



Technischer Bericht: 2DGameSDK

Bachelorthesis: A. Markóczy

| | |
|---------------|-------------------------|
| Studiengang: | Bsc. Informatik |
| Autor: | Alain Alistair Markóczy |
| Betreuer: | Dr. Urs Künzler |
| Auftraggeber: | Student |
| Datum: | 16.06.2019 |

Dieses Dokument ist für den Druck in Farbe optimiert, der Druck dieses Dokuments in Graustufen ist nicht empfohlen.

Versionskontrolle

| Version | Datum | Beschreibung | Autor |
|---------|------------|--------------------------|-------|
| 0.1 | 09.12.2019 | Outline erste Version | A.M. |
| 0.2 | 20.12.2019 | Design | A.M. |
| 0.3 | 02.01.2020 | Anforderungen | A.M. |
| 0.4 | 05.01.2020 | Marktanalyse | A.M. |
| 0.5 | 12.01.2020 | Physik Engine Evaluation | A.M. |
| 0.6 | 13.01.2020 | Revision | A.M. |
| 1.0 | 15.01.2020 | Letzte Revision | A.M. |

Management Summary

Das Ziel der vorliegenden Bachelorthesis war es, eine Game Engine für 2D Computerspiele in der Programmiersprache C++ zu erstellen. Die Game Engine ermöglicht es, 2D Spiele der unterschiedlichsten Kategorien und Genres für die Betriebssysteme Windows, Linux und Mac OS zu entwickeln. Das Produkt der Arbeit ist letztendlich ein Software Development Kit (SDK) in Form einer C++ Library, das vorgefertigte und erweiterbare Lösungen für die vielseitigen Problemstellungen der 2D Spieleentwicklung zur Verfügung stellt. Das Produkt wird unter einer Open Source Lizenz (MIT Lizenz) bereitgestellt und richtet sich an Spielentwickler, die eine einfache, vielseitige und gut erweiterbare Game Engine als Grundlage ihres 2D Computerspiels verwenden möchten. Die Arbeit baut auf den Leistungen eines Vorprojekts auf, in dem im Rahmen einer Projektarbeit bereits einige Features der Game Engine entwickelt wurden.

Im Rahmen dieser Thesis wurden verschiedene Varianten für die Simulation einer physikalischen Spielumgebung mit einander verglichen, mit dem Ergebnis, dass die Simulation auf Basis der Physik Engine Chipmunk Physics am besten für das Projekt geeignet ist. Die Game Engine, welche im Rahmen dieser Thesis erarbeitet wurde, stellt unter anderem eine vollumfänglich simulierte physikalische Umgebung auf Basis der Chipmunk Physik Engine bereit. Weitere Features der Game Engine sind unter anderem eine Scene Graph Datenstruktur zur Verwaltung von Spielobjekten, eine Tilemap Datenstruktur für die Definition einer statischen Spielwelt und einen Event Controlling Mechanismus zur Definition und Verarbeitung von Spielereignissen.

Zusätzlich zur Game Engine selbst, wurden im Rahmen dieser Bachelorthesis zwei unterschiedliche Prototypen von funktionsfähigen Computerspielen auf Basis der Game Engine implementiert. Die Prototypen zeigen unter anderem, dass es möglich ist, Spiele für unterschiedliche Spiel Genres zu erstellen und dienen auch als Referenzbeispiele für den praktischen Umgang mit der Game Engine.

Inhaltsverzeichnis

| | |
|-----------------------------------------------|----|
| 1 Einleitung | 5 |
| 1.1 Zweck des Dokuments | 5 |
| 1.2 Daten und Dokumentation | 5 |
| 2 Ausgangslage | 5 |
| 2.1 Produktbeschreibung | 5 |
| 2.2 Produktübersicht | 5 |
| 2.3 Projektziele | 6 |
| 3 Umsetzung | 6 |
| 3.1 Anforderungsspezifikation | 6 |
| 3.2 Qualitätssicherung | 6 |
| 4 Marktanalyse | 7 |
| 4.1 Game Engines auf dem Markt | 7 |
| 4.2 Marktpositionierung 2DGameSDK | 13 |
| 4.3 Distributionskonzept | 13 |
| 5 Evaluation: Physik Engine | 14 |
| 5.1 Ausgangslage | 14 |
| 5.2 Varianten | 14 |
| 5.3 Evaluationskriterien | 15 |
| 5.4 Benchmarking | 15 |
| 5.5 Evaluation | 17 |
| 6 Design | 20 |
| 6.1 UML Kontextdiagramm | 20 |
| 6.2 UML Paketdiagramm | 21 |
| 6.3 UML Klassendiagramm: Global | 22 |
| 6.4 Design Patterns | 24 |
| 6.5 Globale Konzepte | 25 |
| 7 Implementation | 28 |
| 7.1 Technologiewahl | 28 |
| 7.2 Externe Komponenten | 28 |
| 7.3 Umsetzung | 29 |
| 7.4 Testing | 40 |
| 8 Projektmanagement | 45 |
| 8.1 Zeitplan | 45 |
| 8.2 Anforderungen | 47 |
| 9 Schlussfolgerungen/Fazit | 53 |
| 9.1 Ergebnisse | 53 |
| 9.2 Herausforderungen | 53 |
| 9.3 Ausblick | 53 |
| 10 Referenzen | 54 |
| 11 Glossar | 55 |
| 12 Abbildungsverzeichnis | 56 |
| 13 Tabellenverzeichnis | 57 |
| 14 Anhang | 58 |
| 14.1 Anhang A1: Benchmarking Chipmunk Physics | 58 |
| 14.2 Anhang A2: Benchmarking Box2D Physics | 59 |
| 15 Selbstständigkeitserklärung | 60 |

1 Einleitung

1.1 Zweck des Dokuments

Dieses Dokument ist die technische Dokumentation zur Applikation, die im Rahmen der Bachelorthesis «2DGameSDK» realisiert wurde, sie beschreibt zum Einen die Vorgehensweise (Zeitplan, Variantenstudien etc.) die zur Implementation der Lösung angewendet wurde, zum anderen die spezifischen Implementationsdetails des fertigen Produkts mit den entsprechenden UML Diagrammen.

1.2 Daten und Dokumentation

- **Offizielle SDK Dokumentation:** <https://markoczy.github.io/2DGameSDK/>
- **Git Repository:** www.Github.com/markoczy/2DGameSDK
- **Dokumente:** <repository>/docs/Thesis
- **Source Code:** <repository>/src
- **Source Code SDK:** <repository>/Project_2DGameSDK/src
- **Source Code Prototyp:** <repository>/Project_Prototype/src
- **3. Partei Bibliotheken:** <repository>/libs

2 Ausgangslage

2.1 Produktbeschreibung

Ziel der Arbeit ist es, ein Software Development Kit für unterschiedliche Arten von 2D Computerspielen zu erstellen. Zur Unterstützung wird die Library SFML [1] verwendet, die es ermöglicht, Multi Plattform fähige Applikationen zu erstellen, wobei die Darstellung von texturierten Elementen auf dem Bildschirm (mit Zoom, Screen Offset etc.) bereits gelöst ist. Der Fokus liegt daher auf der Implementation einer Gamelogik, die individuell angepasst werden kann, sodass unterschiedliche Arten von 2D Games (Shooter, Adventure, Roleplay Game) mit wenig Aufwand erstellt werden können. Im Rahmen dieser Thesis wurde zusätzlich eine Marktanalyse durchgeführt, bei der die meistverwendeten Game Engines mit einander verglichen wurden. Ausgehend von dieser Analyse, wurde die mögliche Marktpositionierung des Endprodukts bestimmt.

2.2 Produktübersicht

Das Produkt ist auf den Definitionen der Frameworks SFML [1] und Chipmunk Physics [2] aufgebaut und erweitert diese mit zusätzlichen Definitionen. Im Rahmen dieser Bachelorthesis steht das zu erstellende Produkt zwischen den Low Level Frameworks (SFML und Chipmunk Physics) und dem Computerspiel, das der jeweilige Spieleentwickler erstellen möchte:

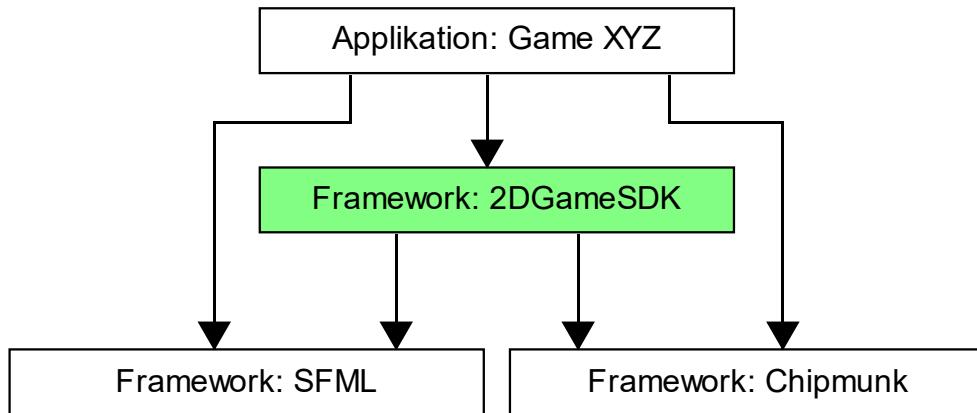


Abbildung 1 Produktübersicht 2DGameSDK

2.3 Projektziele

2.3.1 Anwender / Spiele Entwickler

Im Kontext dieses Projekts ist der Anwender ein Spieleentwickler, der das SDK verwendet, um ein beliebiges 2D Spiel zu erstellen. Diesem Anwender soll eine Sammlung von Entwicklertools in Form einer C++ Library bereitgestellt werden, wodurch der Entwicklungsaufwand für die Erstellung eines 2D Computerspiels massgeblich verringert werden kann. Die Tools sollen den unterschiedlichsten Ansprüchen von unterschiedlichen Spielkonzepten und -Ideen gerecht werden, um einen möglichst hohen Nutzwert für den Spieleentwickler darzustellen.

2.3.2 Code Qualität und –Wartbarkeit

Der Entwickler des Produkts möchte den Source Code so implementieren, dass das Produkt über einen längeren Zeitraum mit geringem Aufwand gewartet und erweitert werden kann. Um dies zu ermöglichen, sollen die für das Projekt gültigen Coding Standards (definiert in [3]) strikt eingehalten werden. Zusätzlich soll für die Wahl der Softwarearchitektur wann immer möglich auf gängige Design Patterns der Industrie (gemäss [4] und [5]) zurückgegriffen werden.

3 Umsetzung

3.1 Anforderungsspezifikation

Die funktionalen, technischen und qualitativen Anforderungen, die für dieses Projekt gelten, sind im Pflichtenheft des Projekts [5] spezifiziert. Zusätzlich enthält das Pflichtenheft die definierten Randbedingungen, die bei der Implementation dieses Projektes gelten.

3.2 Qualitätssicherung

Für die Qualitätssicherung werden die im Pflichtenheft [5] genannten Coding Standards angewendet. Die Verifikation der Software erfolgt anhand der Test Szenarien, die im Pflichtenheft spezifiziert sind.

4 Marktanalyse

4.1 Game Engines auf dem Markt

Die folgende Statistik basiert auf einer Umfrage des Wirtschaftsverbands für Spielehersteller TIGA [6] im Jahr 2014 und zeigt welche Game Engines in diesem Jahr bei den verschiedenen Spielehersteller in Grossbritannien zum Einsatz kamen. Die meisten dieser Game Engines sind zwar hauptsächlich auf die Erstellung von 3D Spielen ausgerichtet, sie enthalten aber auch ein 2D Modul mit dem 2D Spiele erstellt werden können:

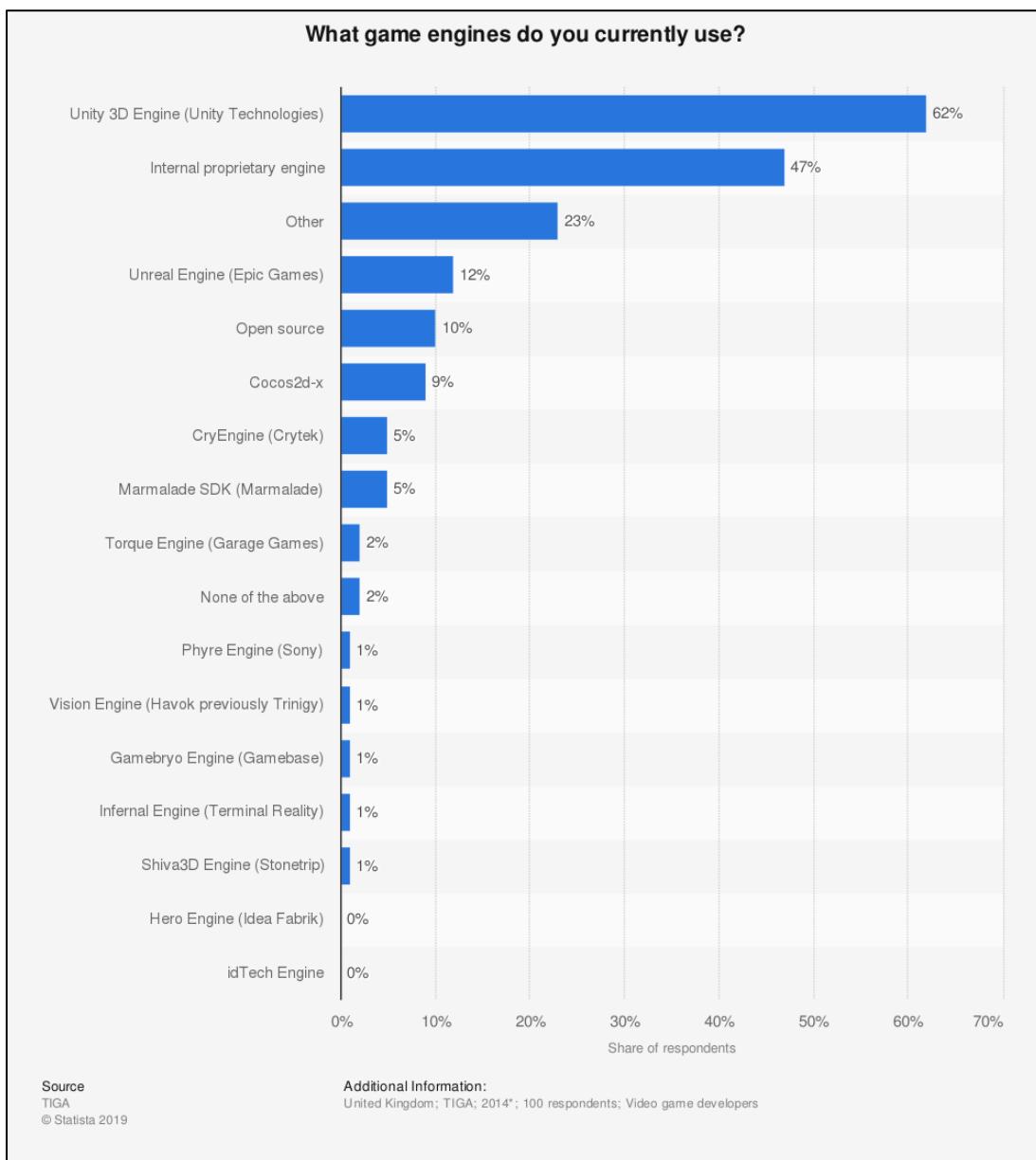


Abbildung 2 Meistverwendete Game Engines in Grossbritannien im Jahr 2014 (Umfrage von: [6], Quelle: [19])

Die Statistik ist zwar eine gute Referenz um die Verhältnisse zwischen den etablierten Game Engines abzuschätzen, sie ist aber nicht genau auf die heutige Zeit übertragbar, da sie aus dem Jahr 2014 stammt und nur auf Firmen in Grossbritannien beschränkt ist. Der Einfluss der Godot Game Engine, [7] die im Jahr 2014 herausgegeben wurde und als Open Source Engine hohe Beliebtheit erfährt, ist zum Beispiel in dieser Statistik nicht abgebildet.

Die Statistik zeigt, dass die kommerziellen Game Engines Unity [8] und die Unreal Engine (auch Unreal Development Kit, UDK) [9] mit Abstand zu den beliebtesten Game Engines gehören. Auf dem zweiten Platz sind die internen Eigenlösungen der Spielehersteller, wie z. B. die Rockstar Advanced Game Engine RAGE, die bei Rockstart North (Schottland) zum Einsatz kam. Die Open Source Game Engines (Aufgeführt unter «Open Source», «Cocos2d-x» und «Torque Engine») besitzen zu dieser Zeit ca. 21 % des Marktanteils. Im folgenden Teil werden die wichtigsten freien und kommerziellen Game Engines beschrieben:

4.1.1 Unity

Die Unity Game Engine ist eine kommerzielle Game Engine der Firma Unity Technologies. Unity ist gemäss Abbildung 2 die am meisten eingesetzte Game Engine. Die Unity Game Engine wird unter den folgenden Lizenzierungsmodellen verbreitet (aus [8]):

- **Personal:** Unity Lizenz ohne Kosten. Für Spielentwicklerfirmen, die weniger als 100'000 US Dollar pro Jahr verdienen.
- **Plus:** Unity Lizenz mit einer Gebühr von 40 US Dollar pro Anwender und Monat. Für Spielentwicklerfirmen, die zwischen 100'000 und 200'000 US Dollar im Jahr verdienen.
- **Pro:** Unity Lizenz mit einer Gebühr von 150 US Dollar pro Monat und Anwender. Diese Lizenz wird benötigt, wenn das jährliche Einkommen der Spielentwicklerfirma über 200'000 US Dollar liegt.

Bei allen Lizenzierungsmodellen gilt, dass keine zusätzlichen Lizenzkosten beim Vertrieb des Produkts anfallen (Royalty-Free Konzept). Die Unity Game Engine bietet die folgenden Kernfunktionalitäten (aus [10]):

- **Real-Time 2D und 3D Applikationen:** Mit Unity können sowohl 2D als auch 3D Computerspiele und Simulationen erstellt werden.
- **Editor:** Die Unity Game Engine stellt eine visuelle Benutzerschnittstelle zum Erstellen und Bearbeiten von Spielen bereit. Ein Editor beinhaltet unter anderem einen Level Editor mit dem die Spielwelten erstellt werden können, einen Asset Manager in dem die Assets des Spiels (3D Modelle, Materialien, Scripts) verwaltet werden können und einen visuellen Scene Graph mit dem die Objektinstanzen in einer hierarchischen Ansicht verwaltet werden können.
- **Multiplattform:** Die Unity Game Engine ermöglicht es, Spiele für über 25 unterschiedliche Plattformen zu erstellen, darunter sind Computer (Windows, Mac OS, Linux), Spielkonsolen (Nintendo Switch, Playstation 4, Xbox One), Smartphones (Android, iOS), Smart TV (Android TV), und VR Geräte (Samsung Gear, Valve Index). Unity ermöglicht auch die Bereitstellung von Spielen auf WebGL, wodurch ermöglicht wird, dass die Spiele in einem HTML 5 fähigen Browser laufen.
- **State of the Art Rendering:** Die Unity Game Engine stellt eine Vielzahl von Rendering Features auf dem Stand der Technik in der sogenannten «High Definition Rendering Pipeline» (HDRP) bereit. Dies ermöglicht es, Computerspiele mit einem hohen grafischen Standard zu erstellen.
- **Asset Store:** Die Unity Game Engine stellt im eigenen Editor einen Asset Store zur Verfügung, mit dem es möglich ist anderen Unity Entwickler verschiedene Assets (3D Modelle, Scripts, Materialien etc.) auf einem Online Markt zum Verkauf (oder gratis) anzubieten. Beim Entwickeln eines Spiels, kann ein Spielehersteller in diesem Asset Store nach geeigneten Assets suchen und diese einfach und bequem ins aktuelle Spiel einbauen.
- **Scripting:** Die Unity Game Engine bietet die Möglichkeit mit der Programmiersprache C# eigene Scripts für das Spiel und die Objekte im Spiel zu schreiben.

Die Unity Game Engine ist eine performante Game Engine, die viele nützliche Hilfsmittel für die Entwicklung von grafisch hochwertigen 2D und 3D Spielen zu einem annehmbaren Preis zur Verfügung stellt. Sie ist auch gut für kleinere Entwicklungsbüros geeignet, weil die kostenpflichtige Lizenz erst ab einem bestimmten Einkommen gemietet werden muss.

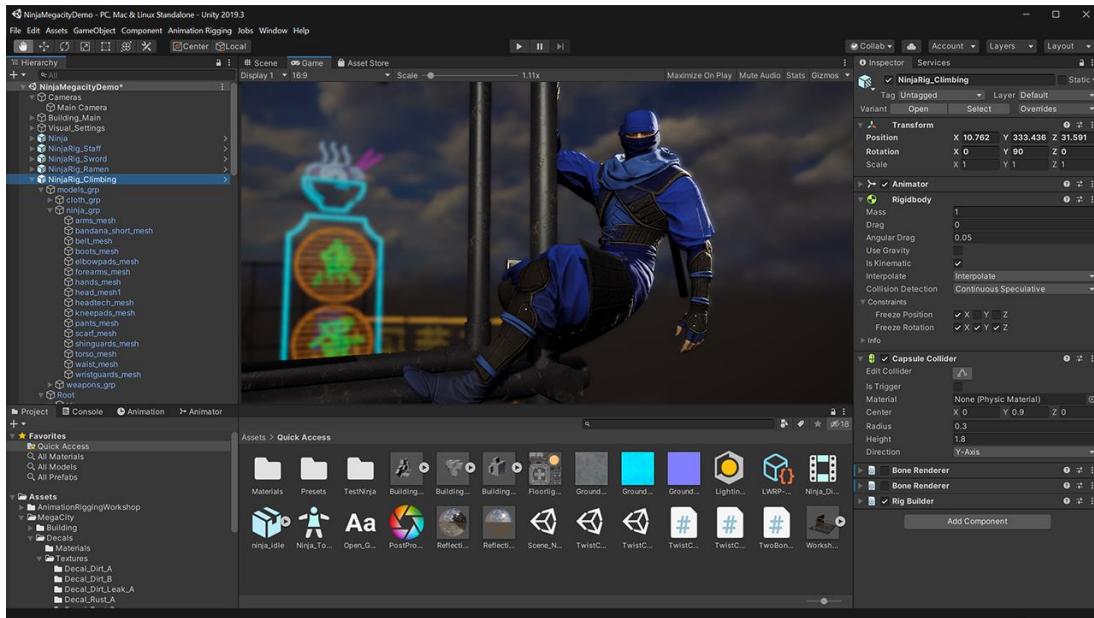


Abbildung 3 Userinterface der Unity Game Engine (aus [8])

4.1.2 UDK (Unreal Development Kit)

Das Unreal Development Kit ist eine kommerzielle Game Engine der Firma Epic Games. Die UDK Game Engine kommt zwar in der Häufigkeit weniger oft zum Einsatz als die Unity Game Engine, wurde aber in der Vergangenheit von grossen Entwicklungsfirmen für die Herstellung von bekannten Spielen (wie z. B. der Devil May Cry Serie von Capcom oder der Borderlands Serie von Gearbox) verwendet. Die UDK Game Engine wird unter den folgenden Lizenzmodellen verbreitet (aus [9]):

- **Frei:** UDK Lizenz ohne Kosten. Darf nur für die Herstellung von nicht-kommerziellen Applikationen verwendet werden.
- **Kommerziell:** UDK Lizenz mit einer initialen (einmaligen) Lizenzgebühr von 99 US Dollar. Bei einem jährlichen Einkommen von über 50'000 US Dollar mit UDK basierten Applikationen, müssen 25% des Einkommens des Vertriebs an den Lizenzgeber Epic Games gezahlt werden (Royalty Konzept).

Das Lizenzierungsmodell der UDK Game Engine unterscheidet sich grundlegend vom Lizenzierungsmodell der Unity Game Engine: Während die Unity Game Engine immer mit einem Fixpreis arbeitet, unabhängig davon ob mit einem bestimmten Unity Produkt bereits Umsatz generiert wurde, (das Lizenzmodell hängt vom Gesamteinkommen der Firma, nicht vom einzelnen Produkt ab) fallen bei der UDK Game Engine erst dann grosse Kosten an, wenn mit einer UDK basierten Applikation ein Umsatz generiert wird. 25 % des gesamten Umsatzes kann zwar bei beliebten Spielen eine grosse Summe bedeuten, dem ist aber die lange Entwicklungszeit eines solchen Spiels gegenüberzustellen, während der man für die Unity Game Engine bereits vor dem Release Lizenzgebühren zahlen würde. Die UDK Game Engine bietet die folgenden Kernfunktionalitäten:

- **Real-Time 2D und 3D Applikationen:** Mit der UDK Game Engine können sowohl 2D als auch 3D Computerspiele und Simulationen erstellt werden.
- **Editor:** Die UDK Game Engine stellt analog zu Unity Game Engine eine visuelle Benutzerschnittstelle zum Erstellen und Bearbeiten von Spielen bereit.

- **Multiplattform:** Die UDK Game Engine ermöglicht es, Spiele für eine Vielzahl unterschiedlicher Plattformen zu erstellen, darunter sind Computer (Windows, Mac OS, Linux), Spielkonsolen (Nintendo Switch, Playstation 4, Xbox One), Smartphones (Android, iOS) und VR Geräte (Samsung Gear, Valve Index). UDK ermöglicht auch die Bereitstellung von Spielen mit WebGL, wodurch ermöglicht wird, dass die Spiele in einem HTML 5 fähigen Browser laufen.
- **State of the Art Rendering:** Die UDK Game Engine stellt eine Vielzahl von Rendering Features auf dem Stand der Technik bereit. Die UDK Game Engine ist bekannt für hochwertige Spieldesigns mit einem hohen grafischen Standard. Die UDK Game Engine stellt auch eine Rendering Technologie bereit, die speziell für VR Technologien entwickelt wurde (bekannt als «Forward Rendering»).
- **Marketplace:** Die UDK Game Engine stellt im eigenen Editor einen online Markt zur Verfügung, mit dem es möglich ist anderen UDK Entwickler verschiedene Assets (3D Modelle, Scripts, Materialien etc.) zum Verkauf (oder gratis) anzubieten. Beim Entwickeln eines Spiels, kann ein Spielehersteller in diesem Marketplace nach geeigneten Assets suchen und diese einfach und bequem ins aktuelle Spiel einbauen.
- **Scripting:** Die UDK Game Engine bietet die Möglichkeit mit der von Epic entwickelten Programmiersprache «Unreal Script» eigene Scripts für das Spiel und die Objekte im Spiel zu schreiben.

Die UDK Game Engine stellt viele nützliche Tools für die Erstellung von hochwertigen 3D Spielen bereit. 2D Spiele sind zwar nicht das Kerngeschäft der Game Engine, werden aber vollumfänglich unterstützt. Die Game Engine ist im produktiven Einsatz gut erprobt, da bereits viele hochwertige Spieldesigns mit der Engine erstellt wurden. Das Finanzierungsmodell der Game Engine kann etwas abschreckend wirken, da es de facto keine obere Grenze für die Summe der Lizenzkosten gibt.



Abbildung 4 Userinterface der UDK Game Engine (aus [9])

4.1.3 Godot Engine

Die Godot Game Engine ist eine Open Source Game Engine, die von einem unabhängigen Entwicklerteam bereitgestellt und gewartet wird. Die Godot Engine wurde im Jahr 2014 veröffentlicht und ist daher relativ jung im Vergleich zu Unity oder UDK. Die Godot Game Engine wird unter der MIT Lizenz verbreitet, diese permissive Open Source Lizenz unterliegt keinen Copyleft Restriktionen und erlaubt es absolut kostenfrei sowohl freie als auch kommerzielle Applikationen mit der Engine zu erstellen. Die Godot Game Engine bietet die folgenden Kernfunktionalitäten:

- **Real-Time 2D und 3D Applikationen:** Mit der Godot Game Engine können sowohl 2D als auch 3D Computerspiele und Simulationen erstellt werden.
- **Editor:** Die Godot Game Engine stellt analog zur Unity Engine und UDK Engine eine visuelle Benutzerschnittstelle zum Erstellen und Bearbeiten von Spielen bereit.
- **Multiplattform:** Die Godot Game Engine ermöglicht es, Spiele für verschiedene Plattformen zu erstellen, darunter sind Computer (Windows, MacOS, Linux) und Smartphones (Android, iOS). Die Godot Engine ermöglicht auch die Bereitstellung von Spielen mit WebGL, wodurch ermöglicht wird, dass die Spiele in einem HTML 5 fähigen Browser laufen.
- **Hochwertiges Rendering:** Die Godot Game Engine stellt eine Vielzahl von hochwertigen Rendering Features bereit. Die Rendering Technologie der Godot Engine ist zwar nicht so fortschrittlich, wie die der Game Engines Unity oder UDK, unterstützt aber die wesentlichen Features, um schöne 2D und 3D Spiele zu erstellen.
- **Asset Library:** Die Godot Game Engine stellt im eigenen Editor einen online Marktplatz zur Verfügung, mit dem es möglich ist anderen Godot Entwickler verschiedene Assets (3D Modelle, Scripts, Materialien etc.) zum Verkauf (oder gratis) anzubieten. Beim Entwickeln eines Spiels kann ein Spielehersteller in dieser Marktplatz nach geeigneten Assets suchen und diese einfach und bequem ins aktuelle Spiel einbauen.
- **Scripting:** Die Godot Game Engine bietet die Möglichkeit mit der vom Godot Entwicklerteam entwickelten Programmiersprache «GDScript» eigene Scripts für das Spiel und die Objekte im Spiel zu schreiben.

Die Godot Game Engine ist eine robuste Game Engine, die sich in den wesentlichen Punkten auch mit den proprietären Game Engines wie Unity oder UDK messen kann. Da die Godot Engine gratis verfügbar ist, ist sie sehr attraktiv für Spielentwickler, die mit einem geringen Kapital ein Spiel herstellen möchten.

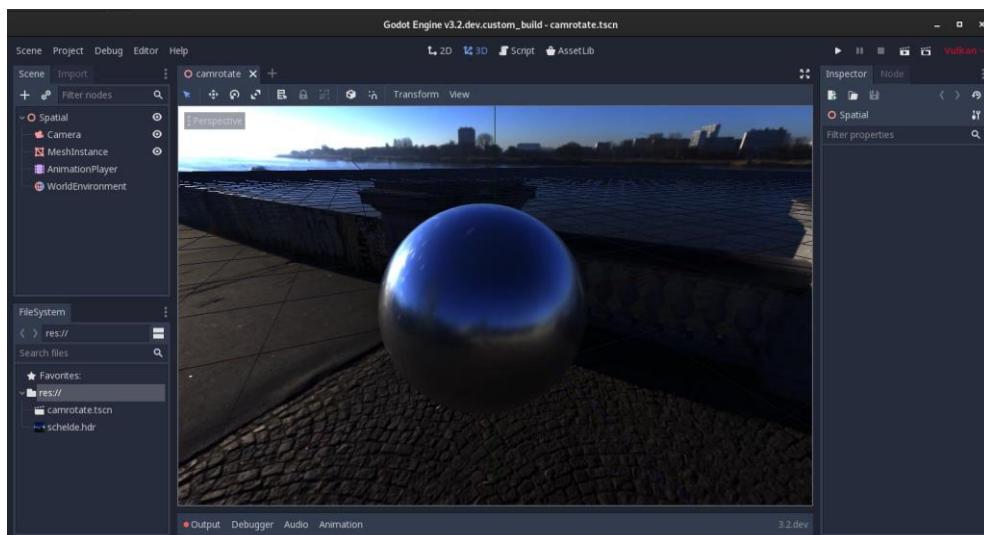


Abbildung 5 Userinterface der Godot Engine (aus [7])

4.1.4 Cocos 2D-X

Die Cocos 2D-X Game Engine [11] ist eine Open Source Game Engine, die von der Spielehersteller Firma Chuckong Technologies gewartet und bereitgestellt wird. Die Engine wird unter der MIT Lizenz verbreitet und ist speziell auf die Entwicklung von 2D Spielen ausgerichtet. Die Cocos 2D-X Game Engine bietet die folgenden Kernfunktionalitäten:

- **Real-Time 2D Applikationen:** Mit der Cocos 2D-X Game Engine können 2D Computerspiele und Simulationen erstellt werden. (3D Applikationen sind nicht unterstützt)
- **Editor:** Die Cocos 2D-X Game Engine stellt analog den anderen Game Engines eine visuelle Benutzerschnittstelle zum Erstellen und Bearbeiten von Spielen bereit. Dieser Editor heisst «Cocos Creator». Anders als bei den anderen Game Engines ist der visuelle Editor nicht notwendig, um Applikationen zu erstellen, sondern ein zusätzliches Feature. Die Engine kann auch vollumfänglich mit der bereitgestellten C++ API verwendet werden.
- **Multiplattform:** Die Cocos 2D-X Game Engine ermöglicht es, Spiele für verschiedene Plattformen zu erstellen, darunter sind Computer (Windows, Mac OS, Linux) und Smartphones (Android, iOS).
- **Scripting:** Die Cocos 2D-X Game Engine bietet die Möglichkeit mit der Programmiersprache Javascript eigene Scripts für das Spiel und die Objekte im Spiel zu schreiben.

Die Cocos 2D-X Game Engine ist auf die Entwicklung von 2D Spielen ausgerichtet und bietet in diesem Bereich alle wichtigen Tools an. Im Vergleich zu anderen Game Engines wie zum Beispiel die Godot Game Engine bietet sie aber weniger Features an. Ein wichtiges Feature, das fehlt ist zum Beispiel ein Marktplatz, an dem die Nutzer untereinander Assets austauschen können. Durch die Bereitstellung einer Low-Level Programmierschnittstelle bietet die Game Engine mehr Möglichkeiten in die Funktionsweise der Game Engine einzutragen als die anderen Game Engines.



Abbildung 6 Userinterface der Cocos 2D-X Game Engine

4.2 Marktpositionierung 2DGameSDK

Es wird sehr schnell klar, dass eine Game Engine die im Rahmen einer Bachelorthesis realisiert wird, bei Weitem nicht den Funktionsumfang anbieten kann, den die etablierten Game Engines nach einer mehrjährigen Entwicklungszeit anbieten. Das Endprodukt der Arbeit soll deshalb eine Game Engine sein, die sich auf ein bestimmtes Nischensegment des Marktes konzentriert. Die Game Engine soll mit möglichst wenig externen Abhängigkeiten und mit einer einfachen Kernstruktur ein vielseitig erweiterbares Software Development Kit bereitstellen, das für die effiziente Entwicklung von verschiedenen Arten von 2D Computerspielen genutzt werden kann. Das folgende sind die wesentlichen Alleinstellungsmerkmale der Game Engine:

- **Ultra-Lightweight:** Die gesamte Game Engine und alle Abhängigkeiten können in einem Library Paket von unter 15 MB bereitgestellt werden. (Die Game Engine selbst ist eine DLL Datei, die kleiner ist als 1 MB)
- **Vielseitig erweiterbar:** Die Game Engine gibt zwar eine fest definierte Struktur vor, (z. B. besteht eine Spielwelt immer aus einer Anordnung von Tiles) die Komponenten der Struktur können aber auf verschiedenen Ebenen an beliebigen Stellen ausgetauscht oder erweitert werden. (z. B. hat der Spielentwickler die Möglichkeit Spielwelten zu definieren, die auf einzelnen Bilder basiert) Der Spielentwickler hat somit die Freiheit in jeden Prozess des Spiels direkt einzutreten und die Funktionalität der Game Engine nach eigenem Belieben zu erweitern.
- **Schnelle Implementation:** Die Game Engine ist darauf ausgerichtet, dass die Entwicklung von einfachen Spielen mit möglichst wenig Codezeilen realisiert werden kann. Anders als andere Game Engines, deren High Level Schnittstelle ein Editor GUI ist, ist die High Level Schnittstelle des Produkts dieser Arbeit eine Sammlung von vordefinierten Objekten für die wichtigsten Anwendungsfälle.
- **Offene Schnittstelle:** Die Game Engine verwendet für Konfigurationsdateien und Assets (wie z.B. die Tilemap der Spielwelt) eine JSON Dateischnittstelle, deren Spezifikation öffentlich bereitgestellt wird. Eine offene Schnittstelle ermöglicht es beliebigen Applikationen von Drittparteien Dateien auszugeben, die mit der Game Engine kompatibel sind.

Die Game Engine konzentriert sich hauptsächlich auf einen Markt von kleinen Spielentwicklerfirmen, die ein frei verfügbares, einfaches und vielseitig erweiterbares Software Development Kit für die effiziente, C++ basierte Implementation eines 2D Computerspiels suchen.

4.3 Distributionskonzept

Da das Produkt dieser Arbeit ein Open Source Projekt ist, das für alle frei verfügbar sein soll, kann es ohne zusätzliche Kosten auf öffentlichen Plattformen bereitgestellt werden. Die kompilierte Library wird im entsprechenden Github Repository bereitgestellt, sowie auf der Open Source Softwareplattform Sourceforge. Darüber hinaus wird das Produkt auf verschiedenen Foren und sozialen Medien präsentiert, sodass möglichst viele potenzielle Anwender erreicht werden können.

5 Evaluation: Physik Engine

5.1 Ausgangslage

Ein wichtiges Feature von Game Engines, ist die Möglichkeit zu erkennen, wenn sich Objekte überschneiden. Erst durch die Erkennung von Überschneidungen zwischen Objekten, ist es zum Beispiel möglich massive, undurchdringbare Objekte zu definieren, oder Objekte zu definieren, die dem Spieler bei Berührung Schaden zufügen können. Viele Spiele der heutigen Zeit sind Simulationen auf Basis von physikalischen Gesetzen, dadurch wird das Spielverhalten realistischer und ein besserer Immersionseffekt kann erzielt werden. Für die Berechnung der Physik eines Spiels, wird meistens eine externe Softwarebibliothek verwendet, diese stellt eine Simulationsumgebung bereit, in der physikalische Objekte erstellt und deren Verhalten über die Zeit simuliert werden kann. Solche Softwarebibliotheken werden auch «Physik Engines» genannt.

5.2 Varianten

Im Rahmen des ersten Sprints der Implementationsphase wurde geprüft, ob es grundsätzlich möglich ist, Überschneidungen zwischen Objekten anhand eines selbst erstellten Algorithmus' zu erkennen. Für diese Variante wurde ein Proof Of Concept erstellt, dieses ist im Repository unter dem Branch «CollisionsPOC» abrufbar. Für die Berechnung der Überschneidungen wurde ein zweistufiges Verfahren implementiert, bei dem zuerst geprüft wird, ob sich die Bounding Boxes der Objekte überschneiden. Wenn sich die Bounding Boxes von zwei Objekten überschneiden, wird weiter im Detail geprüft, ob sich die exakten Formen der beiden Objekte überschneiden, hierzu werden für alle Kanten der ersten Form die Schnittpunkte mit allen Kanten der zweiten Form berechnet. Eine Kollision hat dann stattgefunden, wenn einer der berechneten Schnittpunkte innerhalb des definierten Bereichs der Form befindet:

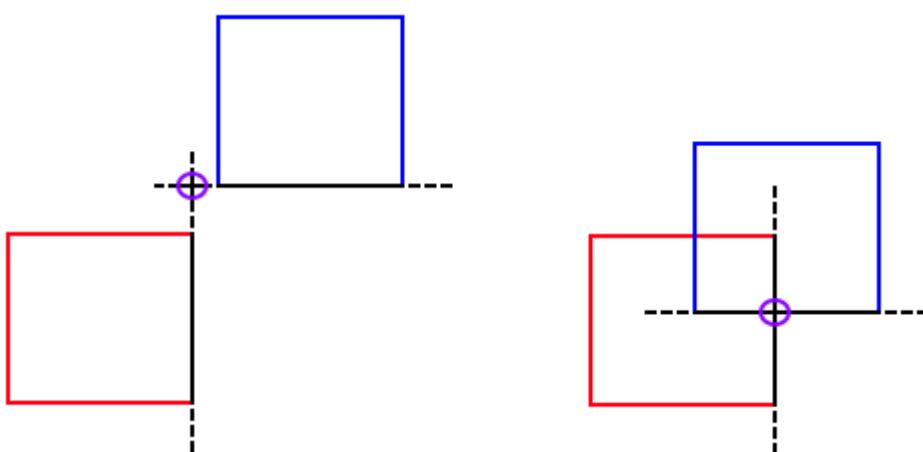


Abbildung 7 Prinzip der Kollisionserkennung auf Basis von Schnittpunkt Berechnungen. Links: Keine Kollision (Schnittpunkt ist ausserhalb der Form), rechts: Kollision (Schnittpunkt ist in der Form)

Die Ergebnisse dieser selbst gemachten Lösung sind akzeptabel. Die Engine ist in der Lage die Kollisionen zu erkennen und die jeweiligen Schnittpunkte im lokalen Koordinatensystem der Objekte oder im Weltkoordinatensystem auszugeben. Für die Berechnung von resultierenden Kräften, die einer Kollision von Objekten entspringen, ist dieses Modell jedoch zu einfach und ungeeignet. Die Möglichkeit selbst ein Modell mit einer volumfähiglichen Physiksimulation zu implementieren wurde in Betracht gezogen, jedoch aus zeitlichen Gründen wieder verworfen. Statt einer selbst gemachten Lösung für die Physiksimulation, wurde entschieden im Rahmen des ersten Sprints dieser Arbeit ein Proof of Concept für physikbasierte Spiele mit den relevantesten C++ Physik Engines zu

erstellen und die jeweiligen Engines dann mit einander zu vergleichen. Die folgenden beiden Physik Engines wurden für die Evaluation in Betracht gezogen:

- **Box2D** [12]: Eine C++ basierte Open Source Physik Engine für Spiele vom Entwickler Erin Catto.
- **Chipmunk Physics** [2]: Eine C basierte Open Source Physik Engine der Firma Howling Moon Software. Chipmunk Physics hat bis vor Kurzem auch eine kostenpflichtige Pro Version mit zusätzlichen Features angeboten, heute sind alle Features der Pro Version in der freien Version enthalten.

5.3 Evaluationskriterien

Für die Evaluation der verschiedenen Physik Engines wurden folgende Kriterien definiert:

- **Performance:** Rechengeschwindigkeit bei unterschiedlichen Lasten, evaluiert anhand selbst erstellter Benchmarking Szenarios (siehe Kapitel 5.4). Folgende Szenarios wurden geprüft:
 - o Performance bei gleichzeitiger Simulation von 100 dynamischen Objekten.
 - o Performance bei gleichzeitiger Simulation von 500 dynamischen Objekten.
 - o Performance bei gleichzeitiger Simulation von 1'000 dynamischen Objekten.
- **Dokumentation:** Qualität und Aktualität der Softwaredokumentation. Folgende Kriterien wurden berücksichtigt:
 - o Qualität und Aktualität des Nutzerhandbuchs.
 - o Qualität und Aktualität der API-Dokumentation.
 - o Anzahl und Relevanz der Anwendungsbeispiele.
- **Kompatibilität:** Schritte die notwendig sind, um die Physik Engine als Teil der Game Engine zu verwenden. Folgende Kriterien wurden in Betracht gezogen:
 - o Numerische Grenzbereiche: Wenn eine Physik Engine nur in einem bestimmten Dimensionsbereich exakt rechnet, ist bei Umrechnungen zwischen visuellen und physikalischen Dimensionen eine zusätzliche Skalierung notwendig.
 - o Verwendetes Koordinatensystem: Die Umrechnung von Koordinaten zwischen dem physikalischen und dem visuellen Koordinatensystem sollte möglichst effizient möglich sein.
 - o Individuelles Handling von Kollisionen: Die Physik Engine muss eine Möglichkeit bereitstellen, um auf Kollisionen zwischen Objekten individuell zu reagieren. Die Game Engine kann so z. B. das Spielobjekt benachrichtigen, wenn es in der physikalischen Welt eine Kollision gab. Idealerweise kann das Standard Kollisionsverhalten sogar unterbrochen werden.

5.4 Benchmarking

5.4.1 Versuchsaufbau

Für das Benchmarking der Physik Engines wurde mit jeder der beiden Physik Engines eine eigene Applikation geschrieben. Die beiden Applikationen sollen den exakt selben Versuchsaufbau unter den gleichen Voraussetzungen simulieren, dadurch sind die Ergebnisse mit einander vergleichbar.

Der Versuchsaufbau sieht wie folgt aus:

1. Die Schwerkraft der simulierten Welt beträgt jeweils 10 m/s^2 .
2. Vor Simulationsbeginn werden n (100, 500 oder 1'000) quadratische Objekte mit einer Kantenlänge von 0.5 m zufällig in der Luft platziert. Zusätzlich wird ein statischer Boden und jeweils Links und Rechts eine Wand platziert, um dafür zu sorgen, dass sich die Objekte während der Simulation immer im gleichen Bereich aufhalten.
3. Die Objekte erhalten eine Textur und werden während der Simulation mit der SFML Library in der richtigen Pose dargestellt.
4. Es wird eine Echtzeitsimulation des Szenarios durchgeführt, dabei wird gezählt wie viele Simulationsschritte pro Sekunde berechnet und dargestellt werden können. Ein guter Wert sollte dabei nie unter 60 liegen, um zu garantieren, dass die Simulation stets schneller läuft als die Bildwiederholrate der meisten Monitore.

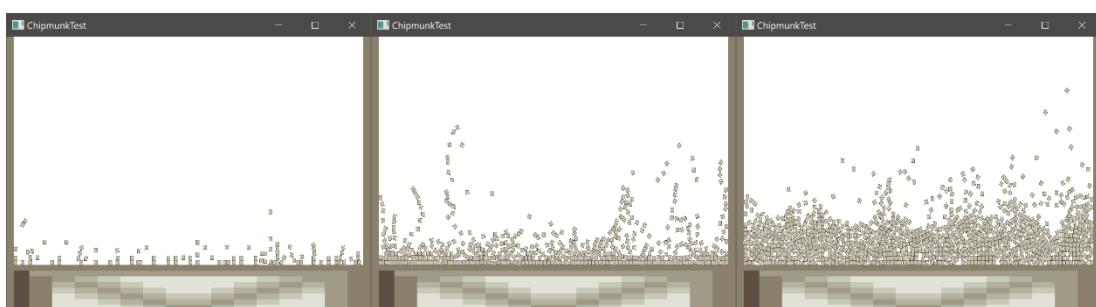


Abbildung 8 Benchmarking Szenarios mit 100 Objekten (links), 500 Objekten (Mitte) und 1'000 Objekten (rechts)

Für die Durchführung des Versuchs wurde ein stationärer Computer mit den folgenden Spezifikationen verwendet:

- **Prozessor:** Intel Core i7-4770, 8 CPUs, 3.4Ghz
- **Arbeitsspeicher:** 16 GB, DDR3 RAM
- **Grafikkarte:** NVIDIA GeForce GTX 970
- **Betriebssystem:** Windows 10 Education 64-bit

5.4.2 Ergebnisse

Die genauen Messergebnisse sind im Anhang aufgeführt. Für die statistische Analyse wurden nur die Messungen ab der 30. Sekunde (bzw. ab Zeitindex 30) berücksichtigt um hohe Schwankungen, die zu Anfangs entstehen auszugrenzen. Die Schwankungen der Messungen über die gesamte Zeit sind separat unter den Extrema aufgeführt. Die finalen Ergebnisse der Messungen sind wie folgt:

Ergebnisse der Messungen mit Chipmunk Physics

Statistische Analyse der Messungen von Zeitindex 30 (30 s) bis Zeitindex 70 (70 s):

Tabelle 1 Statistische Analyse des Benchmarking mit Chipmunk Physics (Zeitindex 30-70)

| Wert | 100 Objekte | 500 Objekte | 1'000 Objekte |
|-----------|-------------|-------------|---------------|
| \bar{x} | 2'282.26 | 626.55 | 297.19 |
| σ | 27.24 | 5.71 | 2.48 |
| k/n | 70.97% | 67.74% | 80.65% |

- ⇒ Der Wert "k/n" sagt aus, welcher Anteil der Messungen innerhalb der einfachen Standardabweichung liegt ($\bar{x} - \sigma \leq x \leq \bar{x} + \sigma$). Eine Messung ist statistisch aussagekräftig, wenn dieser Wert bei ca. 68% oder höher liegt.

Extrema der Messungen über die gesamte Zeit:

Tabelle 2 Extrema des Benchmarking mit Chipmunk Physics (Zeitindex 0–70)

| Wert | 100 Objekte | 500 Objekte | 1'000 Objekte |
|---------|-------------|-------------|---------------|
| Minimum | 2200 | 602 | 292 |
| Maximum | 2510 | 1070 | 473 |

Ergebnisse der Messungen mit Box2D

Statistische Analyse der Messungen von Zeitindex 30 (30 s) bis Zeitindex 70 (70 s):

Tabelle 3 Statistische Analyse des Benchmarking mit Box2D (Zeitindex 30–70)

| Wert | 100 Objekte | 500 Objekte | 1'000 Objekte |
|-----------|-------------|-------------|---------------|
| \bar{x} | 2'398.48 | 1'566.84 | - |
| σ | 19.51 | 16.74 | - |
| k/n | 87.10% | 74.19% | - |

- ⇒ Für die Messungen mit 1000 Objekten konnten keine Daten erhoben werden, weil die Simulationsgeschwindigkeit ab Zeitindex 10 unterhalb vom messbaren Bereich von einem Frame pro Sekunde ist.

Extrema der Messungen über die gesamte Zeit:

Tabelle 4 Extrema des Benchmarking mit Box2D (Zeitindex 0–70)

| Wert | 100 Objekte | 500 Objekte | 1'000 Objekte |
|---------|-------------|-------------|---------------|
| Minimum | 1'737 | 231 | - |
| Maximum | 2'518 | 1'588 | - |

Schlussfolgerung

Bei den Messungen mit 100 Objekten sind keine signifikanten Unterschiede zu erkennen, beide Physik Engines sind in der Lage hohe Geschwindigkeiten von über 2'000 Frames pro Sekunde zu erzielen. Bei den Messungen des Szenarios mit 500 Objekten liegt die Box2D Engine am Zeitpunkt der Messung deutlich vorn, unter Betrachtung der Extrema über die gesamte Zeit fällt jedoch auf, dass das Minimum der Messungen mit Box2D deutlich unter dem Minimum der Messungen mit Chipmunk Physics liegt. Die Messungen des Szenarios mit 1'000 Objekten konnten nur mit der Chipmunk Engine erhoben werden, da Box2D in diesem Szenario eine Geschwindigkeit von unter 1 Frame pro Sekunde aufwies. Insgesamt ist festzustellen, dass die Chipmunk Physics Engine stabiler läuft (die Abweichung zwischen dem Minimum und Maximum ist kleiner) und mit grossen Lasten besser umgehen kann als Box2D.

5.5 Evaluation

Für die Evaluation wurden die in Kapitel 5.3 genannten Evaluationskriterien gegen einander abgewogen und mit einer relativen Gewichtung von 1 (nützlich), 2 (notwendig), oder 3 (unbedingt notwendig) versehen. Anschliessend wurden die Eigenschaften der beiden Physik Engines in Bezug auf ein bestimmtes Kriterium betrachtet und mit einer Punktzahl von 1 (Kriterium nicht erfüllt), 2 (Kriterium teilweise erfüllt), oder 3 (Kriterium erfüllt) versehen.

5.5.1 Evaluation Chipumunk Physics

Die Ergebnisse der Evaluation der Chipmunk Physics Engine sind wie folgt:

Tabelle 5 Evaluation Chipmunk Physics (G = Gewichtung, P = Punkte, P * G = Gewichtetes Ergebnis)

| Kriterium | G (1-3) | P (1-3) | P*G | Kommentar |
|---------------------------|---------|-----------|-----|---------------------------------------------------------------------------|
| Performance | | | | |
| 100 Objekte | 3 | 3 | 9 | |
| 500 Objekte | 3 | 3 | 9 | Minimum 600 FPS |
| 1000 Objekte | 2 | 3 | 6 | Minimum 300 FPS |
| Dokumentation | | | | |
| Nutzerhandbuch | 3 | 2 | 6 | Ausführliche Dokumentation, einige Beispiele fehlen, keine Illustrationen |
| API Dokumentation | 2 | 3 | 6 | Doxygen generierte Web Dokumentation, gut kommentiert |
| Beispiele | 2 | 3 | 6 | 32 Einfache Beispiele mit unterschiedlichen Szenarien |
| Kompatibilität | | | | |
| Numerische Grenzbereiche | 2 | 3 | 6 | Keine Grenzbereiche |
| Koordinatensystem | 2 | 3 | 6 | Bottom-Up, Objektzentrisch |
| Indiv. Kollisionshandling | 3 | 3 | 9 | Unterstützt |
| Total: | | 63 | | |

Anmerkungen zu den Ergebnissen:

- **Performance:** Die Chipmunk Physics Engine läuft stabil auch bei einer hohen Anzahl von dynamischen Objekten.
- **Dokumentation:** Die Dokumentation der Chipmunk Physik Engine ist sehr gut und es werden auch viele Beispiele zur Verfügung gestellt. Das Nutzerhandbuch könnte an bestimmten Stellen etwas mehr Codebeispiele und Illustrationen enthalten.
- **Kompatibilität:** Laut eigenen Angaben der Hersteller (siehe [2]) gibt es keine Begrenzungen für die Dimensionen der Objekte, eine Skalierung der Objekte ist deshalb nicht notwendig. Das objektzentrische Koordinatensystem ist gut mit dem SFML Koordinatensystem kompatibel und ein individuelles Kollisionshandling wird vollumfänglich unterstützt.

5.5.2 Evaluation Box2D

Die Ergebnisse der Evaluation der Box2D Physics Engine sind wie folgt:

Tabelle 6 Evaluation Box2D (G = Gewichtung, P = Punkte, P * G = Gewichtetes Ergebnis)

| Kriterium | G (1-3) | P (1-3) | P*G | Kommentar |
|---------------------------|---------|-----------|-----|----------------------------------------------------------------|
| Performance | | | | |
| 100 Objekte | 3 | 3 | 9 | |
| 500 Objekte | 3 | 3 | 9 | Hohe Schwankung, Minimum 230 FPS |
| 1000 Objekte | 2 | 1 | 2 | Nicht simulierbar |
| Dokumentation | | | | |
| Nutzerhandbuch | 3 | 3 | 9 | Ausführliche Dokumentation, gute Illustrationen und Beispiele |
| API Dokumentation | 2 | 3 | 6 | Dxygen generierte Web Dokumentation, gut kommentiert |
| Beispiele | 2 | 2 | 4 | 1 Komplexes Beispiel mit Input Handling, OpenGL Rendering etc. |
| Kompatibilität | | | | |
| Numerische Grenzbereiche | 2 | 2 | 4 | Optimiert für Objekte mit Dimensionen im Bereich 0.1 m - 10 m |
| Koordinatensystem | 2 | 3 | 6 | Bottom-Up, Objektzentrisch |
| Indiv. Kollisionshandling | 3 | 3 | 9 | Unterstützt |
| Total: | | 58 | | |

Anmerkungen zu den Ergebnissen:

- **Performance:** Leider hat die Box2D Physik Engine beim Test mit 1'000 dynamischen Objekten versagt, da die minimal notwendige Geschwindigkeit von 60 Frames pro Sekunde nicht erreicht werden konnte. Bei kleineren Objektzahlen kann die Physik Engine jedoch gut mit der Chipmunk Physik Engine mithalten.
- **Dokumentation:** Die Dokumentation der Box2D Physik Engine ist sehr gut und enthält verschiedene Codebeispiele und Illustrationen. Das Beispiel, das zur Verfügung gestellt wird, ist sehr gut ausgebaut und zeigt die Entwicklung eines 2D Spiels auf Basis der Physik Engine. Leider gibt es aber nur ein einziges Beispiel.
- **Kompatibilität:** Die Box2D Physik Engine ist gemäss eigenen Angaben (siehe [13]) optimiert für dynamische Objekte im Dimensionsbereich von 0.1 m - 10 m, es ist somit in bestimmten Fällen eine Skalierung der Objekte notwendig. Das objektzentrische Koordinatensystem ist gut mit dem SFML Koordinatensystem kompatibel und ein individuelles Kollisionshandling wird vollumfänglich unterstützt.

5.5.3 Ergebnisanalyse und Entscheid

Die Ergebnisse zeigen, dass die Chipmunk Physics Engine in den Punkten Performance und Kompatibilität deutlich besser abschneidet als die Box2D Physik Engine und darum auch eine bessere Gesamtpunktzahl erzielt. Es ist zusätzlich festzuhalten, dass unter Einsatz der Box2D Physik Engine das zuvor definierte Qualitätskriterium Q3 (siehe [5] oder Kapitel 8.2) wegen mangelnder Performance nicht eingehalten werden kann. Die Wahl der Physik Engine, die im Rahmen des Projekts verwendet wird fällt somit auf die Chipmunk Physics Engine.

6 Design

6.1 UML Kontextdiagramm

Das folgende UML Kontextdiagramm zeigt die Komponenten der Game Engine sowie die externen Komponenten von denen die Engine abhängt:

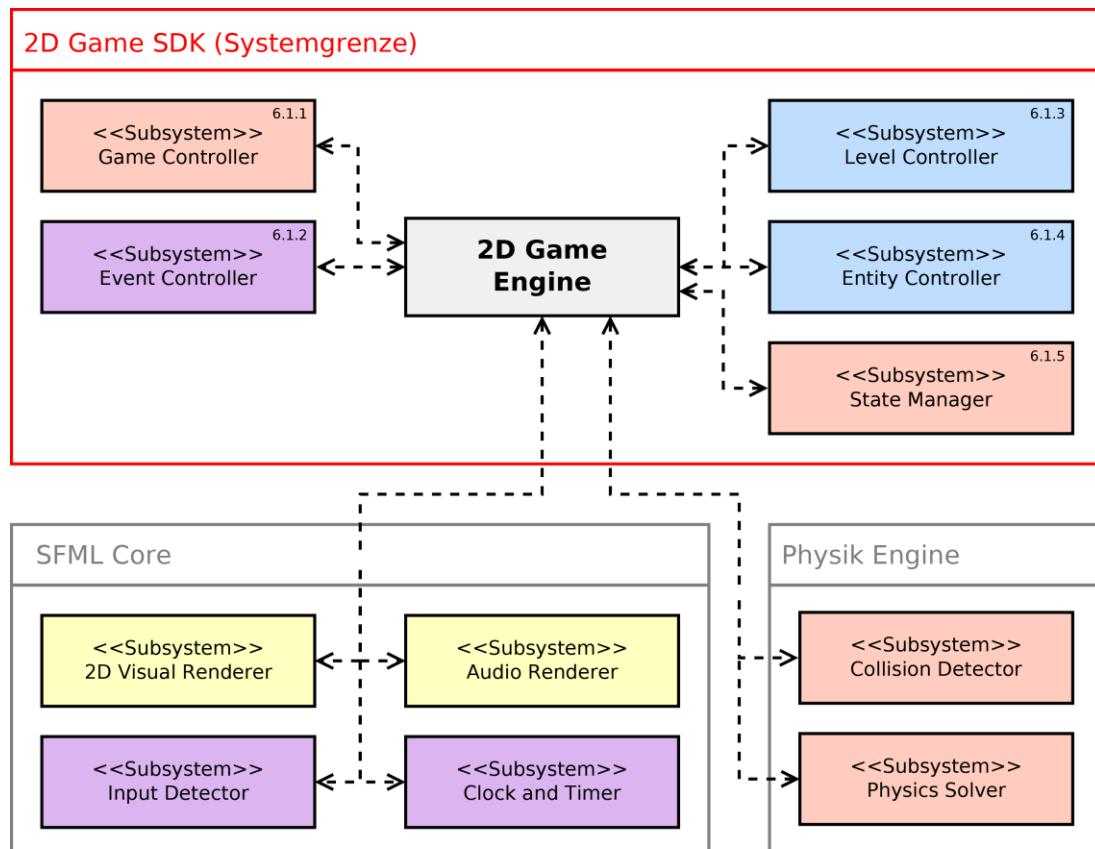


Abbildung 9 UML Kontextdiagramm 2DGameSDK

Die einzelnen Subsysteme des Projekts sind in den folgenden Unterkapitel beschrieben.

6.1.1 Game Controller

Der Game Controller ist für die Initialisierung des Spiels und für die Ausführung des Spielablaufs zuständig. Der Spielablauf definiert sich durch den sogenannten Game Loop in dem fortlaufend die Events verarbeitet werden und dann der Spielzustand (bzw. die Spielwelt und alle Spielobjekte) aktualisiert wird. Die Darstellung des Spiels geschieht unabhängig davon in einem separaten Thread. Der Game Controller arbeitet eng mit dem State Manager zusammen, diese garantieren gemeinsam, dass im Thread, mit dem das Spiel dargestellt wird, immer ein garantierter zulässiger Spielzustand dargestellt wird. Das Game Controller Subsystem ist Teil des «Core» Pakets. (Siehe Kapitel 6.2)

6.1.2 Event Controller

Der Event Controller erlaubt es verschiedene Events im Spiel zu registrieren und zu verarbeiten. Events können zum Beispiel die Tastatur- und Mauseingaben des Spielers oder auch komplexe selbst erstellte Events sein. Das Event Controller Subsystem ist Teil des «Event» Pakets. (Siehe Kapitel 6.2)

6.1.3 Level Controller

Der Level Controller ist für den Aufbau der Spielwelt zuständig. Er lädt die visuelle Repräsentation der Spielwelt sowie die unterschiedlichen Materialeigenschaften und stellt sie für die physikalische und visuelle Repräsentation zur Verfügung. Das Level Controller Subsystem ist Teil des «World» Pakets. (Siehe Kapitel 6.2)

6.1.4 Entity Controller

Der Entity Controller, ist für die Verwaltung aller beweglichen Objekte zuständig. Durch ihn können Objekte auf einem Scene Graph platziert werden, sodass sie in der visuellen und physikalischen Repräsentation berücksichtigt werden. Das Entity Controller Subsystem ist Teil des «Scene» Pakets. (Siehe Kapitel 6.2)

6.1.5 State Manager

Der State Manager verwaltet den aktuellen Spielstatus und stellt ihn während der Laufzeit des Spiels für alle Klassen zur Verfügung. Der State Manager verfügt über Mechanismen, die die sichere Löschung eines bestimmten Spielobjekts garantieren, so ist es z. B. auch möglich, dass ein Spielobjekt sich selbst löschen kann. Das State Manager Subsystem ist Teil des «Core» Pakets. (Siehe Kapitel 6.2)

6.2 UML Paketdiagramm

Die einzelnen Pakete der Game SDK und deren jeweilige Abhängigkeiten sind im folgenden UML Paketdiagramm dargestellt:

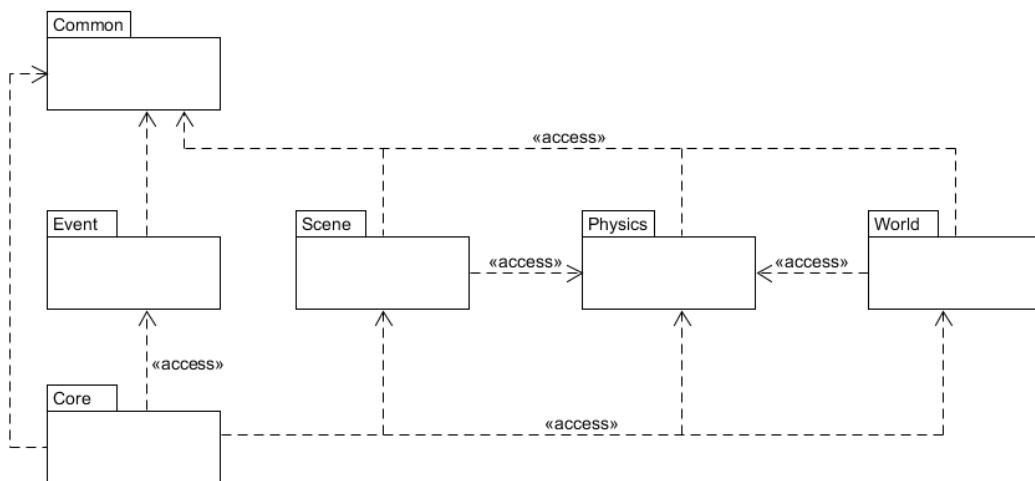


Abbildung 10 UML Paketdiagramm

Die einzelnen Pakete sind in den folgenden Unterkapitel beschrieben.

6.2.1 Paket «Common»

Das Paket «Common» hat selbst keine Abhängigkeiten und wird von jedem anderen Paket verwendet. Es enthält unter anderem Typdefinitionen (Basistypen und Plain Data Objekte), verschiedene Hilfsklassen und -Funktionen sowie Hilfsmittel für grafische Berechnungen.

6.2.2 Paket «World»

Das Paket «World» enthält alle Klassen und Hilfsmittel im Zusammenhang mit der Spielwelt. Dies beinhaltet unter anderem die Datenstrukturen der Spielwelt, (Kacheln und Materialien) die Parser Algorithmen zum Einlesen der Spielweltdaten, sowie die Hilfsklasse für die Erstellung der Spielwelt.

6.2.3 Paket «Event»

Das Paket «Event» enthält alle Klassen und Hilfsmittel im Zusammenhang mit Events und der Verarbeitung von Events. Dies beinhaltet unter anderem die Event Datentypen und den Event Controller.

6.2.4 Paket «Scene»

Das Paket «Scene» enthält alle Klassen und Hilfsmittel im Zusammenhang mit beweglichen Spielobjekten. Dies beinhaltet unter anderem den Scene Graph und die Objekte auf dem Scene Graph.

6.2.5 Paket «Physics»

Im Paket «Physics» sind die Interaktionen der Game Engine mit der Physik Engine definiert. Das Paket enthält unter anderem die Shape Klasse, die eine geometrische Form in der physikalischen Welt definiert und die ShapePhysics Klassen, die das jeweilige Physikalische Verhalten der Form definieren.

6.2.6 Paket «Core»

Das Paket «Core» ist das Kernstück der Game Engine und greift auf die Definitionen von allen anderen Paketen zu. Im Paket sind unter anderem die Game und die StateManager Klasse definiert, diese sind für den Ablauf des Update Loops und des Render Loops zuständig. Das Paket beinhaltet zahlreiche weitere Funktionalitäten, wie zum Beispiel den AudioController, (Ausgabe von Musik und Tönen) den CameraController (Kamerasteuerung) das OverlayDisplay (Statusanzeige) und einfache Spielobjekte, die nicht Teil des SceneGraphs sind (z. B. Effect und Projectile). Ein wichtiger Bestandteil des Pakets sind die unterschiedlichen Rendering Strategien, die definieren, wie ein bestimmtes Spielobjekt dargestellt wird.

6.3 UML Klassendiagramm: Global

Das folgende UML Klassendiagramm zeigt die Klassen und Abhängigkeiten zwischen den Klassen aus allen Packages. Dieses Klassendiagramm ist nur eine Übersicht und enthält daher keine Informationen zu Attributen und Methoden der Klassen. Das Paket «Common» ist zwecks Übersicht nicht dargestellt:

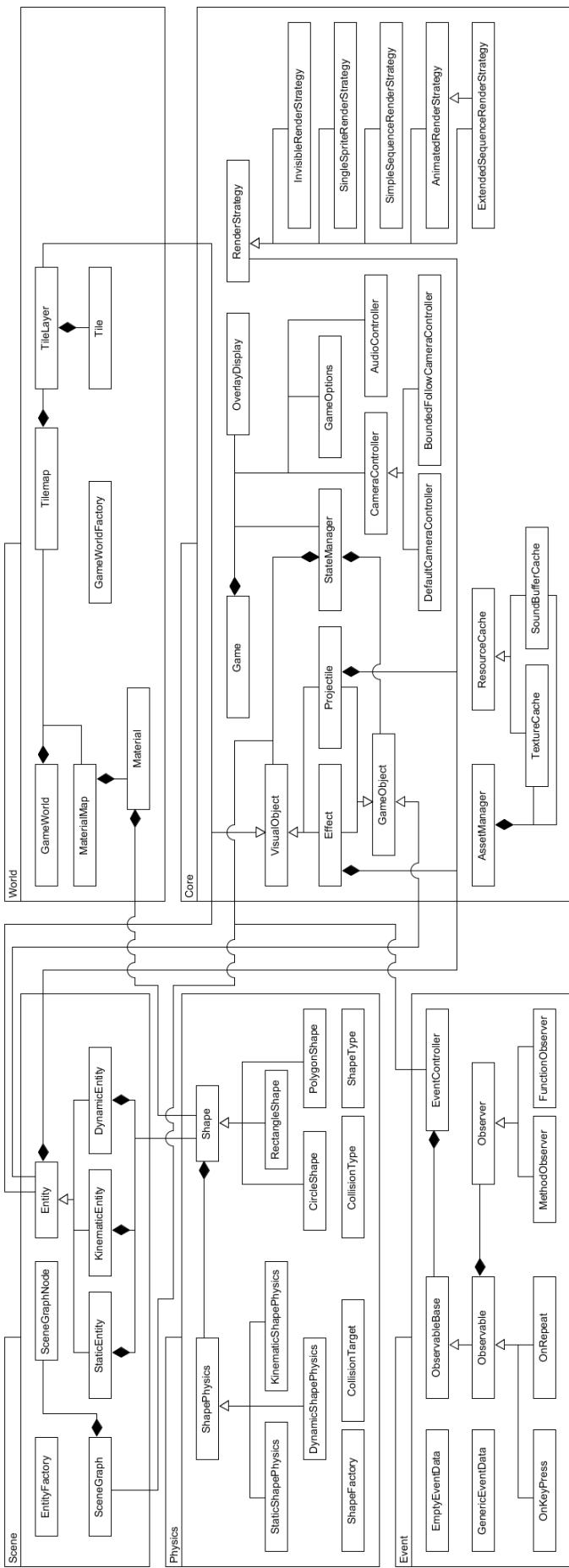


Abbildung 11 UML Klassendiagramm:
Highlevel Ansicht

Da das «Common» Package in der Grafik nicht dargestellt ist, ist zusätzlich zu erwähnen, dass viele der Klassen eine indirekte Referenz zur Klasse «Game» haben. Diese Abhängigkeit ist notwendig, weil die «Game» Klasse Informationen über den aktuellen Zustand des Spiels enthält und auch viel genutzte zustandsabhängige Hilfsmittel (wie zum Beispiel den PoseConverter) bereitstellt. In den detaillierten Klassendiagrammen ist diese Abhängigkeit ersichtlich.

6.4 Design Patterns

Gemäss Vorgabe aus dem Pflichtenheft dieser Arbeit [5] wurden bei der Umsetzung des Projekts verschiedene gut etablierte Design Patterns der Softwareindustrie (gemäss [14]) eingesetzt. Im folgenden Teil sind die verschiedenen Design Patterns aufgelistet, die im Rahmen der Arbeit eingesetzt wurden:

6.4.1 Creational Design Patterns

- **Factory Method:** Das Factory Method Design Pattern erlaubt es mit einfachen Hilfsmethoden komplexe Objekte zu erstellen. Die Game Engine stellt Factory Methoden für die Erstellung der Spielwelt (Klasse GameWorld) und für die Erstellung von Physikalischen Formen (Klasse Shape) zur Verfügung.

6.4.2 Behavioural Design Patterns

- **Observer/Observable:** Das Observer/Observable Design Pattern wird eingesetzt, um Ereignisse zu verarbeiten und an jene Klassen weiterzuleiten, die am jeweiligen Ereignis interessiert sind. Die Game Engine implementiert das Observer/Observable Design Pattern für die Verarbeitung von Spiel Ereignissen.
- **Strategy:** Das Strategy Design Pattern wird eingesetzt, um eine Familie von Algorithmen zu kapseln und die einzelnen Implementationen über eine gemeinsame Schnittstelle (bzw. Strategie) bereitzustellen. Die Game Engine implementiert dieses Design Pattern durch die Bereitstellung von Strategien zum Darstellen von Objekten (Klasse: RenderStrategy) und Strategien zum Initialisieren der physikalischen Eigenschaften von physikalischen Formen. (Klasse ShapePhysics)

- **Template Method:** Mit dem Template Method Design Pattern wird ein abstraktes Skelett für eine Methode einer Klasse definiert, dieses Skelett kann anschliessend von Erweiterungen dieser Klasse (bzw. den Child Klassen) konkretisiert werden. Dies bietet den Erweiterungen der Klasse mit der Template Methode die Möglichkeit ein polymorphes Verhalten zu definieren. Die Game Engine stellt an verschiedenen Orten Basisklassen mit Template Methoden bereit, die es dem Spielentwickler erlauben grundlegend in die Funktionsweise der Game Engine einzugreifen. Das Ziel dabei ist es, dem Entwickler die maximal mögliche Freiheit zur Erweiterung der Game Engine zu gewährleisten. Beispiele von Klassen, die das Template Method Pattern implementieren sind überall in der Game Engine anzutreffen. Es sind unter anderem die Spielobjekte, (Inklusive Effekte, Projektiler und Entitäten) die visuellen Objekte, der Kamera Controller, die Rendering Strategien, die Events und viele weitere Klassen.

6.5 Globale Konzepte

6.5.1 Wahl des Koordinatensystems

Im Rahmen des Vorprojekts wurde mit dem Koordinatensystem gearbeitet, dass das Framework SFML zur Verfügung stellt. Da SFML (hauptsächlich) ein Rendering Framework ist, folgt dieses der Konvention von Bildschirmkoordinaten. Das SFML Koordinatensystem ist so aufgebaut, dass der Nullpunkt oben links liegt, die X-Achse nach rechts gerichtet ist und die Y-Achse nach unten. Die Maßeinheit bei diesem System sind Pixel, wobei aber auch Dezimalzahlen zulässig sind. (1 Pixel der Spielwelt entspricht nicht zwingend einem Pixel auf dem Bildschirm) Im Rahmen des Vorprojekts war es sinnvoll, mit diesem SFML Koordinatensystem zu arbeiten, weil die Koordinaten (bzw. die geometrischen Transformationen) so unverändert an die Rendering Engine geschickt werden konnten. Im Rahmen dieser Thesis kommt zusätzlich die Chipmunk Physik Engine zum Einsatz, diese verwendet ein eigenes Koordinatensystem das näher an der physikalischen Welt ist. Beim Chipmunk Koordinatensystem ist der Nullpunkt unten links, die X-Achse ist nach rechts gerichtet und die Y-Achse nach oben. Die Maßeinheit des Chipmunk Koordinatensystems ist 1 Meter, da die Physik Engine auf das SI-Einheitensystem ausgerichtet ist.

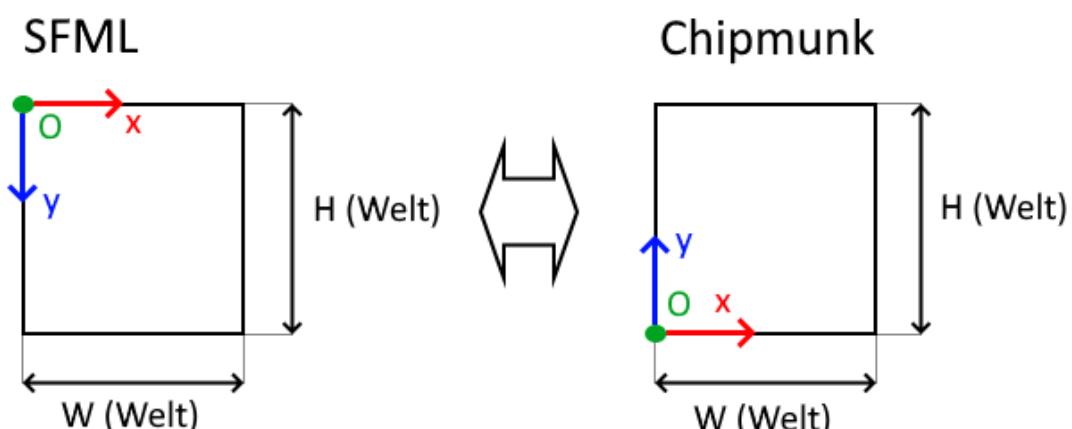


Abbildung 12 Vergleich des visuellen Koordinatensystems von SFML (links) mit dem physikalischen Koordinatensystem von Chipmunk (rechts)

Um diese beiden Koordinatensysteme zu vereinen, war es in einem ersten Schritt notwendig, das Referenzkoordinatensystem zu bestimmen, sodass alle Berechnungen der Game Engine in diesem Koordinatensystem durchgeführt werden und die Resultate ggf. in das andere Koordinatensystem umgerechnet werden. Die Wahl des Referenzkoordinatensystems fiel aus den folgenden Gründen auf das Koordinatensystem der Physik Engine:

- **Performance:** Wenn die Anzahl physikalischer Ereignisse (wie z. B. Kollisionen) pro Zeitschritt gross ist, hat dies zur Folge, dass auch viele Berechnungen innerhalb der Game Engine getätigt werden müssen. Würden diese Berechnungen im visuellen Koordinatensystem durchgeführt werden, müssten bei jedem physikalischen Ereignis zuerst die Koordinaten umgerechnet werden. Visuelle Berechnungen erfolgen jeweils nur ein Mal pro Objekt und Zeitschritt, wenn das jeweilige Objekt dargestellt werden muss. Es ist daher vorteilhaft mit dem physikalischen Koordinatensystem zu rechnen und erst zuletzt die Umrechnung ins visuelle Koordinatensystem durchzuführen. Dies wird auch ausdrücklich in der Dokumentation des Chipmunk Frameworks [15] empfohlen.
- **Intuitive Bedienung:** Für einen Spielentwickler, der mit der Game Engine arbeitet, macht es mehr Sinn, wenn der Nullpunkt des Koordinatensystems am Boden anfängt statt am höchsten Punkt der Weltkarte.

Bei der Umrechnung der Koordinaten und geometrischen Transformationen von einem Koordinatensystem ins andere ist Folgendes zu beachten: Wenn man die Koordinaten ohne irgend eine Skalierung vom einen Koordinatensystem ins andere umrechnet, bedeutet dies dass man voraussetzt, dass eine Einheit aus dem visuellen Koordinatensystem (1 Pixel) immer genau einer Einheit aus dem physikalischen Koordinatensystem (1 Meter) entspricht. Diese Restriktion kann zu Problemen führen, deshalb wurde in der Game Engine als Option der Umrechnungsfaktor «Meter pro Pixel» eingeführt, der bei jeder Umrechnung appliziert wird.

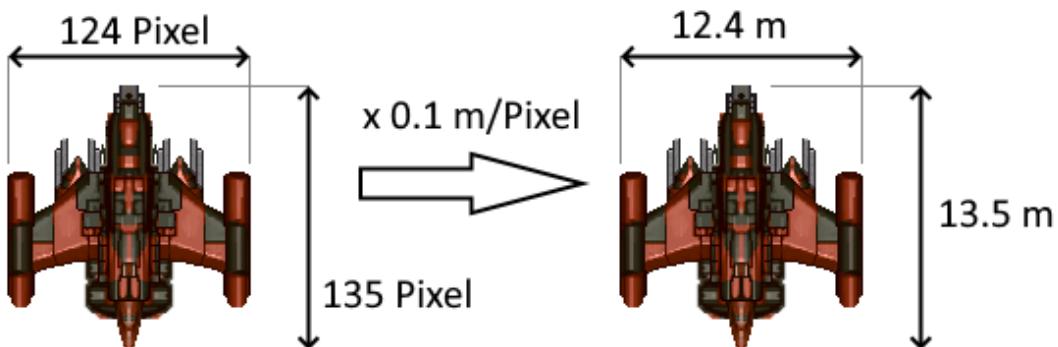


Abbildung 13 Prinzip der Skalierung von Dimensionen zur Umwandlung von Pixel des SFML Koordinatensystems in Meter des Chipmunk Koordinatensystems

Eine letzte Hürde, die hauptsächlich bei der Umrechnung von geometrischen Transformationen zwischen den einzelnen Koordinatensystemen zum Vorschein kommt, ist die Tatsache, dass die Referenzpunkte (lokaler Ursprung) der Objekte nicht gleich sind: Während im SFML Framework der Referenzpunkt eines Objekts immer oben Links ist, ist er im Chipmunk Framework in der geometrischen Mitte vom Objekt. Dieses Problem konnte letztendlich durch manuelles Setzen des Referenzpunktes im SFML Framework umgangen werden.

6.5.2 Rendering Konzept

Das Rendering Konzept, welches im Rahmen des Vorprojekts umgesetzt wurde, birgt noch gewisse Einschränkungen und Probleme:

- **Fehlende Z-Index Sortierung:** Die Reihenfolge, in der die visuellen Objekte gezeichnet werden, kann nicht frei gewählt werden, somit hat der Spielentwickler keine Möglichkeit zu bestimmen welche Objekte sich im Vordergrund und welche Objekte sich im Hintergrund befinden.
- **Timing und Performance:** Da das Rendering im selben Thread läuft, in dem auch der Spielablauf berechnet wird, können Verzögerungen beim Rendering (z. B. auch hardwarebedingt durch vertikale Synchronisation) zu einer Verfälschung der Simulation führen.
- **Fehlende Kapselung:** Die Information wie ein bestimmtes Spielobjekt dargestellt wird (z.B. Statisch oder animiert) ist fest in den jeweiligen Spielobjekten verankert, dies führt bei mehreren Arten von Spielobjekten zu unnötig viel dupliziertem und schlecht wartbarem Code.

Die Erweiterungen des Rendering Konzepts, die im Rahmen dieses Projekts umgesetzt wurden, bieten Lösungen für diese Probleme: Für die fehlende Z-Index Sortierung wurde neu das Konzept der visuellen Objekte (VisualObject Klasse) eingeführt. Ein visuelles Objekt besitzt eine Funktion zum Setzen und Erhalten des Z-Indexes sowie eine Funktion für die Darstellung des Objekts. Alle Spielobjekte, die während dem Rendering Zyklus berücksichtigt werden sollen, müssen zwingend auch visuelle Objekte sein (bzw. die VisualObject Klasse erweitern). Mit diesem Mechanismus können die Objekte nach ihrem Z-Index sortiert und in der richtigen Reihenfolge dargestellt werden. Zur Lösung der Timing und Performanceprobleme, die dadurch entstehen, dass die Berechnung der Spiellogik und das Rendering im selben Thread läuft, wurde das Rendering in einen eigenen Thread ausgelagert. Zuletzt wurde für das Problem der fehlenden Abstraktion zwischen Spielobjekten und deren Darstellung neu das Konzept der Rendering Strategien eingeführt. Eine Rendering Strategie übernimmt sämtliche Rendering Funktionalitäten eines jeweiligen Objekts. Durch diese Entkopplung der Rendering Strategie von den Objekten, können die selben Rendering Strategien für verschiedene Objekttypen verwendet werden.

7 Implementation

7.1 Technologiewahl

7.1.1 C++ Standard

Der im Rahmen dieses Projekts erstellte Source Code wurde mit dem aktuellsten C++ Standard C++17 implementiert und ist auf die Kompilierung mit der GNU Compiler Collection (GCC) [16] ausgerichtet.

7.1.2 SFML

Die sogenannte «Simple and Fast Multimedia Library» SFML [1] ist ein C++ Framework, das die Erstellung von Multimedia-Applikationen für unterschiedliche Betriebssysteme wie Windows, Mac OS und Linux ermöglicht. Das Framework ist spezialisiert auf 2D Applikationen, basiert aber auf OpenGL und bietet daher auch Schnittstellen für die Darstellung von 3D Objekten an. SFML wird mit der Lizenz zlib/png verbreitet, diese Lizenz erlaubt unter anderem auch die kommerzielle Nutzung des Produkts. Für dieses Projekt wurde die neuste Version der SFML Library (SFML 2.5.1) verwendet.

7.1.3 Chipmunk Physics

Das Chipmunks Physics Library [2] ist eine in C geschriebene Softwarebibliothek für die Simulation von verschiedenen Szenarien auf einer physikalischen (2-dimensionalen) Ebene. Die Chipmunk Physics Library ist optimiert für schnelle Berechnungen und daher gut für Echtzeitsimulationen, wie z.B. Computerspiele geeignet. Für dieses Projekt wurde die neuste Version der Chipmunk Physics Library (Chipmunk 7.0.3) verwendet.

7.1.4 JSON Parser

Zum Lesen von JSON Dateien wurde auf die open Source JSON Parser Library von Niels Lohmann (siehe [17]) zurückgegriffen. Diese Library wird unter der MIT Lizenz verbreitet (open source, jedoch auch kommerziell einsetzbar).

7.1.5 Build Tools

Für das Projekt werden CMake Konfigurationsdateien bereitgestellt, dies ermöglicht es, die Applikation mit unterschiedlichen build Tools und IDE's wie MAKE oder Visual Studio zu kompilieren. Im Rahmen dieses Projekts wurde das Produkt mit dem Build Tool mingw32-make der MinGW Softwaresuite auf Windows (Version 7.3.0-posix-seh-rt_v5) kompiliert und getestet.

7.2 Externe Komponenten

7.2.1 Komponenten des SFML Frameworks

Im Rahmen einer vorgelagerten Machbarkeitsstudie wurde entschieden, dass das Projekt mithilfe der Softwarebibliothek SFML [1] erstellt werden soll, die Bibliothek bietet eine einfache, plattformunabhängige Lösung für folgende Aufgabenbereiche:

- **Maus und Tastatureingabe:** Registrieren der Benutzereingabe.
- **Visuelle Ausgabe 2D:** Applikationsfenster, Darstellung texturierter Elemente, Zoom, Welt-/ Bildschirmkoordinatensystem.
- **Visuelle Ausgabe 3D:** Integrierte Schnittstelle zu OpenGL.
- **Audio Ausgabe:** Buffern und Abspielen von Audio Dateien.
- **Uhr:** Messung der vergangenen Zeit.

7.2.2 Komponenten des Chipmunk Physics Frameworks

Die Chipmunk Physics Library ist im Wesentlichen ein Framework zur Simulation von Interaktionen zwischen starren Körpern im 2-dimensionalen Raum. Im folgenden Teil sind die Kernfunktionalitäten des Chipmunk Frameworks beschrieben:

- **Kollisionserkennung:** Erkennen von Kollisionen zwischen zwei oder mehreren Objekten. Registrieren eines Callbacks als Folge einer Kollision.
- **Kollisionssimulation:** Simulieren von Reaktionen als Folge von Kollisionen. (resultierende Kräfte und Geschwindigkeiten)
- **Flexible optimierte Simulation:** Möglichkeit statische, (unbewegliche) kinetische (Code gesteuerte) und dynamische (physikgesteuerte) Körper zu definieren und dynamisch mit Kräften und Geschwindigkeiten zu versehen.
- **Verbindungen:** Möglichkeit Verbindungen wie Gelenke und Rotationsachsen zwischen Körpern zu definieren.

7.3 Umsetzung

Im folgenden Teil sind die Implementationsdetails der 2D Game SDK beschrieben.

7.3.1 UML Klassendiagramm: Core Package

Das folgende UML Klassendiagramm zeigt die Klassen aus dem «Core» Package im Detail: Der folgende Teil beschreibt die wichtigsten Klassen des «Core» Packages und deren Kernfunktionalität:

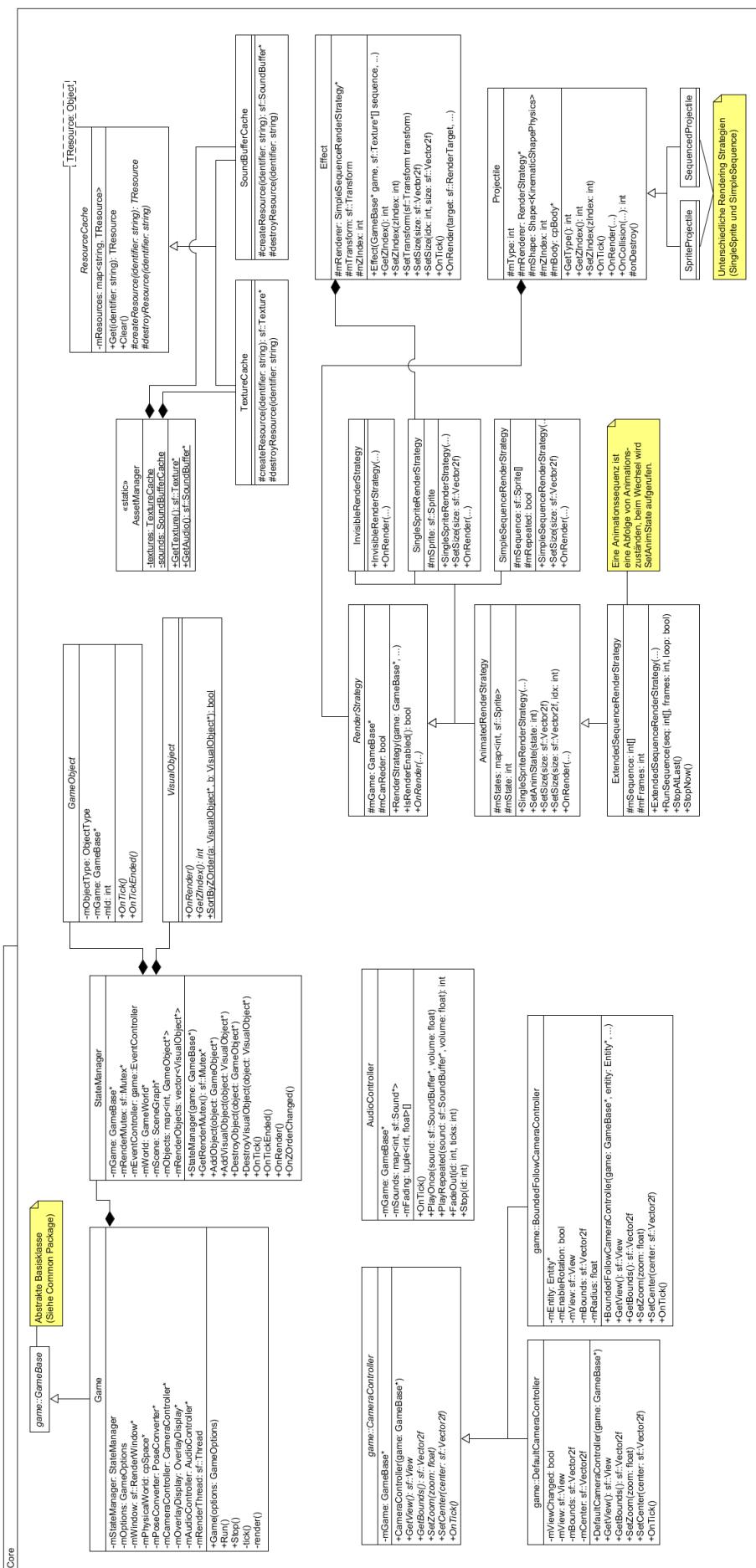


Abbildung 14 UML Klassendiagramm: Core Package

- **Game:** Die Game Klasse ist die zentrale Schnittstelle für alle Klassen der Game Engine, sie ist zuständig für die Spiel Initialisierung und für die Durchführung des Game Loops, der den gesamten Spielablauf regelt. Die Game Klasse stellt den StateManager, den CameraController, den AudioController und weitere Spielelemente bereit.
- **StateManager:** Die StateManager Klasse ist hauptsächlich für die Verwaltung des Spielzustands zuständig, der Spielzustand ist im Wesentlichen definiert durch die Spielobjekte, die visuellen Objekte und die Spielwelt. Eine wichtige Funktion des StateManagers ist die Synchronisation des Game Loops mit dem Rendering Loop, letzterer läuft aus Performancegründen in einem separaten Thread. (Siehe Kapitel 6.5.2) Der StateManager stellt eine Methode zum Löschen von Spielobjekten bereit, da die Objekte, die mit dieser Methode als gelöscht markiert wurden erst am Ende eines Zyklus gelöscht werden, ist es so auch möglich, dass ein Spielobjekt sich selbst löschen kann.
- **CameraController:** Die CameraController Klasse ist für die Steuerung der Kamera im Spiel zuständig. Für die Klasse werden von der Game Engine unterschiedliche Erweiterungen bereitgestellt, zusätzliche Erweiterungen können auch vom Spielentwickler selbst erstellt werden. (Siehe Beispiel in Kapitel 7.4.1)
- **GameObject:** Die GameObject Klasse ist die Basisklasse für alle Objekte im Spiel. Die Klasse stellt die Template Methode «OnTick» bereit, diese wird jeweils einmal pro Zyklus des Game Loops ausgeführt. Mit der «OnTick» Methode kann für ein Objekt ein dynamisches Verhalten definiert werden, sodass das Objekt (u. A.) auf Änderungen des Spielzustands reagieren kann.
- **VisualObject:** Die VisualObject Klasse ist die Basisklasse von allen visuellen Objekten im Spiel. Die Klasse stellt die Template Methode «OnRender» bereit, diese wird jeweils einmal pro Zyklus im Game Loop ausgeführt. Die «OnRender» Methode wird verwendet, um das visuelle Objekt darzustellen.
- **RenderStrategy:** Die RenderStrategy Klasse stellt durch dessen Erweiterungen unterschiedliche Rendering Algorithmen mit unterschiedlicher Komplexität bereit. Die wichtigsten Rendering Strategien sind unter anderem die SingleSpriteRenderStrategy (Objekt hat immer die gleiche Textur) und die AnimatedRenderStrategy (Objekt besitzt mehrere Texturen, die aktive Textur kann zur Laufzeit geändert werden) Die RenderStrategy Klasse wird verwendet, um die Algorithmen zur Darstellung von visuellen Objekten zu kapseln.
- **Effect:** Die Klasse Effect ist ein Spielobjekt, das für die Darstellung von visuellen Effekten verwendet wird. Ein Effekt ist definiert durch eine Abfolge von einzelnen Bildern, sobald diese Abfolge durchgelaufen ist, wird der Effekt automatisch zerstört. (Effekte können aber auch so konfiguriert werden, dass sie endlos laufen) Effekte besitzen keinen physikalischen Körper, folglich sind keine Kollisionen mit Effekten möglich.
- **Projectile:** Die Klasse Projectile ist ein Spielobjekt, das sich in eine frei definierbare Richtung fortbewegt und bei Kollision mit einer Entität (siehe Klasse «Entity» in Kapitel 7.3.4) bei dieser Entität einen bestimmten Callback auslöst. Wenn nichts anderes konfiguriert ist, zerstört sich ein Projektil selbst sobald es mit einem anderen Objekt kollidiert. Ein Projektil kann auch eine maximale Lebensdauer haben, was bedeutet, dass das Projektil nach Ablauf der Lebenszeit automatisch zerstört wird.

7.3.2 UML Klassendiagramm: World Package

Das folgende UML Klassendiagramm zeigt die Klassen aus dem «World» Package im Detail:

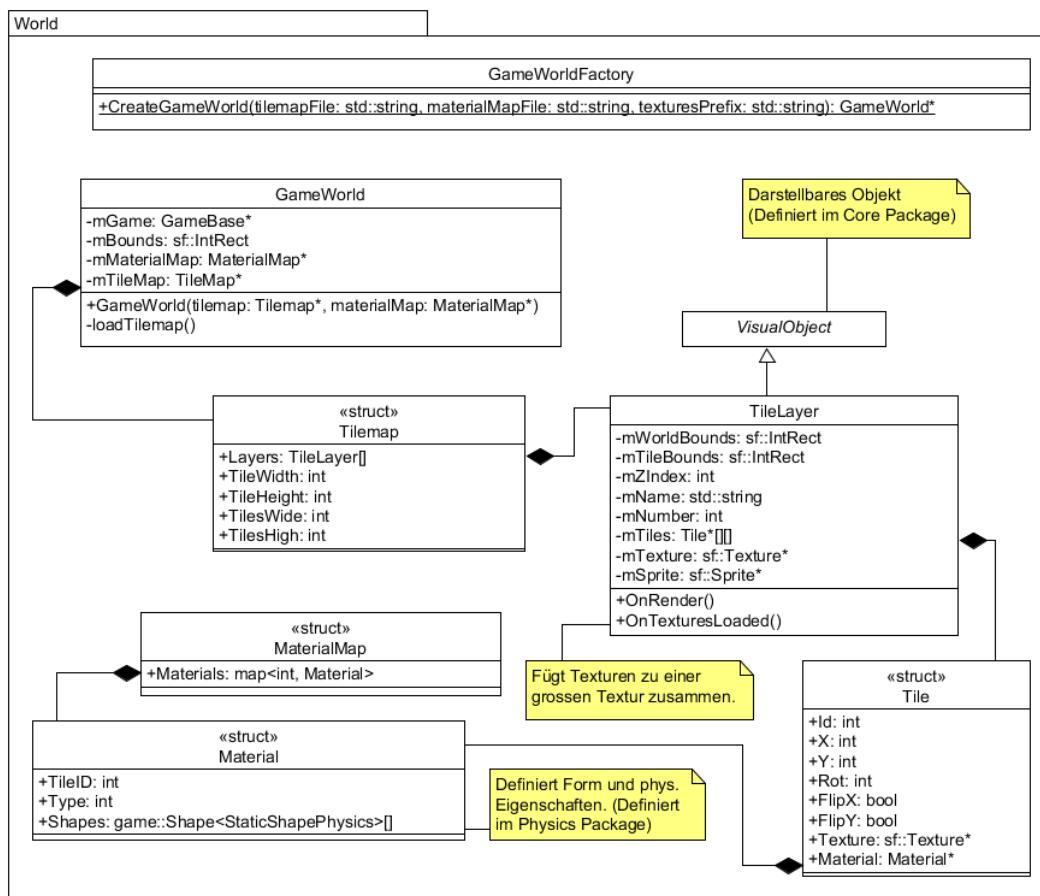


Abbildung 15 UML Klassendiagramm: World Package

Der folgende Teil beschreibt die wichtigsten Klassen des «World» Packages und deren Kernfunktionalität:

- **Tilemap**: Die Klasse Tilemap definiert eine mehrschichtige 2-dimensionale Anordnung von Tiles, welche die visuelle Repräsentation einer Spielwelt definieren. Die Kacheln können auch mit einer Materialdefinition erweitert werden, hierzu wird die Klasse MaterialMap verwendet.
- **TileLayer**: Die Klasse TileLayer definiert eine einzelne Schicht einer Tilemap. Die Klasse TileLayer ist eine Erweiterung der Klasse VisualObject und somit ein Element, das auf dem Bildschirm gezeichnet werden kann.
- **MaterialMap**: Die Klasse MaterialMap definiert die Zuordnung der Tiles zu einem Material. Ein Material hat in diesem Kontext physikalische Eigenschaften sowie eine oder mehrere physikalische Formen. Dass ein Material mehrere Formen enthalten kann ist notwendig, damit die Spielwelt auch komplexere geometrische Formen unterstützt.
- **GameWorld**: Die Klasse GameWorld definiert eine Spielwelt. Eine Spielwelt besteht aus einer Tilemap und einer MaterialMap.
- **GameWorldFactory**: Die Klasse GameWorldFactory stellt Methoden zum Erstellen einer Spielwelt bereit.

7.3.3 UML Klassendiagramm: Physics Package

Das folgende UML Klassendiagramm zeigt die Klassen aus dem «Physics» Package im Detail:

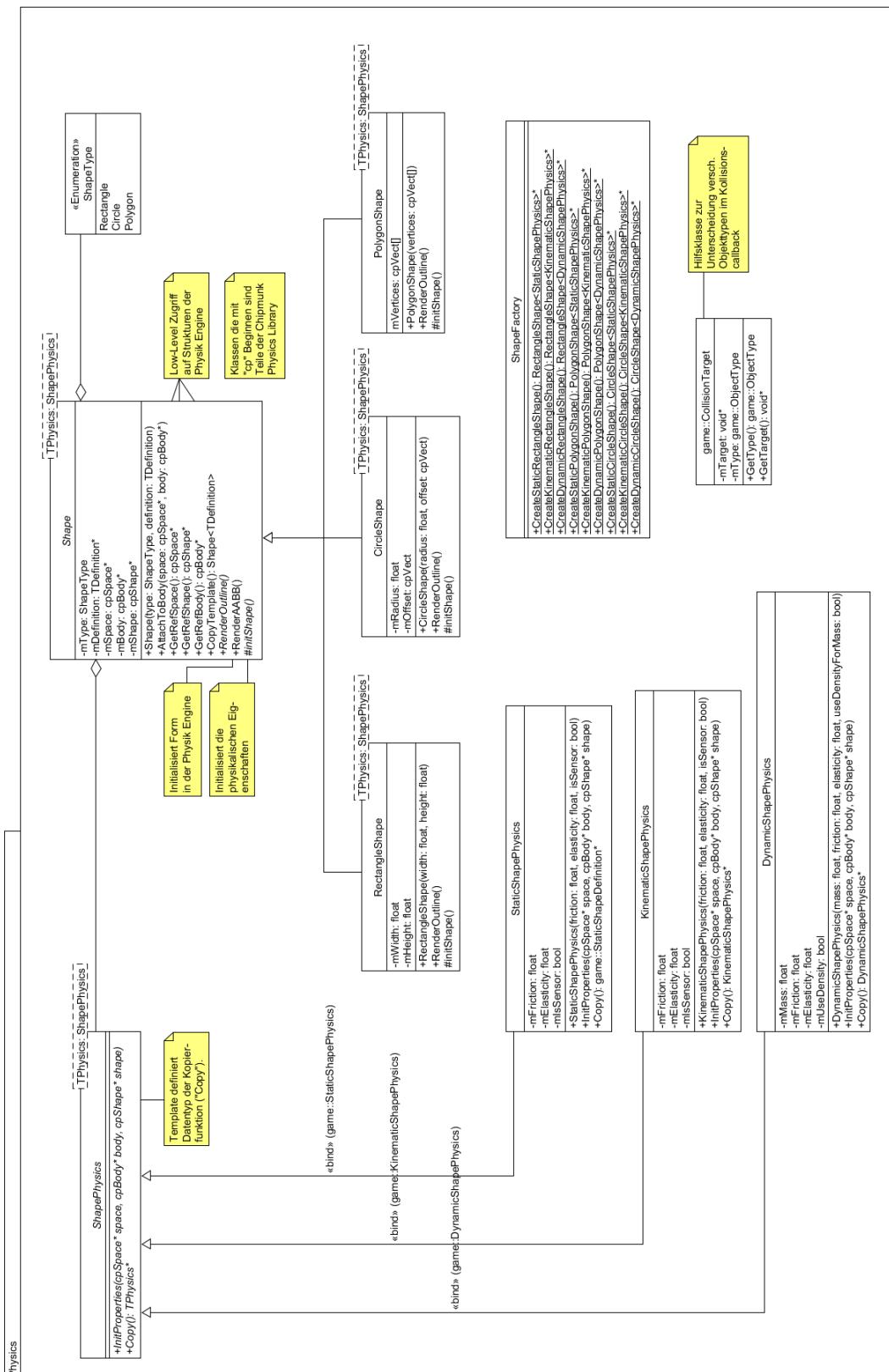


Abbildung 16 UML Klassendiagramm: Physics Package

Der folgende Teil beschreibt die wichtigsten Klassen des «Physics» Packages und deren Kernfunktionalität:

- **Shape:** Die Klasse Shape definiert eine Form, die Teil eines starren Körpers ist. Ein starrer Körper besteht aus einer oder mehreren Formen, diese Formen sind definiert durch deren Typ (siehe Erweiterungen StaticShape, KinematicShape und DynamicShape), Attribute, welche die Form selbst beschreiben (Form Art und Dimensionen) und aus physikalischen Attributen, wie z. B. Masse und Elastizität.
- **ShapePhysics:** Die Klasse ShapePhysics definiert eine Strategie zum Setzen der physikalischen Eigenschaften einer Form eines starren Körpers. Die Klasse besitzt für jeden Typ von starren Körper (statisch, kinematisch, dynamisch) eine eigene Erweiterung. Die Klasse unterliegt dem Strategy Design Pattern und stellt die Algorithmen zum Erstellen der entsprechenden Formen in der Chipmunk Physik bereit.
- **StaticShapePhysics:** Die Klasse StaticShapePhysics wird verwendet, um eine Form eines statischen starren Körpers zu definieren. Statische Körper können keine dynamischen Kräfte erfahren und sollten nach der initialen Platzierung nicht mehr verschoben werden. Statische Körper belasten den Prozessor aufgrund ihrer Unbeweglichkeit i. d. R. deutlich weniger als kinematische oder dynamische Körper.
- **KinematicShapePhysics:** Die Klasse KinematicShapePhysics wird definiert, um eine Form eines kinematischen starren Körpers zu definieren. Kinematische Körper können selbst keine dynamischen Kräfte erfahren, sind aber im Gegensatz zu statischen Körpern mit effizienten Methoden bewegbar. Ein kinematischer Körper repräsentiert ein Objekt, das vom Code des Spielprogrammierers und nicht von der Physik Engine gesteuert wird. Die Masse eines kinematischen Körpers wird von der Physik Engine als unendlich gross angenommen, somit kann ein dynamischer Körper von einem kinematischen Körper weggestossen werden, jedoch nicht umgekehrt.
- **DynamicShapePhysics:** Die Klasse DynamicShapePhysics wird verwendet, um eine Form eines dynamischen starren Körpers zu definieren. Dynamische Körper erfahren dynamische Kräfte von anderen dynamischen oder kinematischen Körpern und können nicht direkt durch den Code des Spielprogrammierers bewegt werden (da dies im physikalischen Sinne einer Teleportation gleichkommt, vgl. [15]). Die Bewegung eines dynamischen Körpers ist nur indirekt durch Setzen der Geschwindigkeit oder Kraft möglich.
- **ShapeFactory:** Die Klasse ShapeFactory stellt verschiedene Methoden zum Erstellen von Formen bereit. Für jede Permutation aus Form Art (Rechteck, Polygon, Kreis) und physikalischem Typ (statisch, kinematisch, dynamisch) wird eine Methode zum Erstellen der entsprechenden Form bereitgestellt.

7.3.4 UML Klassendiagramm: Scene Package

Das folgende UML Klassendiagramm zeigt die Klassen aus dem «Scene» Package im Detail:

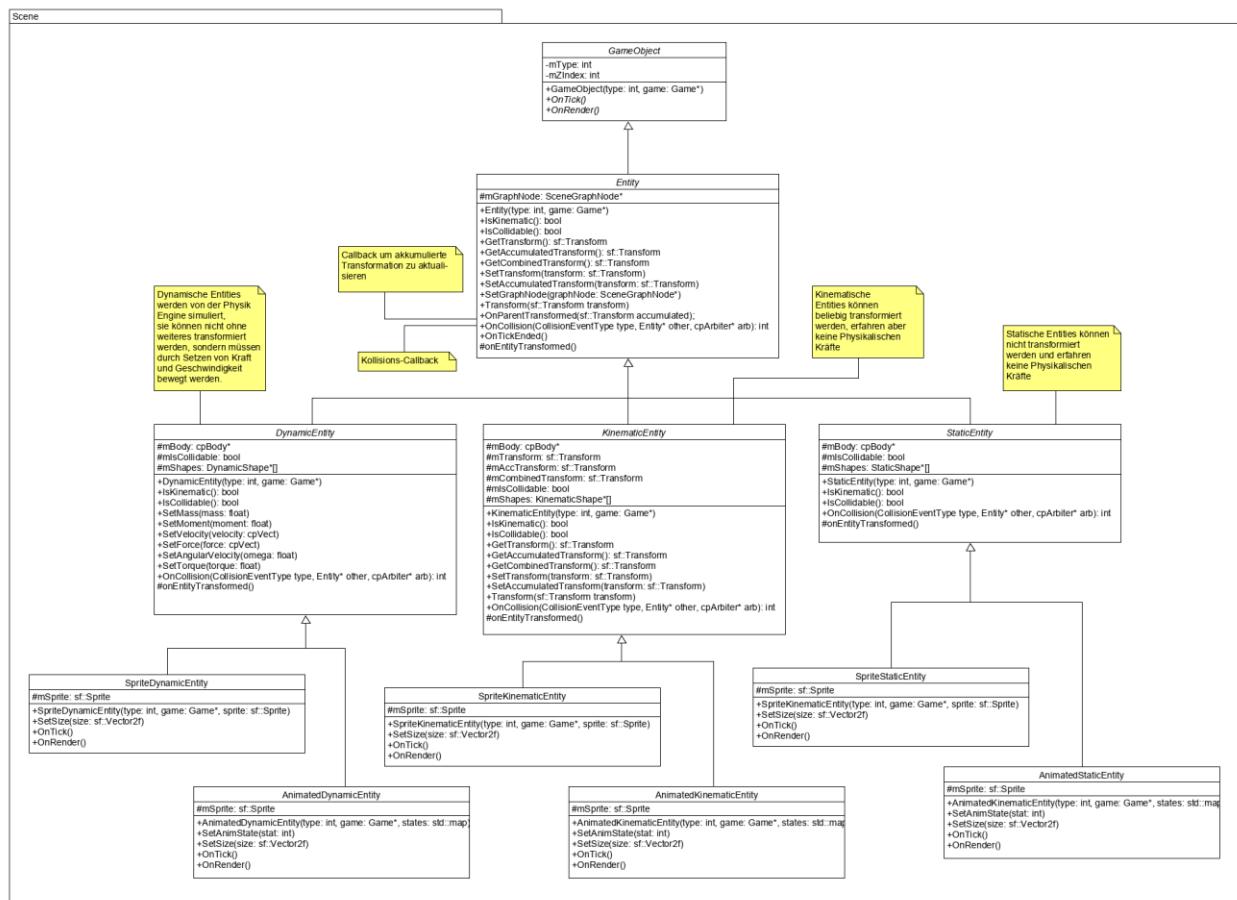


Abbildung 17 UML Klassendiagramm: Scene Package

Der folgende Teil beschreibt die wichtigsten Klassen des «Scene» Packages und deren Kernfunktionalität:

- **Entity:** Die Klasse Entity definiert ein Spielobjekt, das entweder durch die Physik Engine oder durch den Code des Spielprogrammierers bewegt werden kann. Entitäten (Instanzen der Entity Klasse oder deren Erweiterungen) implementieren eine Methode zum Erhalten der aktuellen Pose (bestehend aus Position und Rotation) und optional auch eine Methode zum Setzen der aktuellen Pose.
- **StaticEntity:** Die Klasse StaticEntity definiert eine Entität mit einem statischen und unbeweglichen Körper. Instanzen des Typs StaticEntity können nur Formen die als statisch definiert sind enthalten. Statische Entitäten sollten folglich nach der initialen Platzierung nicht mehr verschoben werden.
- **KinematicEntity:** Die Klasse KinematicEntity definiert eine Entität mit einem kinematischen, codegesteuerten Körper. Instanzen des Typs KinematicEntity können nur Formen die als kinematisch definiert sind enthalten. Kinematische Entitäten werden direkt vom Spielprogrammierer (durch Setzen der Position) gesteuert.
- **DynamicEntity:** Die Klasse DynamicEntity definiert eine Entität mit einem dynamischen physikgesteuerten Körper. Instanzen des Typs DynamicEntity können nur Formen die als dynamisch definiert sind enthalten. Dynamische Entitäten werden von der Physik Engine gesteuert, der Spieleprogrammierer hat aber die Möglichkeit die Bewegung durch Setzen der Geschwindigkeit und Kraft zu beeinflussen.

7.3.5 UML Klassendiagramm: Event Package

Das folgende UML Klassendiagramm zeigt die Klassen aus dem «Event» Package im Detail:

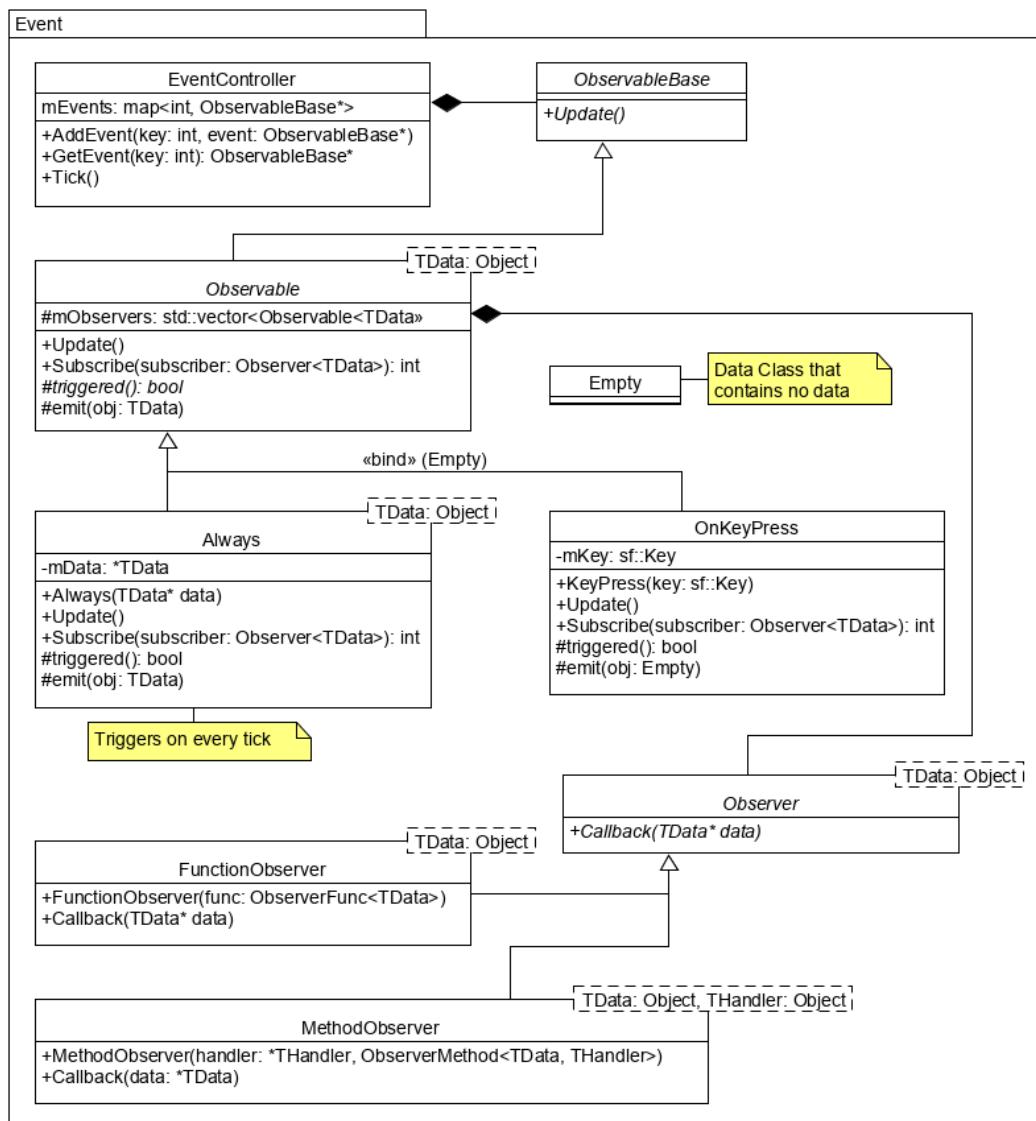


Abbildung 18 UML Klassendiagramm: Event Package

Der folgende Teil beschreibt die wichtigsten Klassen des «Event» Packages und deren Kernfunktionalität:

- **Observable**: Die Observable Klasse definiert im Kontext der Game Engine ein Ereignis. Ein Ereignis kann zum Beispiel eine Tastatur- oder Mauseingabe des Spielers sein, oder auch ein komplexes frei definierbares Ereignis, wie zum Beispiel, wenn der Spielercharakter eine bestimmte Zone betreten hat. Ereignisse sind definiert als Observables gemäss dem Observer/Observable Design Pattern. Um ein Ereignis mit einer bestimmten Reaktion zu verknüpfen, registriert man einen Observer mit einer entsprechenden Callback Methode bei diesem Ereignis.
- **Observer**: Die Observer Klasse definiert im Kontext der Game Engine eine Reaktion auf ein bestimmtes Ereignis. Ein Observer implementiert eine Callback Methode, die bei Auslösung des Ereignisses ausgeführt wird. Die Observer Klasse ist als abstrakt definiert und wird durch die Klassen FunctionObserver und MethodObserver konkretisiert.

- **FunctionObserver:** Die Klasse FunctionObserver definiert einen Observer mit einer Callback Methode im globalen Scope. Bei Aufruf der Funktion sind nur die Variablen im globalen Scope verfügbar. Wenn ein Observer dynamische Zustandsvariablen verwalten oder bereitstellen muss, wird empfohlen statt der Klasse FunctionObserver die Klasse MethodObserver zu verwenden.
- **MethodObserver:** Die Klasse MethodObserver definiert einen Observer mit einer Callback Methode innerhalb einer Klasse. Eine Instanz der Klasse wird bei der Erstellung des Observers zugewiesen, dieser hat danach Zugriff auf die öffentlichen Funktionen und Variablen dieser Instanz.
- **EventController:** Die Klasse EventController verwaltet die verschiedenen Ereignisse in einem Spiel. Der EventController garantiert, dass jedes registrierte Ereignis einmal pro Zeitschritt an einem fest definierten Zeitpunkt geprüft wird. Erst zu diesem Zeitpunkt kann ein ausgelöstes Ereignis die registrierten Observer benachrichtigen, die dann entsprechend auf das Ereignis reagieren.

7.3.6 UML Sequenzdiagramm: Initialisierung

Aus Platzgründen wurde das Sequenzdiagramm der Spielinitialisierung in zwei Phasen unterteilt, die erste Phase zeigt wie das Spiel, die Spielwelt, der StateManager und die Spielobjekte erstellt werden:

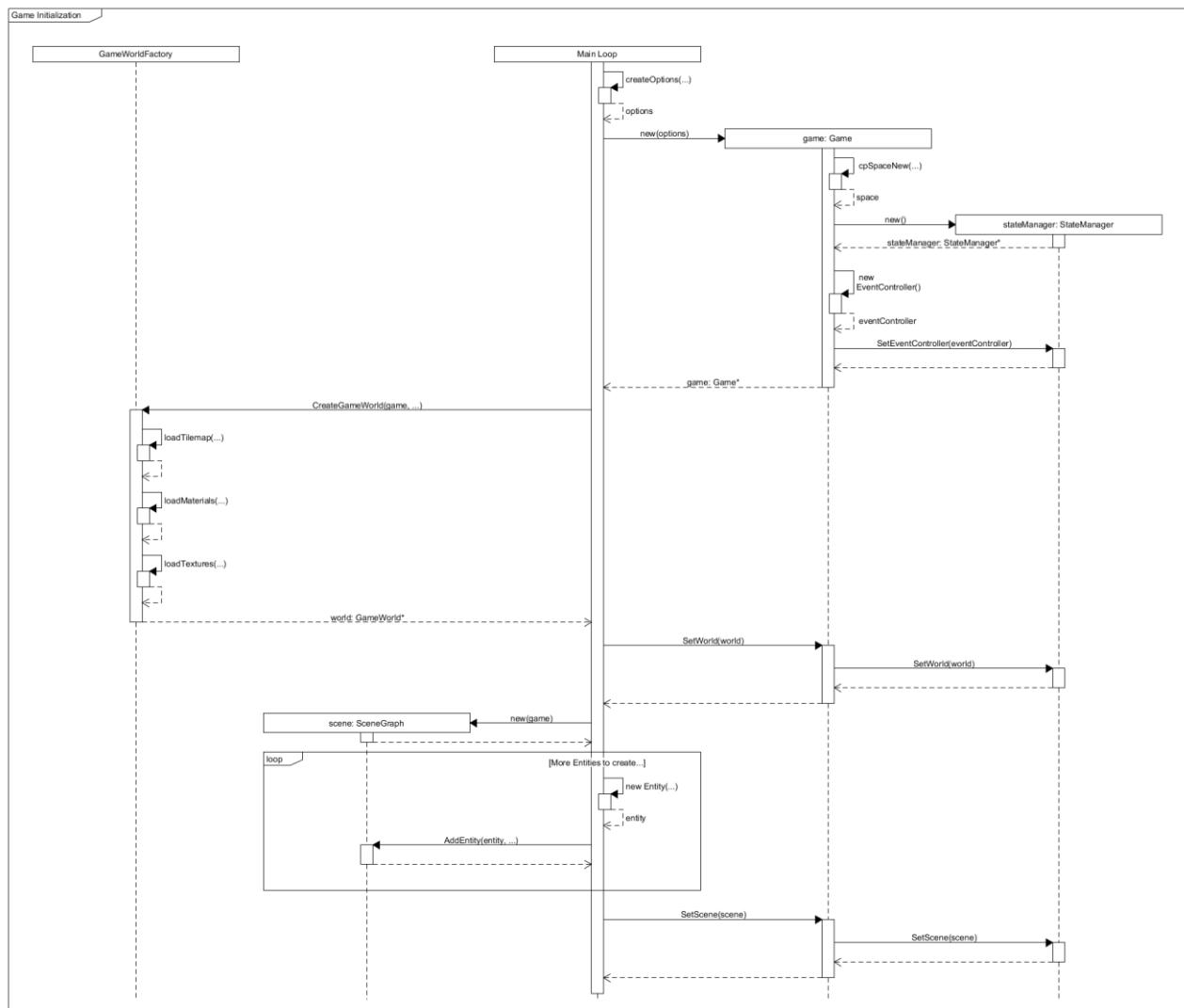


Abbildung 19 UML Sequenzdiagramm: Spiel Initialisierung (Teil 1)

In der zweiten Phase werden alle Events registriert und das Spiel wird gestartet:

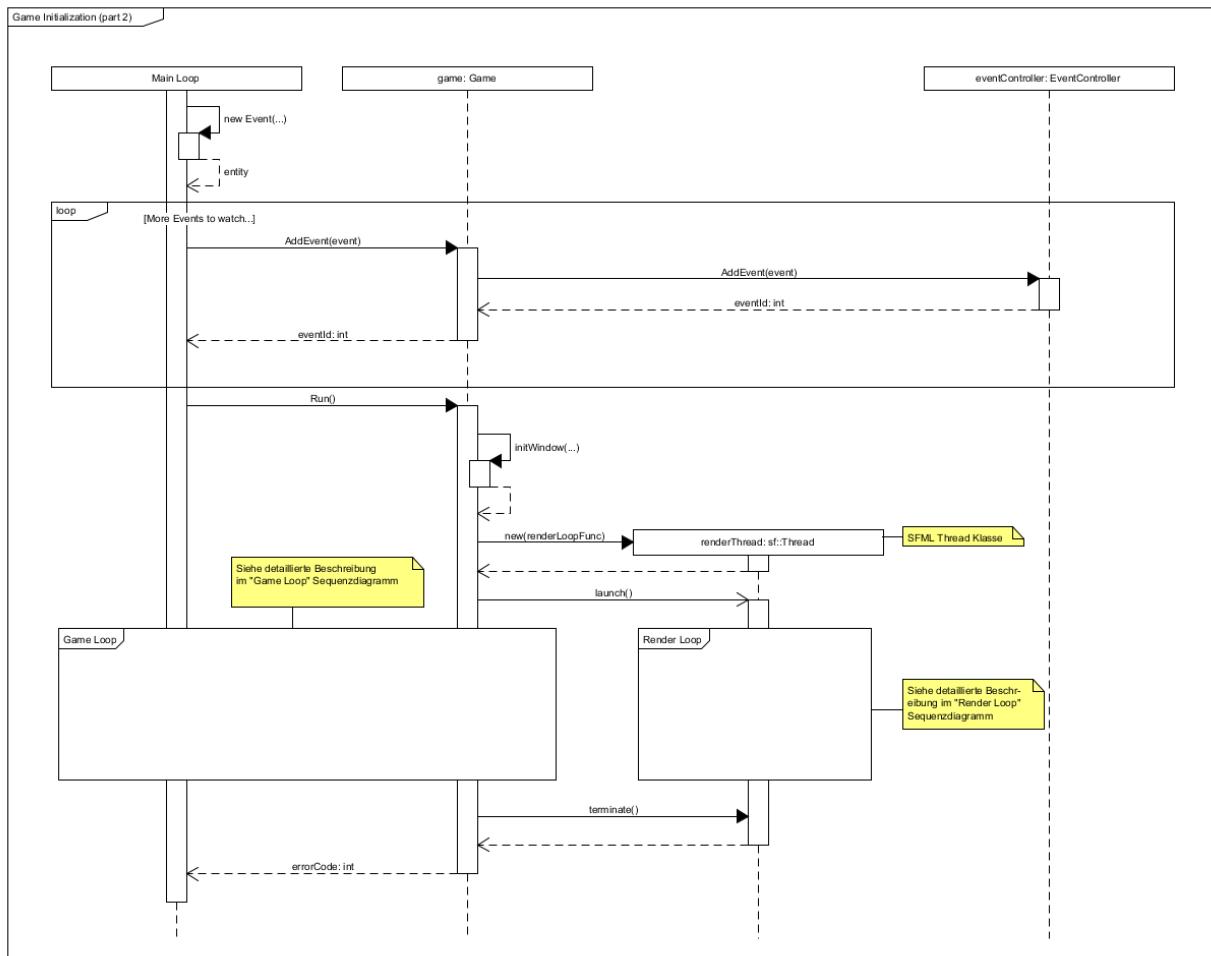


Abbildung 20 UML Sequenzdiagramm: Spiel Initialisierung (Teil 2)

7.3.7 UML Sequenzdiagramm: Game Loop

Der Game Loop sorgt dafür, dass das Spiel fortlaufend aktualisiert wird:

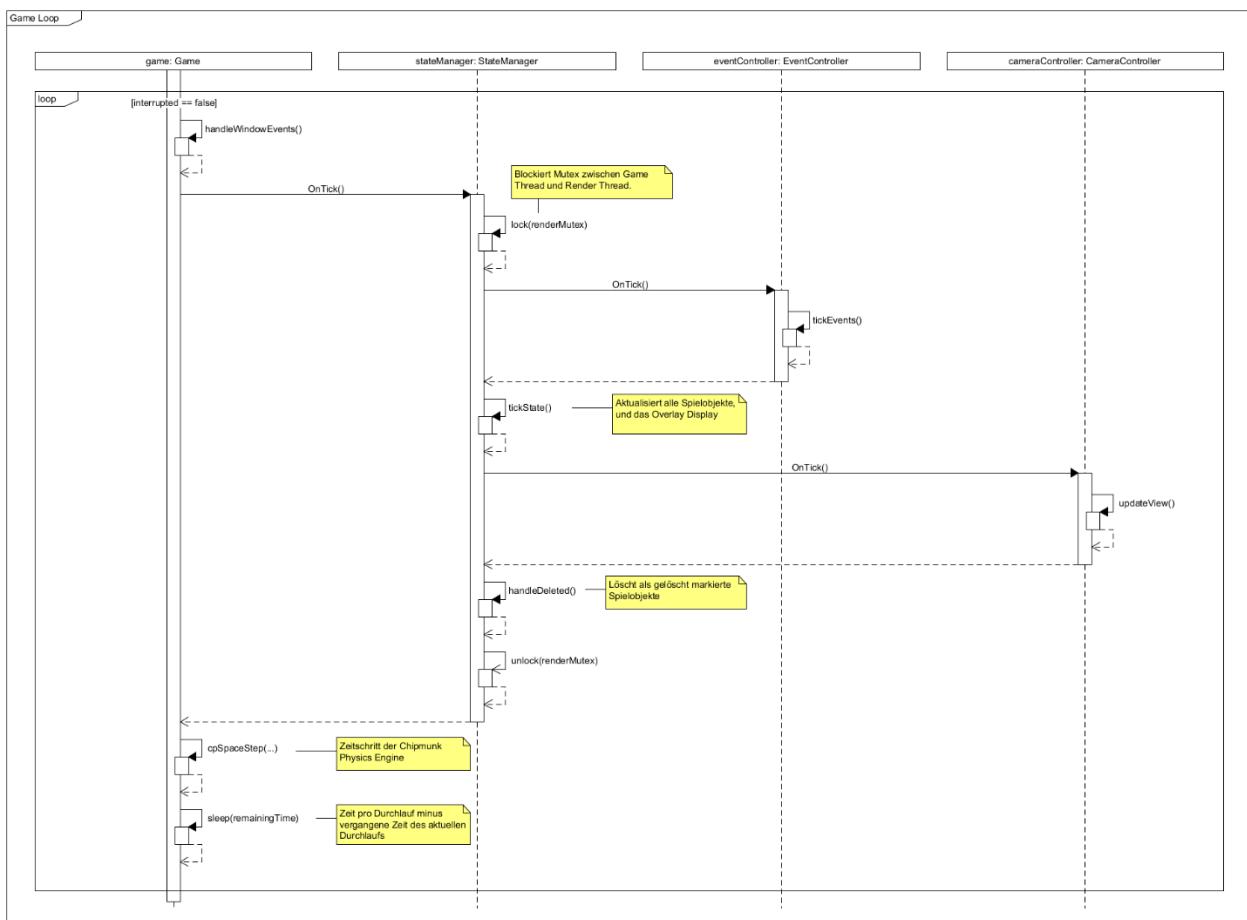


Abbildung 21 UML Sequenzdiagramm: Game Loop

7.3.8 UML Sequenzdiagramm: Render Loop

Der Render Loop sorgt dafür, dass das Spiel fortlaufend neu dargestellt wird:

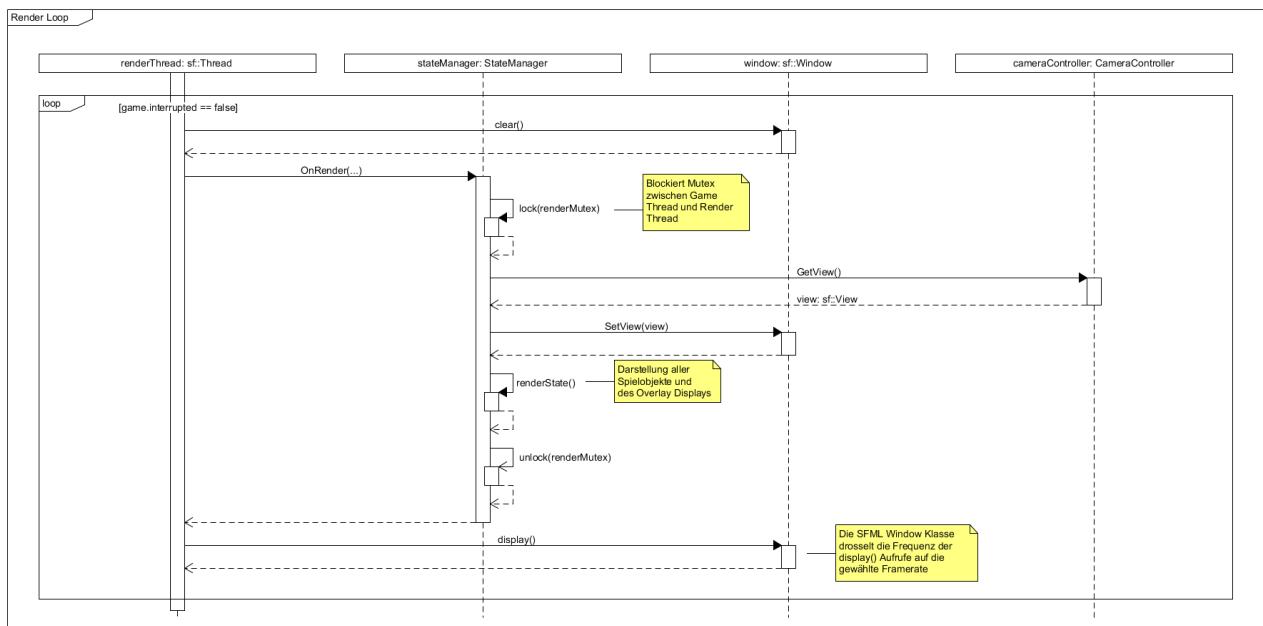


Abbildung 22 UML Sequenzdiagramm: Render Loop

7.4 Testing

Um die Funktionalitäten der Game Engine zu testen und um zu verifizieren ob die Game Engine in der Praxis auch funktioniert und einen Mehrwert bietet, wurden im Rahmen dieser Thesis zwei unterschiedliche Prototypen erstellt. Beide Prototypen unterscheiden sich anhand des Spieltyps und anhand der Technologien, die für den Aufbau der Spiellogik eingesetzt wurden. Die folgenden Unterkapitel beschreiben diese beiden Prototypen im Detail.

7.4.1 Prototyp 1: Arcade Shooter Game

Der erste Prototyp, der zum Testen und Demonstrieren der Game Engine erstellt wurde, ist ein Prototyp eines typisches Arcade Shooter Spiels im Weltraum Thema. Im Spiel kontrolliert der Spieler ein Raumschiff, das in der Lage ist sich in alle Richtungen zu bewegen (4-Achsen Bewegung) und Projektil auf feindliche Raumschiffe zu feuern. Während dem Spiel erscheinen fortlaufend feindliche Raumschiffe, die selbst in regelmässigen Abständen Projektil in die Richtung des Spielers feuern und so versuchen das Raumschiff des Spielers zu zerstören. Der Prototyp implementiert folgende Funktionalitäten:

- **Projektile:** Sowohl der Spieler als auch die feindlichen nicht-Spieler Charaktere sind in der Lage Projektil zu feuern. Die Projektil sind animiert und ändern ihr Erscheinungsbild bei jedem neuen Frame.
- **Verwundbarkeit:** Sowohl der Spieler als auch die feindlichen nicht-Spieler Charaktere sind verwundbar und ändern kurzzeitig die Farbe, wenn sie getroffen wurden. Sie haben eine fest definierte Anzahl Lebenspunkte und werden zerstört, wenn diese bei 0 ist.
- **Effekte:** Sowohl der Spieler als auch die feindlichen nicht-Spieler Charaktere erzeugen einen Explosionseffekt, wenn sie zerstört werden.
- **Bewegungssequenzen:** Die feindlichen Raumschiffe bewegen sich anhand einer frei definierbaren Bewegungssequenz, wie zum Beispiel wiederholt nach links und nach rechts.

- **Kamera:** Für die Steuerung der Spielerkamera wurde ein einzigartiger CameraController implementiert, der sowohl die Funktionalitäten einer Scrollenden (sich stetig in eine Richtung bewegen) Kamera implementiert, als die Funktionalitäten einer Entität, (siehe Klasse «Entity») die hierarchische geometrische Transformationen unterstützt. Letztere ermöglichen es, dass das Raumschiff des Spielers die geometrischen Transformationen der Kamera erbt und sich so automatisch mit der Kamera bewegt. Die Steuerung des Spielers ist somit immer relativ zur Bewegung der Kamera.
- **Begrenzungen:** Um zu verhindern, dass das Raumschiff des Spielers den Fokus der Kamera verlassen kann, wird im lokalen Koordinatensystem der Kamera geprüft, ob die Bewegungen des Spielers die Begrenzungen der Kamera übertreten. Wenn die aktuelle Bewegung des Spielers die Begrenzungen der Kamera übertreten würde, wird der Bewegungsvektor so korrigiert, dass die Übertretung verhindert wird. Diese Implementation funktioniert unabhängig von der Physik Engine mit einem Algorithmus, der allein auf Geometrischen Vergleichen basiert. Eine Implementation basierend auf der Physik Engine wird im zweiten Prototyp umgesetzt.
- **Punktanzeige:** Mit der Overlay Display Funktionalität der Game Engine wurde eine Statusanzeige implementiert, die über die gesamte Zeit des Spiels die aktuelle Punktzahl des Spielers anzeigt. Jedes zerstörte feindliche Raumschiff fügt diesem Punktestand eine gewisse Anzahl Punkte hinzu. (Die Anzahl ist abhängig vom Typ des feindlichen Raumschiffs)
- **Lebensanzeige:** Mit der Overlay Display Funktionalität der Game Engine wurde eine Statusanzeige implementiert, die über die gesamte Zeit des Spiels die verbleibenden Leben des Spielers anzeigt. Der Spieler hat zu anfangs drei Leben, sind diese verbraucht, hat der Spieler das Spiel verloren.
- **Spawn Sequenz:** Die feindlichen Raumschiffe werden anhand einer zeitgesteuerten Logik erstellt bevor sie in den Fokus der Kamera kommen und zerstört sobald sie den Fokus der Kamera wieder verlassen. Es wurde eine zeitgesteuerte Logik verwendet, um die Implementation des Prototyps einfach zu halten. Eine bessere, aber auch komplexere Lösung wäre anhand eines Vergleichs der geometrischen Positionen (analog zur Logik der Kamerabegrenzung des Spielers) zu implementieren.
- **Endgegner:** Wenn der Spieler den gesamten Level durchlaufen hat, stoppt die Kamera und es erscheint ein Endgegner. Der Endgegner hat deutlich mehr Trefferpunkte als die einfachen Gegner im Spiel und kann anders als die einfachen Gegner mehrere Arten von Projektilen mit unterschiedlichen Frequenzen abfeuern. Der Endgegner bewegt sich in einem rechteckigen Muster mit der zuvor erläuterten Logik der Bewegungssequenzen.
- **Abschlussbedingung:** Wenn das Raumschiff des Spielers zerstört wurde, erscheint eine «Game Over» Anzeige, die signalisiert, dass das der Spieler das Spiel verloren hat. Die Kamera bewegt sich an diesem Punkt nicht mehr weiter.



Abbildung 23 Screenshot des ersten Prototyps (Farben wurden verändert um den Schwarzanteil zu reduzieren)

7.4.2 Prototyp 2: Roleplay Game

Der zweite Prototyp, der zum Testen und Demonstrieren der Game Engine erstellt wurde, ist ein Prototyp eines typischen 2D Rollenspiels im mittelalterlichen Thema. Im Spiel sieht der Spieler die Spielwelt aus einer leicht abgeflachten Vogelperspektive und kontrolliert einen Helden, der sich mit einer animierten Laufbewegung durch ein Dorf (bzw. die Spielwelt) bewegen kann. Der Prototyp implementiert die folgenden Funktionalitäten:

- **Nicht-Spieler Charakter:** Der Prototyp enthält einen nicht-Spieler Charakter, der sich zufällig durch den Raum bewegt. Wenn sich der Spieler in der Nähe dieses nicht-Spieler Charakters befindet, ist der Spieler in der Lage eine Dialogbox zu öffnen, in der ein zufällig ausgewählter Text angezeigt wird. Wenn die Dialogbox offen ist, sind sowohl der Spieler als auch der nicht-Spieler Charakter immobil und der nicht-Spieler Charakter schaut in die Richtung des Spielers.
- **Animierte Laufbewegung:** Sowohl der Spieler als auch der nicht-Spieler Charakter spielen eine entsprechende Laufanimation ab, wenn sie sich durch den Raum bewegen. Die Laufanimation erzeugt einen Effekt, der den Anschein macht, als ob die Charaktere die Beine bewegen, um sich fortzubewegen. Wenn die Charaktere stillstehen, nehmen sie eine ruhende Position ein.
- **Nicht-passierbare Objekte:** Die Spielwelt enthält Objekte, wie z. B. Mauern, Tische oder Fässer, die vom Spieler nicht passiert werden können. Dieses dynamische Verhalten basiert auf der Simulation von dynamischen Körpern innerhalb der Physik Engine: Der Spieler ist ein dynamischer Körper, die Spielwelt besteht aus statischen Körpern, die eine feste Grenze darstellen.

- **2.5-dimensionales Rendering:** Für den Prototyp wurde das Layering System der Spielwelt (Tilemap und Tilelayer) erweitert, sodass die Spielwelt in mehrere horizontale Reihen aufgeteilt werden kann. Jeder Reihe kann so ein anderer Z-Index zugeordnet werden, (der Z-Index bestimmt die Reihenfolge beim Rendering) so wird ermöglicht dass die Spielwelt sich fast so verhält wie eine 3-dimensionale Spielwelt: Wenn der Spieler sich in der Reihe unterhalb eines Objekts der Spielwelt befindet, erhält dieser einen höheren Z-Index als das Objekt und wird deshalb so dargestellt, als ob es sich vor dem Objekt befindet. Wenn der Spieler sich in der Reihe oberhalb eines Objektes der Spielwelt befindet, erhält dieser einen kleineren Z-Index als das Objekt und wird deshalb so dargestellt, als ob er sich hinter dem Objekt befindet. Diese Logik funktioniert sowohl für die Interaktion der Spielcharakter mit der Spielwelt als auch bei der Interaktion zwischen den Spielcharakter.



Abbildung 24 Screenshot des zweiten Prototyps

7.4.3 Schlussfolgerungen

Die Prototypen zeigen, dass es gut möglich ist mit der Game Engine funktionsfähige Spiele zu erstellen, aus zeitlichen Gründen konnten aber keine komplexeren Prototypen erstellt werden. Bei der Erstellung der Prototypen wurde festgestellt, dass die von der Game Engine bereitgestellte Event Logik im praktischen Einsatz zu aufwändig ist im Vergleich zum Nutzen, den sie tatsächlich mit sich bringt. Für die Implementation der beiden Prototypen wurde deshalb vollständig auf die Event Logik verzichtet. Stattdessen wurde ein neues Konzept mit modularen Verhaltensdefinitionen implementiert, dieses hat in der Praxis einen grossen Nutzen gebracht und wird in zukünftigen Versionen der Game Engine auch integriert werden. Das folgende UML Klassendiagramm (Ausschnitt aus dem ersten Prototyp) zeigt wie durch Vererbung aus einer Kombination von modularen Verhaltensdefinitionen die Spieler- und Gegnerklasse erstellt werden:

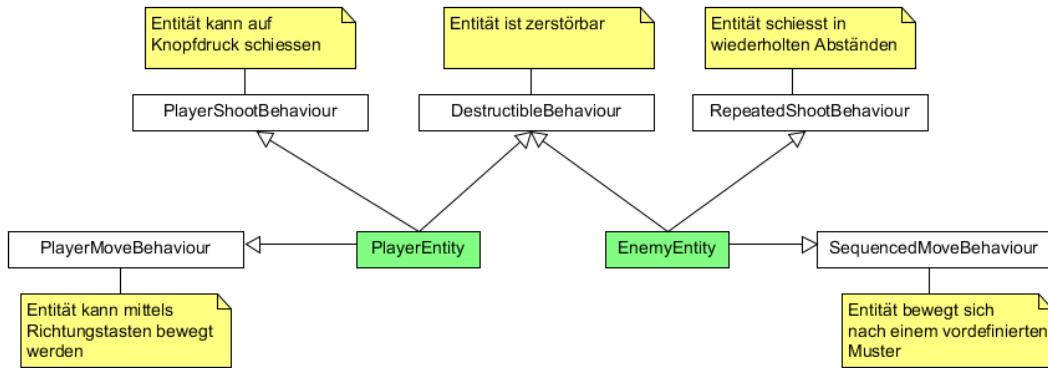


Abbildung 25 Demonstratives UML Klassendiagramm der Behaviour Logik des ersten Prototyps

Wie bereits in den Kapitel 7.4.1 und 7.4.2 angedeutet, basieren die beiden Prototypen auf grundlegend unterschiedlichen Technologien: Während für die Implementation des ersten Prototyps nur kinematische (codegesteuerte) Körper verwendet werden, basiert der zweite Prototyp auf einer Implementation, die dynamische (physikgesteuerte) Körper für die Spielcharaktere und statische Körper für die Spielwelt verwendet. Die beiden Prototypen zeigen somit, dass es sowohl möglich ist ein rein physikbasiertes Spiel zu machen oder ein Spiel, das nur vom Spielcode gesteuert wird. Eine Kombination aus beiden Technologien ist theoretisch auch möglich.

8 Projektmanagement

8.1 Zeitplan

8.1.1 Zeitplan gemäss Pflichtenheft (SOLL):

| | 16.09.2019 | 23.09.2019 | 30.09.2019 | 07.10.2019 | 14.10.2019 | 21.10.2019 | 28.10.2019 | 04.11.2019 | 11.11.2019 | 18.11.2019 | 25.11.2019 | 02.12.2019 | 09.12.2019 | 16.12.2019 | 23.12.2019 | 30.12.2019 | 06.01.2020 | 13.01.2020 |
|-------------------------------------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|
| Kickoff | | | | | | | | | | | | | | | | | | |
| Sprint 1: Physik POC, Kollisionsdetection | | | | | | | | | | | | | | | | | | |
| Sprint 2: Refactoring und Vorbereitungen | | | | | | | | | | | | | | | | | | |
| Sprint 3: Physik Engine, Kollisionen | | | | | | | | | | | | | | | | | | |
| Sprint 4: Kamera und Audio Rendering | | | | | | | | | | | | | | | | | | |
| Sprint 5: In-Game Overlay Display | | | | | | | | | | | | | | | | | | |
| Sprint 6: Shooter Prototyp | | | | | | | | | | | | | | | | | | |
| Sprint 7: Roleplay Prototyp | | | | | | | | | | | | | | | | | | |
| Sprint 8: Platformer Prototyp | | | | | | | | | | | | | | | | | | |
| MS1 01.11.19 | | | | | | | | | | | | | | | | | | |
| MS2 29.11.19 | | | | | | | | | | | | | | | | | | |
| MS3 14.06.19 (Projektende) | | | | | | | | | | | | | | | | | | |

Abbildung 26 Zeitplan gemäss Pflichtenheft

8.1.2 Zeitplan tatsächlich umgesetzt (IST):

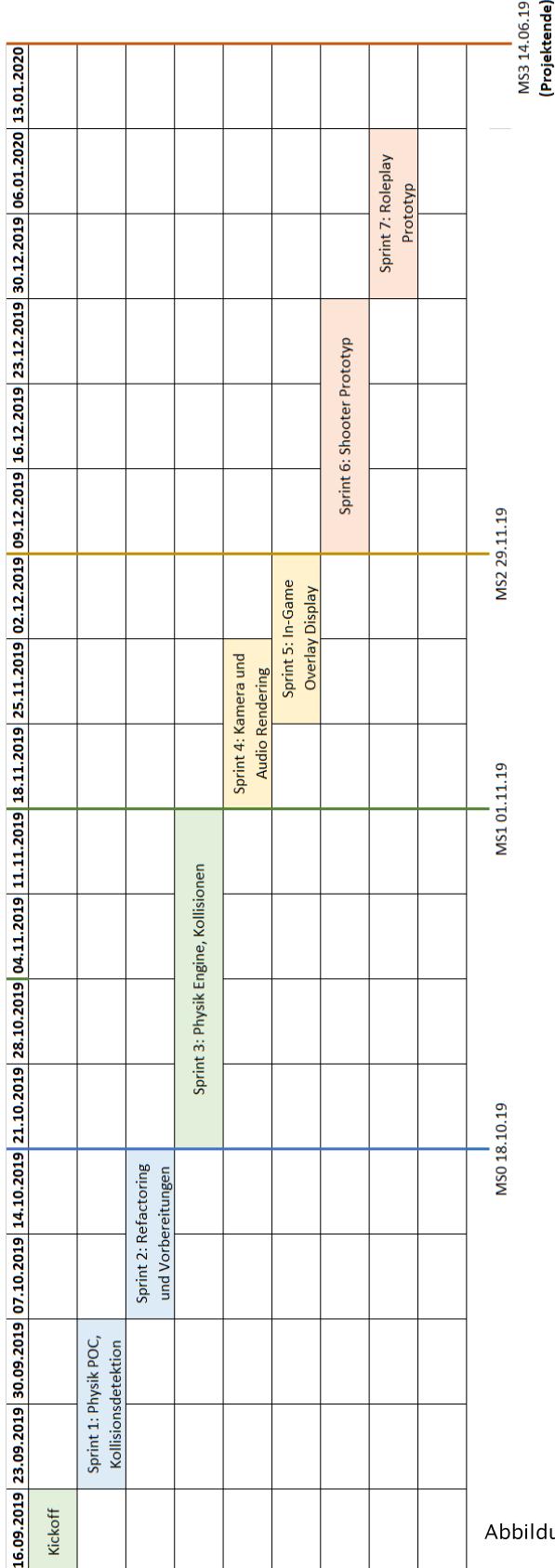


Abbildung 27 Zeitplan tatsächlich umgesetzt

8.1.3 Erläuterungen

Die Integration der Physik Engine war mit einem Sprint etwas zu knapp bemessen, aufgrund der Komplexität der Aufgabe und aufgrund ungeahnter Konsequenzen, konnte die Physik Engine Integration erst nach zwei Sprints abgeschlossen werden. Dies führte zu einer generellen Verzögerung von zwei Wochen. Aufgrund dieser Verzögerung, wurde später entschieden auf die Implementation eines dritten Prototyps zu verzichten. Die Zeit konnte zwar während der Implementation der Sprints 4 (Kamera und Audio) und 5 (In-Game Overlay Display) etwas aufgeholt werden, dennoch reichte die Zeit letztendlich nur für die Implementation von zwei der drei Prototypen.

8.2 Anforderungen

8.2.1 Funktionale Anforderungen

Im folgenden Teil sind die funktionalen Anforderungen gemäss Pflichtenheft [5] beschrieben, mit der Information, ob diese Anforderungen umgesetzt werden konnten oder nicht. Die Anforderungen sind unterteilt in die jeweiligen Subsysteme des Projekts:

Game Controller (Functional Game Controller)

Tabelle 7 Funktionale Anforderungen: Game Controller Subsystem

| ID | Prio | Beschreibung | Erfüllt? |
|-----|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| FG1 | Muss | Der Spieleanwickler möchte die Möglichkeit haben bei Kollisionen zwischen den Spielobjekten ein bestimmtes Verhalten zu definieren, sodass auf Basis der Kollisionserkennung eine komplexere Spiellogik realisiert werden kann. | Ja |
| FG2 | Muss | Der Spieleanwickler möchte die Möglichkeit haben Elemente eines In-Game Overlay Displays zu definieren, sodass globale Spielinformationen (z.B. Abgelaufene Levelzeit, Spielerleben) statisch auf dem Bildschirm dargestellt werden können. | Ja |
| FG3 | Muss | Der Spieleanwickler möchte die Möglichkeit haben, Audioeffekte und eine Hintergrundmusik zu definieren, sodass das Spielerlebnis mit akustischen Elementen erweitert werden kann. | Ja |
| FG4 | Muss | Der Spieleanwickler möchte die Möglichkeit haben ein Kameraverhalten zu definieren, sodass die Kamera während des Spielablaufs gesteuert werden kann. (Dies ermöglicht zum Beispiel, dass die Kamera dem Spieler folgen kann) | Ja |
| FG5 | Wunsch | Der Spieleanwickler möchte die Möglichkeit haben die Taktfrequenz des Spiels unabhängig von der Bildwiederholrate zu steuern, sodass mehrere Spielschritte pro Darstellungsschritt berechnet werden können und das Spiel somit präziser berechnet wird. | Ja |

Die funktionalen Anforderungen des Game Controller Subsystems konnten vollständig erfüllt werden, darüber hinaus wurden zusätzliche Funktionalitäten umgesetzt, die sich während der Implementation der Game Engine und der Prototypen als wichtig herausgestellt haben und nicht im Pflichtenheft spezifiziert waren:

- **Projektile:** Der Spieleanwickler hat die Möglichkeit animierte Projektile zu erstellen, die sich mit konstanter Geschwindigkeit in eine frei definierbare Richtung bewegen und bei Kollision mit einem Objekt ein entsprechendes Signal auslösen.

- **Effekte:** Der Spielentwickler hat die Möglichkeit animierte Effekte (wie z. B. Explosionen oder Feuer) darzustellen. Effekte haben keinen physikalischen Körper und zerstören sich selbst, wenn die Animation fertig ist (mit Ausnahme von endlosen Effekten, die manuell zerstört werden müssen). Event Controller (Functional Event Controller)
- **Rendering Strategien:** Der Spielentwickler hat die Möglichkeit individuelle Rendering Strategien für die Darstellung der verschiedenen Spielobjekte zu definieren oder eine der vorgefertigten Rendering Strategien der Game Engine zu verwenden.

Event Controller (Functional Event Controller)

Tabelle 8 Funktionale Anforderungen: Event Controller Subsystem

| ID | Prio | Beschreibung | Erfüllt? |
|-----|--------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| FE1 | Muss | Der Spieleanwickler möchte das Ereignis , dass ein Spielobjekt zerstört wurde, mit möglichen Aktionen im Spiel verknüpfen können, sodass mit diesem Ereignis eine komplexere Spiellogik definiert werden kann. | Nein |
| FE2 | Muss | Der Spieleanwickler möchte das Ereignis , dass ein Spielobjekt eines bestimmten Typs erstellt wurde, mit möglichen Aktionen im Spiel verknüpfen können, sodass mit diesem Ereignis eine komplexere Spiellogik definiert werden kann. | Nein |
| FE3 | Wunsch | Der Spieleanwickler möchte die Möglichkeit haben Bereiche zu definieren, die beim Betreten durch Spielobjekte ein Ereignis auslösen , sodass mit diesem Ereignis eine komplexere Spiellogik definiert werden kann. | Nein |

Die Anforderungen des Event Controller Subsystems wurden während der Implementation der Game Engine aus den folgenden Gründen verworfen:

- Während der Implementation der Prototypen hat sich herausgestellt, dass das Event Controller Subsystem nur bedingt praxistauglich ist und in vielen Fällen einen zu grossen Overhead generiert. Es gibt jedoch Anwendungsfälle, in denen das Event Controller Subsystem sinnvoll ist, zum Beispiel wenn Ereignisse definiert werden müssen, die den gesamten Spielzustand betreffen. (z. B. Spielabschluss Ereignisse wie Game Over)
- Die Game Engine verwendet das Event Controller Subsystem intern nicht, sie implementiert für die Verarbeitung von Ereignissen einen Ansatz mit polymorphen Callback Methoden, der sich in der Praxis als performanter herausgestellt hat. Ein Beispiel hierfür sind die «OnCollision» Callbacks, welche signalisieren, dass ein Objekt in der Physik Engine mit einem anderen Objekt kollidiert ist.
- Die beiden Prototypen zeigen, dass die Verarbeitung von High-Level Events (inklusive der Events, die in den Anforderungen FE1-FE3 beschrieben wurden) sehr einfach mit einer selbst implementierten Klasse realisiert werden kann. Ein Beispiel dafür ist die Klasse «GameController» des ersten Prototyps, dieser verarbeitet das Ereignis, wenn ein feindliches Raumschiff zerstört wurde und fügt dem Punktestand des Spielers (abhängig vom Typ des Gegners) eine bestimmte Anzahl Punkte hinzu.
- Aufgrund zusätzlicher Anforderungen, die während der Implementation wichtiger eingestuft wurden, wurden die Anforderungen an das Event Controller Subsystem zurückgestellt. Das Subsystem ist jedoch im aktuellen Zustand vollumfänglich funktional.

Level Controller (Functional Level Controller)

Tabelle 9 Funktionale Anforderungen: Level Controller Subsystem

| ID | Prio | Beschreibung | Erfüllt? |
|-----|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| FL1 | Muss | Der Spieleanentwickler möchte die Möglichkeit haben für Texturflächen der Spielwelt (Tiles) Materialeigenschaften zu definieren, sodass diese Eigenschaften das Verhalten mit anderen Spielobjekten festlegen. Dies umfasst bspw. ob das Material solide oder durchdringbar ist, oder den Z-Index beim Rendering. | Ja |
| FL2 | Wunsch | Der Spieleanentwickler möchte die Möglichkeit haben die Elemente der Spielwelt mit physikalischen Materialeigenschaften zu versehen, sodass die Interaktion zwischen den Spielobjekten und der Spielwelt durch physikalische Gesetze definiert ist. Dieses Feature bedingt, dass eine Physik Engine integriert wird. | Ja |
| FL3 | Wunsch | Der Spieleanentwickler möchte die Möglichkeit haben Hintergrundtexturen zu definieren, die unabhängig vom Vordergrund bewegt werden können, sodass durch die unterschiedlichen Bewegungen ein 3D Effekt erzeugt werden kann. | Nein |

Die Anforderungen des Level Controller Subsystems, die als «muss» Kriterien definiert wurden, konnten vollständig erfüllt werden. Für die Anforderung **FL3**, die als optionales Kriterium definiert wurde, blieb leider nicht genug Zeit übrig. Die Anforderungen **FL1** und **FL2** konnten mit der Integration der Physik Engine realisiert und sogar übertroffen werden, zum Beispiel können die Levelobjekte mehrere individuelle Formen mit unterschiedlichen physikalischen Eigenschaften besitzen. Die folgende Anforderung wurde im Rahmen des Projektes zusätzlich umgesetzt obwohl sie zu Anfangs nicht Teil der Anforderungsspezifikation war:

- **Multiple Z-Indizes (2.5D Rendering):** Der Spieleanentwickler hat die Möglichkeit eine Spielwelt zu erstellen, bei der jede Reihe (y-Achse) einen eigenen Z-Index erhält, sodass ein Objekt der Spielwelt abhängig von der Position des Spielers einmal vordergründig und einmal hintergründig zum Spieler ist.

Entity Controller (Functional Entity)

Tabelle 10 Funktionale Anforderungen: Entity Controller Subsystem

| ID | Prio | Beschreibung | Erfüllt? |
|-----|--------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| FI1 | Muss | Der Spieleanentwickler möchte die Möglichkeit haben den Spielobjekten einen Z-Index Wert zuzuordnen, sodass Objekte mit einem höheren Z-Index vor den Objekten mit niedrigerem Z-Index erscheinen. | Ja |
| FI2 | Wunsch | Der Spieleanentwickler möchte die Möglichkeit haben physikalische Eigenschaften für Spielobjekte zu definieren, sodass die Interaktion zwischen den Spielobjekten durch physikalische Gesetze definiert ist. Dieses Feature bedingt, dass eine Physik Engine integriert wird. | Ja |
| FI3 | Wunsch | Der Spieleanentwickler möchte die Möglichkeit Licht emittierende Spielobjekte zu definieren, sodass die Darstellung der Spielwelt und der Spielobjekte durch die Lichtröhrer beeinflusst wird. | Nein |

Die Anforderungen des Entity Controller Subsystems, die als «muss» Kriterien definiert wurden, konnten vollständig erfüllt werden. Des Weiteren wurde durch die Integration der Physik Engine die folgende Anforderung erfüllt, die nicht Teil der Anforderungsspezifikation war:

- **Verschiedene Physik Strategien:** Der Spielentwickler hat die Möglichkeit Spielobjekte zu erstellen, die entweder unbeweglich, codegesteuert, oder physikgesteuert sind.

State Manager (Functional State Manager)

Tabelle 11 Funktionale Anforderungen: State Manager Subsystem

| ID | Prio | Beschreibung | Erfüllt? |
|-----|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| FS1 | Muss | Der Spieleanwickler möchte zu jeder Zeit Zugang zum aktuellen Spielzustand haben, sodass die Spiellogik auf dem Spielzustand aufgebaut werden kann. Der Spielzustand umfasst den aktuellen Level, die Spielobjekte, die registrierten Events und die Spieloptionen. | Ja |
| FS2 | Muss | Der Spieleanwickler möchte die Möglichkeit haben individuelle Attribute und Klassen im Spielzustand zu speichern, sodass diese jederzeit im Spiel verfügbar sind. | Ja |

Die Anforderungen des State Managers, konnten vollständig erfüllt werden. Darüber hinaus wurde die folgende Anforderung zusätzlich umgesetzt, die nicht im Teil der Anforderungsspezifikation war:

- **Asynchrone Löschung:** Der Spieleanwickler ist in der Lage Spielobjekte asynchron zu löschen, dies ermöglicht unter anderem, dass ein Spielobjekt sich selbst löschen kann.

Prototypen (Functional Prototypes)

Tabelle 12 Funktionale Anforderungen: Prototypen

| ID | Prio | Beschreibung | Erfüllt? |
|-----|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| FP1 | Muss | <p>Der Spieleentwickler möchte einen Prototyp eines Arcade Bottom Up Shooter Spiels als Referenz zur Verfügung haben, sodass der Einstieg in die Erstellung eines solchen Spiels deutlich leichter ist. Der Prototyp soll folgende Features beinhalten:</p> <ul style="list-style-type: none"> - Feindliche nicht-spieler Charakter, die sich in einem bestimmten Muster bewegen - Kamera, die sich konstant nach Oben bewegt - Abschlussbedingungen («Game Over») | Ja |
| FP2 | Muss | <p>Der Spieleentwickler möchte einen Prototyp eines 2D Rollenspiels als Referenz zur Verfügung haben, sodass der Einstieg in die Erstellung eines solchen Spiels deutlich leichter ist. Der Prototyp soll folgende Features beinhalten:</p> <ul style="list-style-type: none"> - Feindliche nicht-spieler Charakter, die sich in einem bestimmten Muster bewegen - Kamera, die dem Spieler folgt - In-Game Overlay Display mit Lebensanzeige - Abschlussbedingungen («Game Over») - (Wunsch) Held mit Schwertschlag Animation | Ja |
| FP3 | Muss | <p>Der Spieleentwickler möchte einen Prototyp eines 2D Platformer Spiels als Referenz zur Verfügung haben, sodass der Einstieg in die Erstellung eines solchen Spiels deutlich leichter ist. Der Prototyp soll folgende Features beinhalten:</p> <ul style="list-style-type: none"> - Feindliche nicht-spieler Charakter, die sich in einem bestimmten Muster bewegen - Kamera, die dem Spieler folgt, wenn bestimmte Bedingungen erfüllt sind (horizontal: immer, vertikal: wenn erlaubt) - Schwerkraft, die entweder durch eine Physik Engine berechnet oder mit der Engine simuliert wird. - Abschlussbedingungen («Game Over») | Nein |

Es wurde aus zeitlichen Gründen entschieden, dass nur die beiden ersten Prototypen (**FP1** und **FP2**) umgesetzt werden. Der erste Prototyp erfüllt alle Anforderungen, die in der Anforderungsspezifikation definiert wurden und die folgenden Anforderungen, die nicht Teil der Anforderungsspezifikation sind:

- **Begrenzung der begehbarer Welt:** Der Spieler kann sich nur im sichtbaren Bereich der Kamera aufhalten. Eine unsichtbare Mauer verhindert das Verlassen des sichtbaren Bereichs.
- **Endgegnerkampf:** Am Schluss des Levels erscheint ein Endgegner, der ein komplexes Flugmuster fliegt und mehrere Arten von Projektilen feuert.
- **Projektil:** Sowohl der Spieler als auch die feindlichen nicht-Spieler Charaktere können Projektil abfeuern.
- **Explosionseffekte:** Sowohl der Spieler als auch die feindlichen nicht-Spieler Charaktere erzeugen einen Explosionseffekt, wenn sie zerstört werden.

Der zweite Prototyp wurde etwas anders ausgeführt, als es in der Anforderungsspezifikation definiert war, es wurden zum Beispiel keine feindlichen nicht-Spieler Charaktere erstellt und daher auch keine Spielabschlussbedingungen. Statt der genannten Anforderungen wurden die folgenden Anforderungen zusätzlich umgesetzt, die nicht Teil der Anforderungsspezifikation waren:

- **2.5-dimensionale Welt:** Die Spielwelt ist so aufgebaut, dass sich der Spieler je nach Position entweder vor, oder hinter einem Objekt der Spielwelt befinden kann.
- **Freundlicher nicht-Spieler Charakter:** Der Prototyp enthält einen nicht-Spieler Charakter. Wenn der Spieler in der Nähe dieses nicht-Spieler Charakters ist, ist er in der Lage ein Dialogfenster zu öffnen, indem ein zufällig ausgewählter Text angezeigt wird. Wenn der Dialog geöffnet ist, ist der Spieler unbeweglich und der nicht-Spieler Charakter schaut in die Richtung des Spielers.
- **Physikalische Begrenzungen:** Die Spielwelt enthält Objekte, wie zum Beispiel Mauern oder Tische, die nicht durchdringbar sind.

8.2.2 Technische Anforderungen

Tabelle 13 Technische Anforderungen

| ID | Prio | Beschreibung | Erfüllt? |
|----|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|
| T1 | Muss | Das Produkt (Softwareentwicklungs paket) soll auf den gängigen Plattformen (Windows, Linux, Mac) bereitgestellt und ausgeführt werden können, sodass eine möglichst grosse Audienz mit dem damit erstellten Endprodukt erreicht werden kann. | Teilweise |
| T2 | Muss | Die Prozesse sollen im sinnvollen Massen parallel ablaufen, sodass moderne Multi-Core Prozessoren möglichst gut ausgelastet werden können. | Ja |
| T3 | Muss | Die Frame Zyklen müssen präzise getaktet sein, sodass ein flüssiges Spielerlebnis realisiert werden kann. | Ja |
| T4 | Wunsch | Bei zu hoher Zeitüberschreitung soll ein Frame übersprungen werden sodass die Darstellung mit der Spiellogik wieder synchron ist. | Ja |

Die Anforderung T1 wurde insofern umgesetzt, dass mit Technologien gearbeitet wurde, die mit den gängigen Plattformen kompatibel sind. Die CMake Build Scripts wurden aber im Rahmen dieser Arbeit nur für Windows erstellt.

8.2.3 Anforderungen rendering Qualität

Tabelle 14 Anforderungen rendering Qualität

| ID | Prio | Beschreibung | Erfüllt? |
|----|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| Q1 | Muss | Das Rendering soll eine Double- oder Triple Buffering Strategie verfolgen sodass die Darstellung nahtlos und flüssig erscheint. | Ja |
| Q2 | Muss | Die Framerate des Spiels soll stabil bei 50Hz sein, wenn auf einem 1680x1050 Pixel grossen Bildschirm 1'000 unterschiedlich texturierte 8x8 Pixel grosse Elemente dargestellt werden (Wunschkriterium: 10'000 Elemente). | Ja |
| Q3 | Muss | Die Framerate des Spiels soll stabil bei 50Hz sein, wenn 1'000 kollisionsfähige Spielobjekte gleichzeitig im Spiel sind (Wunschkriterium 10'000 Spielobjekte). | Ja |

9 Schlussfolgerungen/Fazit

9.1 Ergebnisse

Die meisten der Anforderungen, die in der Anforderungsspezifikation definiert waren, konnten erfüllt werden. Bei der Realisation des Projekts mussten einige der Anforderungen zurückgestellt werden, um den Fokus auf neue Anforderungen zu legen, die zu Anfangs unbekannt waren, die aber eine essenzielle Voraussetzung für die Implementation der beiden Prototypen darstellten. Im Rahmen dieser Thesis konnte eine nützliche und vielseitig einsetzbare Game Engine erarbeitet werden. Die Funktionalitäten der Game Engine konnte mit den beiden Prototypen gut getestet und demonstriert werden. Ein dritter Prototyp, wie es zu Anfangs im Anforderungsspezifikationsdokument definiert war, konnte leider aufgrund mangelnder Zeit (bedingt durch die wachsenden Anforderungen) nicht realisiert werden. Insgesamt kann das Projekt als Erfolg betrachtet werden, denn es konnte letztendlich ein hochwertiges Produkt mit einem gut erprobten Nutzen erarbeitet werden.

9.2 Herausforderungen

Eine wesentliche Herausforderung stellte vor allem die Integration der Physik Engine dar. Die Physik Engine stellt eine eigene Umgebung für die Simulation von Objekten bereit, die Vereinigung dieser Simulationsumgebung mit der Scene Graph Datenstruktur der Game Engine war mit einigen Komplikationen verbunden. In der Mitte des Projekts wurde zum Beispiel der Scene Graph so umgebaut, dass er statt des visuellen Koordinatensystems von SFML das Koordinatensystem der Physik Engine verwendet. Die Initialisierung der Objekte, die von der Physik Engine bereitgestellt werden, musste in entsprechenden Wrapper Klassen (den «ShapePhysics» Klassen) definiert werden. Eine weitere Herausforderung stellten die ständig wachsenden Anforderungen dar, zum Beispiel wurde zu Anfangs nicht berücksichtigt, dass es sinnvoll wäre auch Objekte wie Projekteile und visuelle Effekte zu unterstützen. Im Rahmen des Projekts mussten die Anforderungen so angepasst werden, dass die Realisierung der beiden Prototypen möglich ist.

9.3 Ausblick

Die Idee ist, dass die Game Engine auch nach Ende dieser Bachelorthesis weiterhin gewartet und weiterentwickelt wird. Einige der Features sollen noch weiter ausgebaut werden und es liegt auch bereits ein Konzept für das erste Spiel mit der Engine vor. Die folgende Liste enthält einige Features, die in der Zukunft im Rahmen einer Version 2.0 geplant sind:

- **3D Rendering:** Im Rahmen des Vorprojekts wurde ein Konzept für das Rendering von 3D Objekten direkt mit OpenGL erstellt, dieses Konzept soll zukünftig in die Engine eingebaut werden. Die Spielwelt soll 2-dimensional bleiben, jedoch soll es zusätzlich möglich sein 3D Objekte aus einer beliebigen Perspektive darzustellen.
- **Behaviours:** Das Konzept der dynamischen Verhalten auf Basis von Klassenvererbung soll weiter ausgebaut und als Teil der Game Engine integriert werden. Es soll z. B. möglich sein, einfache Funktionalitäten wie die Spielerbewegung durch reine Klassenvererbung zu realisieren.
- **Benutzerhandbuch:** In der zukünftigen Version soll ein Benutzerhandbuch zur Verfügung gestellt werden, um neuen Spielentwickler einen schnelleren Zugang zur Engine zu verschaffen.
- **Drittpartei Klassen:** Die Signaturen der Klassen der Frameworks SFML und Chipmunk, die von der Game Engine verwendet werden, sollen von der Game Engine selbst zur Verfügung gestellt werden, sodass ein Spiel, das die Game Engine verwendet diese beiden Frameworks nicht separat importieren muss.

10Referenzen

- [1] L. Gomila, «SFML Home Page,» 2018. [Online]. Available: <https://www.sfml-dev.org/>. [Zugriff am 15 August 2019].
- [2] Howling Moon Software, «Chipmunk Physics Homepage,» 2013. [Online]. Available: <https://chipmunk-physics.net>. [Zugriff am 10 September 2019].
- [3] B. Stroustrup und H. Sutter, «Isocpp C++ Core Guidelines,» 7 März 2019. [Online]. Available: <http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>. [Zugriff am 22 März 2019].
- [4] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, New Jersey: Addison-Wesley, 2001.
- [5] E. Gamma, R. Helm, R. Johnson und J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Boston: Addison-Wesley Professional, 1994.
- [6] A. Markóczy, Pflichtenheft 2DGameSDK, Bern: Berner Fachhochschule, 2019.
- [7] TIGA, «About TIGA and our Industry,» 2019. [Online]. Available: <https://tiga.org/about-tiga-and-our-industry>. [Zugriff am 06 Oktober 2019].
- [8] Godot, «Godot Engine,» 2019. [Online]. Available: <https://godotengine.org/>. [Zugriff am 06 Oktober 2019].
- [9] Unity Technologies, «Unity Real-Time Development Platform,» 2019. [Online]. Available: <https://unity.com/>. [Zugriff am 06 Oktober 2019].
- [10] Epic Games, «What is Unreal Engine 4,» 2019. [Online]. Available: <https://www.unrealengine.com/>. [Zugriff am 06 Oktober 2019].
- [11] Unity Technologies, «Unity Home Page,» [Online]. Available: <https://unity.com/>. [Zugriff am 20 März 2019].
- [12] Chukong Technologies, «Cocos 2D-X Home Page,» 2019. [Online]. Available: <https://www.cocos.com>. [Zugriff am 10 September 2019].
- [13] E. Catto, «Box2D Home Page,» 2019. [Online]. Available: <https://box2d.org>. [Zugriff am 6 August 2019].
- [14] E. Catto, «Box2D v2.3.0 User Manual,» 2019. [Online]. Available: <https://github.com/erincatto/box2d>. [Zugriff am 6 August 2019].
- [15] Howling Moon Software, «Chipmunk2D 7.0.3,» 2019. [Online]. Available: <https://chipmunk-physics.net/release/ChipmunkLatest-Docs>. [Zugriff am 6 August 2019].
- [16] Free Software Foundation, Inc., «GCC, the GNU Compiler Collection,» 22 Februar 2019. [Online]. Available: <https://gcc.gnu.org/>. [Zugriff am 20 März 2019].
- [17] N. Lohmann, «JSON for Modern C++,» 24 Mai 2019. [Online]. Available: <https://nlohmann.github.io/json/>. [Zugriff am 2 Juni 2019].
- [18] Khronos Group Inc., «OpenGL Home Page,» [Online]. Available: <https://www.opengl.org/>. [Zugriff am 20 März 2019].
- [19] Statista Ltd., «What game engines do you currently use?,» 2016. [Online]. Available: <https://www.statista.com/statistics/321059>. [Zugriff am 15 August 2019].

11 Glossar

| Begriff | Erläuterung |
|--------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bounding Box | Eine Bounding Box ist eine rechteckige Form, die ein bestimmtes Objekt vollständig umschliesst. Bounding Boxes werden für effiziente Vorabkalkulationen (z.B. bei Kollisionsrechnungen) verwendet. |
| CMake | CMake ist eine plattformübergreifende Gruppe von Open-Source Tools zum Erstellen, Testen und Bereitstellen von Software. |
| Copyleft | Copyleft ist eine Eigenschaft von manchen Open Source Lizzen (wie zum Beispiel GPL). Produkte, die mit einer Copyleft Lizenz lizenziert sind, dürfen nur verwendet werden, wenn die damit erstellten Produkte auch unter einer Copyleft Lizenz verbreitet werden. |
| GCC (GNU Compiler Collection) | Die GNU Compiler Collection ist eine Sammlung von Open Source Compiler für Sprachen wie C, C++, Go oder Fortran |
| JSON | Die JavaScript Object Notation, kurz JSON, ist ein kompaktes Datenformat in einer einfach lesbaren Textform zum Zweck des Datenaustauschs zwischen Anwendungen. |
| Library (Software-bibliothek) | Eine Softwarebibliothek ist eine Sammlung von Ressourcen, die für Computerprogramme bereitgestellt werden. Dazu gehören unter anderem Klassen, Typdefinitionen und Konfigurationsdaten. |
| Plain Data Object | Ein Plain Data Object ist ein Objekt, das selbst keine Methoden hat und nur zum Zweck der Datenspeicherung verwendet wird. |
| Proof of Concept | Ein Proof of Concept ist im Kontext der Softwareentwicklung ein Prototyp, mit dem die Machbarkeit eines bestimmten Vorhabens (z.B. eines neuen Features) abgeklärt wird. |
| Scene Graph | Ein Szenengraph ist eine Datenstruktur, mit der die logische, in vielen Fällen auch die räumliche Anordnung der Objekte in einer darzustellenden zwei- oder dreidimensionalen Szene beschrieben wird. |
| SDK | Ein Software Development Kit ist eine Sammlung von Tools zur Erstellung einer Software in einem bestimmten Aufgabenbereich. |
| SFML | Simple and Fast Multimedia Library SDML ist eine Open Source C++ Softwarebibliothek die gewisse low-level Tools für die Erstellung von 2D Computerspielen zur Verfügung stellt [1]. |
| SI Einheiten | Das Internationale Einheitensystem oder SI (franz.: Système international d'unités) ist ein Einheitensystem für physikalische Größen. |
| Tiles (Kachelgrafik) | Tiles oder Kachelgrafik ist eine Methode in der Computergrafik, bei der ein Bild aus mehreren kleinen Kacheln mosaikartig zusammengesetzt wird. |
| VSync | Vertikale Synchronisation ist ein Prozess der Computergrafik, bei dem sichergestellt wird, dass ein Bild das auf dem Monitor dargestellt wird, nicht während der Darstellung ausgewechselt werden kann. |

12 Abbildungsverzeichnis

| | |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Abbildung 1 Produktübersicht 2DGameSDK | 6 |
| Abbildung 2 Meistverwendete Game Engines in Grossbritannien im Jahr 2014 (Umfrage von: [6], Quelle: [19]) | 7 |
| Abbildung 3 Userinterface der Unity Game Engine (aus [8]) | 9 |
| Abbildung 4 Userinterface der UDK Game Engine (aus [9]) | 10 |
| Abbildung 5 Userinterface der Godot Engine (aus [7]) | 11 |
| Abbildung 6 Userinterface der Cocos 2D-X Game Engine | 12 |
| Abbildung 7 Prinzip der Kollisionserkennung auf Basis von Schnittpunkt Berechnungen. Links: Keine Kollision (Schnittpunkt ist ausserhalb der Form), rechts: Kollision (Schnittpunkt ist in der Form) | 14 |
| Abbildung 8 Benchmarking Szenarios mit 100 Objekten (links), 500 Objekten (Mitte) und 1'000 Objekten (rechts) | 16 |
| Abbildung 9 UML Kontextdiagramm 2DGameSDK | 20 |
| Abbildung 10 UML Paketdiagramm | 21 |
| Abbildung 11 UML Klassendiagramm: Highlevel Ansicht | 23 |
| Abbildung 12 Vergleich des visuellen Koordinatensystems von SFML (links) mit dem physikalischen Koordinatensystem von Chipmunk (rechts) | 25 |
| Abbildung 13 Prinzip der Skalierung von Dimensionen zur Umwandlung von Pixel des SFML Koordinatensystems in Meter des Chipmunk Koordinatensystems | 26 |
| Abbildung 14 UML Klassendiagramm: Core Package | 30 |
| Abbildung 15 UML Klassendiagramm: World Package | 32 |
| Abbildung 16 UML Klassendiagramm: Physics Package | 33 |
| Abbildung 17 UML Klassendiagramm: Scene Package | 35 |
| Abbildung 18 UML Klassendiagramm: Event Package | 36 |
| Abbildung 19 UML Sequenzdiagramm: Spiel Initialisierung (Teil 1) | 37 |
| Abbildung 20 UML Sequenzdiagramm: Spiel Initialisierung (Teil 2) | 38 |
| Abbildung 21 UML Sequenzdiagramm: Game Loop | 39 |
| Abbildung 22 UML Sequenzdiagramm: Render Loop | 40 |
| Abbildung 23 Screenshot des ersten Prototyps (Farben wurden verändert um den Schwarzanteil zu reduzieren) | 42 |
| Abbildung 24 Screenshot des zweiten Prototyps | 43 |
| Abbildung 25 Demonstratives UML Klassendiagramm der Behaviour Logik des ersten Prototyps | 44 |
| Abbildung 26 Zeitplan gemäss Pflichtenheft | 45 |
| Abbildung 27 Zeitplan tatsächlich umgesetzt | 46 |

13 Tabellenverzeichnis

| | |
|--------------------------------------------------------------------------------------------------------|----|
| Tabelle 1 Statistische Analyse des Benchmarking mit Chipmunk Physics (Zeitindex 30–70) | 16 |
| Tabelle 2 Extrema des Benchmarking mit Chipmunk Physics (Zeitindex 0–70) | 17 |
| Tabelle 3 Statistische Analyse des Benchmarking mit Box2D (Zeitindex 30–70)..... | 17 |
| Tabelle 4 Extrema des Benchmarking mit Box2D (Zeitindex 0–70) | 17 |
| Tabelle 5 Evaluation Chipmunk Physics (G = Gewichtung, P = Punkte, P * G = Gewichtetes Ergebnis) | 18 |
| Tabelle 6 Evaluation Box2D (G = Gewichtung, P = Punkte, P * G = Gewichtetes Ergebnis) | 19 |
| Tabelle 7 Funktionale Anforderungen: Game Controller Subsystem..... | 47 |
| Tabelle 8 Funktionale Anforderungen: Event Controller Subsystem | 48 |
| Tabelle 9 Funktionale Anforderungen: Level Controller Subsystem..... | 49 |
| Tabelle 10 Funktionale Anforderungen: Entity Controller Subsystem..... | 49 |
| Tabelle 11 Funktionale Anforderungen: State Manager Subsystem | 50 |
| Tabelle 12 Funktionale Anforderungen: Prototypen..... | 51 |
| Tabelle 13 Technische Anforderungen | 52 |
| Tabelle 14 Anforderungen rendering Qualität | 52 |

14 Anhang

14.1 Anhang A1: Benchmarking Chipmunk Physics

| Zeitindex (s) | Framerate (1/s) | | |
|---------------|-----------------|-------------|--------------|
| | 100 Objekte | 500 Objekte | 1000 Objekte |
| 30 | 2291 | 636 | 294 |
| 31 | 2279 | 636 | 303 |
| 32 | 2289 | 624 | 295 |
| 33 | 2311 | 633 | 299 |
| 34 | 2310 | 635 | 297 |
| 35 | 2285 | 636 | 297 |
| 36 | 2285 | 630 | 295 |
| 37 | 2286 | 622 | 298 |
| 38 | 2270 | 630 | 299 |
| 39 | 2259 | 632 | 301 |
| 40 | 2299 | 625 | 299 |
| 41 | 2305 | 624 | 296 |
| 42 | 2330 | 629 | 292 |
| 43 | 2278 | 621 | 297 |
| 44 | 2333 | 623 | 298 |
| 45 | 2283 | 631 | 299 |
| 46 | 2252 | 630 | 299 |
| 47 | 2260 | 630 | 297 |
| 48 | 2278 | 620 | 299 |
| 49 | 2249 | 622 | 292 |
| 50 | 2266 | 619 | 293 |
| 51 | 2200 | 623 | 295 |
| 52 | 2271 | 625 | 299 |
| 53 | 2327 | 629 | 298 |
| 54 | 2250 | 630 | 298 |
| 55 | 2283 | 627 | 297 |
| 56 | 2286 | 616 | 299 |
| 57 | 2257 | 620 | 296 |
| 58 | 2281 | 623 | 295 |
| 59 | 2288 | 625 | 298 |
| 60 | 2309 | 617 | 299 |
| \bar{x} | 2282.26 | 626.55 | 297.19 |
| σ | 27.24 | 5.71 | 2.48 |
| k/n | 70.97% | 67.74% | 80.65% |

- Nur die Messungen ab Zeitindex 30 (30 Sekunden) wurden für die statistische Auswertung berücksichtigt, da die Schwankungen ab dort minimal sind.
- Der Wert "k/n" sagt aus, welcher Anteil der Messungen innerhalb der einfachen Standardabweichung liegt ($\bar{x} - \sigma \leq x \leq \bar{x} + \sigma$). Eine Messung ist aussagekräftig, wenn dieser Wert bei 68% liegt.

14.2 Anhang A2: Benchmarking Box2D Physics

| Zeitindex (s) | Framerate (1/s) | | |
|---------------|-----------------|-------------|--------------|
| | 100 Objekte | 500 Objekte | 1000 Objekte |
| 30 | 2407 | 1518 | 0 |
| 31 | 2405 | 1578 | 0 |
| 32 | 2388 | 1581 | 0 |
| 33 | 2406 | 1579 | 0 |
| 34 | 2405 | 1564 | 0 |
| 35 | 2396 | 1573 | 0 |
| 36 | 2409 | 1537 | 0 |
| 37 | 2393 | 1541 | 0 |
| 38 | 2399 | 1569 | 0 |
| 39 | 2414 | 1571 | 0 |
| 40 | 2377 | 1583 | 0 |
| 41 | 2402 | 1582 | 0 |
| 42 | 2374 | 1578 | 0 |
| 43 | 2388 | 1576 | 0 |
| 44 | 2381 | 1580 | 0 |
| 45 | 2320 | 1548 | 0 |
| 46 | 2401 | 1541 | 0 |
| 47 | 2393 | 1562 | 0 |
| 48 | 2417 | 1569 | 0 |
| 49 | 2406 | 1546 | 0 |
| 50 | 2399 | 1568 | 0 |
| 51 | 2405 | 1543 | 0 |
| 52 | 2400 | 1577 | 0 |
| 53 | 2399 | 1581 | 0 |
| 54 | 2410 | 1577 | 0 |
| 55 | 2415 | 1568 | 0 |
| 56 | 2402 | 1568 | 0 |
| 57 | 2409 | 1576 | 0 |
| 58 | 2438 | 1584 | 0 |
| 59 | 2413 | 1577 | 0 |
| 60 | 2382 | 1577 | 0 |
| \bar{x} | 2398.48 | 1566.84 | 0.00 |
| σ | 19.51 | 16.74 | 0.00 |
| k/n | 87.10% | 74.19% | 0.00% |

- Nur die Messungen ab Zeitindex 30 (30 Sekunden) wurden für die statistische Auswertung berücksichtigt, da die Schwankungen ab dort minimal sind.
- Der Wert “k/n” sagt aus, welcher Anteil der Messungen innerhalb der einfachen Standardabweichung liegt ($\bar{x} - \sigma \leq x \leq \bar{x} + \sigma$). Eine Messung ist aussagekräftig, wenn dieser Wert bei 68% oder höher liegt.
- Für die Messungen mit 1000 Objekten konnten keine Daten erhoben werden, weil die Simulationsgeschwindigkeit ab Zeitindex 10 unter dem messbaren Bereich von einem Frame pro Sekunde ist.

15 Selbstständigkeitserklärung

Selbständige Arbeit / Travail autonome

Ich bestätige mit meiner Unterschrift, dass ich meine vorliegende Bachelor-Thesis selbstständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.) und anderen Hilfsmittel, die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt. Sämtliche Inhalte, die nicht von mir stammen, sind mit dem genauen Hinweis auf ihre Quelle gekennzeichnet.

Par ma signature, je confirme avoir effectué ma présente thèse de bachelor de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.) et autres ressources qui m'ont fortement aidé-e dans mon travail sont intégralement mentionnées dans l'annexe de ma thèse. Tous les contenus non rédigés par mes soins sont dûment référencés avec indication précise de leur provenance.

Name/Nom, Vorname/Prénom
.....

Datum/Date
.....

Unterschrift/Signature
.....



Bachelorthesis-Aufgabe

für Alistair Markóczy
Abteilung Informatik
Betreuung durch Urs Künzler

2D Game engine mit C++ und SFML

Aufbauend auf den Ergebnissen der Projekt2 Arbeit, soll die Entwicklung eines Software Development Kits für 2D Computerspielen weiter geführt und abgeschlossen werden.

Zur Unterstützung wird die Library SFML verwendet, die es ermöglicht Multi-Plattform fähige Applikationen zu erstellen, wobei die Darstellung von Texturierten Elementen auf dem Bildschirm (mit Zoom, Screen Offset etc.) bereits gelöst ist. Der Fokus liegt daher auf der Implementation einer Gamelogik die individuell angepasst werden kann, so dass unterschiedliche Arten von 2D Games (Shooter, Adventure, Roleplay Game) mit wenig Aufwand erstellt werden können.

Die Arbeit soll die folgenden Schwerpunkte umfassen:

- Scene Graph
- Game Loop (Tick Render Ziklus)
- Levels: Tilemap Loader anhand von JSON Dateien
- Material Eigenschaften: Solid, Background, Foreground
- Layering, Z-Order
- Game Events und Handler
- Input Handling und Spieler Steuerung
- Engine Support für Künstliche Intelligenz
- In-Game Overlay Display
- Untersuchung wie weit physikalische Eigenschaften integriert werden können.

Für die Implementation von Animationen soll überprüft werden, ob allenfalls eine 3D Darstellung besser geeignet ist als eine reine 2D Darstellung. Das gewählte Framework SFML bietet hierzu auch eine geeignete Schnittstelle zu OpenGL an.

Beginn der Arbeit 16. September 2019
Abschluss der Arbeit 16. Januar 2020

Der Betreuer:

A handwritten signature in blue ink, appearing to read "Urs Künzler".

Der Abteilungsleiter:

A handwritten signature in blue ink, appearing to read "Urs Künzler".

Berner Fachhochschule | Informatik + Medizininformatik

1/1