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)

erst Prüfer: Prof. Dr. Christian Kohls

zweit Prüfer: Alexander Dobrynin

Remscheid, XX.XX.XXXX

Inhaltsverzeichnis

[Inhaltsverzeichnis B](#_Toc103703435)

[Abbildungsverzeichnis E](#_Toc103703436)

[1 Abstract 1](#_Toc103703437)

[2 Einleitung 2](#_Toc103703438)

[2.1 Relevanz 2](#_Toc103703439)

[2.2 Zielsetzung 2](#_Toc103703440)

[2.3 Recherchephase 3](#_Toc103703441)

[2.4 Planung 3](#_Toc103703442)

[2.4.1 Definitionsphase 3](#_Toc103703443)

[2.4.2 Definition der Architektur 5](#_Toc103703444)

[2.4.3 Planung der Thesenüberprüfung 5](#_Toc103703445)

[2.4.4 Testumgebung 5](#_Toc103703446)

[2.5 Grundaufbau der Anwendung 5](#_Toc103703447)

[2.6 Veranschaulichung 6](#_Toc103703448)

[3 Hauptteil 7](#_Toc103703449)

[3.1 Parallels iterieren auf einer Liste 7](#_Toc103703450)

[3.1.1 Herangehensweise 7](#_Toc103703451)

[3.1.2 Funktionsaufbau 8](#_Toc103703452)

[3.2 Sweep-and-prune Algorithmus 8](#_Toc103703453)

[3.2.1 Datenstruktur 8](#_Toc103703454)

[3.2.2 Kollisionserkennung sequenziell 10](#_Toc103703455)

[3.2.3 Kollisionserkennung parallel 11](#_Toc103703456)

[3.2.4 Verändern der Objektpositionen 11](#_Toc103703457)

[3.2.5 Teststruktur sequenziell 12](#_Toc103703458)

[3.2.6 Teststruktur parallel 12](#_Toc103703459)

[3.3 Das Gravitationssystem 13](#_Toc103703460)

[3.3.1 Gravitationsobjekt 13](#_Toc103703461)

[3.3.2 Gravitationsmanager 13](#_Toc103703462)

[3.3.3 Anwenden der Gravitation 14](#_Toc103703463)

[3.3.4 Parallelisierung 14](#_Toc103703464)

[3.3.5 Teststruktur 15](#_Toc103703465)

[3.4 Anpassung und Erweiterung der Sprache 17](#_Toc103703466)

[3.4.1 Feature Auflistung 17](#_Toc103703467)

[3.4.2 Kontext 18](#_Toc103703468)

[3.4.3 Variablen: Einfache Typen und Objekte 18](#_Toc103703469)

[3.4.4 Überlagerte Funktionen 19](#_Toc103703470)

[3.4.5 Generische Klassen und Funktionen 19](#_Toc103703471)

[3.4.6 Dateiverwaltung 19](#_Toc103703472)

[3.4.7 Interne Funktionen 19](#_Toc103703473)

[3.5 Zusammenführung und Rendering 19](#_Toc103703474)

[3.5.1 Initialisierung 20](#_Toc103703475)

[3.5.2 Update und OnKey 20](#_Toc103703476)

[3.5.3 Umwandlung von Objekten 20](#_Toc103703477)

[3.6 Einfaches Beispiel 20](#_Toc103703478)

[3.7 Implementierung der Anwendungsszenarien 21](#_Toc103703479)

[3.7.1 Sortieralgorithmen 21](#_Toc103703480)

[3.7.2 Perlin Noise 22](#_Toc103703481)

[3.7.3 Random Walk 23](#_Toc103703482)

[3.8 Testfälle 24](#_Toc103703483)

[3.9 Vergleich zu Alternativen 24](#_Toc103703484)

[3.9.1 OpenGL 24](#_Toc103703485)

[3.9.2 Three-Dimensional Plotting in Matplotlib 25](#_Toc103703486)

[3.9.3 Unity 25](#_Toc103703487)

[4 Fazit 26](#_Toc103703488)

[4.1 Ergebnisse 26](#_Toc103703489)

[4.2 Erkenntnisse 26](#_Toc103703490)

[4.3 Ausblick 27](#_Toc103703491)

[Quellenverzeichnis I](#_Toc103703492)

[Q1 Perlin Noise Web.Archive.Org I](#_Toc103703493)

[Q2 15 Sorting Algorithms in 6 Minutes I](#_Toc103703494)

[Q3 Projekt OuterSpace I](#_Toc103703495)

[Q4 3DCode – 3 Sorting Algorithms I](#_Toc103703496)

[Q5 3DCode – Perlin Noise I](#_Toc103703497)

[Q6 3DCode – Random walk I](#_Toc103703498)

[Q7 Projekt PuC\_C-- I](#_Toc103703499)

[Q8 OpenGL Webseite II](#_Toc103703500)

[Q9 Merge Sort Algorithm II](#_Toc103703501)

[Q10 Lightweight Java Game Library II](#_Toc103703502)

[Q11 Matplotlib documentation II](#_Toc103703503)

[Q12 OpenGL Wikipedia II](#_Toc103703504)

[Q13 Perlin-Noise Wikipedia II](#_Toc103703505)

[Anhang III](#_Toc103703506)

[A1 Sweep-and-prune Testfälle III](#_Toc103703507)

[A2 Github repository Link V](#_Toc103703508)

[A3 Testfälle V](#_Toc103703509)

[A4 Codebeispiel Link [3 Sorting Algorithms] V](#_Toc103703510)

[A5 Bubblesortfunktion VI](#_Toc103703511)

[A6 Mergesortfunktion VIII](#_Toc103703512)

[A7 Combosortfunktion X](#_Toc103703513)

[A8 Codebeispiel Link [Perlin Noise] XI](#_Toc103703514)

[A9 Codebeispiel Link [Random Walk 3D] XI](#_Toc103703515)

[A10 Performens Test [Kotlin vs 3D-Code] XII](#_Toc103703516)

Abbildungsverzeichnis

[Abbildung 1: [Listenzerteilung zur Parallelisierung] 7](#_Toc103425705)

[Abbildung 2: [Erweiterungsfunktion foreachParallel] 8](#_Toc103425706)

[Abbildung 3: [Klassendefinition SAP ohne Funktionen] 9](#_Toc103425707)

[Abbildung 4: [Klassendefinition EndPoint] 9](#_Toc103425708)

[Abbildung 5: [Schnittstelle IHitbox] 9](#_Toc103425709)

[Abbildung 6: [SAP-Kollisionsüberprüfung der X-Achse] 10](#_Toc103425710)

[Abbildung 7: [SAP-Kollisionsüberprüfung der Y /Z-Achse] 11](#_Toc103425711)

[Abbildung 8: [Schnittstelle IGravity] 13](#_Toc103425712)

[Abbildung 9 [Variablen Zuweisung] 15](#_Toc103425713)

[Abbildung 10 [Einfaches Beispiel] 17](#_Toc103425714)

[Abbildung 11 [Perlin Noise Output] 20](#_Toc103425715)

[Abbildung 12 [Random Walk 3D] 21](#_Toc103425716)

# 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. (Quelle)

## 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 geeignete Daten wie zum Beispiel Größe, Flugbahn und die Geschwindigkeit zu recherchieren.

## 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 [1.4 Grundaufbau der Anwendung S.6] 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 [Q10 Lightweight Java Game Library S.17]. 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 verschieden skalierten Testdurchläufen die Performance des jeweiligen Systems hervorbringen.

### Testumgebung

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

## 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]. 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. Ein Beispiel hierfür ist das UI-System, welches keinen dynamischen Text unterstützt.

Der Grundaufbau der Anwendung besteht aus einem fertiggestellten Projekt, welches als Grundlage dient.

Das erste Projekt ist im Rahmen des Moduls *Programmiersprachen und Compilerbau* entstanden. Dieses Projekt hatte das Ziel eine Programmiersprache ähnlich wie *C* von Grund auf zu entwickeln. Hierfür musste ein *Lexer* der den Quellcode in logische zusammengehörige Einheiten zerlegt entwickelt werden. Darauffolgend wurden diese *Tokens* durch einen *Parser* in einen *Abstract Syntax Tree*, kurz *AST*, umgewandelt. Dieser Teil des Modulprojektes wurde von dem Studenten Lukas Momberg entwickelt. Weiterdessen wurde der *AST* evaluiert und auf seine Typisierung überprüft. [Q7 Projekt PuC\_C-- S.16]

Der Ablauf gilt, wie in folgender Abbildung veranschaulicht anzupassen.

Das zweite Projekt ist im Modul *Computergrafik und Animation* entstanden. In diesem Modul wurde eine dreidimensionale Weltraumsimulation geschaffen, welche es ermöglicht verschiedene Sonnensysteme zu generieren und diese zu animieren. Das Projekt wurde in Zusammenarbeit mit Frau Anastasia Chouliaras erstellt. [Q3 Projekt OuterSpace S.16]

Es ist geplant nur die grobe Struktur des Ladens und Rendern der Objekte für dieses Projekt zu verwenden.

## Veranschaulichung

Nach der Fertigstellung der Programmierumgebung sind verschiedene Veranschaulichungen zu entwickelt. Als Inspiration wurde zum Beispiel das YouTube Video von *TimoBingmann* [Q2 15 Sorting Algorithms in 6 Minutes S.16] genutzt, welches 15 verschiede Sortieralgorithmen grafisch darstellt. Ein weiteres Beispiel ist die Visualisierung von Perlin Noise unter verschiedenen Parametern.

# 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 Zugriffselement enthält. Um 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 Algorithmus zu effizienten Kollisionserkennung von Objekten im dreidimensionalen Raum. Ein Objekt (*Hitbox*) im *SAP* definiert sich durch seinen achsenorientierten Begrenzungskasten. Jedes Objekt besitzt hierbei einen minimalen (*min*) und einem maximalen *(max*) Wert, je Achse. Daher hat ein Objekt in einer dreidimensionalen Umgebung sechs Wert. Alle Werte einer Achse werden in einer sortierten Liste gespeichert.

### Datenstruktur

Um den Algorithmus darzustellen werden gewisse Datenstrukturen benötigt. Hierbei wurde sich stark an den Strukturen aus dem Paper siehe [Q2 Sweep and prune S.I] orientiert.

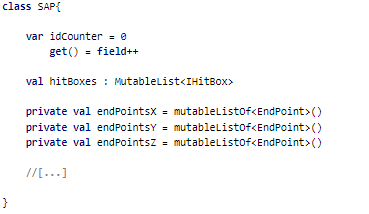


Abbildung 3: [Klassendefinition SAP ohne Funktionen]

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 die Referenz zu seinem Besitzer, den Koordinatenwert und ob es sich um den *min* Wert seines Besitzers handelt.

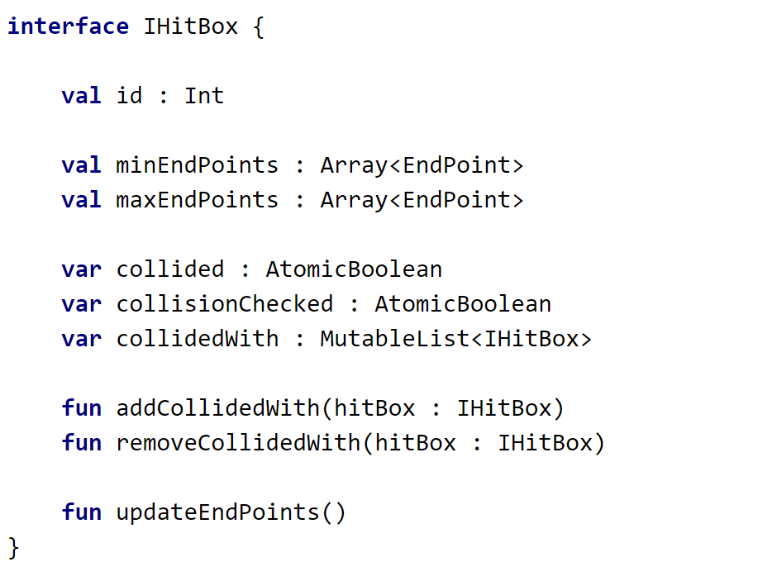


Abbildung 5: [Schnittstelle IHitbox]

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

### Kollisionserkennung sequenziell

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 wurde. 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 nacheinander überprüft. Die Überprüfung vergleich die minimalen und maximalen Werte der potenziell kollidieren Objekte und entscheidet, ob eine Kollision auf der Y und Z-Achse stattfindet.



Abbildung 7: [SAP-Kollisionsüberprüfung der Y /Z-Achse]

### Kollisionserkennung parallel

Beim Ausführen des Algorithmus wird statt der normalen *foreachIndexedSchleife* die neu entwickelte *foreachParallelIndex* Erweiterungsfunktion der Listklasse verwendet siehe [3.1 Parallels iterieren auf einer Liste S.7].

Da währet des Durchlaufens der List auf andere Listenobjekte zugegriffen wird, muss gewährlistet 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.

### Verändern der Objektpositionen

Wie im Abschnitt [3.2 Sweep-and-prune Algorithmus S.8] geschrieben müssen alle Endpunktelisten vor der Ausführung der Kollisionserkennung stehts sortiert sein. Wenn eine oder mehrere Objekte verschoben oder skaliert werden müssen seine Endpunktepositionen 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 performanter die drei Listen komplett durchzusortieren.

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

### Teststruktur sequenziell

Um sicherzustellen, dass der SAP-Algorithmus sequenziell sowie parallel einbahnfrei funktioniert, sind verschiedenste Tests entwickelt, siehe [A1 Sweep-and-prune Testfälle S.II]. 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 überprüft werden. Diese Anordnungen wurden mit der Software *Blender[[1]](#footnote-1)* erstellt. Nach der Ausführung wird jedes Objekt auf seine Kollisionspartner überprüft. Wenn nun zu wenig oder zu viele Kollisionen stattgefunden haben, gilt der Test als fehlgeschlagen.

### Teststruktur parallel

Die parallelisierte Ausführung wird ähnlich getestet wie die sequenzielle, aber hierbei muss zusätzlich beachtet werden, dass die Variablen bei der Ausführung des Algorithmus Threadsicher gesetzt werden. Wenn eine Variable nicht Threadsicher gesetzt wird, kann es passieren, dass gewisse Kollisionen nicht richtig erfasst werden.

Um auf diese Problemstellung zu testen, wird in einem 500 x 500 x 500 großen Bereich 5000 zufällig platzierte Testobjekte erschaffen. Diese werden vorab mit dem sequenziellen Algorithmus auf Kollisionen überprüft. Der Test vergleich die sequenziellen Ergebnisse mit den parallelen Ausführungen. Der Algorithmus wird mit einer Jobanzahl von 1 bis 100 getestet.

## Das Gravitationssystem

Wie in Abschnitt [LINK] sollen die Himmelskörper einer Gravitation, ähnlich wie die von der Erde, ausgesetzt sein. Hierfür ist das Newtonsches Gravitationsgesetz 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 List 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 Newtonsches Gravitationsgesetz an. Um die Kraft (*F*) zwischen zwei Objekten zu ermitteln wird Folgende Formel angewendet:

Hierbei ist (*G*) die Gravitationskonstante, (*m*) die Masse eines Objektes und (*r*) die Distanz zwischen den beiden Objekten.

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, welche eine 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 siehe [Link] auf die List 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.

Quelle der Formeln siehe [Q1 Mathematics of Satellite Motion S.I]

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].

## Anpassung und Erweiterung der Sprache

Wie im Kapitle [1.4 Grundaufbau der Anwendung S.6] beschrieben, baut das Projekt auf der eigen entwickelten Programmiersprache *C--* auf. Dieser Abschnitt beschäftigt sich mit der Anpassung und den Erweiterungen der Sprache.

### Feature Auflistung

Folgende Aufzählung beinhaltet alle Features, die die *3D Code* Sprache besitzt, welche nicht in der Sprache *C--* schon enthalten waren.

* Syntax
  + Variablen besitzen kein Anfangszeichen, wie z.B. ‘$‘
  + Semikolon optional
  + Inkludierung von Dateien
  + Additionszuweisungsoperator
* Schleifen
  + For-Schleife
* Definitionen
  + Klassen definieren
  + Private Klassenattribute/-funktionen
  + Überladene Funktionen
  + Überladene Konstruktoren
  + Generische Funktionen/ Klassen
* Typen
  + Float
  + Nullwerte
  + Arrays
  + Listen / Verlinkte Listen
  + Pair, Tripple und Vector3f
* Fließkommazahl Arithmetik
  + Division
* Sonstiges
  + Mathebibliothek

### Kontext

Der Kontext speichert und verwaltet globale und lokale Variablen. Jede Datei besitzt einen globalen Kontext. Weiterdessen besitzt jeder Codeabschnitt, wie zum Beispiel eine Funktion oder ein Block einen eigenen Kontext. Ein Block in einer Funktion besitzt somit eine Teilmenge des Funktionskontext.

Jedes Laufzeitobjekt besitzt dementsprechend auch einen eigenen Kontext, der bei Funktionsaufrufen genutzt werden kann.

### Variablen: Einfache Typen und Objekte

Variablen mit einfachen Typen wie zum Beispiel: Integer, Fließkommerzahlen und Strings werden als konstante Werte in dem aktuellen Kontext gespeichert. Im Gegensatz zu Objekten, die mit einem Konstruktor erzeugt und als *DynamicValue* gespeichert werden. In diesem Fall wird nur eine Referenz in der Variablen abgelegt. Dies hat zur Folge, dass mehrere Referenzen auf dasselbe Objekt, und dessen Inhalt, zeigen könnten.

Die Abbildung [Abbildung 2 [Variablen Zuweisung] S.8] zeigt Beispiele der Zuweisung und Verwendung der verschiedenen Variablentypen.



Abbildung 11 [Variablen Zuweisung]

Der Code in der oberen Abbildung, würde folgendes ausgeben:

Ausgabe:

6 | 7

b | b

### Überlagerte Funktionen

Überladene Funktionen besitzen den gleichen Namen aber unterschiedliche Parameter. Es werden alle Funktionen einer Datei in Form einer *Map*, bestehend aus dem Namen und dessen Deklaration, gespeichert.

Wenn nun eine Funktion aufgerufen wird, die mehrfach mit dem gleichen Namen in einer Datei existiert, muss nun der *Evaluator* die Funktion herausfiltern, die die gleichen Parameter besitzt, wie die aufgerufene.

### Generische Klassen und Funktionen

Die Sprache *3DCode* enthält die Möglichkeit Klassen oder Funktionen mit generischen Eigenschaften zu deklarieren. Hierfür ist der *Type Checker* besonders angepasst. Dieser muss zunächst prüfen, ob die generischen Typen richtig verwendet werden. Der *Evaluator* ist dementsprechend auch angepasst, indem er nun bei überlagerten Funktionen auch den generischen Anteil berücksichtigen muss.

### Dateiverwaltung

Da es möglich ist in mehrere Dateien zu programmieren und diese sich jeweils inkludieren können, muss vor dem Evaluieren und der Typüberprüfung eine Verlinkung stattfinden. Hierfür wurde ein neues Modul namens *Parsermanager* eingeführt. Dieser durchläuft jeweils eine Datei und bei einem *include* wird die jeweilige Datei auch überprüft. Somit werden alle Dateien einmalig rekursiv überprüft und miteinander verlinkt.

### Interne Funktionen

Da es oftmals sehr aufwendig oder unmöglich ist manche Features in der Sprache zu entwickeln, wird auf gewisse *Kotinfeatures* zurückgegriffen.

Ein wichtiges Beispiel wäre hierfür das Array. Diese Klasse ist in der *AST* Syntax vorhanden und wird bei einem Include durch den *Parsermanager* hinzugefügt. Beim Aufruf dessen Konstruktors, wird wiederrum zusätzlich eine integrierte Funktion namens *\_integratedFunctionSetArray* aufgerufen. Diese erzeugt ein *DynamicValue* Objekt, welches ein *Kotlin* Array enthält.

## Zusammenführung und Rendering

Dieser Abschnitt erläutert wie die Schnittstelle zwischen Programmiersprache und der Anzeige von den dreidimensionalen Objekten funktioniert.

### Initialisierung

Beim Starten der Anwendung wird eine Datei namens *App* gesucht und alle Inhalte geparst. Die Datei und alle Verbunddateien werden, in der *AST Form*, zwischengespeichert und nach der erfolgreichen Typüberprüfung evaluiert.

### Update und OnKey

Wenn eine Tastatureingabe stattfindet, wird jedes Mal die Funktion *OnKey*, mit dem vorherigen Kontext, evaluiert. Durch dieses Vorgehen kann die statische dreidimensionale Welt animiert werden. Dafür können die Eigenschafften der erschaffenen Objekte angepasst werden.

Nach dem *Update/ OnKey* Event werden die Objekte jeweils in ein *Renderable* umgewandelt und dem Nutzer ausgegeben.

### Umwandlung von Objekten

Im Umwandlungsprozess werden alle Eigenschaften des erstellten Objektes so umgewandelt, dass Sie kompatible mit der erstellten 3D-Engine sind. Hierfür werden Attribute wie die Skalierung, Rotation, Position und Einfärbung genutzt.

## Einfaches Beispiel

Das folgende Beispiel soll veranschaulichen, wie man ein dreidimensionales Objekt erschafft und dieses Transformiert. Dafür wird ein Stern geladen und dieser gelb eingefärbt. In jedem Updatezyklus wird dieser um 1° in seiner Y-Achse gedreht.

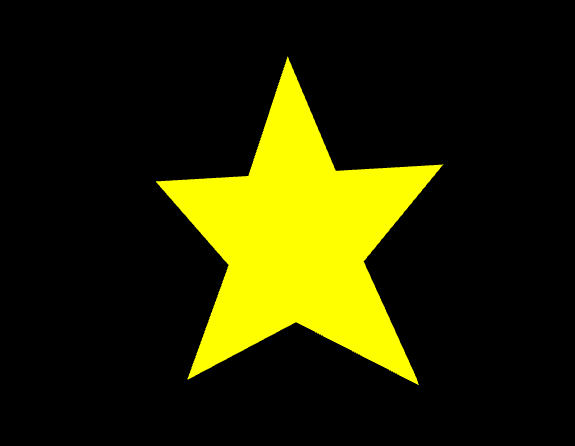


Abbildung 12 [Einfaches Beispiel]

Folgender Code wurde für die obere Ausgabe genutzt.

include "lib.List"

include "lib.Vector3f"

include "Object"

List objects<Object> = List<Object>(1)

Void Init(){

Object o = Object("assets/objects/star.obj")

o.color = Vector3f(255.0,255.0,0.0)

o.position = Vector3f(0.0,0.0,-30.0)

objects.Add(o)

}

Void Update(Float deltaTime, Float time){

Object o = objects.Get(0)

o.rotation = Vector3f(0.0,o.rotation.y + 1.0,0.0)

}

## Implementierung der Anwendungsszenarien

In diesem Abschnitt werden die entwickelten Beispiele erklärt und dessen Ausgabe erläutert. Weitergehend werden die Erkenntnisse, die bei der Programmierung entstanden sind, dargestellt und an Ihren Beispielen erläutert.

### Sortieralgorithmen

Dieses Beispiel soll verdeutlichen, wie Veränderungen von bereits erschaffenen Objekten in der Programmiersprache realisiert werden könnte. Hierfür sind drei Sortieralgorithmen ausgewählt, die das Szenario auf verschiedensten Arten abbilden.

Dafür ist eine Klasse entstanden, namens *SortObj*, die sich bei Ihrer Konstruktion selbst einen *Score*, eine zugehörige Farbe und eine Skalierung zuweist. Sodass beim Sortieren nur die Position verändert werden muss.

Der *Bubblesort* Algorithmus wurde mittels 26 Zeilen Code implementiert, wobei nur zwei Zeilen für das Aktualisieren der Position benötigt werden. [A4 Bubblesortfunktion S.19]

Bei dem zweiten Sortieralgorithmus handelt sich es um *Mergesort*. Dieser Algorithmus sortiert die Elemente zunächst in kleinen Paketen, wenn zwei Pakete der gleichen Größe sortiert wurden, werden diese zu einem größeren Paket zusammengefasst, welches dementsprechend auch sortiert wird. Dieses Verfahren wird so lange wiederholt, bis ein Paket der Größe der sortieren Menge vorliegt. Der Algorithmus besitzt eine Komplexität von O(n\*log(n)).[[2]](#footnote-2) Der implementierte Algorithmus [A5 Mergesortfunktion S.20] ist allerdings auf Sortiergrößen von glatten zweier Potenzen limitiert.

Der dritte Algorithmus ist eine Abwandlung von *Combsort*. Diese Implantation vergleicht wie der *Bubblesortalgorithmus* Elemente, die sich nebeneinander befinden, und vertauscht diese, wenn nötig. Der große Unterschied ist, dass dieser Algorithmus das größere Element bis zur nächsten schleifen Iteration nichtmehr verschiebt. [A6 Combosortfunktion S.21]

Die Ausführung, von den oben genannten drei Algorithmen, ist per Videoform [Q4 3DCode – 3 Sorting Algorithms S.16] abrufbar.

### Perlin Noise

Das zweite Beispiel beschäftigt sich mit dem *Perlin Noise* Algorithmus. Dieser Algorithmus soll zeigen, wie sich komplexe Formen konstruieren und sich in der dreidimensionalen Umgebung betrachteten lassen.

Der Algorithmus lässt sich wie folgt beschreiben:„Perlin Noise ist eine [Rauschfunktion](https://de.wikipedia.org/wiki/Rauschen_(Physik)) auf der Basis von seudozufälligen [Gradientenwerten](https://de.wikipedia.org/wiki/Gradient" \o "Gradient) an Gitterpunkten. […] Perlin Noise wird häufig in der [Bildsynthese](https://de.wikipedia.org/wiki/Bildsynthese) eingesetzt, um natürliche Phänomene wie Wolken, Landschaftstopologien oder Wasser zu simulieren.“ *[[3]](#footnote-3)*

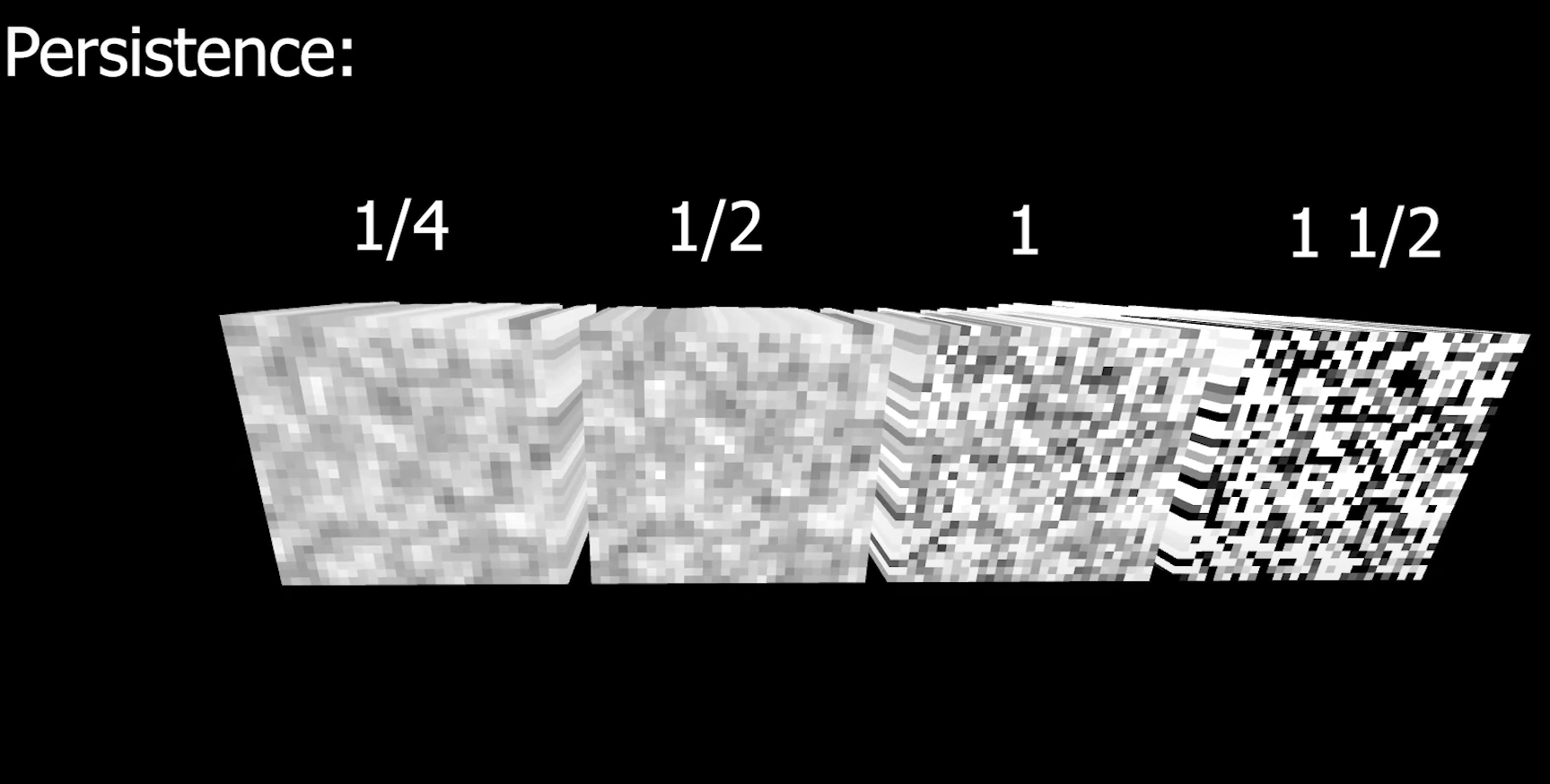


Abbildung 13 [Perlin Noise Output]

Die Abbildung [Abbildung 4 [Perlin Noise Output]] zeigt die Ausgabe des Algorithmus. Hierbei sind vier verschiedene Werte für die Persistenz gewählt. Das Video [Q5 3DCode – Perlin Noise S.16] veranschaulicht zusätzlich die dreidimensionale Ansicht, die aus einem anderen Blickwinkel hervorgeht.

Die Implementierung und das generelle Verständnis gehen aus Quelle [Q1 Perlin Noise Web.Archive.Org S.16] hervor. Der entwickelte Code ist im Anhang zu finden [A7 Codebeispiel Link [Perlin Noise] S.22].

### Random Walk

Der Random Walk ist als drittes Beispiel gewählt, da er ermöglicht eine stetig verändernde Welt dazustellen.

Bei jedem Aufruf der Updatefunktion wird ein neuer Block generiert. Dieser kann an jeder Seite und an jeder Ecke seines Vorgängers generiert werden. Somit wird bei jedem Aufruf eine von 26 möglichen zufälligen Position gewählt.[[4]](#footnote-4)

Ein Bild, das Karte enthält.

Automatisch generierte Beschreibung

Abbildung 14 [Random Walk 3D]

Die Abbildung [Abbildung 5 [Random Walk 3D] S.13] stellt die Ausgabe des oben beschrieben Algorithmus im Zeitpunkt t+2min 10s da. Die animierte Ausgabe ist unter [Q6 3DCode – Random walk S.16] zu finden.

## Testfälle

Um nach Veränderungen des Programmcodes weitergehend sicherzustellen, dass die Veränderung keine Fehler auslösen, sind insgesamt 170 Tests entstanden. Die Tests decken einen Großteil der Funktionalitäten der Programmiersprache ab. Über folgenden Link sind alle Tests abrufbar [A2 Testfälle S.18].

## Vergleich zu Alternativen

Es befinden sich mehrere Tools auf dem freien Markt, um dreidimensionale Objekte zu rendern und diese anzuzeigen. Die Unterkapitel erklären drei andere Ansätze und listen Vor und -Nachteile auf.

### OpenGL

„OpenGL […] ist eine Spezifikation einer plattform- und programmiersprachenübergreifenden Programmierschnittstelle (API) zur Entwicklung von 2D- und 3D-Computergrafikanwendungen. Der OpenGL-Standard beschreibt etwa 250 Befehle, die die Darstellung komplexer 3D-Szenen in Echtzeit erlauben.“ [[5]](#footnote-5) OpenGL ist als vergleich genutzt, da es erlaubt zwei und -dreidimensionale Objekte sehr hardwarenah abzubilden. Dieser Ansatz erfordert wiederrum ein erweitertes mathematisches Wissen, da er keine vorgefertigten Kameras oder Beleuchtungsmedien beinhaltet und somit nur sehr rudimentäre Features bietet. OpenGL wird oft in der Videospielindustrie eingesetzt, um sehr komplexe und ressourcenintensive Szenen darzustellen.

### Three-Dimensional Plotting in Matplotlib

Matplotlib ist eine Python Bibliothek. Die Bibliothekswebseite beschreibt sich übersetzt als: „Matplotlib ist eine umfassende Bibliothek zur Erstellung statischer, animierter und interaktiver Visualisierungen in Python.“ [[6]](#footnote-6) Sie ermöglicht ähnlich wie dieses Projekt, das Darstellen von dreidimensionalen Szenen. Diese Bibliothek wird, aber eher für eine frontale zweidimensionale Ansicht genutzt und ist weitestgehend hierfür auch optimiert. Außerdem ist es nicht möglich eigene dreidimensionale Objekte zu nutzen. (Stand: 10.12.2021)

### Unity

Das dritte Vergleichsobjekt ist die Spiel-Engine Unity. Die Engine bietet eine Vielzahl von vorgefertigten Features wie zum Beispiel Kameras, Bedeutung, Materialien und vieles weitere. Es kann wie dieses Projekt auch eine dreidimensionale Welt darstellen und interaktiv verändern. Wer in Unity ein Projekt realisieren möchte, muss dies mit der Unity-Applikation machen. Da Unity eine sehr große Anzahl von Features enthält kann, dies leicht abschreckend wirken.

# Fazit

Dieses Kapitel berichtet über aufgetretene Schwachstellen, sowie Vorteile, die die entwickelte Sprache im Gegensatz zu den herkömmlichen Alternativen bietet. Außerdem wird über mögliche Features berichtet die zu signifikant Verbesserungen in der Benutzung führen könnten.

## Ergebnisse

Grundsätzlich hat dieses Projekt gezeigt, dass es durchaus sinnvoll sein kann gewisse Problemstellung mit der Sprache *3D-Code* zu visualisieren, da sie ohne großes Vorwissen oder Kenntnisse im Bereich *Open GL* auskommt. Sie ermöglich auf verschiedene Weisen mit den selbstdefinierten Objekten zu interagieren und diese zu verändern. Außerdem lässt sich die Sprache auf verschiedensten Endgeräten ausführen, da sie auf *Kotlin* basiert.

## Erkenntnisse

Wie die Beispiele [2.4 Implementierung der Anwendungsszenarien S.10] zeigen ist es mit wenig Aufwand möglich verschiedene komplexe Gebilde darzustellen. Dies wird erreicht, indem der Nutzer einer Liste Objekte hinzufügt, diese werden nach der Vollendung des Codes gerendert.

Ein wesentlicher Vorteil der eigenentwickelten Sprache ist, dass besondere Features sich leicht hinzufügen lassen, die sonst unüblich in normalen Programmiersprachen sind. Des Weiteren ist die Syntax leicht zu verstehen und komplexe System vor dem Nutzer abgekapselt. Ein Nachteil der Sprache *3Dcode* ist, dass sie ungefähr 938-mal langsamer ist als die Sprache, worauf *3D-Code* aufbaut. Dies ist durch mehrere Testdurchläufe Einens Priemzahlenermittlungsalgorithmus festgehalten. Siehe Anhang [A9 Performens Test [Kotlin vs 3D-Code] S.23]

In Anbetracht der oben genannten Punkte, wäre eine andere valide Option einer Entwicklung einer *Kotlin* Bibliothek, die ähnlich wie die entwickelte Sprache fungiert. Diese Option hätte zu dem jetzigen Entwicklungsstand noch Vorteile gegenüber der selbstentwickelten Sprache. Durch zusätzliche Features der Sprache *3D-Code* und weiteren Optionen könnte, könnten die Vorteile kompensiert werden. Siehe [3.3 Ausblick S.15]

## Ausblick

Damit die Sprache *3D Code* einen wesentlichen Vorteil gegenüber anderen Optionen [2.6 Vergleich zu Alternativen S.13] hätte, müsste sie in manchen Punkten ergänzt werden.

* Statische Funktionen / Klassen
* Vererbung/ Polymorphie
* Beleuchtung und Kollisionserkennung
* Injektion von eigenen Shaderprogrammen
* Überschreiben der Kamera Bewegung
* Umwandlung des *3D-Codes* in Maschinensprache, statt Evaluation in Kotlin
* Das manuelle Aktualisieren des Anzeigebildes, während der Code Ausführung
* Ausarbeitung der Bibliotheken
* Eine Entwicklungsumgebung

Quellenverzeichnis

1. Mechanik und Wärme

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.>

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. Github repository Link

<https://github.com/DennisGoss99/PraxProj_3DCode>

1. Testfälle

<https://github.com/DennisGoss99/PraxProj_3DCode/tree/main/3DCode/src/test/kotlin>

1. Codebeispiel Link [3 Sorting Algorithms]

<https://github.com/DennisGoss99/PraxProj_3DCode/blob/main/3DCode/code/Sort_App.3dc>

1. Bubblesortfunktion

Void BubbleSortStep(){

if(!finished){

if(sortPos < sortSize){

SortObj x = sortObjects.Get(sortPos)

SortObj x1 = sortObjects.Get(sortPos + 1)

//Swap Position

if(x.score > x1.score){

sortObjects.InsertAt(sortPos, x1)

sortObjects.InsertAt(sortPos + 1, x)

x.Move(Vector3f( 3.0, 0.0, 0.0))

x1.Move(Vector3f( -3.0, 0.0, 0.0))

}

sortPos += 1

}else{

if(sortSize <= 1){

finished = true

Println("Finished sorting: BubbleSort")

}

sortSize -= 1

sortPos = 0

}

}

}

1. Mergesortfunktion

Void MergeSortStep(){

if(!finished){

if(sortPos < sortSize){

SortObj x = sortObjects.Get(sortPos)

SortObj x1 = sortObjects.Get(sortPos + 1)

if(x.score > x1.score){

sortObjects.InsertAt(sortPos, x1)

sortObjects.InsertAt(sortPos + 1, x)

x.Move(Vector3f( 3.0, 0.0, 0.0))

x1.Move(Vector3f( -3.0, 0.0, 0.0))

}

sortPos += 2

for(Int i = 4 ; i <= elementCount ; i \*= 2){

if(m.Mod(sortPos,i) == 0){

Int pos1 = sortPos - 1

Int pos2 = (sortPos - (i/2)) - 1

Int pos1L = i / 2

Int pos2L = i / 2

Int count = i - 1

Array tempArray<SortObj> = Array<SortObj>(i)

SortObj x = sortObjects.Get(pos1)

SortObj x1 = sortObjects.Get(pos2)

while(pos1L >= 1 || pos2L >= 1){

if(pos1L >= 1){

x = sortObjects.Get(pos1)

}

if(pos2L >= 1){

x1 = sortObjects.Get(pos2)

}

if(pos1L >= 1 && (pos2L < 1 || x.score > x1.score)) tempArray.Set(count,x)

pos1 -= 1

pos1L -= 1

}else{

if(pos2L >= 1 && (pos1L < 1 || x.score <= x1.score)){

tempArray.Set(count,x1)

pos2 -= 1

pos2L -= 1

}

}

count -= 1

}

for(Int j = 0 ; j < tempArray.size ; j += 1){

SortObj sObj = tempArray.Get((tempArray.size - 1) - j)

sObj.SetMove( Vector3f((((sortPos - 1) - j) \* 3) -((elementCount / 2.0) \* 3), sObj.pos.y, sObj.pos.z))

sortObjects.InsertAt((sortPos - 1) - j, sObj)

}

}

}

}else{

finished = true

Println("Finished sorting: MergeSort")

}

}

}

1. Combosortfunktion

Bool combSortSortet = true

Void CombSortStep(){

if(!finished){

if(sortPos < sortSize){

SortObj x = sortObjects.Get(sortPos)

SortObj x1 = sortObjects.Get(sortPos + 1)

//Swap Position

if(x.score > x1.score){

sortObjects.InsertAt(sortPos, x1)

sortObjects.InsertAt(sortPos + 1, x)

x.Move(Vector3f( 3.0, 0.0, 0.0))

x1.Move(Vector3f( -3.0, 0.0, 0.0))

combSortSortet = false

sortPos += 1

}

sortPos += 1

}else{

if(combSortSortet){

finished = true

Println("Finished sorting: Comb Sort")

}

combSortSortet = true

sortPos = 0

}

}

}

1. Codebeispiel Link [Perlin Noise]

<https://github.com/DennisGoss99/PraxProj_3DCode/blob/main/3DCode/code/PerlinNoise_App.3dc>

1. Codebeispiel Link [Random Walk 3D]

<https://github.com/DennisGoss99/PraxProj_3DCode/blob/main/3DCode/code/RandomWalk_App.3dc>

1. Performens Test [Kotlin vs 3D-Code]

Um die beiden Sprachen miteinander zu vergleichen, werden Primzahlen in einem gewissen Bereich gesucht. Beide Sprachen erhalten ein Programm mit der gleichen Code Semantik.

Int Mod(Int n, Int k)  
{  
 while(n >= k){  
 n -= k  
 }  
 return n  
}  
   
Int Main()  
{  
 Int x = 2  
 Int i = 2  
 Bool quitFlag = true  
 Int foundPrimes = 0  
   
 while(x <= $limit)  
 {   
 while(i <= x && quitFlag)  
 {  
 if((Mod(x,i) == 0) && (x != i))  
 {  
 quitFlag = false // break  
 }  
 else  
 {  
 if(i == x)  
 {  
 foundPrimes += 1  
 }  
 }   
   
 i += 1   
 }  
 quitFlag = true  
 i = 2  
 x += 1  
 }   
 return foundPrimes  
}

|  |  |  |  |
| --- | --- | --- | --- |
| Bereich von 0 bis | Kotlin | 3D-Code | Faktor |
| 5000 | 10 ms | 9912 ms | 991x |
| 7500 | 22 ms | 22372 ms | 1016x |
| 10000 | 53 ms | 38952 ms | 734x |
| 15000 | 93 ms | 90467 ms | 972x |
| 20000 | 169 ms | 161270 ms | 954x |
| 25000 | 251 ms | 161270 ms | 954x |
| 30000 | 370 ms | 304294 ms | 822x |
| 40000 | 670 ms | 655132 ms | 977x |
| 50000 | 1092 ms | 1119378 ms | 1025x |

Die obere Tabelle zeigt das 3D-Code ungefähr im Durchschnitt 938-mal langsamer ist als Kotlin.

1. *Blender* ist eine Software um dreidimensionale Bühnenbilder/ Modelle zu modellieren und zu gestalten. [↑](#footnote-ref-1)
2. 1. [Q8 OpenGL Webseite

   Khronos Group. (o. D.). OpenGL - The Industry Standard for High Performance Graphics. Copyright (c) 2000 - 2021 Khronos Group. Abgerufen am 16. Dezember 2021, von https://www.opengl.org/

   Merge Sort Algorithm], S.19 [↑](#footnote-ref-2)
3. 1. [Q11 Matplotlib documentation

   Matplotlib development team. (o. D.). Matplotlib documentation — Matplotlib 3.5.1 documentation. Matplotlib 3.5.1 documentation. Abgerufen am 16. Dezember 2021, von https://matplotlib.org/stable/index.html

   1. OpenGL Wikipedia

   Seite „OpenGL“. In: Wikipedia – Die freie Enzyklopädie. Bearbeitungsstand: 8. November 2021, 14:54 UTC. URL: https://de.wikipedia.org/w/index.php?title=OpenGL&oldid=217099756 (Abgerufen: 16. Dezember 2021, 22:44 UTC)

   Perlin-Noise Wikipedia], siehe zusätzlich [Q1 Perlin Noise Web.Archive.Org] [↑](#footnote-ref-3)
4. Zugehöriger Code: [A8 Codebeispiel Link [Random Walk 3D] S.21] [↑](#footnote-ref-4)
5. 1. [Q11 Matplotlib documentation

   Matplotlib development team. (o. D.). Matplotlib documentation — Matplotlib 3.5.1 documentation. Matplotlib 3.5.1 documentation. Abgerufen am 16. Dezember 2021, von https://matplotlib.org/stable/index.html

   OpenGL Wikipedia S.16] siehe auch [Q8 OpenGL Webseite S.15] [↑](#footnote-ref-5)
6. [Q11 Matplotlib documentation S.16] [↑](#footnote-ref-6)