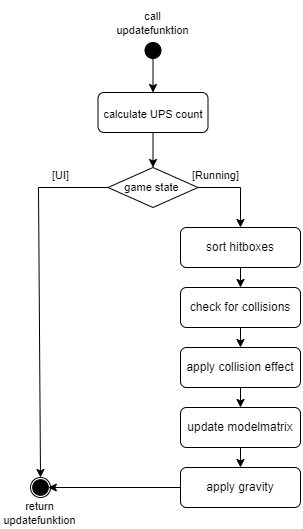


Abbildung : [Aktivitätsdiagramm Aktualisierungsfunktion]

### 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.26] noch genauer beschrieben.

# 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.

## Paralleles Iterieren auf einer Liste

In der Anwendung wird oftmals über eine Liste iteriert und auf das ausgewählte Objekt einer 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.

### 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.

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.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung : [Listenzerteilung zur Parallelisierung]

### 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. Um einen redundanten Code zu verhindern, wird die Funktion *foreachParallelIndexed* überladen und somit kann die Funktion auch ohne Zugriffsindex ausgeführt werden.



Abbildung : [Erweiterungsfunktion foreachParallel]

## Sweep and Prune Algorithmus

Der Sweep and Prune Algorithmus (*SAP*) 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.

### Datenstruktur

Um den Algorithmus darzustellen, werden gewisse Datenstrukturen benötigt. Hierbei orientiert man sich stark an den Strukturen aus dem Paper [Q10 Sweep and prune S.II].

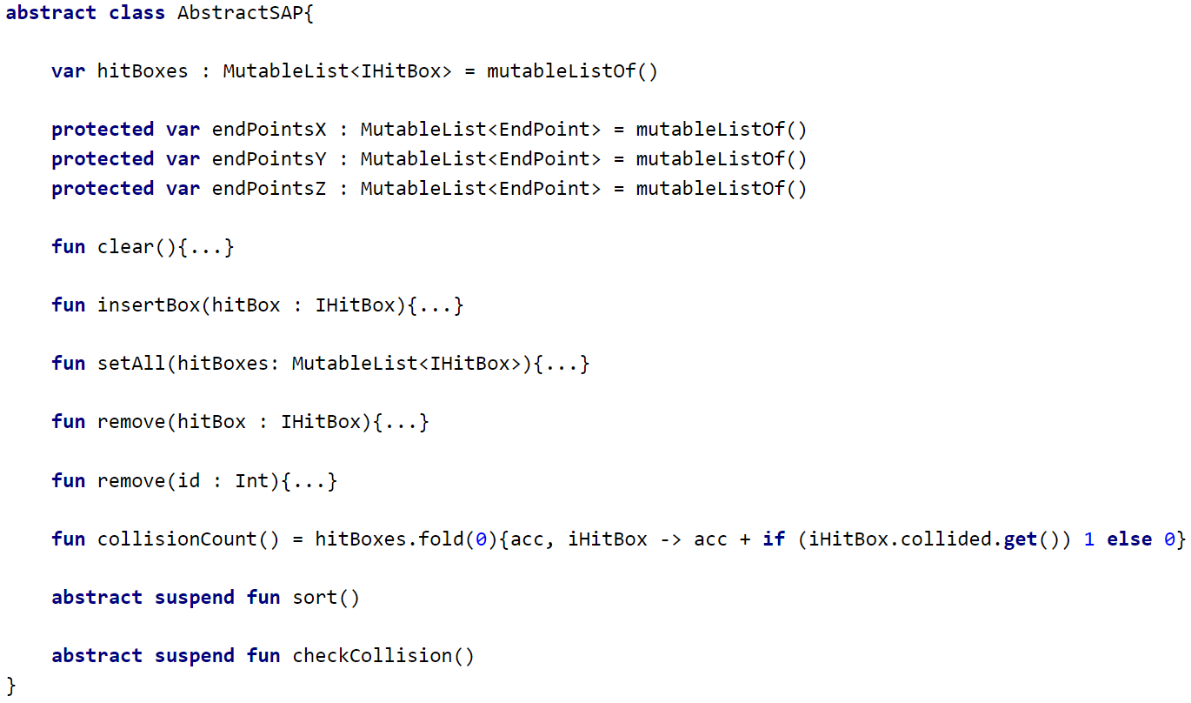


Abbildung : [Klassendefinition SAP ohne Funktionen]

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



Abbildung : [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.

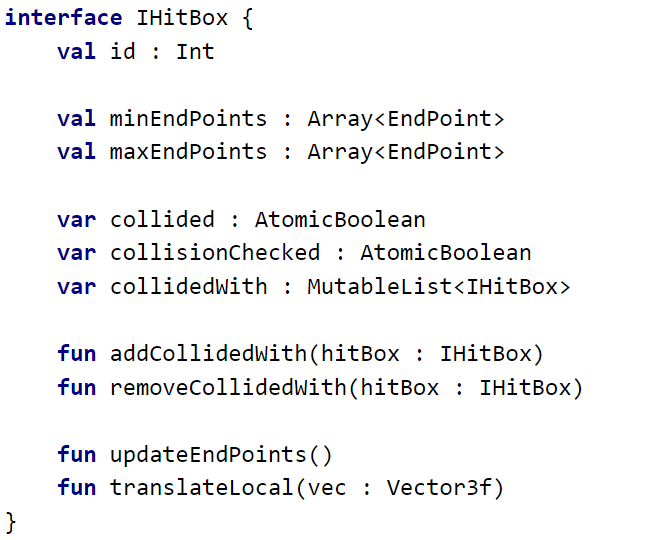


Abbildung : [Schnittstelle IHitbox]

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.

### 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.



Abbildung : [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.



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

### Parallele Kollisionserkennung

Beim Ausführen des Algorithmus wird statt der normalen *foreachIndexedSchleife* die neu entwickelte *foreachParallelIndex* Erweiterungsfunktion der Listklasse verwendet [3.1 Paralleles iterieren auf einer Liste S.12].

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.

### Veränderung der Objektpositionen

Wie im Abschnitt [3.2 Sweep and Prune Algorithmus S.13] 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.

### Sequenzielle Teststruktur

Um sicherzustellen, dass der SAP - Algorithmus sequenziell als auch parallel fehlerfrei funktioniert, sind verschiedene Tests entwickelt [A1 Sweep-and-prune Testfälle S.III]. Die einzelnen Tests benutzen statt den Hitboxobjekten eigene Testobjekte, die 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]](#footnote-3)* erstellt. Die Anzahl der stattgefundenen Kollisionen entscheidet über den Erfolg oder Misserfolg eines Tests.

### 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.

## Das Gravitationssystem

Wie in Abschnitt [2.6.1 Gravitation S.5] beschrieben, sollen Himmelskörper einer Gravitation ausgesetzt werden. Hierbei wird das Newtonsche Gravitationsgesetz[[4]](#footnote-4) auf die einzelnen Objekte angewendet. Der folgende Absatz beschäftigt sich mit der Implementierung und dem Testen des Algorithmus.

### 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.

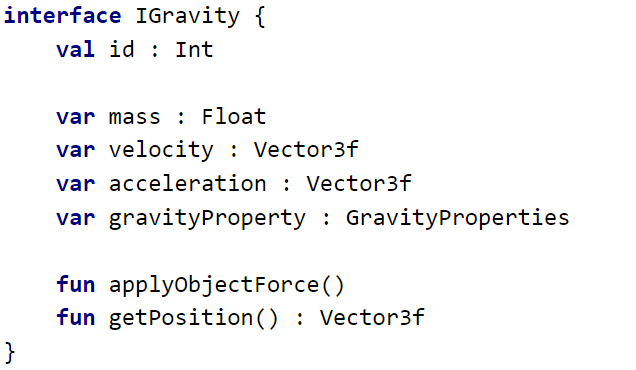


Abbildung : [Schnittstelle IGravity]

### 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.

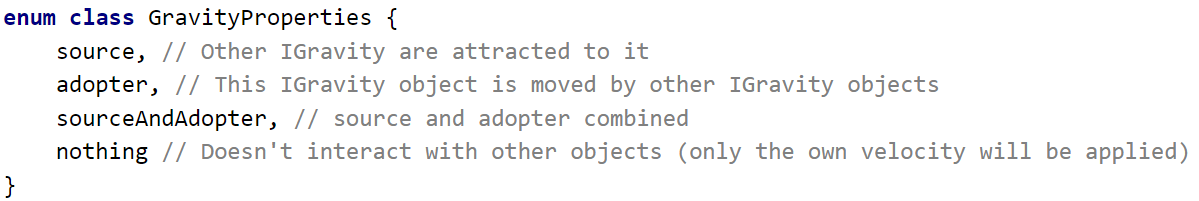


Abbildung : [Enum GravityProperties]

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.

### Umsetzung des Gravitationsalgorithmus

Wie schon im Abschnitt [3.3 Das Gravitationssystem S.18] 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.

Um die Geschwindigkeit an Zeitpunkt t+1 zu berechnen, wird die berechnete Kraft (*F*), die Richtung () von *obj1* zu *obj2* und die Beschleunigung () verwendet.

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

Der zugehörige Quellcode ist im Anhang siehe [LINK] einsehbar.

### Parallelisierung

Um das Modul zu parallelisieren, wird die *ForeachParallel* Erweiterungsfunktion [3.1 Paralleles iterieren auf einer Liste S.12] auf die Liste mit den Objekten, die von der Gravitation beeinflusst werden, angewendet.

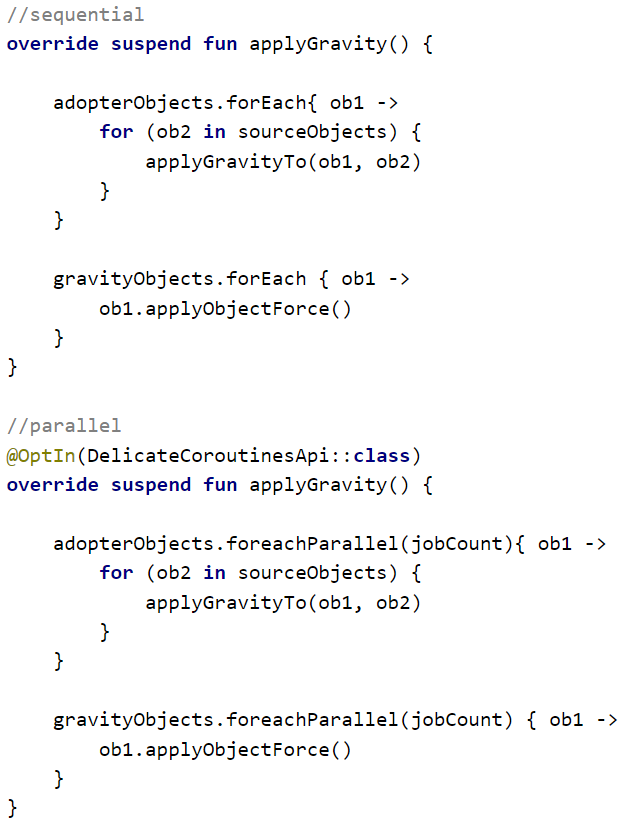


Abbildung : [Aufbau der Gravitationsfunktion]

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

### 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) liegen. 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 () befindet. Hierfür werden die jeweiligen Variablen, d.h. der Radius (*r*), die Masse des Testobjektes im Ursprung () und die Gravitationskonstante (*G*) festgelegt. Die initiale Geschwindigkeit () und die orbitale Periode (*T*) sind mittels der folgenden Formeln bestimmt.

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.

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

## Kollisionsbearbeiter

Nachdem zwei Objekte miteinander kollidieren, verändert der Kollisionsbearbeiter die Objekte, welche die *IApplier* Schnittstelle nach den vorgegebenen Parametern implementieren. Hierbei wird entschieden, ob die beiden Objekte voneinander abprallen [3.4.2 Abprall von Objekten S.23] oder in viele Splitter zerlegt werden [3.4.4 Zersplitterung S.24]. Zudem müssen die Positionen so verändert werden, dass die Boxen sich nicht mehr überschneiden [3.4.3 Überschneidung der Objektboxen S.23].

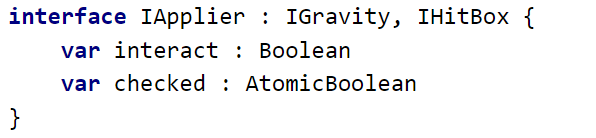


Abbildung : [Schnittstelle IApplier]

### Kollisionsachse

Wenn zwei Objekte aufeinander zufliegen und kollidieren gibt es immer eine oder mehrere Kollisionsachsen. Diese Achsen werden verwendet, um zum Beispiel im Abschnitt [3.4.2 Abprall von Objekten S.23] die Geschwindigkeit auf einer oder mehreren Achsen anzupassen.

Um die Kollisionsachse zu bestimmen, werden die *Hitboxpositionen* auf den Zeitpunkt umgerechnet. Hierfür wird die jeweilige Objektgeschwindigkeit () von den minimalen und maximalen Endpunktpositionen abgezogen. Die Positionen am Zeitpunkt werden dann miteinander verglichen.

Wenn auf einer Achse keine Kollision am Zeitpunkt stattgefunden hat, 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 auf allen Achsen kollidieren lässt. Demnach kann die Kollisionsachse nicht bestimmt werden.

### Abprall von Objekten

Wenn zwei Objekte miteinander kollidieren, prallen sie voneinander ab. Um die daraus resultierenden Geschwindigkeiten () zu berechnen, wird folgende Formel[[5]](#footnote-5) verwendet. Dabei ist die Masse eines Objektes und die Geschwindigkeit vor der Kollision.

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

### 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 zurücksetzt werden, bis sie sich nur noch berühren. Dieses Verfahren benutzt ebenfalls die ermittelte Kollisionsachse [3.4.1 Kollisionsachse S.22].

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 ein 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.

### 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 () einen gewissen Wert, kommt es zu einer Zersplitterung der Objekte.

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

Die beiden kollidierenden Objekte werden nach den Berechnungen entfernt.

### Parallelisierung

Damit alle Funktionen des Kollionsbearbeiters parallel sowie sequenziell das gleiche Resultat liefern, muss insbesondere darauf geachtet werden, dass keine *race condition* auftreten. Hierfür wird beispielsweise 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 voraus, 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.)

### Teststruktur

Um zu überprüfen, ob der Algorithmus fehlerfrei funktioniert, sind einige Tests entstanden. 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.

## OpenGL Rendering

Beim Rendering werden innerhalb eines *frame‘s* 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]](#footnote-6), die Farbe und weitere Eigenschaften, die die Grafikkarte für den Zeichnungsprozess benötigen kann. Das aktuelle Kapitel beschäftigt sich mit den einzelnen Möglichkeiten, ein Objekt möglichst effizient an die Grafikkarte zu transferieren und darzustellen.

Außerdem ist die Leistung der verschiedenen Renderverfahren mittels Testversuchen dargestellt.

### Simpler Renderprozess

Die separate Behandlung mehrerer Würfel ermöglicht eine vereinfachte Form der Darstellung. Bei dieser Verfahrensweise wird jedes 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 Hinzufügen einzelner Eigenschaften. Aufgrund des hohen Anteils an redundanten Daten ist dieser Prozess zwar ohne großen Aufwand zu implementieren, aber sehr schlecht skalierbar.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Objektanzahl** | **Durchschnitte FPS** | **Arbeitsspeicher** | **Grafikspeicher** |
| 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 : [Leistungstest 1: redundante Objekte]

### Renderprozess am selben Modell

Zur Vermeidung von redundanten Daten und Hochladeprozesse kann es von Vorteil sein, dass Modell lediglich einmal hochzuladen und vor jedem Rendern einmal zu aktivieren. Um Verzögerungen auf der Grafikkarte zu vermeiden, sollten die Modelle, die ein gleiches Modell verwenden, hintereinander dargestellt werden. Diese Methode ist immer noch sehr flexibel, 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.29]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Objektanzahl** | **Durchschnitte FPS** | **Arbeitsspeicher** | **Grafikspeicher** |
| 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 : [Leistungstest 2: gleiches Modell]

### Instancing

Für die Darstellung einer Vielzahl von ähnlichen Objekten eignet sich das *Instancing*. Dieses Verfahren benutzt ähnlich wie Abschnitt [3.5.2 Renderprozess selbes Modell S.27] nur ein Modell. Das Charakteristische an diesem Verfahren ist, dass nur eine Übermittlung pro Frame von der CPU aus zur Grafikkarte durch ein *rendercall* 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 kann ein *Floatarray* der Grafikkarte mit den dazugehörigen Daten zur Verfügung gestellt werden, so dass die Objekte dennoch eine eigene Transformationsmatrix erhalten. Dabei muss zusätzlich die Informationsstruktur des *Floatarray* einmal der Grafikkarte mitgeteilt werden. Dieses Prozedere eignet sich insbesondere für besonders große Darstellungsmengen desselben Objektmodells.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Objektanzahl** | **Durchschnitte FPS** | **Arbeitsspeicher** | **Grafikspeicher** |
| 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 : [Leistungstest 3: Instancing]

### Testdurchführung

Die Auswertung jedes der drei Renderverfahren geschieht mittels eines Testskripts [Abbildung 17 S.28]. Das Testskript beschreibt die jeweils 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 er von jeglichen nicht genutzten IO-Geräten getrennt werden. Alle Leistungstests werden mittels desselben Rechners (1) [A3 Testsysteme S.VIII] evaluiert.

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

Abbildung : [Testskript Renderverfahren]

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

## 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 dieses Systems ein.

### 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.

### 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 lediglich mit relativen Angaben in Abhängigkeit zur Fenstergröße.

### Schachtelung von Elementen

Wie im Anschnitt [3.6.1 Grundelement S.29] beschrieben, besitzt jedes Benutzeroberflächenelement eine Liste von inneren Grundelementen. Dieses Schachtelungssystem wird inspiriert von *HTML[[7]](#footnote-7).* Dieses erlaubt eine Schachtelung von Elementen innerhalb anderer Elemente.



Abbildung : [Schachtelung von UI-Elementen]



Abbildung : [Zugehöriger Code von Abbildung 11]

Die Abbildung 18 zeigt die Visualisierung des Codes aus Abbildung 19 S.30. Diese exemplarische Darstellung verdeutlicht den Aufbau eines UI-Systems und zeigt die anschauliche Darstellung von komplexen Szenarien und deren Positionierung durch das Einschränkungssystem.

### Schrift

Um eine komplexe Interaktion von Benutzer und dem jeweiligen Programm zu gewährleisten, ist die Ausgabe von Text oftmals erforderlich. OpenGL bietet keine native Unterstützung zur Ausgabe von Text. Ein Implementierungsbeispiel [Q11 Font Rendering S.II] dient als Vorlage.

Um eine 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 ein 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. Auf diese Rechtecke bezogen, 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.

### 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 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, welches 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.

## Aufbau der Thesenüberprüfung

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

### Einstellungen

In der Anwendung können verschiedenste Einstellungen vorgenommen werden, um das System ausgiebig zu testen. 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 [Q6 Kotlinx serialization S.I].

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung : [Beispiel einer settings.json Datei]

### Testkonfigurationsdatei

Um die verschiedenen Systeme [A3 Testsysteme VIII] möglichst ohne großen manuellen Aufwand testen zu können, wird eine Testkonfigurationsdatei verwendet. 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.

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

Abbildung : [Beispiel der Test - Konfigurationsdatei]

### 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 in Nanosekunden. Die Testergebnisse werden zeilenweise an die Resultatdatei angehangen.

### Testsysteme

Um die aufgestellte These zu überprüfen, werden XXX 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 Durchläufe vornehmen und dass sie vor dem Starten des Tests neugestartet werden. Eine Liste aller Testsysteme ist im Anhang [A3 Testsysteme S.VIII] zu finden.

# Fazit

Aus dem Fazit geht die Analyse der Resultate aus den verschiedenen Testdurchläufen hervor. Zudem wird beschrieben inwiefern sich die gewonnenen Informationen mit der aufgestellten These überschneiden.

## Ergebnisse

## Erkenntnisse

### Parallelisierung minimaler aufwand

## Ausblick

Quellenverzeichnis

1. 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>

1. Demtröder2006\_Book\_Experimentalphysik1

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

1. 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>

1. 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>

1. JUnit

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

1. 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

1. 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

1. Lightweight Java Game Library

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

1. 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/](Computing.%20https:/www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/)

1. Sweep and prune

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

1. 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

Anhang

1. 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: {}

1. Gravitationssystem Testfälle

Bild zum 1 Testfall Bild zum 2,3 und 4 Testfall

Bild zum 5 Testfall

*Zugehöriger Code:* [<https://github.com/DennisGoss99/BachelorThesis/blob/main/Application/src/test/kotlin/GravitySystemTest.kt>]

() = Initiale Geschwindigkeit (G) = Gravitationskonstante = 6.674

() = Masse des Testobjektes im Zentrum (r) = Radius der Umlaufbahn

(T) = Periodenzeit

*Quelle der Formeln zur Berechnung:*

[*QX LINK*]

1. **Testfall [zwei stationäre beeinflussen ein Testobjekt]**

***Objekt 1:*** *; wird nicht von anderen Objekten beeinflusst*

***Objekt 2:*** *; wird von anderen Objekten beeinflusst*

***Objekt 3:*** *; wird nicht von anderen Objekten beeinflusst*

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

***Objekt 1:***

*wird nicht von anderen Objekten beeinflusst*

***Objekt 2:***

*wird von anderen Objekten beeinflusst*

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

***Objekt 1:***

*wird nicht von anderen Objekten beeinflusst*

***Objekt 2:***

*wird von anderen Objekten beeinflusst*

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

***Objekt 1:***

*wird nicht von anderen Objekten beeinflusst*

***Objekt 2:***

*wird von anderen Objekten beeinflusst*

1. **Testfall [testen des parallelen Systems]**

***Objekt [1 …200]:***

*Alle Objekte werden und können andere Objekte beeinflussen*

1. 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 | U Pc |  |  |  |  |  |  |
| 4 | AMD Ryzen 5 2600  @ 3.7 GHz | 6 | 12 | Radeon RX 590 | WIN-H 10 x64 | 16  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 | S Pc |  |  |  |  |  |  |
| 7 | Mac |  |  |  |  |  |  |
| 8 | Intel Pentium 4 630  @ 2796.84 MHz | 1 | 2 | NVIDIA GTX 460 | WIN-H 7 x64 | 3 GB | 1 GB |

1. [Q2 Demtröder2006\_Book\_Experimentalphysik1 S.I] (S.80 2.47a) [↑](#footnote-ref-1)
2. Ist die Abkürzung für *frames per second* (die Bildrate). [↑](#footnote-ref-2)
3. *Blender* ist eine Software, um dreidimensionale Bühnenbilder/ Modelle zu modellieren und zu gestalten. [↑](#footnote-ref-3)
4. Q2 Demtröder2006\_Book\_Experimentalphysik1 [S.68 (2.57b)] [↑](#footnote-ref-4)
5. Quelle der Formel: [Q3 Collisions in 1-dimension S.I] (229) [↑](#footnote-ref-5)
6. Die Transformationsmatrix beinhaltet die Skalierung, Verschiebung und die Rotation. [↑](#footnote-ref-6)
7. Hypertext Markup Language [↑](#footnote-ref-7)