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](#_Toc103942405)

[Abbildungsverzeichnis D](#_Toc103942406)

[1 Abstract 1](#_Toc103942407)

[2 Einleitung 2](#_Toc103942408)

[2.1 Relevanz 2](#_Toc103942409)

[2.2 Zielsetzung 2](#_Toc103942410)

[2.3 Recherchephase 3](#_Toc103942411)

[2.4 Planung 3](#_Toc103942412)

[2.4.1 Definitionsphase 3](#_Toc103942413)

[2.4.2 Definition der Architektur 5](#_Toc103942414)

[2.4.3 Planung der Thesenüberprüfung 5](#_Toc103942415)

[2.4.4 Testumgebung 5](#_Toc103942416)

[2.5 Grundaufbau der Anwendung 5](#_Toc103942417)

[2.5.1 Gravitation 6](#_Toc103942418)

[2.5.2 Grundaufbau der Anwendung 6](#_Toc103942419)

[2.5.3 Grundaufbau der Anwendung 6](#_Toc103942420)

[2.6 Veranschaulichung 6](#_Toc103942421)

[3 Hauptteil 7](#_Toc103942422)

[3.1 Parallels iterieren auf einer Liste 7](#_Toc103942423)

[3.1.1 Herangehensweise 7](#_Toc103942424)

[3.1.2 Funktionsaufbau 8](#_Toc103942425)

[3.2 Sweep-and-prune Algorithmus 8](#_Toc103942426)

[3.2.1 Datenstruktur 8](#_Toc103942427)

[3.2.2 Sequenzielle Kollisionserkennung 10](#_Toc103942428)

[3.2.3 Parallele Kollisionserkennung 11](#_Toc103942429)

[3.2.4 Veränderung der Objektpositionen 11](#_Toc103942430)

[3.2.5 Sequenzielle Teststruktur 12](#_Toc103942431)

[3.2.6 Parallele Teststruktur 12](#_Toc103942432)

[3.3 Das Gravitationssystem 13](#_Toc103942433)

[3.3.1 Gravitationsobjekt 13](#_Toc103942434)

[3.3.2 Gravitationsmanager 13](#_Toc103942435)

[3.3.3 Anwenden der Gravitation 14](#_Toc103942436)

[3.3.4 Parallelisierung 14](#_Toc103942437)

[3.3.5 Teststruktur 15](#_Toc103942438)

[Q1 Quelle der Formeln siehe [Q1 Projekt OuterSpace 16](#_Toc103942439)

[Q2 Demtröder2006\_Book\_Experimentalphysik1 16](#_Toc103942440)

[4 Fazit 17](#_Toc103942441)

[4.1 Ergebnisse 17](#_Toc103942442)

[4.2 Erkenntnisse 17](#_Toc103942443)

[4.3 Ausblick 17](#_Toc103942444)

[Quellenverzeichnis I](#_Toc103942445)

[Q3 Projekt OuterSpace I](#_Toc103942446)

[Q4 Demtröder2006\_Book\_Experimentalphysik1 I](#_Toc103942447)

[Q5 Mathematics of Satellite Motion I](#_Toc103942448)

[Q6 JUnit I](#_Toc103942449)

[Q7 Lightweight Java Game Library I](#_Toc103942450)

[Q8 42 Years of Microprocessor Trend Data I](#_Toc103942451)

[Q9 Sweep and prune II](#_Toc103942452)

[Anhang III](#_Toc103942453)

[A1 Sweep-and-prune Testfälle III](#_Toc103942454)

[A2 Gravitationssystem Testfälle IV](#_Toc103942455)

Abbildungsverzeichnis

[Abbildung 1: [Listenzerteilung zur Parallelisierung] 7](#_Toc103942456)

[Abbildung 2: [Erweiterungsfunktion foreachParallel] 8](#_Toc103942457)

[Abbildung 3: [Klassendefinition SAP ohne Funktionen] 9](#_Toc103942458)

[Abbildung 4: [Klassendefinition EndPoint] 9](#_Toc103942459)

[Abbildung 5: [Schnittstelle IHitbox] 9](#_Toc103942460)

[Abbildung 6: [SAP-Kollisionsüberprüfung der X-Achse] 10](#_Toc103942461)

[Abbildung 7: [SAP-Kollisionsüberprüfung der Y /Z-Achse] 11](#_Toc103942462)

[Abbildung 8: [Schnittstelle IGravity] 13](#_Toc103942463)

[Abbildung 9: [Enum GravityProperties] 14](#_Toc103942464)

[Abbildung 10: [Aufbau der Gravitationsfunktion] 15](#_Toc103942465)

# Abstract

# Einleitung

Die im Zuge der Bachelorarbeit zu entwickelnde Anwendung, zeigt dem Nutzer eine dreidimensionale Abbildung unserer Welt und dessen orbitalen Himmelskörper, wie zum Beispiel Satelliten oder Schrottobjekten.

Der Nutzer wird Einstellungen über die Menge der Objekte und deren Verhalten vornehmen können. Über diese Einstellungsmethode ist es auch möglich bei potenziellen Kollisionen die Menge und Streuung der entstandenen Kind-Objekten zu bestimmen. Der Nutzer kann aus verschiedenen Kameraperspektiven wählen und somit die Simulation aus unterschiedlichen Blickwinkeln betrachten. Außerdem ist die Wiedergabegeschwindigkeit der Simulation bestimmbar und der Nutzer kann sich zusätzlich frei im dreidimensionalen Raum bewegen.

## 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 auf mehr Kerne in ihren CPUs statt schnelleren Taktfrequenzen, welches das parallele bearbeiten von Aufgaben zusätzlich begünstigt. [Q6 42 Years of Microprocessor Trend Data S.I]

## 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.

## Planung

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 [2.5 Grundaufbau der Anwendung S.5] noch ausführlicher behandelt.

### Definitionsphase

Die erste Phase besteht zum Großteil aus dem Planen der Architektur, 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 ausgeführtes Kollisionssystem mit einem sequenziellen Kollisionssystem zu ermöglichen. [siehe Architektur].

Um gewisse Abläufe und Prozesse zu veranschaulichen sind ..., … und Diagramme 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.

Hierbei werden alle Features definiert und getestet. Da das Grundgerüst der zu entwickelnden Programmiersprache in der Sprache Kotlin geschrieben wurde, wird diese Programmiersprache weiterhin beibehalten. Ein wesentlicher Bestandteil ist die Planung der Umsetzung von einer prozeduralen Programmiersprache in die gewünschte objektorientieren Programmiersprache.

### Definition der Architektur

Das Framework des dreidimensionalen Outputs ist die Lightweight Java Game Library in kurz LWJGL [Q5 Lightweight Java Game Library S.I]. Da in einem anderen Praktikum bereits eine Grundstruktur in Kotlin entstanden ist, gilt es diese anzupassen.

### Planung der Thesenüberprüfung

Um zu überprüfen, ob und inwiefern sich das Parallelisieren des Kollisionssystems 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.

### Testumgebung

Um gewisse Bereiche der Anwendung zu überprüfen sind Test zu erstellen. Diese Tests verwenden die JUnit Bibliothek [Q4 JUnit S.I]. Da manche Objekte nicht direkt zum Testen geeignet sind, werden separate Testobjekte erstellt, die auf die zugehörige Schnittstelle zugreifen.

## 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 Kollision System. Wie im Abschnitt [2.3 Recherchephase S.3] beschrieben, gilt es ein geeigneter Algorithmus auszuwählen, der die gegebenen Anforderungen erfüllt.

### 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.

### 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.

Der Nutzer sollte Text, sowie Zahlen eingeben, Knöpfe drücken, Schieberegler verschieben und Optionen an und ausschalten können.

## Benutzeroberflächendesign

## Parallelisierung

## Ablauf der Applikation

# Hauptteil

Der Hauptteil beschäftigt sich mit der Entwicklung und den daraus folgenden Problemstellungen der einzelnen Implementierungsphasen.

## 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 1: [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 2: [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 [Q7 Sweep and prune S.II].

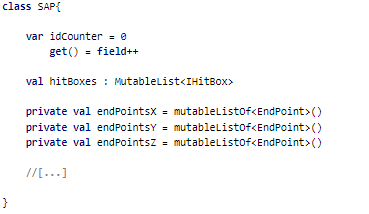


Abbildung 3: [Klassendefinition SAP ohne Funktionen]

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 4: [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.

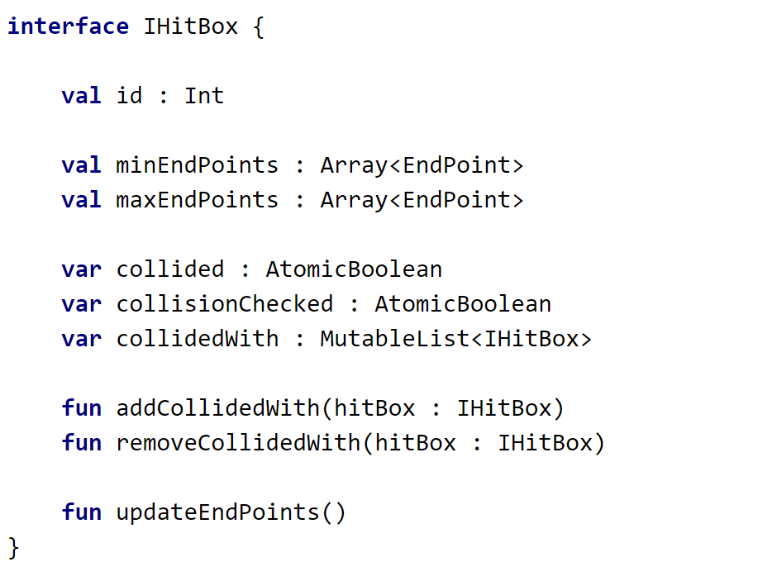


Abbildung 5: [Schnittstelle IHitbox]

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 6: [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 7: [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.7].

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.8] 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 [LINK] beschrieben, sollen Himmelskörper einer Gravitation ausgesetzt sein. Hierbei wird das Newtonsche Gravitationsgesetz (LINK) auf die einzelnen Objekte anzuwenden. 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. Wenn ein Objekt diese Schnittstelle implementiert, kann es einem *GravityObjectContainer* zugeordnet werden. Jedes *IGravity* Objekt besitzt eine Masse, eine Geschwindigkeit und eine Beschleunigung. Außerdem muss ein Objekt, welches die Schnittstelle implementiert, seine aktuelle Position wiedergeben können.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 8: [Schnittstelle IGravity]

### Gravitationsmanager

Der *GravityObjectContainer* beinhaltet zwei Listen von *IGravity* Objekten. Die erste Liste beinhaltet alle Objekte, die der Gravitation ausgesetzt sind. Die zweite Liste enthält alle Objekte, die eine Anziehung auf andere Objekte ausüben.

Via der *add* Funktion lassen sich *IGravity* Objekte dem Container hinzufügen. Diese Funktion nimmt ein *IGravity* Objekt und eine *GravityProperty* auf. Die *GravityProperty* entscheidet, ob ein Objekt in eine oder in beide Listen aufgenommen wird.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 9: [Enum GravityProperties]

### Anwenden der Gravitation

Wie schon im Abschnitt [3.3 LINK] erwähnt, wendet der Algorithmus das Newtonsche Gravitationsgesetz an. Um die Kraft (*F*) zwischen zwei Objekten zu ermitteln, wird folgende Formel angewendet, wobei (*G*) die Gravitationskonstante, (*m*) die Masse eines Objektes und (*r*) die Distanz zwischen den beiden Objekten ist.

Die ausgerechnete Kraft (*F*), die Richtung () von Obj1 zu Obj2 und die Beschleunigung () werden verwendet, um die Geschwindigkeit an Zeitpunkt t+1 zu berechnen.

Diese Berechnungen werden für jedes Objekt, welches einer Gravitation ausgesetzt werden soll, *n*-mal angewendet. (n*:* Anzahl der Objekte, die eine Gravitation ausüben)

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

### Parallelisierung

Die Parallelisierung in diesem Modul wird realisiert, indem die *ForeachParallel* Erweiterungsfunktion [Link] auf die Liste mit den Objekten, die von der Gravitation beeinflusst werden, angewendet wird.

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

Automatisch generierte Beschreibung

Abbildung 10: [Aufbau der Gravitationsfunktion]

Die obere Abbildung zeigt den sequenziellen und parallelen Aufbau der Gravitationsfunktionen.

### Teststruktur

Für den Gravitation Algorithmus sind insgesamt fünf Test entstanden. Diese Tests benutzen spezielle *TestGravityObjekte*, welche 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 liegen. Im Ursprung (0,0,0) befindet sich ein weiteres Objekt, welches von den beiden Objekten angezogen wird. Da beide äußeren Objekte nicht von der Gravitation beeinflusst werden und die gleiche Masse besitzen, sollte das Objekt im Ursprung sich nicht bewegen.

Der zweite, dritte und vierte Testfall überprüfen, ob ein Objekt, welches auf einer bestimmten Achse ein weiteres Objekt im Ursprung umkreist, nach einer bestimmten Zeit (t), sich wieder an seinem Ausgangspunkt () befindet. Hierfür ist jeweils 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, wenn das Zentrale Objekt eine Masse von 2 Messeinheiten besitz, dass das umkreisende Objekt mit einem Abstand von 20 eine Periode von 154 Zeiteinheiten besitzt. (G = 6.674)

Test 5 überprüft die parallele Variante des Gravitationsalgorithmus, hierfür werden 200 Testobjekte mit einer zufälligen Masse und Position generiert. Um diese Variante zu überprüfen wird der Algorithmus einmal parallel und einmal sequenziell ausgeführt. Die Ausgabe des sequenziellen Algorithmus, wird anschließend mit der Ausgabe des parallelen Algorithmus verglichen.

Die Abbildungen und genutzten Werten der Testfälle sind im Anhang hinterlegt, siehe [ANHANG LINK].

# 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. 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#:%7E:text=As%20seen%20in%20the%20equation,2.](https://www.physicsclassroom.com/class/circles/Lesson-4/Mathematics-of-Satellite-Motion%23:%7E:text=As%20seen%20in%20the%20equation,2.)

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>

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. [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)