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](#_Toc105068421)

[Abbildungsverzeichnis E](#_Toc105068422)

[Tabellenverzeichnis F](#_Toc105068423)

[1 Abstract 1](#_Toc105068424)

[2 Einleitung 2](#_Toc105068425)

[2.1 Relevanz 2](#_Toc105068426)

[2.2 Zielsetzung 2](#_Toc105068427)

[2.3 Recherchephase 3](#_Toc105068428)

[2.4 Planung 3](#_Toc105068429)

[2.4.1 Definitionsphase 3](#_Toc105068430)

[2.4.2 Definition der Architektur 5](#_Toc105068431)

[2.4.3 Planung der Thesenüberprüfung 5](#_Toc105068432)

[2.4.4 Testumgebung 5](#_Toc105068433)

[2.5 Grundaufbau der Anwendung 5](#_Toc105068434)

[2.5.1 Gravitation 6](#_Toc105068435)

[2.5.2 Kollisionen 6](#_Toc105068436)

[2.6 Benutzeroberfläche 7](#_Toc105068437)

[2.6.1 Elemente Auflistung 7](#_Toc105068438)

[2.6.2 Schrift 7](#_Toc105068439)

[2.7 Benutzeroberflächendesign 8](#_Toc105068440)

[2.8 Parallelisierung 9](#_Toc105068441)

[2.9 Ablauf der Applikation 9](#_Toc105068442)

[3 Hauptteil 10](#_Toc105068443)

[3.1 Parallels iterieren auf einer Liste 10](#_Toc105068444)

[3.1.1 Herangehensweise 10](#_Toc105068445)

[3.1.2 Funktionsaufbau 11](#_Toc105068446)

[3.2 Sweep-and-prune Algorithmus 11](#_Toc105068447)

[3.2.1 Datenstruktur 11](#_Toc105068448)

[3.2.2 Sequenzielle Kollisionserkennung 13](#_Toc105068449)

[3.2.3 Parallele Kollisionserkennung 14](#_Toc105068450)

[3.2.4 Veränderung der Objektpositionen 14](#_Toc105068451)

[3.2.5 Sequenzielle Teststruktur 15](#_Toc105068452)

[3.2.6 Parallele Teststruktur 15](#_Toc105068453)

[3.3 Das Gravitationssystem 16](#_Toc105068454)

[3.3.1 Gravitationsobjekt 16](#_Toc105068455)

[3.3.2 Gravitationsmanager 16](#_Toc105068456)

[3.3.3 Anwenden der Gravitation 17](#_Toc105068457)

[3.3.4 Parallelisierung 18](#_Toc105068458)

[3.3.5 Teststruktur 18](#_Toc105068459)

[3.4 OpenGL Rendering 20](#_Toc105068460)

[3.4.1 Simpler Renderprozess 20](#_Toc105068461)

[3.4.2 Renderprozess selbes Modell 21](#_Toc105068462)

[3.4.3 Instancing 21](#_Toc105068463)

[3.4.4 Testdurchführung 22](#_Toc105068464)

[3.5 Benutzeroberfläche 23](#_Toc105068465)

[3.5.1 Grundelement 23](#_Toc105068466)

[3.5.2 Einschränkungen der Größe und Position 23](#_Toc105068467)

[3.5.3 Schachtelung von Elementen 23](#_Toc105068468)

[3.5.4 Schrift 24](#_Toc105068469)

[3.5.5 Interaktion 25](#_Toc105068470)

[3.6 Analyse 26](#_Toc105068471)

[3.6.1 Einstellungen 26](#_Toc105068472)

[3.6.2 Konfigurationsdatei 26](#_Toc105068473)

[3.6.3 Testsysteme 26](#_Toc105068474)

[4 Fazit 27](#_Toc105068475)

[4.1 Ergebnisse 27](#_Toc105068476)

[4.2 Erkenntnisse 27](#_Toc105068477)

[4.3 Ausblick 27](#_Toc105068478)

[Quellenverzeichnis I](#_Toc105068479)

[Q1 Projekt OuterSpace I](#_Toc105068480)

[Q2 Demtröder2006\_Book\_Experimentalphysik1 I](#_Toc105068481)

[Q3 Mathematics of Satellite Motion I](#_Toc105068482)

[Q4 JUnit I](#_Toc105068483)

[Q5 Lightweight Java Game Library I](#_Toc105068484)

[Q6 42 Years of Microprocessor Trend Data I](#_Toc105068485)

[Q7 Sweep and prune II](#_Toc105068486)

[Q8 Font Rendering II](#_Toc105068487)

[Anhang III](#_Toc105068488)

[A1 Sweep-and-prune Testfälle III](#_Toc105068489)

[A2 Gravitationssystem Testfälle IV](#_Toc105068490)

[A3 Testsysteme VIII](#_Toc105068491)

Abbildungsverzeichnis

[Abbildung 1: [Virtueller Prototyp der Einstellungsoberfläche] 8](#_Toc105068492)

[Abbildung 2: [Listenzerteilung zur Parallelisierung] 10](#_Toc105068493)

[Abbildung 3: [Erweiterungsfunktion foreachParallel] 11](#_Toc105068494)

[Abbildung 4: [Klassendefinition SAP ohne Funktionen] 12](#_Toc105068495)

[Abbildung 5: [Klassendefinition EndPoint] 12](#_Toc105068496)

[Abbildung 6: [Schnittstelle IHitbox] 12](#_Toc105068497)

[Abbildung 7: [SAP-Kollisionsüberprüfung der X-Achse] 13](#_Toc105068498)

[Abbildung 8: [SAP-Kollisionsüberprüfung der Y /Z-Achse] 14](#_Toc105068499)

[Abbildung 9: [Schnittstelle IGravity] 16](#_Toc105068500)

[Abbildung 10: [Enum GravityProperties] 17](#_Toc105068501)

[Abbildung 11: [Aufbau der Gravitationsfunktion] 18](#_Toc105068502)

[Abbildung 12: [Testskript Renderverfahren] 22](#_Toc105068503)

[Abbildung 13: [Schachtelung von UI-Elementen] 24](#_Toc105068504)

[Abbildung 14: [Zugehöriger Code von Abbildung 11] 24](#_Toc105068505)

Tabellenverzeichnis

[Tabelle 1: [Leistungstest 1: redundante Objekte] 20](#_Toc105068506)

[Tabelle 2: [Leistungstest 2: gleiches Modell] 21](#_Toc105068507)

[Tabelle 3: [Leistungstest 3: Instancing] 22](#_Toc105068508)

# 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 weitestgehend 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.

## 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 [2.5 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 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. [2.7 Benutzeroberflächendesign S.8]

### 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. [2.4.5 Testumgebung S.4]

Die Projektdurchführung ist nochmals detaillierter im Abschnitt [3 Hauptteil S.10] beschrieben.

### 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. 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 Signifikaten Vorteil bietet.

## Testumgebung

Um die verschiedenen Algorithmen 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.

## 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

…

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

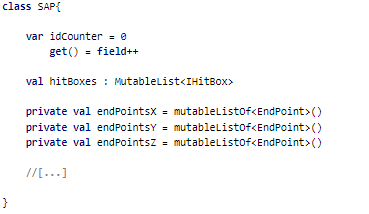


Abbildung : [Klassendefinition SAP ohne Funktionen]

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.

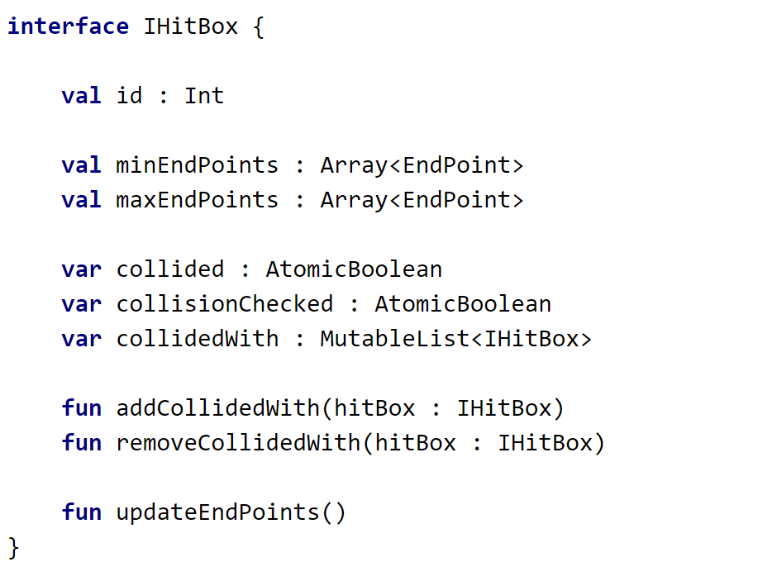


Abbildung : [Schnittstelle IHitbox]

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.10].

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.11] 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 : [Schnittstelle IGravity]

### Gravitationsmanager

Der *GravityObjectContainer* beinhaltet zwei Listen von *IGravity* Objekten. Die erste Liste *L1* beinhaltet alle Objekte, die der Gravitation ausgesetzt sind. Die zweite Liste *L2* 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 : [Enum GravityProperties]

### Anwenden der Gravitation

Wie schon im Abschnitt [3.3 Das Gravitationssystem S.16] 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 *obj1* in der List *L1* auf jedes andere Objekt *obj2* in der List *L2* angewendet.

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

### Parallelisierung

Die Parallelisierung in diesem Modul wird realisiert, indem die *ForeachParallel* Erweiterungsfunktion [3.1 Parallels iterieren auf einer Liste S.10] 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 : [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].

## OpenGL Rendering

Die Applikation rendert jeden *Frame* alle Objekte neu. In OpenGL gibt es verschiedene Möglichkeiten die Daten an die Grafikkarte zu übermitteln. Unter den Daten befinden sich zum Beispiel die Transformationsmatrix[[4]](#footnote-4), die Farbe und weitere Eigenschaften, die die Grafikkarte für den Zeichnungsprozess benötigen könnte. Dieses 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. Dieser Prozess ist zwar ohne großen Aufwand zu implementieren, aber sehr schlecht skalierbar, da sehr viele redundante Daten anfallen.

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

Tabelle : [Leistungstest 1: redundante Objekte]

### Renderprozess selbes Modell

Um die Redundanten Daten und hochlade Prozesse zu vermeiden kann es Sinnvoll sein, dass Modell nur einmal hochzuladen und vor jedem rendern einmal zu aktivieren. Dabei müssen alle Objekte, die dasselbe Modell benutzen hintereinander dargestellt werden, da das Verändern des ausgewählten Modells auf der Grafikkarte auch zu Verzögerungen führt. 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 Rechteck mit verschieden Eigenschaften verwendet wird. [3.5 Benutzeroberfläche S.23]

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

Um eine Vielzahl von Objekten darzustellen, welche oftmals sehr ähnliche Eigenschaften besitzen eignet sich das *Instancing*. Dieses Verfahren benutzt ähnlich wie Abschnitt [LINK] nur ein Modell. Das Besondere am *Instancing* ist, dass OpenGL von der CPU aus nur ein *redercall* übermittelt wird. 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 Rederprozess 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 für besonders für sehr 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 12 S.22] ausgewertet. Das Testskript beschreibt die jeweils 17 Testdurchläufe. Bei jedem durchlauf ist die Objektanzahl jeweils erhört. Vor dem Wechseln zu einem anderen Verfahren ist der genutzte Rechner neugestartet und von jeglichen nicht genutzten IO-Geräten getrennt wurden. Alle Leistungstests wurden mittels demselben 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 [3.5.1 Grundelement S.23] 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 13 S.24] zeigt die Visualisierung vom Code aus Abbildung [Abbildung 14 S.24]. 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 [Q8 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. 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>

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 | i5-4460@ 3.20GHz | 4 | 4 | AMD RX 5700 | WIN-H 10 x64 | 16 GB  1600MHz | 8 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. Die Transformationsmatrix beinhaltet die Skalierung, Verschiebung und die Rotation. [↑](#footnote-ref-4)