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: Alexander Dobrynin

Remscheid, XX.XX.XXXX

Inhaltsverzeichnis

[Inhaltsverzeichnis B](#_Toc106101425)

[Abbildungsverzeichnis E](#_Toc106101426)

[Tabellenverzeichnis F](#_Toc106101427)

[1 Abstract 1](#_Toc106101428)

[2 Einleitung 2](#_Toc106101429)

[2.1 Relevanz 2](#_Toc106101430)

[2.2 Zielsetzung 2](#_Toc106101431)

[2.3 Recherchephase 3](#_Toc106101432)

[2.4 Projektplan 3](#_Toc106101433)

[2.4.1 Definitionsphase 3](#_Toc106101434)

[2.4.2 Durchführung 4](#_Toc106101435)

[2.4.3 Thesenüberprüfung 4](#_Toc106101436)

[2.4.4 Auswertung 4](#_Toc106101437)

[2.5 Testumgebung 4](#_Toc106101438)

[2.6 Grundaufbau der Anwendung 5](#_Toc106101439)

[2.6.1 Gravitation 5](#_Toc106101440)

[2.6.2 Kollisionen 5](#_Toc106101441)

[2.6.3 Kollisionsbearbeiter 6](#_Toc106101442)

[2.7 Benutzeroberfläche 6](#_Toc106101443)

[2.7.1 Elemente Auflistung 6](#_Toc106101444)

[2.7.2 Schrift 7](#_Toc106101445)

[2.8 Benutzeroberflächendesign 7](#_Toc106101446)

[2.9 Parallelisierung 8](#_Toc106101447)

[2.9.1 SAP 8](#_Toc106101448)

[2.9.2 Gravitationssystem 9](#_Toc106101449)

[2.9.3 Kollisionsbearbeiter 9](#_Toc106101450)

[2.10 Ablauf der Applikation 9](#_Toc106101451)

[2.10.1 Aktualisierungsfunktionen 10](#_Toc106101452)

[2.10.2 Renderfunktion 10](#_Toc106101453)

[3 Hauptteil 11](#_Toc106101454)

[3.1 Parallels iterieren auf einer Liste 11](#_Toc106101455)

[3.1.1 Herangehensweise 11](#_Toc106101456)

[3.1.2 Funktionsaufbau 12](#_Toc106101457)

[3.2 Sweep-and-prune Algorithmus 12](#_Toc106101458)

[3.2.1 Datenstruktur 13](#_Toc106101459)

[3.2.2 Sequenzielle Kollisionserkennung 14](#_Toc106101460)

[3.2.3 Parallele Kollisionserkennung 15](#_Toc106101461)

[3.2.4 Veränderung der Objektpositionen 16](#_Toc106101462)

[3.2.5 Sequenzielle Teststruktur 16](#_Toc106101463)

[3.2.6 Parallele Teststruktur 16](#_Toc106101464)

[3.3 Das Gravitationssystem 17](#_Toc106101465)

[3.3.1 Gravitationsobjekt 17](#_Toc106101466)

[3.3.2 Gravitationsmanager 17](#_Toc106101467)

[3.3.3 Umsetzung des Gravitationsalgorithmus 18](#_Toc106101468)

[3.3.4 Parallelisierung 19](#_Toc106101469)

[3.3.5 Teststruktur 19](#_Toc106101470)

[3.4 Kollisionsbearbeiter 21](#_Toc106101471)

[3.4.1 Kollisionsachse 21](#_Toc106101472)

[3.4.2 Abprall von Objekten 22](#_Toc106101473)

[3.4.3 Überschneidung der Objektboxen 22](#_Toc106101474)

[3.4.4 Zersplitterung 23](#_Toc106101475)

[3.4.5 Parallelisierung 23](#_Toc106101476)

[3.4.6 Teststruktur 24](#_Toc106101477)

[3.5 OpenGL Rendering 25](#_Toc106101478)

[3.5.1 Simpler Renderprozess 25](#_Toc106101479)

[3.5.2 Darstellung des Renderprozesses am gleichen Modell 26](#_Toc106101480)

[3.5.3 Instancing 26](#_Toc106101481)

[3.5.4 Testdurchführung 27](#_Toc106101482)

[3.6 Benutzeroberfläche 28](#_Toc106101483)

[3.6.1 Grundelement 28](#_Toc106101484)

[3.6.2 Einschränkungen der Größe und Position 28](#_Toc106101485)

[3.6.3 Schachtelung von Elementen 28](#_Toc106101486)

[3.6.4 Schrift 29](#_Toc106101487)

[3.6.5 Interaktion 30](#_Toc106101488)

[3.7 Analyse 31](#_Toc106101489)

[3.7.1 Einstellungen 31](#_Toc106101490)

[3.7.2 Konfigurationsdatei 31](#_Toc106101491)

[3.7.3 Testsysteme 31](#_Toc106101492)

[4 Fazit 32](#_Toc106101493)

[4.1 Ergebnisse 32](#_Toc106101494)

[4.2 Erkenntnisse 32](#_Toc106101495)

[4.3 Ausblick 32](#_Toc106101496)

[Quellenverzeichnis I](#_Toc106101497)

[Q1 Projekt OuterSpace I](#_Toc106101498)

[Q2 Demtröder2006\_Book\_Experimentalphysik1 I](#_Toc106101499)

[Q3 Collisions in 1-dimension I](#_Toc106101500)

[Q4 Mathematics of Satellite Motion I](#_Toc106101501)

[Q5 JUnit I](#_Toc106101502)

[Q6 Lightweight Java Game Library I](#_Toc106101503)

[Q7 42 Years of Microprocessor Trend Data II](#_Toc106101504)

[Q8 Sweep and prune II](#_Toc106101505)

[Q9 Font Rendering II](#_Toc106101506)

[Anhang III](#_Toc106101507)

[A1 Sweep-and-prune Testfälle III](#_Toc106101508)

[A2 Gravitationssystem Testfälle IV](#_Toc106101509)

[A3 Testsysteme VIII](#_Toc106101510)

Abbildungsverzeichnis

[Abbildung 1: [Virtueller Prototyp der Einstellungsoberfläche] 7](#_Toc106101511)

[Abbildung 2: [Klassendiagramm SAP] 8](#_Toc106101512)

[Abbildung 3: [Klassendiagramm Gravitationssystem] 9](#_Toc106101513)

[Abbildung 4: [Listenzerteilung zur Parallelisierung] 12](#_Toc106101514)

[Abbildung 5: [Erweiterungsfunktion foreachParallel] 12](#_Toc106101515)

[Abbildung 6: [Klassendefinition SAP ohne Funktionen] 13](#_Toc106101516)

[Abbildung 7: [Klassendefinition EndPoint] 13](#_Toc106101517)

[Abbildung 8: [Schnittstelle IHitbox] 14](#_Toc106101518)

[Abbildung 9: [SAP-Kollisionsüberprüfung der X-Achse] 14](#_Toc106101519)

[Abbildung 10: [SAP-Kollisionsüberprüfung der Y /Z-Achse] 15](#_Toc106101520)

[Abbildung 11: [Schnittstelle IGravity] 17](#_Toc106101521)

[Abbildung 12: [Enum GravityProperties] 18](#_Toc106101522)

[Abbildung 13: [Aufbau der Gravitationsfunktion] 19](#_Toc106101523)

[Abbildung 14 [Schnittstelle IApplier] 21](#_Toc106101524)

[Abbildung 15: [Testskript Renderverfahren] 27](#_Toc106101525)

[Abbildung 16: [Schachtelung von UI-Elementen] 29](#_Toc106101526)

[Abbildung 17: [Zugehöriger Code von Abbildung 11] 29](#_Toc106101527)

Tabellenverzeichnis

[Tabelle 1: [Leistungstest 1: redundante Objekte] 25](#_Toc106101528)

[Tabelle 2: [Leistungstest 2: gleiches Modell] 26](#_Toc106101529)

[Tabelle 3: [Leistungstest 3: Instancing] 27](#_Toc106101530)

# Abstract

# Einleitung

Die Bachelorarbeit bildet den Prozess einer Programmentwicklung ab. 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 somit zeigen welchen Mehraufwand und welche Vorteile eine Parallelisierung hervorbringt.

Die zu entwickelnde Anwendung zeigt eine dreidimensionale Abbildung unserer Welt und dessen orbitalen Himmelskörpern, wie zum Beispiel Satelliten oder Schrottobjekten.

Es können Einstellungen über die Menge der Objekte und deren Verhalten gesetzt werden. Über diese Einstellungsmethode ist es auch möglich bei potenziellen Kollisionen die Menge und Streuung der entstandenen Kind-Objekten zu bestimmen. Durch verschiedenen Kameraperspektiven kann die Simulation aus unterschiedlichen Blickwinkeln betrachten werden. Außerdem ist die Wiedergabegeschwindigkeit der Simulation bestimmbar und es kann sich zusätzlich frei im dreidimensionalen Raum bewegt werden.

## Relevanz

Viele unserer genutzten Anwendungen werden weitestgehend noch sequenziell ausgeführt. Dies kann oftmals zu verlängerten Bearbeitungszeiten führen. Eine Parallelisierung würde dabei helfen diese Wartezeiten zu verkürzen. Zudem setzen die größten CPU-Hersteller weitestgehend auf mehr Kerne in ihren CPUs statt schnelleren Taktfrequenzen, welches das parallele bearbeiten von Aufgaben zusätzlich begünstigt. [Q7 42 Years of Microprocessor Trend Data S.II]

## Zielsetzung

Das Ziel dieser Bachelorarbeit ist es, eine Simulation zu entwickeln, welche die Kollisionsberechnungen und Bewegungen der Agenten stark parallelisiert und die Frage beantwortet, ob Leistungsverbesserung durch die Parallelisierung erkennbar werden. Hierfür ist eine geeignete Projektarchitektur zu bestimmen und daraus folgend eine *Kotlinanwendung* zu erstellen, die mithilfe der *Kotlinx* Bibliothek die vorgegebenen Prozesse parallelisiert.

## Recherchephase

In der Recherchephase ist ein Kollisionsalgorithmus zu finden, welcher es erlaub im dreidimensionalen Raum Kollisionen effizient zu ermitteln. Der Algorithmus soll eine Parallelisierung ermöglichen.

Außerdem gilt es für die Himmelskörper eine geeignete Formel zu Berechnung einer Gravitation zu finden. Zusätzlich sollen Daten wie zum Beispiel Größe, Flugbahn und die Geschwindigkeit genutzt werden, um die Himmelskörper akkurat darzustellen.

## Projektplan

Die Planung ist in mehreren Schritten zerlegt. Je Meilenstein ist das weitere Vorgehen agil geplant. Die Anwendung der Bachelorarbeit ist von einer Person geplant, entwickelt und getestet. Außerdem ist das gesamte Projekt in vier Hauptphasen unterteilt. Da die Anwendung auf einem weiteren bereits entwickelten Projekt aufbaut, 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 zum Großteil aus dem Planen der Architektur, dem Erstellen von Diagrammen und das Gestalten des User Interfaces.

Die Architektur ist so zu gestalten, dass einzelne Module austauschbar sind. Somit ist das schnelle Auswechseln eines parallel ausgeführten Systems mit einem sequenziellen System zu ermöglichen. [siehe Architektur].

Um gewisse Abläufe und Prozesse zu veranschaulichen sind UML Diagramme mittels *Draw.io* zu erstellen. [siehe Diagramme].

Für das Gestalten des Userinterfaces wird ein virtueller Prototyp erstellt, dieser zeigt eine Vorabversion der UI und soll eine grobe Idee über das Layout geben. Dieser virtuelle Prototyp des Userinterfaces ist mit *Figma* zu erstellen. [2.8 Benutzeroberflächendesign S.7]

### Durchführung

In dieser Phase werden die Applikation und die zugehörigen Algorithmen entwickelt. Zudem werden die verwendeten Algorithmen, in ihrer sequenziellen und parallelen Form, mittels *JUnit* getestet. [2.5 Testumgebung S.4]

Die Projektdurchführung ist nochmals detaillierter im Abschnitt [3 Hauptteil S.11] beschrieben.

### Thesenüberprüfung

Um zu überprüfen, ob und inwiefern sich das Parallelisieren des Systems eignet, wird es der herkömmlichen weise gegenübergestellt. Dies soll anhand von verschiedenen skalierten Testdurchläufen die Performance des jeweiligen Systems hervorbringen. Es ist geplant das System mit vielen verschiedenen CPUs zu Testen. Die getesteten Systeme sind im Anhang [A3 Testsysteme S.VIII] aufgelistet.

### Auswertung

In dieser Phase werden die gesammelten Daten zusammengetragen und evaluiert. Es ist zu überprüfen, inwiefern sich die Parallelisierung auf gewisse Systeme ausübt und ob sie auch bei CPUs mit vergleichsweise wenig Kernen dennoch einen signifikanten Vorteil bietet.

## Testumgebung

Um die verschiedenen Algorithmen der Anwendung zu überprüfen, sind Test zu erstellen. Diese Tests verwenden die JUnit Bibliothek [Q5 JUnit S.I]. Da manche Objekte nicht direkt zum Testen geeignet sind, werden separate Testobjekte erstellt, die auf die zugehörige Schnittstelle zugreifen. Zudem werden gewisse private Methoden eines Algorithmus durch Reflexion getestet.

## Grundaufbau der Anwendung

Die zu entwickelnde Simulation basiert auf einer Projektarbeit, die im Zuge des WPFs *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 Projekt wurde in Zusammenarbeit mit Frau *Anastasia Chouliaras* erstellt [Q1 Projekt OuterSpace S.I]. Die Anwendung dient als Grundstruktur und ist auf die gegebene Problemstellung anzupassen.

Viele der entwickelten Features des Projektes sind nicht für die Simulation geeignet und müssen verändert oder umgeschrieben werden.

### Gravitation

Das Projekt Outer Space [Q1 Projekt OuterSpace S.I] besitzt zwar Planeten und Monde, die sich auf Umlaufbahnen um ein zentrales Objekt bewegen, aber dieses System eignet sich nur sehr bedingt für die zu entwickelnde Applikation. Das alte System nutzt Trigonometrische Funktionen zum Platzieren der orbitalen Himmelskörper an einem bestimmten Zeitpunkt.

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

### Kollisionen

Der Grundaufbau des zu entwickelnden Systems besitzt kein Kollisionssystem. Wie im Abschnitt [2.3 Recherchephase S.3] beschrieben, gilt es ein geeigneter Algorithmus auszuwählen, der die gegebenen Anforderungen erfüllt.

### Kollisionsbearbeiter

Nach einer Kollision sollen die kollidierten Objekte miteinander interagieren. Bei diesem Prozess soll entschieden werden ob, die beiden Objekt voneinander abprallen oder ob beide in kleinere Objekte zerlegt werden.

## Benutzeroberfläche

Das Projekt Outer Space [Q1 Projekt OuterSpace S.I] besitzt ein sehr rudimentäres Benutzeroberflächensystem. Hierbei lassen sich Elemente nur prozentual der Bildschirmbreite und höhe anzeigen. Außerdem ist das System nur Ausgabe von Bilddateien geeignet und gewährleistet dementsprechend keine direkte Interaktion. Dieses System gilt es so anzupassen, dass es ermöglicht wird auch sehr komplexe Benutzeroberfläche zu designen.

### Elemente Auflistung

Der Nutzer soll Text, sowie Zahlen eingeben, Knöpfe drücken, Schieberegler verschieben und Optionen an und ausschalten können. Folgende Aufzählung beinhaltet alle UI Elemente, welche unterstützt werden.

* **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 so eine Textausgabe sind zum Beispiel die *fps[[2]](#footnote-2)*, welche dem Nutzer einen Leistungsindikator über ausgeführte Applikation geben.

## Benutzeroberflächendesign

Um Einstellungen über 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 erschafft werden sollen. Die folgende Abbildung dient als Vorlage der Einstellungsoberfläche und beschreibt das Farbschema der Applikationselemente.



Abbildung : [Virtueller Prototyp der Einstellungsoberfläche]

## Parallelisierung

Damit die Applikation parallel sowie sequenziell ausführbar ist, wird an diesen Stellen eine Abstrakte Klasse eingesetzt. Diese Klasse definiert nur die Methoden, welche sequenziell ausgeführt werden. Wenn eine Methode auch parallel ausgeführt werden soll, wird diese ebenfalls als abstrakt deklariert und in den Kindklassen implementiert. Zudem besitzen die parallelen Kindklassen eine Variable namens *jobCount*. Diese Variable dient dazu, um zu entschieden wie viele Job eingesetzten werden.

### SAP

Das untere UML-Klassendiagramm veranschaulicht, dass genutzte System anhand des Kollisionsalgorithmus. Hierbei werden die Methoden *sort()* und *checkCollision()* in der *ParallelSAP* Klasse parallel sowie in der SAP Klasse sequenziell ausgeführt.

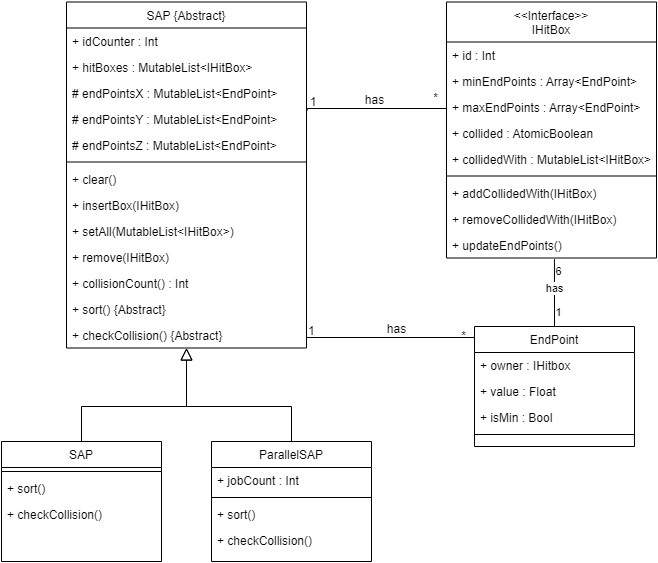


Abbildung : [Klassendiagramm SAP]

### Gravitationssystem

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

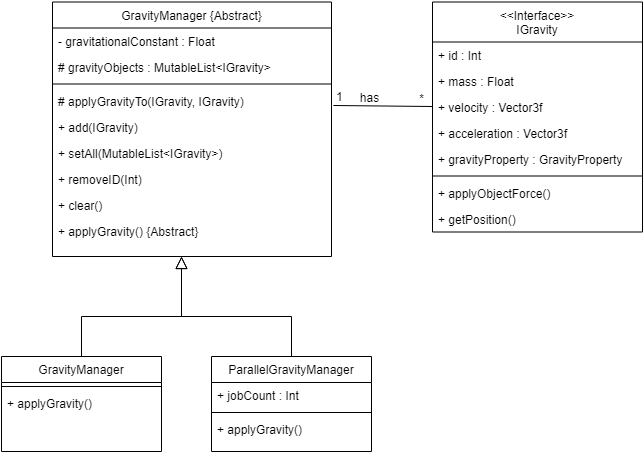


Abbildung : [Klassendiagramm Gravitationssystem]

### Kollisionsbearbeiter

## Ablauf der Applikation

Beim 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, 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 bestimme Anzahl von Aktualisierungen pro Sekunde (*UPS*) begrenzt. Die UPS der Benutzeroberfläche sind auf 60 festgesetzt. Jedoch können die maximale UPS der Objekte in der Benutzeroberfläche auf 1 bis 600 Aktualisierungen pro Sekunde gesetzt werden.

Die *Renderfunktion* wird jedoch so oft wie möglich aufgerufen. Wenn der Fall eintritt, dass die festgelegten UPS nicht in einer Sekunde berechnet werden, dann werden die UPS gedrosselt, um ein Aufstauen von Aktualisierungen zu vermeiden.

### Aktualisierungsfunktionen

Die Aktualisierungsfunktion der Benutzeroberfläche (*UI*) aktualisiert die einzelnen UI-Elemente. Dabei wird zum Beispiel überprüft, ob die Computermaus sich über einem Element befindet oder ob der Textpositionsindikator eines Textfeldes eingeblendet oder ausgeblendet wird.

In der Aktualisierungsfunktion der Objekte wird die aktuale 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.

### Renderfunktion

In der *Renderfunktion* werden die Objektrohdaten an die Grafikkarte hochgeladen, damit ein Bild ausgegeben werden kann. Außerdem wird die aktuelle Framerate berechnet und angezeigt. Der genaue Prozess zum Anzeigen der Objekte wird im Kapitel [3.5 OpenGL Rendering S.25] 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.

## Parallels iterieren auf einer Liste

In dem Projekt wird oftmals über eine List iteriert und auf das ausgewählte Objekt eine Operation angewendet. Wenn eine Operation eine längere Zeitspanne benötigt, 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 Aufgabenstellung auf ein Objekt 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, welches 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 [Q8 Sweep and prune S.II].

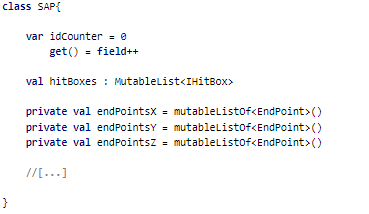


Abbildung : [Klassendefinition SAP ohne Funktionen]

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



Abbildung : [Klassendefinition EndPoint]

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

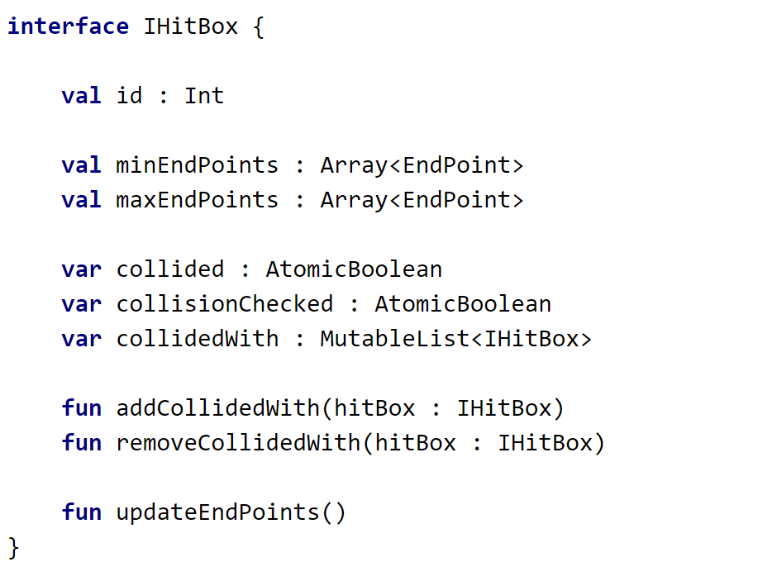


Abbildung : [Schnittstelle IHitbox]

Jede *Hitbox* besitzt einen unikalen Identifizierer, der bei der Erstellung vom *Sap* zugewiesen wird. Außerdem enthält jede *Hitboxstruktur* die 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. Wenn 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 nun 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 Parallels iterieren auf einer Liste S.11].

Da beim Durchlaufen der Liste auf andere Listenobjekte zugegriffen wird, muss gewährleistet sein, dass ein Job nicht mit einem anderen Job gleichzeitig versucht, auf eine Variable zuzugreifen. 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.12] 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, 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]](#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. Wenn eine Variable nicht threadsicher gesetzt wird, 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 überprüft. Die sequenziellen Testergebnisse ermöglichen einen Vergleich 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 sein. 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. Nur dann, wenn ein Objekt diese Schnittstelle implementiert, 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.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

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.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung : [Enum GravityProperties]

Ein *IGravity* Objekt kann zum Beispiel als Quelle dienen, sodass andere Objekte angezogen werden. 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.17] 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 Parallels iterieren auf einer Liste S.11] auf die Liste mit den Objekten, die von der Gravitation beeinflusst werden, angewendet.

Ein Bild, das Text, Person, Screenshot, Dokument enthält.

Automatisch generierte Beschreibung

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 [ANHANG LINK].

## Kollisionsbearbeiter

Nachdem zwei Objekte miteinander kollidieren, verändert der Kollisionsbearbeiter die Objekte, welche die *IApplier* Schnittstelle implementieren, nach den vorgegebenen Parametern. Hierbei wird entschieden, ob die beiden Objekte voneinander abprallen [3.4.2 Abprall von Objekten S.22] oder ob sie in viele Splitter zerlegt werden [0 Da sich nur die Geschwindigkeit auf den Kollisionsachsen verändert, werden nur die Werte auf den Kollisionsachsen ausgetauscht.

### Überschneidung der Objektboxen

Bei einer Kollision überschneiden sich die beiden Hitboxen der Objekte. Da es in der realen Welt, bei einer Kollision, nicht zu so einer Überschneidung kommt, müssen die beiden Objekte auf ihren Flugbahnen so weit zurücksetzt werden bis sie sich nur noch berühren, aber nicht schneiden. Dieses Verfahren benutzt ebenfalls die ermittelte Kollisionsachse [3.4.1 Kollisionsachse S.21].

Wenn die Kollisionsachse der beiden Objekte ermittelt werden kann, wird die Verschiebung auf den Kollisionsachsen bestimmt. Wenn eines der beiden Objekte die Eigenschaft besitzt, dass es nicht mit anderen Objekten interagiert, wird nur ein Objekt beweget.

Sollte der Fall eintreten, dass die Kollisionsachse zwischen den beiden Objekten nicht bestimmt werden kann, wird ein andere Funktion genutzt um die Objekte so zu translatieren. Diese Funktion bestimmt die kleinstmögliche Translation auf einer Achse, um die Objekte aus dem jeweiligen Begrenzungsrahmen zu verschieben.

Zersplitterung S.22]. Zudem müssen die Positionen so verändert werden, dass die Boxen sich nichtmehr überschneiden [3.4.4 Überschneidung der Objektboxen S.22].

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

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 [3.4.3 Abprall von Objekten S.22] 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 alle Achse 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 verändert, werden nur die Werte auf den Kollisionsachsen ausgetauscht.

### Überschneidung der Objektboxen

Bei einer Kollision überschneiden sich die beiden Hitboxen der Objekte. Da es in der realen Welt, bei einer Kollision, nicht zu so einer Überschneidung kommt, müssen die beiden Objekte auf ihren Flugbahnen so weit zurücksetzt werden bis sie sich nur noch berühren, aber nicht schneiden. Dieses Verfahren benutzt ebenfalls die ermittelte Kollisionsachse [3.4.1 Kollisionsachse S.21].

Wenn die Kollisionsachse der beiden Objekte ermittelt werden kann, wird die Verschiebung auf den Kollisionsachsen bestimmt. Wenn eines der beiden Objekte die Eigenschaft besitzt, dass es nicht mit anderen Objekten interagiert, wird nur ein Objekt beweget.

Sollte der Fall eintreten, dass die Kollisionsachse zwischen den beiden Objekten nicht bestimmt werden kann, wird ein andere Funktion genutzt um die Objekte so zu translatieren. Diese Funktion bestimmt die kleinstmögliche Translation auf einer Achse, um die Objekte aus dem jeweiligen Begrenzungsrahmen zu verschieben.

### Zersplitterung

Bei einer Kollision von zwei Objekte, besteht die Möglichkeit, dass diese nicht voneinander abprallen, sondern in viele kleine Objekte zerlegt werden. Das Zersplittert wird ausgelöst, indem die Aufprallgeschwindigkeit () einen gewissen Wert überschreitet.

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

Beiden kollidieren Objekte werden nach den Berechnungen gelöscht.

### Parallelisierung

Damit alle Funktionen des Kollionsbearbeiter parallel sowie sequenziell das gleiche Resultat liefern musste insbesondere darauf geachtet werden, dass keine *race condition* auftreten. Hierfür ist zum Beispiel ein *AtomicBoolean* verwendet, um zu überprüfen, ob ein Objekt abgearbeitet wurde. Außerdem werden alle resultierenden Geschwindigkeiten nicht während dem parallelen Iterieren gesetzt, sondern in einer *ConcurrentHashMap* zwischengespeichert und am Ende sequenziell gesetzt.

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

Ein weiteres Problem ist, das zufällige Verteilen der Objektpositionen, nach der Zersplitterung. 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, der kollidierenden Objekte addiert. (Dadurch, dass durch die Parallelisierung nicht festgelegt ist, wann welcher Würfel erschaffen wird, kann nicht garantiert werden, dass die Identifikationsnummer eines Würfels bei unterschiedlichen Durchläufen der Anwendung gleichbleibt.)

### 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 des parallelen und sequenziellen ausführen des Algorithmus sind in zwei verschiedenen Test aufgeteilt. Dabei muss sichergestellt sein, dass die sequenziellen Resultate akkurat sind, da sie als Vergleichswerte für die parallelen Resultate dienen.

Der erste Test überprüft das Abprallen von 50000 Objekten in einem Berechnungszyclus. 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 zersplitern. Bei der Überprüfung werden die Positionen und Geschwindigkeiten verglichen, ob für jedes sequenziell berechnete Objekt ein paralleles Objekt vorliegt. Hierbei ist die Position der Objekte in den jeweiligen Listen und die Identifikationsnummer nicht immer gleich.

## 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 zum Beispiel die Transformationsmatrix[[6]](#footnote-6), die Farbe und weitere Eigenschaften, die die Grafikkarte für den Zeichnungsprozess benötigen könnte. Das aktuelle Kapitel beschäftigt sich mit den einzelnen Möglichkeiten, ein Objekt möglichst effizient an die Grafikkarte zu übergeben und darzustellen.

Außerdem ist die Leistung der verschiedenen Renderverfahren mittels Testversuchen dargestellt.

### Simpler Renderprozess

Eine einfache Form mehrere Würfel abzubilden, ist es, sie separat zu behandeln. Hierbei wird jedes Modell des Würfels, jede Transformationsmatrix und jede weitere Eigenschaft für jeden Würfel separat hochgeladen und vom zuständigen *Shader* bearbeitet. Dieser Prozess hat den Vorteil, dass einzelne Eigenschaften sehr einfach bearbeitet und hinzugefügt werden können. 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]

### Darstellung des Renderprozesses am gleichen Modell

Um redundante Daten und Hochladeprozesse zu vermeiden, kann es sinnvoll sein, dass Modell nur 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 Code und bietet 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 verwendet wird. [3.6 Benutzeroberfläche S.28]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 Objekten, welche oftmals sehr ähnliche Eigenschaften besitzen, eignet sich das *Instancing*. Dieses Verfahren benutzt ähnlich wie Abschnitt [LINK] nur ein Modell. Das Besondere an diesem Verfahren ist, dass OpenGL von der CPU aus nur ein *redercall* übermittelt. Dieser Aufruf enthält zusätzlich die Anzahl der zu rendernen Objekte. Zwischen den einzelnen Renderprozessen, können jedoch keine Eigenschaften der Objekte bearbeitet werden. Damit nun die Objekte jeweils eine eigene Transformationsmatrix erhalten, muss vor dem Renderprozess ein *Floatarray* mit den zugehörigen Daten der Grafikkarte übermittelt werden. Zusätzlich wird der Grafikkarte übermittelt, wo ein Attribut eines Würfelns anfängt und aufhört. Dieses Verfahren eignet sich insbesondere für besonders große Mengen desselben Objektes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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

Jedes der drei Verfahren ist mittels eines Testskripts [Abbildung 15 S.27] ausgewertet. Das Testskript beschreibt die jeweils 17 Testdurchläufe. Bei jedem Durchlauf wird die Objektanzahl jeweils erhöht. Vor dem Wechseln zu einem anderen Verfahren wird der genutzte Rechner neugestartet und von jeglichen nicht genutzten IO-Geräten getrennt. 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 sollen.

## Benutzeroberfläche

Um die Applikation auf mehreren Systemen zu testen, ist ein Benutzeroberflächensystem entwickelt. Dieses System verwendet verschiedenste Elemente, um eine 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 besitzt immer Einschränkungen der Größe, Einschränkungen der Position, eine Farbe, eine Liste von inneren grundlegenden Elementen und verschiedenste Methoden. [LINK]

### Einschränkungen der Größe und Position

Um die relativen Größen und Positionen der einzelnen Elemente festzulegen wird ein *Constraintsystem* verwendet. Diese Einschränkungen liefern entweder eine relative Größe oder eine Relative Position. Jedes Element besitzt jeweils ein *widthConstraint*, *heightConstraint, positionXConstraint* und ein *positionYConstraint*.

Die Einschränkungen sorgen vor allem dafür, dass die einzelnen Größen und Positionen vom OpenGL System, welches in relativen Angaben abhängig von Fenstergrößen arbeitet, in andere Messgrößen umgewandelt werden, wie zum Beispiel Pixelangaben.

### Schachtelung von Elementen

Wie im Anschnitt [3.6.1 Grundelement S.28] beschrieben, besitzt jedes Benutzeroberflächenelement eine Liste von inneren Grundelementen. Dieses Schachtelungssystem ist inspiriert von *html* welches auch ermöglicht, dass Objekte andere Objekte beinhalten.



Abbildung : [Schachtelung von UI-Elementen]



Abbildung : [Zugehöriger Code von Abbildung 11]

Die Abbildung [Abbildung 16 S.29] zeigt die Visualisierung vom Code aus Abbildung [Abbildung 17 S.29]. Dieses Beispiel verdeutlich den Aufbau des UI-Systems und zeigt, wie das Einschränkungssystem die Positionierung von komplexen Szenerien ermöglicht.

### Schrift

Um eine komplexe Interaktion von Benutzer und dem Programm zu gewährleisten ist die Ausgabe von Text sehr wichtig. OpenGL bietet keine native Unterstützung zur Ausgabe von Text an. Deshalb ist sich an einem Implementierungsbeispiel [Q9 Font Rendering S.II] orientiert.

Um eine Schrift darzustellen, wird zunächst eine Einstellungsdatei (*.fnt*) und eine zugehörige Bilddatei geladen. In der Einstellungsdatei befinden sich Daten, wie sich das Bild zusammensetzt. Diese Daten beinhalten zum Beispiel die Pixelpositionen der einzelnen Buchstaben und seine zugehörigen 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 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 zum Beispiel auf einen Rechts-/ Linksklick reagieren. Dies wird erreicht indem bei einem Klicken mit der Maus in dem Haupt-benutzeroberflächenelement eine Klicküberprüfungsfunktion ausgelöst wird. In dieser Funktion eines *GUIElement* wird die aktuelle Mausposition mit den Eckpunkten des Elements verglichen. Wenn die Maus sich im Element befindet, werden alle inneren Elemente auf die gleiche Weise überprüft. Es wird anschließen die *OnClick* Funktion aufgerufen von dem Element, welches in der Schachtelung am tiefsten ist und den Mauszeiger in seinem Bereich hat. Zusätzlich wird der Fokus auf dieses Element gelegt.

Das UI-System kann zusätzlich Tastatureingaben verarbeiten, diese Tastatureingaben werden an das Element weitergegeben, welches im Fokus der Anwendung ist. Wenn kein Element angeklickt wurde, werden die Eingaben nicht weiter behandelt.

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

## Analyse

### Einstellungen

### Konfigurationsdatei

### Testsysteme

# Fazit

## Ergebnisse

## Erkenntnisse

## 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. 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 | Arbeits PC |  |  |  |  |  |  |
| 3 | U Pc |  |  |  |  |  |  |
| 4 | F Pc |  |  |  |  |  |  |
| 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)