



Anforderungsspezifikation: 2DGameSDK

Markóczy Alain Alistair

Projekt: 2D Game SDK

Markóczy Alain Alistair

Version: 1.0

Status: Freigegeben

Datum: 18.10.2019

Versionskontrolle

Version	Datum	Beschreibung	Autor
0.1	30.09.2019	Entwurf	A.M.
0.2	13.10.2019	Anforderungen, Systembeschreibung	A.M.
1.0	18.10.2019	Finale Version	A.M.

Inhaltsverzeichnis

1	Zweck des Dokuments	1
2	Ausgangslage	1
2.1	Projektvision	1
2.2	Projektziele	1
3	Stand Vorprojekt	1
3.1	Realisierte Konzepte	1
3.2	Variantenstudium: Animationen	3
4	Marktanalyse	5
4.1	Game Engines auf dem Markt	5
4.2	Marktpositionierung 2DGameSDK	7
4.3	Distributionskonzept	7
5	Systembeschreibung	7
5.1	Prozessumfeld	7
5.2	Systemumfeld	9
5.3	Prototypen	11
5.4	Variantenstudium	12
5.5	Randbedingungen	12
5.6	Qualitätssicherung	13
6	Anforderungen	13
6.1	Funktionale Anforderungen	13
6.2	Technische Anforderungen	16
6.3	Anforderungen rendering Qualität	16
7	Testing	16
7.1	Konzept	16
7.2	Technologien	16
8	Projektmanagement	17
8.1	Projektorganisation	17
8.2	Zeitplan	18
9	Referenzen	19
10	Glossar	20
11	Abbildungsverzeichnis	21
12	Tabellenverzeichnis	21

1 Zweck des Dokuments

Dieses Dokument beschreibt die Ziele und Anforderungen für das Projekt 2DGameSDK.

2 Ausgangslage

2.1 Projektvision

Ziel der Arbeit ist es, ein Software Development Kit für unterschiedliche Arten von 2D Computerspielen zu erstellen. Zur Unterstützung soll die Library SFML [1] verwendet werden, 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.

Für die Implementation von Kollisionserkennung und Kollisionsreaktionen, soll zusätzlich im Rahmen eines Variantenstudiums evaluiert werden, ob die Integration einer Open Source Physik Engine sinnvoll ist, oder ob es besser ist mit einer eigenen Lösung zu arbeiten.

2.2 Projektziele

2.2.1 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.

2.2.2 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 [3] strikt eingehalten werden. Zusätzlich soll für die Wahl der Softwarearchitektur wann immer möglich auf gängige Design Patterns der Industrie [4] zurückgegriffen werden.

3 Stand Vorprojekt

3.1 Realisierte Konzepte

Im Rahmen des Vorprojekts wurden einige der Elemente, die für dieses Projekt verwendet werden, bereits umgesetzt, diese sind im Folgenden Teil beschrieben.

3.1.1 Prozess: Game Initialisierung

Der Initialisierungsprozess ist fundamental für den Aufbau des Spiels. Bei der Initialisierung des Spiels werden alle Ressourcen geladen und der Spielkontext wird erstellt. Der Spielkontext beinhaltet zum einen die Spielwelt und alle beweglichen Objekte, zum anderen den Spiel Controller, der den Ablauf des Spiels regelt:



Abbildung 1 UML Activity Diagramm: Spiel Initialisierung (Stand Vorprojekt)

Der aufgeführte Prozess, kann im Rahmen dieses Projekts wiederwendet und erweitert werden (z.B. mit der Erstellung eines physikalischen Kontexts).

3.1.2 Prozess: Game Loop

Der Game Loop ist das Kernstück der Spiellogik. Beim Game Loop werden die Benutzereingaben (Tastatur, Maus) sowie weitere vom Spiel ausgelöste Events abgefragt und zum richtigen Zeitpunkt verarbeitet. Sobald die Events verarbeitet wurden wird der Spielzustand aktualisiert und zuletzt wird der Spielzustand auf dem Bildschirm dargestellt:

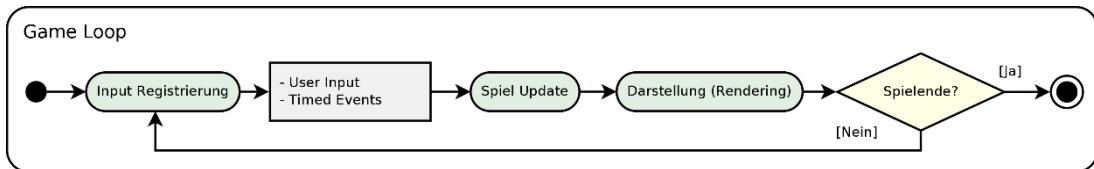


Abbildung 2 UML Activity Diagramm: Game Loop (Stand Vorprojekt)

Der aufgeführte Prozess kann im Rahmen dieses Projekts wiederwendet werden, jedoch werden bestimmte Erweiterungen vorausgesetzt, wodurch ermöglicht wird, dass das Spiel auf Kollisionen zwischen Objekten reagieren kann.

3.1.3 Konzept: Event Verarbeitung

Damit für die Spiellogik beliebige Events ausgelöst und verarbeitet werden können, braucht es einen geeigneten Event Controlling Mechanismus. Der Event Controlling Mechanismus wurde im Rahmen des Vorprojekts realisiert und basiert auf dem Observer/Observable Design Pattern.

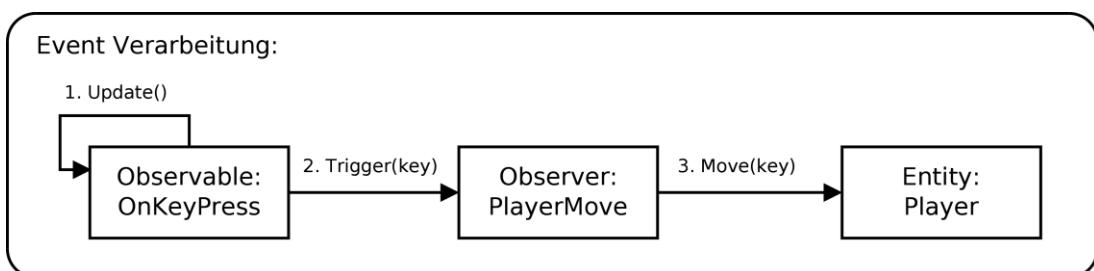


Abbildung 3 UML Kommunikationsdiagramm: Event Verarbeitung (Stand Vorprojekt)

3.1.4 Konzept: Tile Grafik

Die Spielwelt des Vorprojekts basiert auf dem Prinzip der Tile Grafik, (auch Kachelgrafik) nach diesem wird eine Textur aus einer Anordnung von mehreren kleineren Texturen („Tiles“) zusammengestellt. Mit dieser Methode ist es möglich ressourcensparend zu arbeiten da jede Tile mehrmals verwendet werden kann, jedoch nur einmal im Speicher abgelegt sein muss.



Abbildung 4 Beispiel Tile Grafik Konzept (Stand Vorprojekt)

3.1.5 Konzept: Scene Graph

Da es in vielen Computerspielen üblich ist Objekte zu erstellen, die sich relativ zu einander bewegen, wurde im Rahmen des Vorprojekts ein Scene Graph umgesetzt. Ein Scene Graph ermöglicht es, die Spielobjekte in einer Hierarchie anzurichten, sodass die untergeordneten Objekte sich relativ zu den übergeordneten Objekten bewegen. Das folgende Beispiel zeigt das anhand eines Hubschraubers. Der Rotor des Hubschraubers bewegt sich relativ zum Hubschrauber, um immer in der Mitte des Hubschraubers zu sein. (übergeordnete Transformation) Der Rotor rotiert zudem noch um die eigene Achse um den Effekt eines fliegenden Hubschraubers zu erzeugen (lokale Transformation):

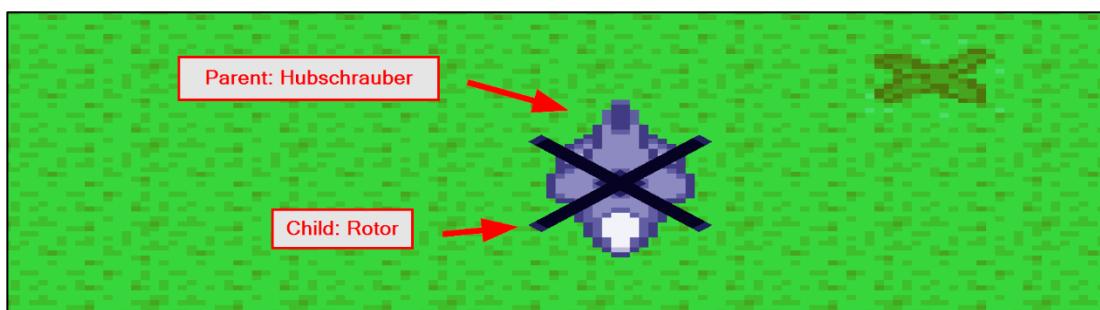


Abbildung 5 Beispiel Scene Graph Konzept (Stand Vorprojekt)

3.2 Variantenstudium: Animationen

Es gibt unterschiedliche Ansätze, um Animationen in einem Spiel zu realisieren, im Rahmen des Vorprojekts wurden die folgenden Varianten geprüft:

1. Animation als Abfolge von Bildern
2. Animation als Bewegung auf dem Scene Graph
3. Animation als Bewegung eines 3D Objekts auf einer 2D Fläche

3.2.1 Animation als Abfolge von Bildern

Wie z.B. in einem Cartoon können Animationen erzeugt werden, indem verschiedene Bilder sequenziell dargestellt werden. Diese Variante wurde vor allem bei älteren Systemen eingesetzt, wo die Größe eines Pixels auf einer Textur genau der Größe eines Pixels auf dem Bildschirm entspricht. (z.B. Nintendo Entertainment System, Neo Geo) Mit dieser Animationsmethode können die vielseitigsten Animationen erzeugt werden, denn jedem Animationsschritt kann ein frei gewähltes Bild zugeordnet werden. Es ist jedoch auch die aufwändigste Animationsmethode, weil jedes einzelne Bild vom Spielehersteller selbst erstellt werden muss.

3.2.2 Animation als Bewegung auf dem Scene Graph

Wenn sich Objekte relativ zu einander bewegen, kann auch der Effekt einer Animation erzeugt werden (Siehe Beispiel Kapitel 3.1.5). Mit dieser Methode können einfache Animationen, wie zum Beispiel die Rotation des Kopfes von einem Menschen, mit wenig Aufwand erzeugt werden. Diese Animationsmethode beschränkt sich auf Animationen, die durch die geometrische Transformation eines 2-dimensionalen Polygons im 2-dimensionalen Raum erzeugt werden können.

3.2.3 Animation mittels 3D Objekten

Diese Animationsmethode basiert ähnlich wie die Animation als Bewegung auf dem Scene Graph darauf, dass sich Objekte relativ zu einander bewegen. Bei dieser Animationsmethode werden jedoch 3D Objekte für die Darstellung der Spielobjekte verwendet, diese Objekte können im 3D Raum bewegt werden, um einen Animationseffekt zu erzeugen. So kann zum Beispiel die Rotation eines Flugzeugs um die eigene Achse simuliert werden.

3.2.4 Fazit

Im Rahmen des Vorprojekts wurden sowohl die Animation als Abfolge von Bildern, (siehe Kapitel 3.2.1) als auch die Animation als Bewegung auf dem Scene Graph (siehe Kapitel 3.2.2) umgesetzt. Beide Konzepte wurden als nützlich und relevant erachtet, dazu kommt, dass die beiden Konzepte auch zusammen in Kombination verwendet werden können. Für die Animation mittels 3D Objekten (siehe Kapitel 3.2.3) wurde ein Proof of Concept erstellt, da die vollständige Umsetzung jedoch deutlich mehr Zeit in Anspruch nehmen würde, wurde dieser Ansatz nicht weiterverfolgt.

4 Marktanalyse

4.1 Game Engines auf dem Markt

Die folgende Statistik basiert auf einer Umfrage des Wirtschaftsverbands für Spielehersteller TIGA [4] im Jahr 2014 und zeigt welche Game Engines in diesem Jahr bei den verschiedenen Spieleherstellern in Grossbrittanien zum Einsatz kamen:

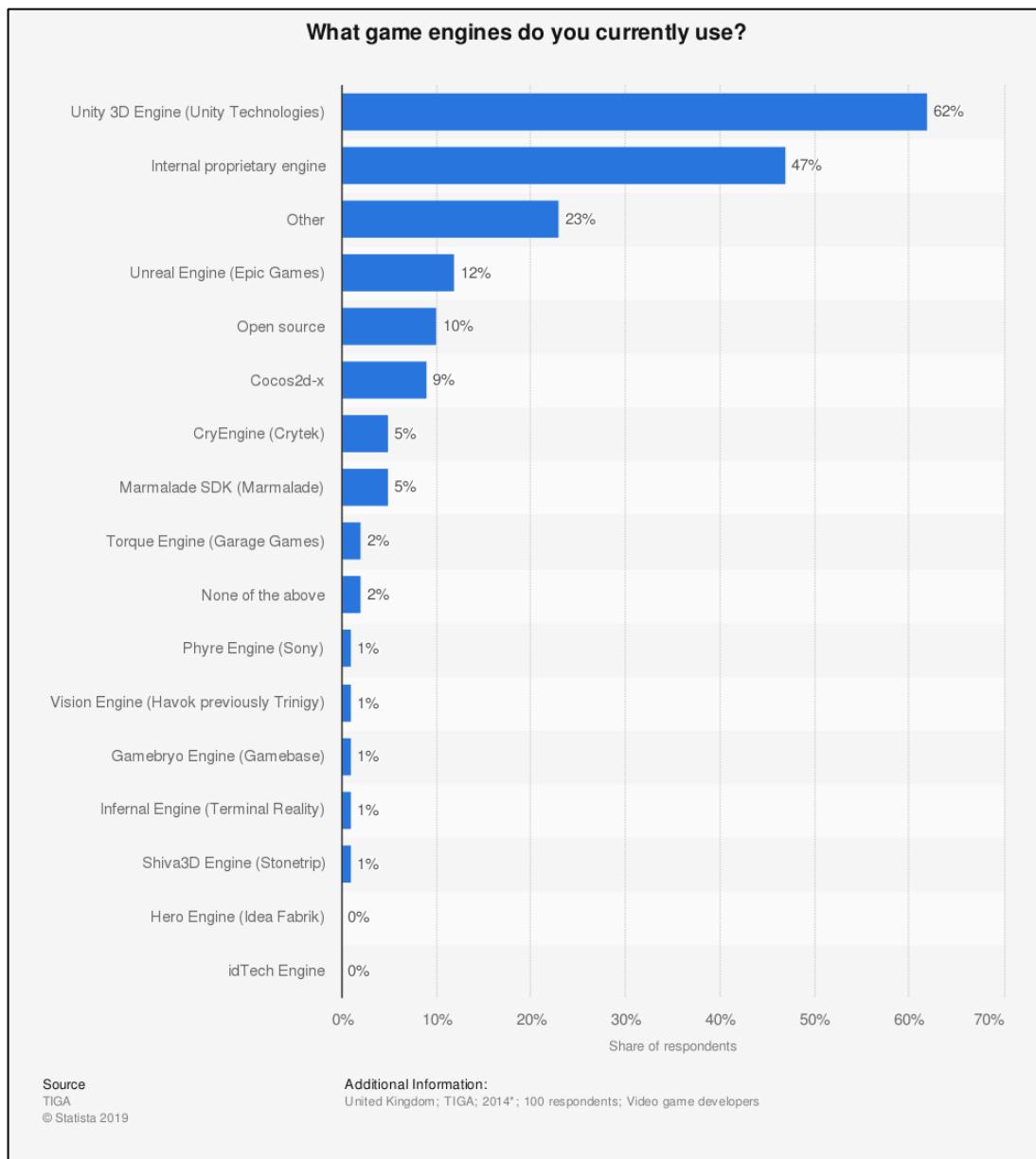


Abbildung 6 Meistverwendete Game Engines in Grossbrittanien im Jahr 2014 gemäss einer Umfrage des Wirtschaftsverbands für Spielehersteller TIGA. Quelle: [12]

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. Der Einfluss der Godot Game Engine, [5] die im Jahr 2014 herausgegeben wurde und als Open Source Engine hohe Beliebtheit erfährt, ist auf der Statistik nicht abgebildet. Die Statistik zeigt, dass die proprietären Game Engines Unity [6] und die Unreal Engine (früher auch Unreal Development Kit, UDK) [7] 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.

4.1.1 Proprietäre Game Engines

Die proprietären Game Engines zeichnen sich dadurch aus, dass sie unter einer restriktiven und meist kostenpflichtigen Lizenz verbreitet werden. Die Gebührenmodelle sind sehr unterschiedlich. Die folgende Liste enthält die bekanntesten proprietären Game Engines und deren Finanzierungsmodelle:

- **Unity:** Eine proprietäre Game Engine der Firma Unity Technologies, die sowohl kostenlose Nutzerlizenzen für Firmen mit einem Jahreseinkommen unter 100'000 Dollar anbietet als kostenpflichtige Lizenzen auf Basis eines Fixpreises von 125 Dollar pro Monat. Für den Spielehersteller fallen keine zusätzlichen Lizenzkosten beim Vertrieb des Produkts an (Royalty-Free Konzept). [8]
- **Unreal Engine:** Eine proprietäre Game Engine der Firma Epic Games, bei der sich die Lizenzgebühr am erreichten Umsatz orientiert. Sobald mit einem Produkt, das mit der Unreal Engine erstellt wurde, mehr als 3'000 Dollar Umsatz pro Quartal erzielt wird, erhebt Epic Games den Anspruch auf 5% des erzielten Bruttoumsatzes. [7]

Die proprietären Game Engines sind weit entwickelt und bieten Features, die in den meisten Open Source Game Engines nicht vorhanden sind, wie zum Beispiel einen Leveleditor oder den Asset Store. Die Rendering Technologie ist bei diesen Game Engines auch sehr nah am Stand der Technik, wodurch die Engines vor Allem für Hersteller von High-End Spielen interessant sind.

4.1.2 Open Source Game Engines

Die Open Source Game Engines sind frei verfügbar und können in der Regel auch für kommerzielle Projekte verwendet werden. (dies hängt jedoch von der Lizenz ab) Die folgende Liste enthält die bekanntesten Open Source Game Engines und deren Lizenztyp:

- **Godot Engine:** Eine Open Source Game Engine, die im Jahr 2014 von einem unabhängigen Entwicklerteam veröffentlicht wurde und daher vergleichsweise jung ist. Lizenziert unter der MIT Lizenz.
- **Cocos 2D-X:** Eine Open Source Game Engine, die von der Spielhersteller Firma Chukong Technologies gewartet und bereitgestellt wird. Linzensiert unter der MIT Lizenz.

Früher war es in der Regel so, dass Open Source Game Engines aufgrund der kleineren Entwicklungsteams etwas einfacher gestrickt waren als die proprietären Game Engines. Heute gibt es Open Source Game Engines wie zum Beispiel die Godot Game Engine, bei denen der Funktionsumfang annähernd so gross ist wie der der proprietären Game Engines. Da die Open Source Game Engines vorzugsweise von kleineren (Indie-) Spielehersteller verwendet werden, stehen sie nur begrenzt in Konkurrenz mit den proprietären Game Engines.

4.2 Marktpositionierung 2DGameSDK

Eine ausführliche Analyse der Marktpositionierungsmöglichkeiten für das erstellte Produkt wird im Rahmen der Projektumsetzung durchgeführt und ist Teil des technischen Berichts der Arbeit.

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 soll das Produkt auf verschiedenen Foren und Sozialen Medien präsentiert werden, sodass möglichst viele potenzielle Anwender erreicht werden können.

5 Systembeschreibung

5.1 Prozessumfeld

5.1.1 Spiel Initialisierung

Der Prozess der Spiel Initialisierung wurde im Rahmen des Vorprojekts bereits definiert (siehe Kapitel 3.1.1). Sofern eine Physik Engine eingebaut würde, würde der Initialisierungsprozess zusätzlich die Initialisierung der physikalischen Welt beinhalten. Das folgende UML Activity Diagramm zeigt den Ablauf der Spielinitialisierung:

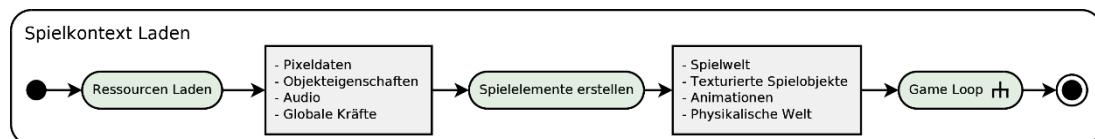


Abbildung 7 UML Activity Diagramm: Spielkontext laden

5.1.2 Game Loop

Der Game Loop ist das Kernstück der Game Engine, er wurde bereits im Rahmen des Vorprojekts definiert (siehe Kapitel 3.1.2) und kann nun entsprechend erweitert werden. So lange das Spiel läuft, wird der Game Loop zu einem fest definierten Takt wiederholt ausgeführt, um das Spiel und deren visuelle Darstellung zu aktualisieren. Das folgende UML Activity Diagramm zeigt die übergeordneten Prozesse des Game Loops, die in den darauffolgenden Kapiteln näher beschrieben werden:

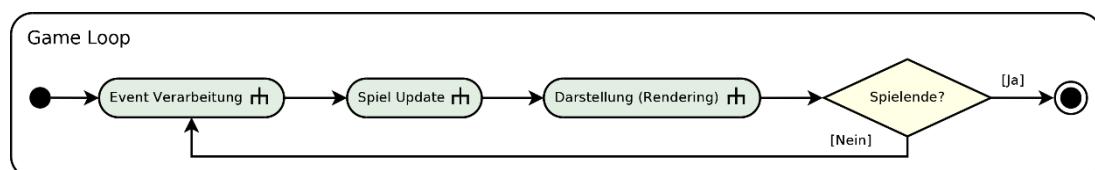


Abbildung 8 UML Activity Diagramm: Game Loop

5.1.3 Event Verarbeitung

Der erste Schritt des Game Loops ist die Verarbeitung der Spielereignisse. Spielereignisse erlauben es dem Spielhersteller, der die Game Engine verwendet eine individuelle Spiellogik aufzubauen, indem zum Beispiel die Spielereingaben mit Maus und Tastatur registriert und gewissen Aktionen im Spiel zugeordnet werden. Im Rahmen des Vorprojekts wurde dieses Konzept auf Basis des Observable Design Patterns umgesetzt (siehe Kapitel 3.1.3). Das folgende UML Activity Diagramm zeigt den Ablauf der Event Verarbeitung:



Abbildung 9 UML Activity Diagramm: Event Verarbeitung

5.1.4 Spiel Update

Der zweite Schritt im Game Loop ist die Aktualisierung des Spiels, bei diesem Schritt werden die Spielobjekte bewegt und die Animationen getaktet. Sobald die Positionen aktualisiert sind, wird die Kollisionskontrolle durchgeführt. Das folgende UML Activity Diagramm zeigt den Ablauf der Spiel Aktualisierung:

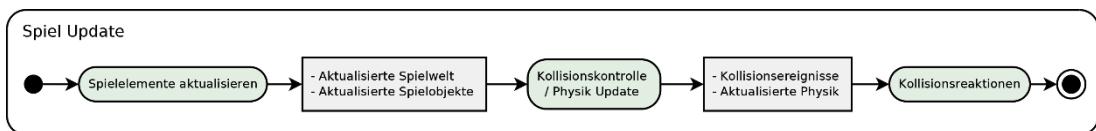


Abbildung 10 UML Activity Diagramm: Spiel Update

5.1.5 Rendering

Der letzte Schritt des Game Loops ist die Darstellung des aktuellen Spielzustands. Der Prozess der Spieldarstellung umfasst zum einen die visuelle Darstellung (Zeichnen von Texturen auf dem Bildschirm) zum anderen die akustische Darstellung (Ausgabe von Musik und Soundeffekten). Das visuelle Rendering wurde im Rahmen des Vorprojekts implementiert, in diesem Projekt wird zusätzlich eine Z-Index Sortierung der Spielelemente durchgeführt. Das folgende UML Activity Diagramm zeigt den Ablauf des visuellen Renderings:



Abbildung 11 UML Activity Diagramm: Visuelles Rendering

Beim Audio Rendering wird zum einen die Spielmusik reguliert (z.B. durch Start und Stop Befehle) zum anderen werden die verschiedenen Soundeffekte, die z.B. von Spielobjekten ausgelöst werden gemäss deren Ursprungsort gefiltert und abgespielt. Das folgende UML Activity Diagramm zeigt den Ablauf des Audio Renderings:

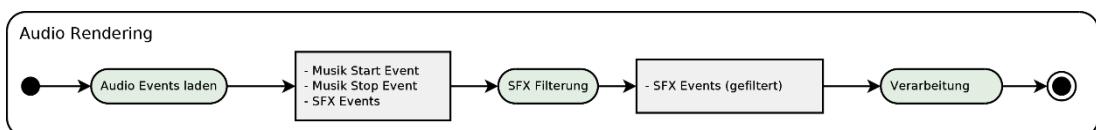


Abbildung 12 UML Activity Diagramm: Audio Rendering

5.1.6 Kamerasteuerung

Im Rahmen des Vorprojekts wurde keine Kamerasteuerung umgesetzt. Die Kamerasteuerung ist essenziell für viele Arten von Spielen, sie ermöglicht es zum Beispiel, dass die Kamera dem Spieler folgen kann. Im Rahmen dieses Projekts soll eine einfache und robuste Lösung für die Kamerasteuerung bereitgestellt werden, diese wird anhand des folgenden UML Activity Diagramms beschrieben:

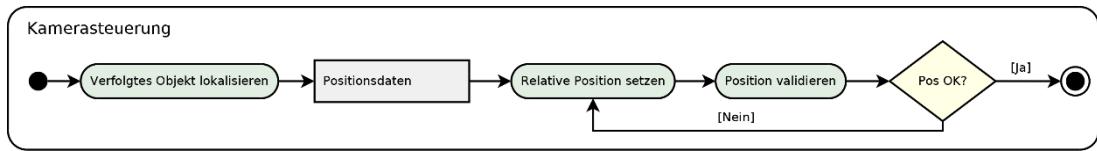


Abbildung 13 UML Activity Diagramm: Kamerasteuerung

5.2 Systemumfeld

5.2.1 Simple and Fast Multimedia Library SFML

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.

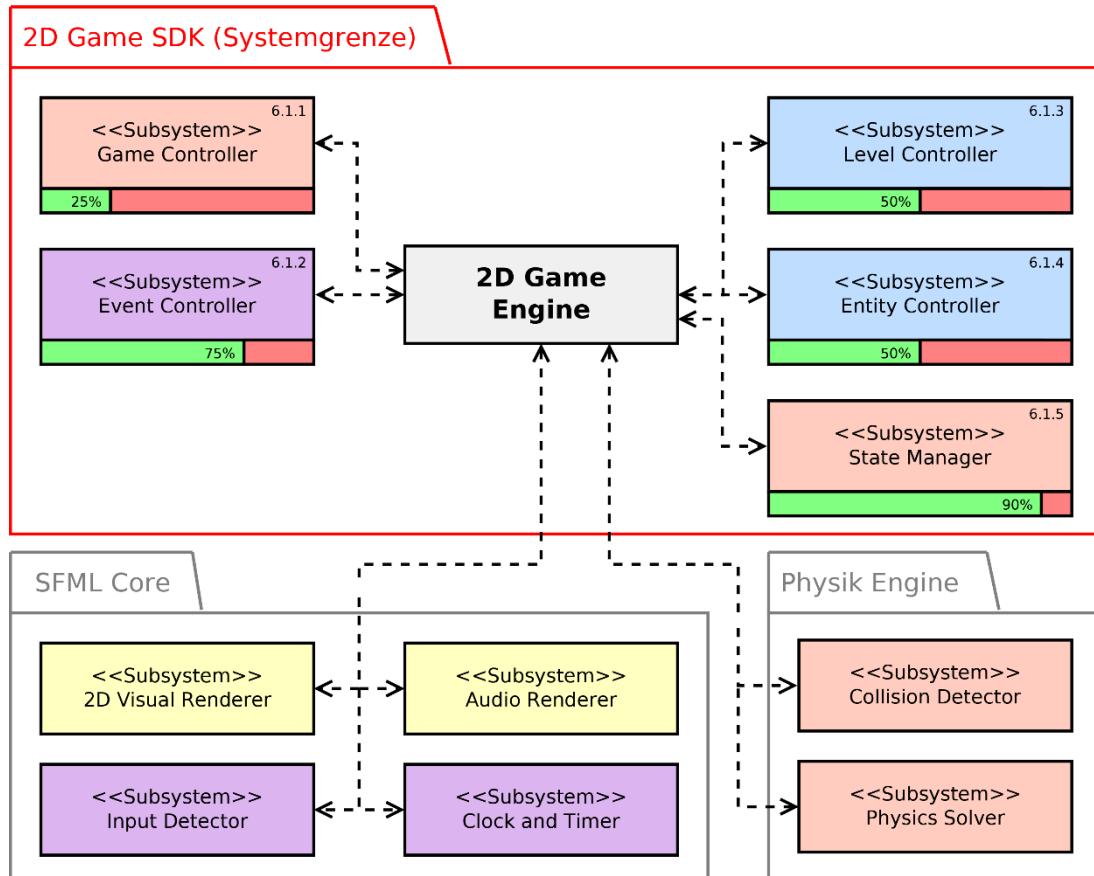
5.2.2 Physik Engines

Auf Basis eines Variantenstudium im Rahmen dieser Arbeit soll darüber entschieden werden, ob auch eine Physik Engine in das Produkt integriert wird. Die Physik Engine würde in diesem Fall die folgenden Aufgaben übernehmen:

- **Kollisionserkennung:** Erkennung der Kollisionen mit Hilfe des Sensor Konzeptes der Physik Engines.
- **Physikalische Simulation:** Materialien und Kräfte können mit der Physik Engine realistisch berechnet werden.

5.2.3 UML Kontextdiagramm

Das folgende Kontextdiagramm zeigt die Hauptkomponenten des zu entwickelnden Game SDK's (innerhalb der Systemgrenze), sowie die Verbindung zu Komponenten der externen Bibliothek SFML und der neuen Physik Engine (sofern umgesetztzt):



- > Das Kapitel im Pflichtenheft mit den Anforderungen an ein jeweiliges Subsystem ist oben rechts angegeben
- > Die Fortschrittsbalken unterhalb der Subsysteme entsprechen dem Entwicklungsstand nach Umsetzung des Vorprojekts

Abbildung 14 UML Kontextdiagramm 2DGameSDK (grün: Datenaufbereitung, gelb: Ausgabe, rot: Spiellogik, violett: Hardware Input)

5.2.4 Beschreibung der Subsysteme

Im folgenden Teil sind die Subsysteme des zu implementierenden Game SDK's und deren Funktionsweise beschrieben:

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 Spielwelt aktualisiert und auf dem Bildschirm dargestellt wird.

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 Spielenden oder auch komplexe selbst erstellte Events sein.

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.

Entity Controller

Der Entity Controller, ist für die Verwaltung aller beweglichen Objekte zuständig. Durch ihn können Objekte auf dem Scene Graph platziert werden, sodass sie in der visuellen und physikalischen Repräsentation berücksichtigt werden.

State Manager

Der State Manager verwaltet den aktuellen Spielstatus und stellt ihn während der Laufzeit des Spiels für alle Klassen zur Verfügung.

5.3 Prototypen

5.3.1 Funktionsumfang

Im Rahmen des Projekts sollen verschiedene Prototypen von 2D Spielen auf Basis des Game SDK's erstellt werden. Die Prototypen dienen einerseits zur Prüfung ob die Konzepte der 2DGameSDK auch praxistauglich sind, andererseits dienen sie als Beispiele, welche die Spieleentwickler zum einfachen Einstieg als Referenz verwenden können. Die verschiedenen Prototypen sollen den folgenden Funktionsumfang abbilden (mind. 1 Beispiel pro Funktionalität):

- **Steuerbarer Spieler:** Animierter Spieler der sich durch Keyboard Steuerung in der Spielwelt fortbewegen kann.
- **Spielwelt:** Eine Spielwelt bestehend aus soliden und nicht-soliden Materialien.
- **Nicht-Spieler-Charaktere:** Gegner die dem Spieler Schaden zufügen können.
- **Gesteuerte Kamera:** Eine Kamera, die dem Spieler durch den Level folgt.
- **In-Game Overlay Display:** Eine Anzeige, die den Spieler über den aktuellen Spielzustand informiert.
- **Abschlussbedingungen:** Konditionen unter welchen das Spiel zu einem erfolgreichen oder nicht erfolgreichen Abschluss gebracht werden.

5.3.2 Spiel Typen

Im Rahmen des Projekts sollen die folgenden Typen von Spielen als Prototypen umgesetzt werden:

- **Arcade Bottom Up Shooter:** Ein einfaches Arcade Spiel, bei dem der Spieler mit einem Raumschiff immer weiter nach oben fliegt und dabei feindliche Raumschiffe zerstört. Für dieses Spiel ist vor allem eine Intakte Kollisionsdetektion und die Implementation von Projektilen notwendig.
- **Roleplay Game:** Ein Spiel, in dem der Spielende den Helden von oben sieht und ihn frei in einer 2D Welt bewegen kann. Ein wichtiges Feature für diesen Spieltyp ist eine gut implementierte Kameraführung.
- **2D Platformer:** Ein Spiel in der Seitenansicht, bei dem der Spieler den Helden durch Laufen und Springen zum Ziel bewegen kann. Für diesen Prototyp wird idealerweise eine Physikengine verwendet, wodurch die Schwerkraft beim Springen einfach simuliert werden kann.

5.4 Variantenstudium

5.4.1 Physik Engine

Wie anfangs erwähnt, soll im Rahmen des Projektes evaluiert werden, ob sich der Einsatz einer externen Physik Engine für die Berechnung von Kollisionen und Reaktionskräften lohnen würde. Konkret sollen die folgenden Optionen mit einander verglichen werden:

1. **Keine Physik Engine:** Die Kollisionen würden auf Basis einer Eigenlösung berechnet werden, Reaktionskräfte könnten so jedoch nicht berechnet werden, da die Implementation einer vollumfänglichen Physik Engine den Rahmen dieser Arbeit überschreiten würde. Die Objekte wären folglich nicht physikalisch.
2. **Box2D Physik Engine:** Die C++ basierte Physik Engine Box2D ist bewährt und wird auch bei der Implementation von bekannten Spielen, wie z.B. Angry Birds eingesetzt. Sie ist gut dokumentiert und steht unter der MIT Lizenz (Permissive Open Source).
3. **Chipmunk Physik Engine:** Die C basierte Physik Engine Chipmunk wird bereits in der Game Engine Cocos 2D-X sowie in einigen Spielen für die Wii und die Playstation Portable eingesetzt. Sie ist laut eigenen Angaben etwas schneller als Box2D, wobei dies noch mit einem eigenen Benchmarking überprüft werden muss. Genau wie Box2D steht Chipmunk ab Version 7.0.0 unter der MIT Lizenz.

Die verschiedenen Varianten sollen auf die folgenden Kriterien überprüft werden:

- **Kompatibilität:** Möglichkeiten und Aufwände, um das Konzept in die bisherige Struktur der Game Engine einzubauen.
- **Features:**
 - o Kollisionssensoren (ohne Physik)
 - o Simulation von physikalischen Kräften und Materialien
 - o Interpolation (mehrere Physik Berechnungen pro TimeStep)
- **Performance:** Maximale Bildwiederholrate mit 1'000 Objekten und 10 (Kraft- und Geschwindigkeits-) Interpolationen pro Zeitschritt.
- **Dokumentation:** Vollständigkeit, Umfang der Beispiele, API Dokumentation.

Die Ausführliche Evaluation wird im Rahmen der Projektumsetzung durchgeführt, die Resultate werden im technischen Bericht aufgeführt.

5.5 Randbedingungen

5.5.1 C++ Standard

Der im Rahmen dieses Projekts erstellte Source Code wird für den aktuellsten C++ Standard C++17 implementiert und auf die Kompilierung mit der GNU Compiler Collection (GCC) [6] ausgerichtet.

5.5.2 SFML Version

Für dieses Projekt wird die neuste Version der SFML Library (SFML 2.5.1) [1] verwendet.

5.5.3 Zusätzliche externe Ressourcen

Die Abhängigkeit zu externen Code Ressourcen (Libraries) soll minimal gehalten werden. Sofern es unvermeidlich ist Code Ressourcen von Drittanbieter einzusetzen, sollen nur Ressourcen eingesetzt werden, die zum einen mit allen gängigen Betriebssystemen kompatibel sind, und zum anderen keinen restriktiven Copyright Lizenzen unterliegen.

5.6 Qualitätssicherung

5.6.1 Source Code

Der Source Code des Produkts soll den Vorgaben der IsoCPP C++ Core Guidelines [3] gerecht werden. Die Namensgebung der Code Strukturelemente (Funktionen, Klassen, Methoden) soll über das gesamte Projekt konsistent gehalten werden.

5.6.2 Tests

Um die Qualität der Software gewährleisten zu können, sollen für die Zentralen Komponenten und Prozesse des Game SDK's Unitests geschrieben werden. Im Rahmen der Unitests sollen die folgenden Punkte abgedeckt werden:

- **Softwareverhalten:** Sicherstellen, dass die Software bei gegebenem Input den erwarteten Output generiert.
- **Speichermanagement:** Sicherstellen, dass keine Objekte unnötig dupliziert werden und dass Objekte nach dem Gebrauch vom Speicher gelöscht werden.
- **Prototypen:** Die verschiedenen Prototypen, die im Rahmen des Projekts erstellt werden, dienen als Test für den praktischen Einsatz der 2DGameSDK. Erkenntnisse aus der Erstellung der Prototypen können so bei der Implementation der Game Engine berücksichtigt werden, um ggf. die Konzepte zu optimieren.

6 Anforderungen

6.1 Funktionale Anforderungen

Die funktionalen Anforderungen werden den einzelnen Subsystemen zugeordnet, sie sind im folgenden Teil aufgelistet:

6.1.1 Game Controller (Functional Game Controller)

Tabelle 1 Funktionale Anforderungen: Game Controller Subsystem

ID	Prio	Beschreibung
FG1	Muss	Der Spieleentwickler 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.
FG2	Muss	Der Spieleentwickler 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.
FG3	Muss	Der Spieleentwickler möchte die Möglichkeit haben, Audioeffekte und eine Hintergrundmusik zu definieren, sodass das Spielerlebnis mit akustischen Elementen erweitert werden kann.
FG4	Muss	Der Spieleentwickler 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)
FG5	Wunsch	Der Spieleentwickler möchte die Möglichkeit haben die Taktfrequenz des Spiels unabhängig von der Bildwiederholrate zu steuern, sodass mehrere Spieldurchläufe pro Darstellungsschritt berechnet werden können und das Spiel somit präziser berechnet wird.

6.1.2 Event Controller (Functional Event Controller)

Tabelle 2 Funktionale Anforderungen: Input / Event Controller Subsystem

ID	Prio	Beschreibung
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.
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.
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.

6.1.3 Level Controller (Functional Level Controller)

Tabelle 3 Funktionale Anforderungen: Level Controller Subsystem

ID	Prio	Beschreibung
FL1	Muss	Der Spieleanwickler 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.
FL2	Wunsch	Der Spieleanwickler 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.
FL3	Wunsch	Der Spieleanwickler 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.

6.1.4 Entity Controller (Functional Entity)

Tabelle 4 Funktionale Anforderungen: Entity Controller Subsystem

ID	Prio	Beschreibung
FI1	Muss	Der Spieleanwickler 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.
FI2	Wunsch	Der Spieleanwickler 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.
FI3	Wunsch	Der Spieleanwickler möchte die Möglichkeit Licht emittierende Spielobjekte zu definieren, sodass die Darstellung der Spielwelt und der Spielobjekte durch die Lichtemitter beeinflusst wird.

6.1.5 State Manager (Functional State Manager)

Tabelle 5 Funktionale Anforderungen: State / Object Manager Subsystem

ID	Prio	Beschreibung
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.
FS2	Muss	Der Spieleanwickler möchte die Möglichkeit haben individuelle Attribute und Klassen im Spielzustand zu speichern, sodass diese jederzeit im Spiel verfügbar sind.

6.1.6 Prototypen (Functional Prototypes)

Tabelle 6 Funktionale Anforderungen: Prototypen

ID	Prio	Beschreibung
FP1	Muss	<p>Der Spieleanwickler 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»)
FP2	Muss	<p>Der Spieleanwickler 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
FP3	Muss	<p>Der Spieleanwickler 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»)

6.2 Technische Anforderungen

Tabelle 7 Technische Anforderungen

ID	Prio	Beschreibung
T1	Muss	Das Produkt (Softwareentwicklungspaket) 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.
T2	Muss	Die Prozesse sollen im sinnvollen Massen parallel ablaufen, sodass moderne Multi-Core Prozessoren möglichst gut ausgelastet werden können.
T3	Muss	Die Frame Zyklen müssen präzise getaktet sein, sodass ein flüssiges Spielerlebnis realisiert werden kann.
T4	Wunsch	Bei zu hoher Zeitüberschreitung soll ein Frame übersprungen werden sodass die Darstellung mit der Spiellogik wieder synchron ist.
NT1	Abgr.	Das Produkt muss nicht für die Bereitstellung auf Mobiltelefonen oder Tablets geeignet sein.

6.3 Anforderungen rendering Qualität

Tabelle 8 Anforderungen rendering Qualität

ID	Prio	Beschreibung
Q1	Muss	Das Rendering soll eine Double- oder Triple Buffering Strategie verfolgen sodass die Darstellung nahtlos und flüssig erscheint.
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).
Q3	Muss	Die Framerate des Spiels soll stabil bei 50Hz sein, wenn 1'000 kollisionsfähige Spielobjekte gleichzeitig im Spiel sind (Wunschkriterium 10'000 Spielobjekte).

7 Testing

7.1 Konzept

Zum Testen der Game Engine werden im Rahmen dieser Arbeit mehrere unterschiedliche Prototypen erstellt, welche die einzelnen Komponenten der Game Engine verwenden und illustrieren. (siehe Kapitel 5.1.3 und 6.1.6) Die Prototypen dienen als Tests für die Praxistauglichkeit der Game Engine. Darüber hinaus werden für einzelne Prozesse, die als fehleranfällig einzustufen sind, entsprechende Unit Tests erstellt. Die Performance der verschiedenen Physik Engines die für die Integration in die Game Engine in Frage kommen, wird anhand selbst erstellter Performancetests evaluiert.

7.2 Technologien

Für die Definition und Durchführung der Unit Tests soll ein geeignetes Testframework verwendet werden. Für die Erstellung der Prototypen und die Performancetests der Physik Engines werden keine zusätzlichen Technologien eingesetzt.

8 Projektmanagement

8.1 Projektorganisation

8.1.1 Prozessmodell

Für die Umsetzung des Projekts wird mit dem agilen Prozessmodell Scrum gearbeitet, die Sprint Länge beträgt jeweils 2 Wochen.

8.1.2 Meetings

Die Projektmeetings mit dem betreuenden Dozenten finden alle zwei Wochen statt. An diesen Meetings werden der aktuelle Projektstand sowie allfällige Fragen zum Projekt diskutiert.

8.1.3 Meilensteine

Im Folgenden die definierten Meilensteine des Projekts basierend auf den in diesem Dokument festgehaltenen Zielsetzungen:

Meilenstein	Beschreibung	Datum
MS0	Finale Festlegung der Anforderungen, Pflichtenheft	18.10.2019
MS1	Spiellogik Erweiterung: Kamera, Kollisionen, Physik Umsetzung gemäss Evaluation	01.11.2019
MS2	Erweiterung Funktionsumfang: In-Game Overlay Display, Audio	29.11.2019
MS3	Prototypen: Platformer, Arcade, Roleplay Game	10.01.2020
MS4	Prototypen, technischer Bericht, Video Pojetabschluss	16.01.2020

Die detaillierte Projektplanung orientiert sich an den Zeitvorgaben dieser Meilensteine.

8.2 Zeitplan

Das Projekt wird gemäss Scrum Planung in 8 Sprints mit jeweils einer Dauer von 2 Wochen unterteilt. Die Sprints an denen parallel die Arbeiten an den Dokumentationen laufen, werden mit 50% Auslastung für die Implementation gewertet. Pro Sprint werden 40 Arbeitsstunden bei Vollzeitbeschäftigung angenommen:

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, Kollisionsdetektion																	
		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						
			Pflichtenheft									Poster, Video, Technischer Bericht						
MS0 18.10.19						MS1 01.11.19	MS2 29.11.19						MS3 14.06.19 (Projektende)					

9 Referenzen

- [1] L. Gomila, «SFML Home Page,» [Online]. Available: <https://www.sfml-dev.org/>. [Zugriff am 15 März 2019].
- [2] 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].
- [3] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, New Jersey: Addison-Wesley, 2001.
- [4] TIGA, «About TIGA and our Industry,» 2019. [Online]. Available: <https://tiga.org/about-tiga-and-our-industry>. [Zugriff am 06 Oktober 2019].
- [5] Godot, «Godot Engine,» 2019. [Online]. Available: <https://godotengine.org/>. [Zugriff am 06 Oktober 2019].
- [6] Unity Technologies, «Unity Real-Time Development Platform,» 2019. [Online]. Available: <https://unity.com/>. [Zugriff am 06 Oktober 2019].
- [7] Epic Games, «What is Unreal Engine 4,» 2019. [Online]. Available: <https://www.unrealengine.com/>. [Zugriff am 06 Oktober 2019].
- [8] Unity Technologies, «Unity Home Page,» [Online]. Available: <https://unity.com/>. [Zugriff am 20 März 2019].
- [9] Free Software Foundation, Inc., «GCC, the GNU Compiler Collection,» 22 Februar 2019. [Online]. Available: <https://gcc.gnu.org/>. [Zugriff am 20 März 2019].
- [10] Khronos Group Inc., «OpenGL Home Page,» [Online]. Available: <https://www.opengl.org/>. [Zugriff am 20 März 2019].
- [11] TIGA, «UK Developers Show Renewed Interest in Consoles,» 27 August 2014. [Online]. Available: <https://tiga.org/news/uk-developers-show-renewed-interest-in-consoles>. [Zugriff am 06 Oktober 2019].
- [12] Statista, «Game engines used by video game developers UK 2014,» 2019. [Online]. Available: <https://www.statista.com/statistics/321059/game-engines-used-by-video-game-developers-uk/>. [Zugriff am 14 Oktober 2019].

10 Glossar

Begriff	Erläuterung
Asset Store	Ein Asset Store ist ein Element von verschiedenen Game Engines das es ermöglicht Assets (Texturen, Materialien, Animationen sonstige Spielelemente) vom Hersteller und meist auch von anderen Anwender zu beziehen. (Gratis oder gegen eine Gebühr)
Entity	Im Kontext dieses Projekts ist eine Entity (Einheit) ein bewegliches Spielobjekt.
Game Engine	Eine Game Engine ist ein Softwareprodukt, das es ermöglicht Computerspiele zu erstellen. Eine Game Engine kann entweder aus einer einfachen Library bestehen, oder gleich eigene Anwendungen mitliefern, um das Spiel zu designen, zu programmieren und zu kompilieren. (Leveleditor, Skript Sprache, Compiler für verschiedene Plattformen)
Game State	Der Game State oder Spielzustand beinhaltet im Kontext dieses Projekts den Zustand der Spielwelt und der Spielobjekte sowie die aktuellen Spieloptionen.
Leveleditor	Ein Leveleditor ist ein ausführbares Programm von verschiedenen Game Engines, mit dem es möglich ist, die unterschiedlichen Spielwelten und andere Spielinhalte zu erstellen.
Library (Softwarebibliothek)	Eine Softwarebibliothek ist eine Sammlung von Ressourcen, die für Computerprogramme bereitgestellt werden. Dazu gehören unter anderem Klassen, Typdefinitionen und Konfigurationsdaten.
Nicht-Spieler Charakter	Ein nicht-spieler-Charakter ist eine Entity die nicht vom Spieler gesteuert wird, sondern durch eine Form von künstlicher Intelligenz.
Physik Engine	Eine Physik Engine ist ein Softwareprodukt, das es ermöglicht physikalische Objekte und Prozesse zu simulieren. Meist in Form einer Library.
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].
Rendering	Rendering ist der Prozess mit dem vorhandene Logische Daten, wie z.B. ein 3D Modell oder eine Audiodatei in visuelle oder akustische Signale umgewandelt werden.
Royalty-Free	Lizenzmodell bei dem kein finanzieller oder rechtlicher Anspruch auf das vom Anwender erstellte Produkt (z.B. Einem Spiel, das mit der Game Engine des Lizenzgebers erstellt wurde) erhoben wird.
Tiles (Kachelgrafik)	Als Kachelgrafik (engl. tiles) wird eine Computergrafik bezeichnet, die mosaikartig zusammengesetzt ein vielfach größeres Gesamtbild ergibt. Die Kachel-Technik wird häufig in Computerspielen eingesetzt, da die Kacheln einfacher zu berechnen sind und weniger Arbeitsspeicher verbrauchen.
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.

11 Abbildungsverzeichnis

Abbildung 1 UML Activity Diagramm: Spiel Initialisierung (Stand Vorprojekt)	2
Abbildung 2 UML Activity Diagramm: Game Loop (Stand Vorprojekt)	2
Abbildung 3 UML Kommunikationsdiagramm: Event Verarbeitung (Stand Vorprojekt)	2
Abbildung 4 Beispiel Tile Grafik Konzept (Stand Vorprojekt)	3
Abbildung 5 Beispiel Scene Graph Konzept (Stand Vorprojekt)	3
Abbildung 6 Meistverwendete Game Engines in Grossbritannien im Jahr 2014 gemäss einer Umfrage des Wirtschaftsverbands für Spielehersteller TIGA. Quelle: [12]	5
Abbildung 7 UML Activity Diagramm: Spielkontext laden	7
Abbildung 8 UML Activity Diagramm: Game Loop	7
Abbildung 9 UML Activity Diagramm: Event Verarbeitung	8
Abbildung 10 UML Activity Diagramm: Spiel Update	8
Abbildung 11 UML Activity Diagramm: Visuelles Rendering	8
Abbildung 12 UML Activity Diagramm: Audio Rendering	8
Abbildung 13 UML Activity Diagramm: Kamerasteuerung	9
Abbildung 14 UML Kontextdiagramm 2DGameSDK (grün: Datenaufbereitung, gelb: Ausgabe, rot: Spiellogik, violett: Hardware Input)	10

12 Tabellenverzeichnis

Tabelle 1 Funktionale Anforderungen: Game Controller Subsystem	13
Tabelle 2 Funktionale Anforderungen: Input / Event Controller Subsystem	14
Tabelle 3 Funktionale Anforderungen: Level Controller Subsystem	14
Tabelle 4 Funktionale Anforderungen: Entity Controller Subsystem	14
Tabelle 5 Funktionale Anforderungen: State / Object Manager Subsystem	15
Tabelle 6 Funktionale Anforderungen: Prototypen	15
Tabelle 7 Technische Anforderungen	16
Tabelle 8 Anforderungen rendering Qualität	16