Technischer Bericht: 2DGameSDK

C++ Game SDK für 2D Games

Semesterarbeit «Projekt 2»

Studiengang:	Bsc. Informatik
Autor:	Alain Alistair Markóczy
Betreuer:	Dr. Urs Künzler
Auftraggeber:	Student
Datum:	14.06.2019

Versionskontrolle

Version	Datum	Beschreibung	Autor
0.1	14.05.2019	Entwurf	A.M.
0.2	02.06.2019	Revision Outline	A.M.
0.3	07.06.2019	Neue UML Diagramme	A.M.
1.0	14.06.2019	Finale Version	A.M.

Management Summary

Das Ziel der vorliegenden Projektarbeit war es, eine Game Engine für 2D Computerspiele in der Programmiersprache C++ zu erstellen. Die Engine soll es ermöglichen 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 SDK enthält unter anderem eine Scene Graph Implementation, Spielobjekte mit Animationen, eine Spielwelt auf Basis des Kachelgrafik Konzepts und ein ausgefeiltes Event Controlling System zur Implementation einer Spiellogik. Um die Funktionalität der SDK zu testen und um den Mehrwert zu bestimmen, den sie bei der Spieleentwicklung aufbringen kann, wurden im Rahmen dieser Arbeit mehrere Prototypen in Form von (nicht-funktionalen) Mini-Spielen erstellt. Im Rahmen eines Variantenstudiums wurde zusätzlich getestet, ob es mit den verwendeten Technologien auch möglich ist neben den 2D Objekten auch 3D Objekte auf dem Bildschirm darzustellen. Die ersten Ergebnisse sind vielversprechend, jedoch sind bei der vollständigen Integration dieses Features noch diverse Probleme zu erwarten.

Inhhaltsverzeichnis

1 Einleitung

1.1 Zweck des Dokuments

Dieses Dokument ist die technische Dokumentation zur Applikation die im Rahmen des Projekts «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`
- **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 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, sodass unterschiedliche Arten von 2D Games (Shooter, Adventure, Roleplay Game) mit wenig Aufwand erstellt werden können.

Für die Implementation von Animationen, soll im Rahmen dieses Projekts zusätzlich abgeklärt werden, ob allenfalls eine 3D Modellierung besser geeignet ist als eine reine 2D Modellierung. Das gewählte Framework SFML bietet hierzu auch eine geeignete Schnittstelle zu OpenGL [2] an.

2.2 Produktübersicht

Das Produkt ist auf den Definitionen des Frameworks SFML aufgebaut und erweitert dieses mit zusätzlichen Definitionen zum Erstellen einer Spiellogik. Im Rahmen dieser Projektarbeit steht das zu erstellende Produkt zwischen dem Framework SFML und dem Computerspiel, das der jeweilige Spieleentwickler erstellen möchte:

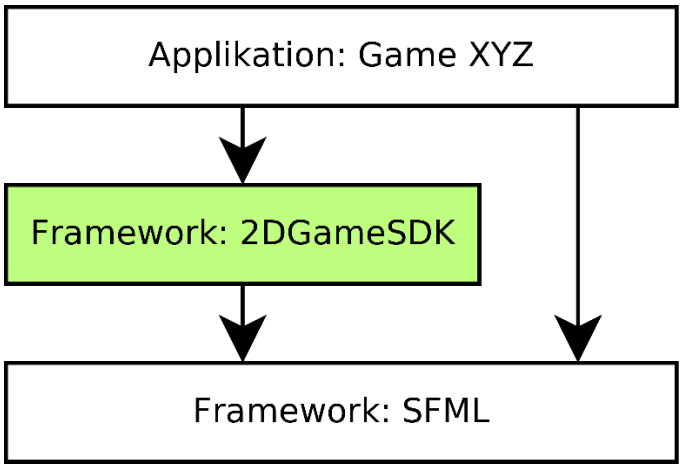

Abbildung 1 Produktübersicht 2DGameSDK

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

4.1 Technologiewahl

4.1.1 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) [5] ausgerichtet.

4.1.2 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 OSX 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.

4.1.3 OpenGL

Das Framework OpenGL [2] stellt eine offene Schnittstelle zur Grafikkarte des Computers bereit, mit der es unter anderem möglich ist 3D Objekte auf dem Bildschirm darzustellen. Um auch Kompatibilität mit älteren Grafikkarten zu gewährleisten, wurde im Rahmen dieses Projekts die OpenGL [2] Version 3.3 verwendet (Die SFML Library setzt voraus, dass eine OpenGL Version höher als 3.0 verwendet wird).

4.1.4 JSON Parser

Zum Lesen von JSON Dateien wurde auf die open Source JSON Parser Library von Niels Lohmann (siehe [6]) zurückgegriffen. Diese Library wird unter der MIT Lizenz verbreitet (open source, jedoch auch kommerziell einsetzbar).

4.1.5 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.

4.2 Globale Konzepte

4.2.1 Design Patterns

Für die Implementation des Event Managements wurde eine Variante des Observer/Observable Patterns angewendet. Alle Events des Spiels (Tick Event, Keyboard Event etc.) sind Observables im Sinne dieses Design Patterns. Ein beliebiges Objekt kann sich bei einem Observable registrieren, indem es dem Observable eine Callback Funktion übergibt. Diese Callback Funktion wird ausgelöst, wenn das entsprechende Event ausgelöst wird. Die Observables werden bei jedem Zyklus des Game Loops zu einem festen Zeitpunkt aktualisiert.

Für die Erstellung von komplexen Objekten wie zum Beispiel die Spielwelt werden Factory Methoden im Sinne des Factory Design Patterns bereitgestellt. Dies ermöglicht es die komplexen Nebenaufgaben bei der Erstellung des entsprechenden Objekts zu kapseln sodass der Nutzer mit einem einfachen Aufruf die Objekte erstellen kann. Factory Methoden erlauben es zusätzlich unterschiedliche Konstruktoren für das selbe Objekt bereitzustellen.

Für die Konzeption einiger Elemente wie zum Beispiel der Spielobjekte wird das Template Method Pattern angewendet, es werden abstrakte Parent Klassen definiert, welche dann bei der Implementation der Game SDK referenziert werden. Die genaue Spezifikation dieser Parent Klassen und die genaue Implementation derer Methoden kann vom Spielentwickler frei gewählt werden. Es werden aber dort wo es sinnvoll ist auch vordefinierte Implementationen zur Verfügung gestellt.

4.3 Architektur

4.3.1 SFML Komponenten

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.

4.3.2 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:

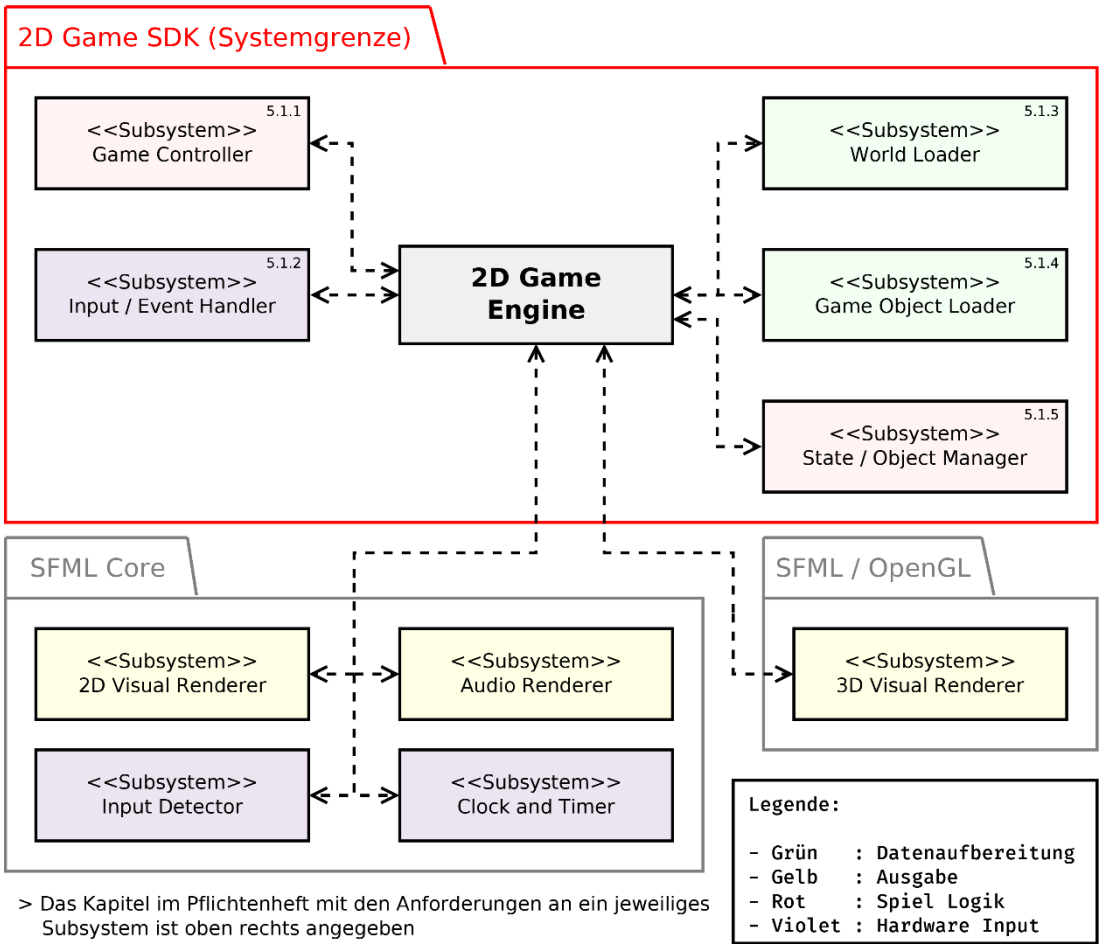

Abbildung 2 UML Kontextdiagramm 2DGameSDK

4.3.3 Beschreibung der Subsysteme

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

Game Controller

Das Game Controller Subsystem ist für die getaktete, wiederholte Abarbeitung des Spiel Kontrollflusses (Game Loops) zuständig (Siehe Kapitel 4.1.2). Bei der Initialisierung des Spiels, werden vom Game Controller Subsystem die notwendigen Ressourcenlader (World/Game Object Loader) initialisiert, welche beim Laden der Spielwelt zum Erstellen der Spielelemente verwendet werden. Der Game Loop wird so lange vom Game Controller Subsystem ausgeführt, bis das Spiel von einem anderen Thread aus unterbrochen wird.

Input / Event Handler

Das Input und Eventhandler Subsystem ist dafür zuständig detektierte Spielereingaben (z.B. mit dem Spielcontroller oder der Tastatur) oder andere Arten von Events konkreten Ereignissen und Interaktionen im Spiel zuzuordnen. Die Spielereingaben können mit dem Input Detector System aus der SFML Library [1] erkannt werden.

World Loader

Das World Loader Subsystem ist dafür zuständig die Spielwelt zu laden. Die Spielwelt besteht im Wesentlichen aus einer mehrschichtigen 2-dimensionalen Anordnung von Level Kacheln (Tiles). Die Anordnung wird anhand einer Datei im JSON Format definiert, welche vom World Loader Subsystem gelesen und für den Aufbau der Spielwelt verwendet wird. Jede Kachel in der Spielwelt besitzt eine Textur aus einer entsprechenden Bilddatei, sowie bestimmte Materialeigenschaften die z.B. beschreiben ob die Kachel fest oder durchlässig ist. Die Materialeigenschaften werden in den jeweiligen Materialklassen (im Source Code) definiert, sie werden analog zu den Texturen durch das World Loader Subsystem den entsprechenden Abschnitten in der Spielwelt zugeordnet.

Game Object Loader

Das Game Object Loader Subsystem ist dafür zuständig die Spielobjekte mit den jeweiligen Attributen und Verhaltensregeln ins Spiel zu laden. Die Texturdaten für die Darstellung der Objekte werden aus Bilddateien gelesen und in den entsprechenden Spielobjektklassen referenziert. Die Spielobjektklassen (Erweiterungen der Klasse „GameObject“), die die jeweiligen Verhaltensregeln enthalten, werden bei Bedarf vom Game Object Loader Subsystem erstellt und an das laufende Spiel übergeben.

State / Object Manager

Das State and Object Manager Subsystem ist dafür zuständig jegliche Daten zum aktuellen Spielzustand festzuhalten, so dass diese für andere Subsysteme zu jedem Zeitpunkt verfügbar und aktuell sind. Der Spielzustand ist unter Anderem definiert durch die Spielerposition, die Kameraposition, die Objekte die aktuell im Spiel sind (und deren Eigenschaften) und durch die abgelaufene Spielzeit in Anzahl Frames.

4.3.4 UML Package Diagramm

Die Game SDK ist in die folgenden Packages (logische Untergruppen) unterteilt:

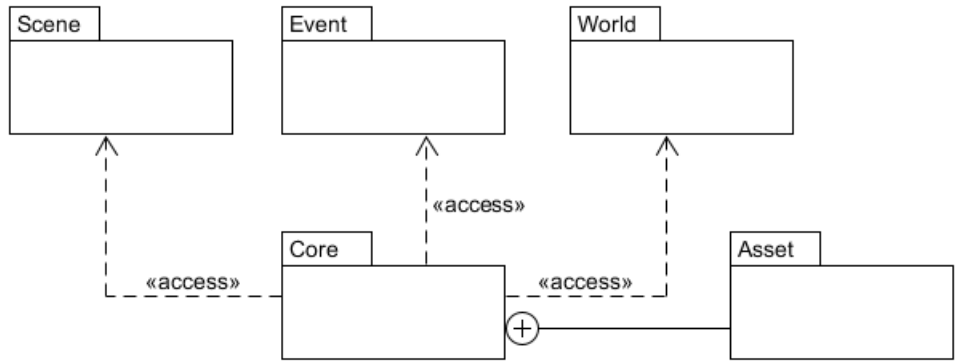

Abbildung 3 UML Package Diagramm 2DGameSDK

Im Paket «Core» ist die Spiellogik definiert, aus diesem Grund werden von dort aus alle anderen Pakete referenziert. Das Paket «Asset» das Teil vom «Core» Paket ist, enthält alle nötigen Operationen zum Verwalten von Ressourcen wie Bild- und Audiodateien. Das Paket «Scene» enthält alle nötigen Implementationen für die Erstellung des Scene Graphs. Im Paket «Event» ist das gesamte Event Handling spezifiziert und im Paket «World» die Objekte und Factory Methoden im Zusammenhang mit der Spielwelt.

4.4 Implemenation

4.4.1 UML Klassendiagramm: Core Package

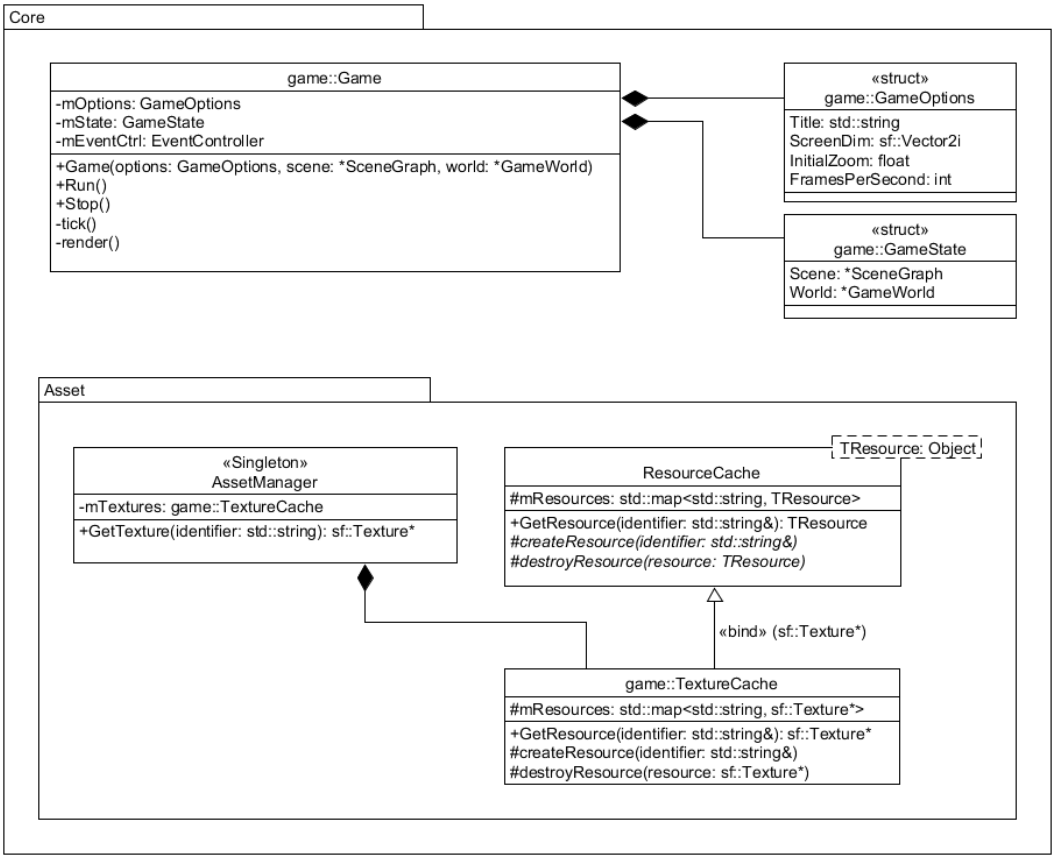

Abbildung 4 UML Klassendiagramm: Core Package

Die sogenannte «AssetManager» Klasse ist für das Laden und Cachen von Ressourcen wie Texturen und Audiodateien zuständig. Das Caching erfolgt durch die übergeordnete abstrakte Klasse «ResourceCache». Der «ResourceCache» speichert alle Ressourcen, die bereits zuvor abgerufen wurden und gibt bei erneutem Abruf der selben Ressource die gespeicherte Ressource zurück. Die Klasse «Game» repräsentiert das Herzstück der Spielengine, von dort aus wird der grafische Kontext (bzw. das Applikationsfenster) erstellt und das Spiel gestartet. Der Game Loop wird auch in der «Game» Klasse ausgeführt. Die Struktur «GameOptions» wird dem Spiel bei Erstellung übergeben, sie definiert alle Optionen, die nicht den Spielablauf beeinflussen. Der «GameState» enthält alle relevanten Elemente, die den aktuellen Zustand des Spiels definieren und wird daher laufend aktualisiert.

4.4.2 UML Klassendiagramm: World Package

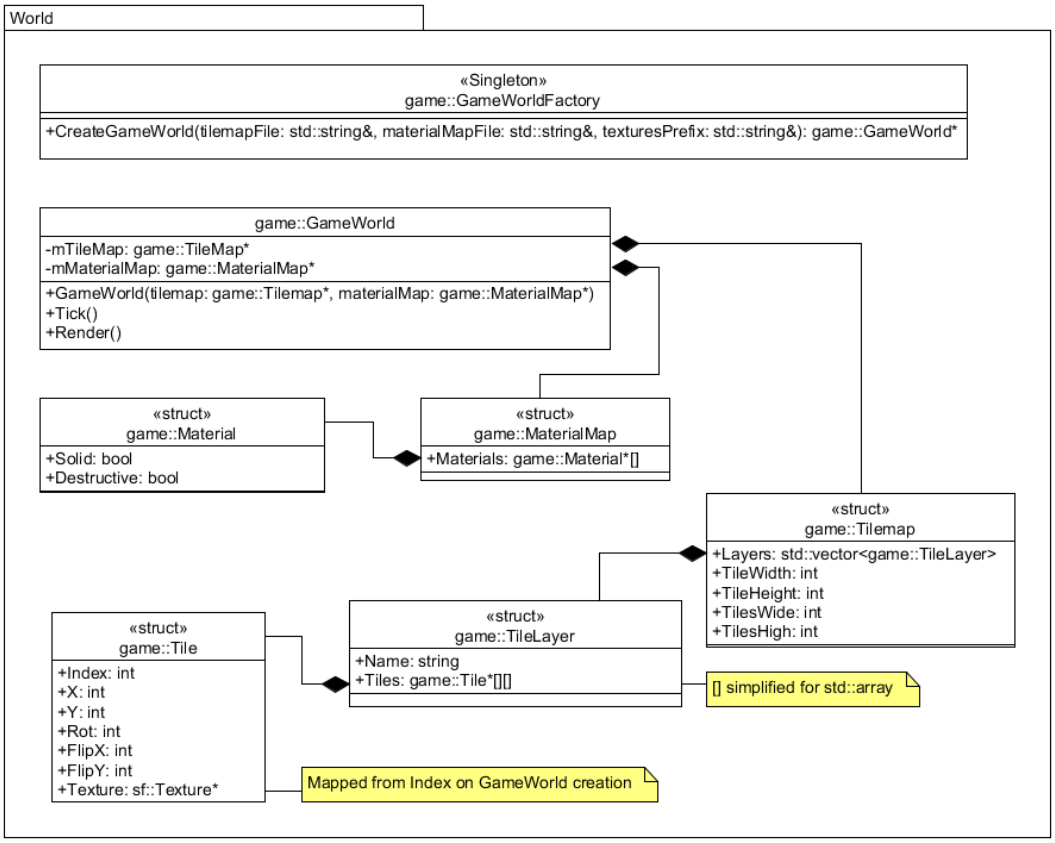

Abbildung 5 UML Klassendiagramm: World Package

Die strukturelle Hauptkomponente der Spielwelt ist die sogenannte «Tilemap» Struktur (Kachelanordnung). Die «Tilemap» ist als eine mehrschichtige Anordnung von texturierten Kacheln einer bestimmten (frei wählbaren, aber konstanten) Grösse zu verstehen. Die Idee stammt aus Implementationen von älteren Spielen und ist eine Methode zur Optimierung des Arbeitsspeichers und der Rendering Geschwindigkeit. Die Kacheln sollen zusätzlich noch an ein Material gebunden werden können, mittels der Materialien ist es möglich Eigenschaften zu definieren, wie zum Beispiel ob ein Material fest oder durchlässig ist. Die Verbindung zwischen Kachel und Material erfolgt über die sogenannte «MaterialMap». Zuletzt werden all diese Komponenten im Spielwelt («GameWorld») Objekt vereint. Um die Erstellung der Spielwelt für den Spielentwickler einfach zu halten wird eine entsprechende Factory Methode in der statischen Klasse «GameWorldFactory» bereitgestellt.

4.4.3 UML Klassendiagramm: Scene Package

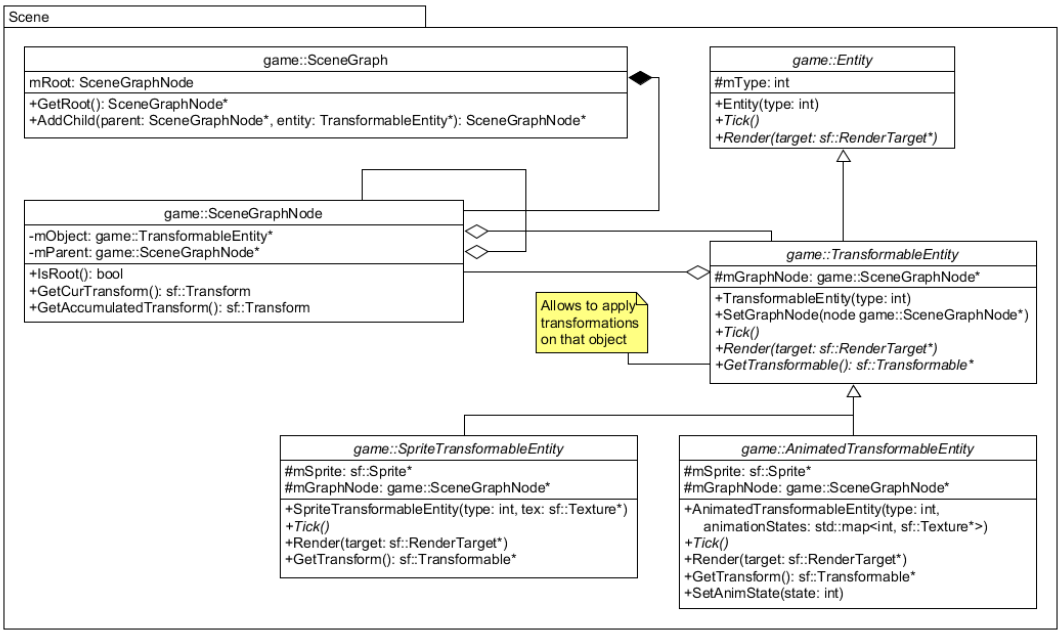

Abbildung 6 UML Klassendiagramm: Scene Package

Die Spielobjekte werden im Kontext der Game SDK als «Entities» (Einheiten) bezeichnet. Die übergeordnete abstrakte Klasse «Entity» verlangt im Konstruktor die Angabe eines Typs, dieser ist eine frei wählbare Integer-Zahl und dient dem Spielentwickler zur Wiedererkennung von Einheiten eines bestimmten Typs. Jedes Spielobjekt erhält bei der Erstellung zusätzlich eine einzigartige ID. Eine abstrakte Erweiterung der Klasse «Entity» ist die Klasse «TransformableEntity» welche neben den Eigenschaften von Spielobjekten noch die Eigenschaften eines transformierbaren Objektes definiert. Diese transformierbaren Einheiten sind die Grundelemente des Scene Graphs. Die Game SDK stellt die folgenden vordefinierten Spezialisierungen der Klasse TransformableEntity bereit:

- **SpriteTransformableEntity:** Die Klasse «SpriteTransformableEntity» ist eine einfache Implementation der Klasse «TransformableEntity», sie erhält bei der Erstellung eine Referenz auf eine Textur, diese Textur definiert die Darstellung des Spielobjekts.
- **AnimatedTransformableEntity:** Die Klasse «AnimatedTransformableEntity» ist eine «TransformableEntity», die im Gegensatz zur «SpriteTransformableEntity» mehrere Texturen enthalten kann. Der Animationsstatus der Klasse (siehe Funktion «SetAnimState») definiert welche Textur im nächsten Zyklus des Game Loops dargestellt werden soll.

Alle Objekte des Spiels sind in der sogenannten «SceneGraph» Klasse gespeichert. Der «SceneGraph» ist ein gerichteter azyklischer Graph, der es ermöglicht die Objekte in einer Hierarchie anzuordnen. Diese Hierarchie definiert unter anderem, ob sich zwei Objekte relativ zu einander bewegen. Dies ist zum Beispiel der Fall, wenn das zweite Spielobjekt ein Teil des ersten Spielobjektes ist (z.B. der Rotor eines Hubschraubers):

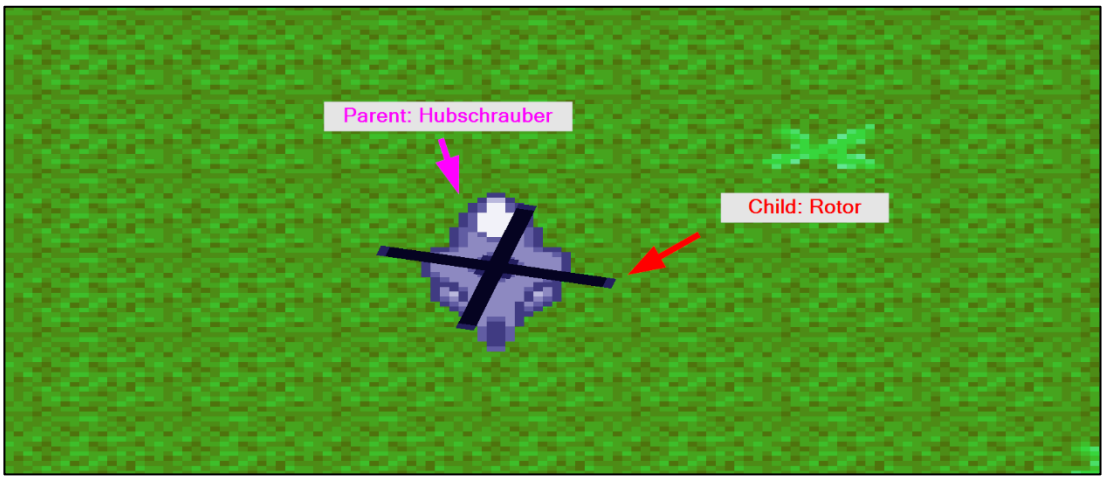

Abbildung 7 Scene Graph Demonstration

Die einzelnen Knoten im Scene Graph enthalten die beweglichen Spielobjekte, die anhand des Scene Graphs hierarchisch zu einander angeordnet werden können. Bei der Erstellung des Scene Graphs erstellt dieser das Root Element, dieses hat keine Transformation und dessen Ursprung liegt am Nullpunkt des Welt Koordinatensystems.

4.4.4 UML Klassendiagramm: Event Package

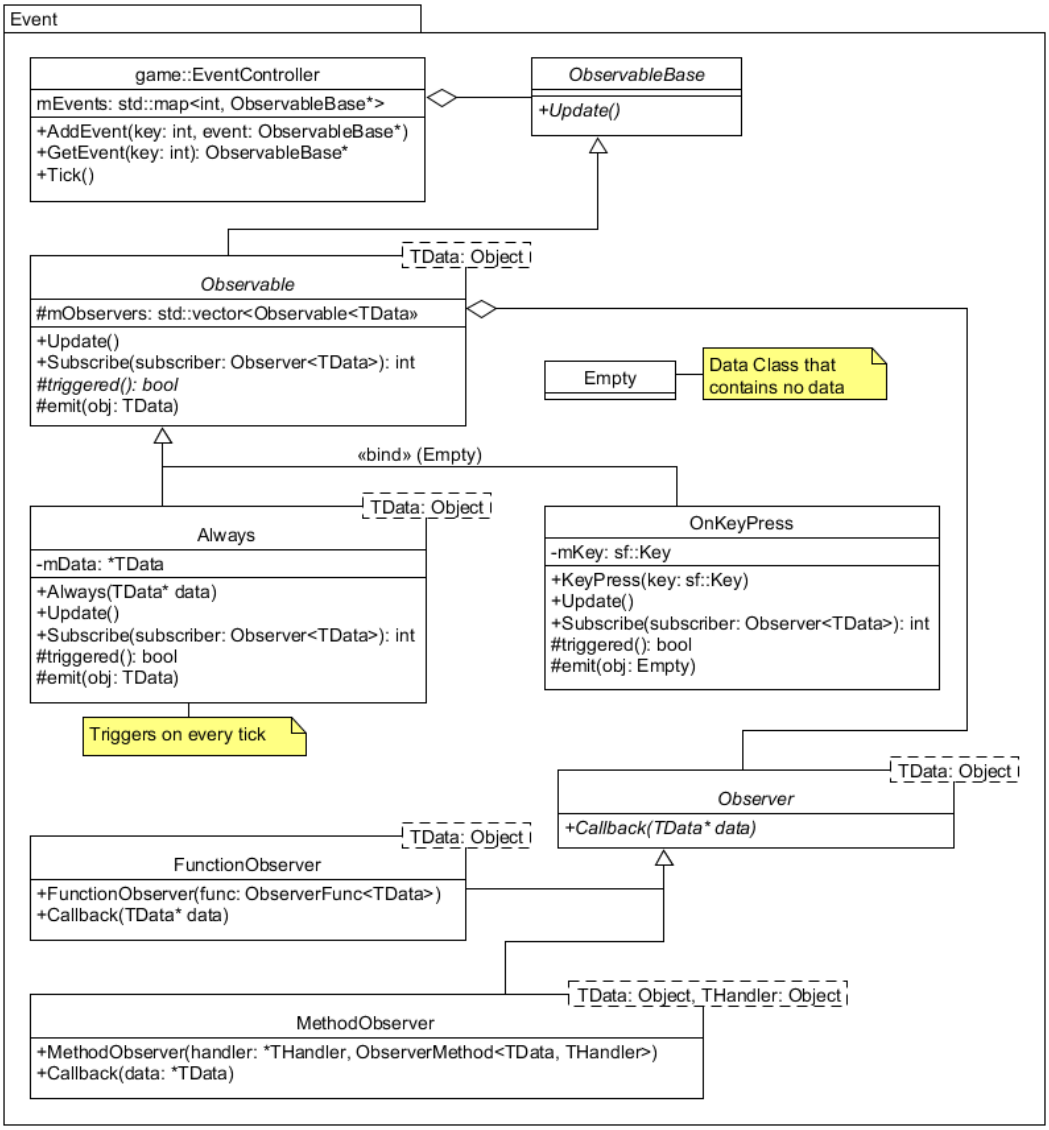

Abbildung 8 UML Klassendiagramm: Event Package

Wie bereits im Kapitel «Design Patterns» erwähnt, basiert das gesamte Event Handling auf dem Observer/Observable Design Pattern. Alle Events im Spiel sind als Observables definiert, diese erlauben sich mittels eines Observers bei diesen Observables zu registrieren und auf die Auslösung des entsprechenden Events zu reagieren. Die Events werden bei jedem Zyklus zu einem festgelegten Zeitpunkt (vor Aktualisierung der Spielwelt und den Spielobjekten) mittels «EventController» Klasse aktualisiert. Ein Event kann nur dann ausgelöst werden, wenn die entsprechende «Update» Funktion ausgeführt wird, so wird durch den «EventController» sichergestellt, dass die Observables immer zum gleichen Zeitpunkt des Game Loops ausgelöst werden. Wenn ein Observable ausgelöst wird, übergibt dieses die Event Daten an den Observer, dieser kann damit entweder eine Callback Funktion im globalen Scope ausführen (Spezialisierung «FunctionObserver») oder eine Callback Methode einer bestimmten Klasse (Spezialisierung «MethodObserver»). Letztendlich können aber durch den Spieleentwickler auch weitere Observer Implementationen definiert werden.

Das folgende Beispiel zeigt eine mögliche Observer<>Observable Interaktion unter Verwendung der «MethodObserver» Spezialisierung. Das Observable reagiert auf eine Tastatureingabe des Spielers («OnKeyPress») und löst dabei die entsprechende Callback Methode vom Observer aus, dieser löst zuletzt die zuvor definierte Callback Methode in der «Player» Klasse aus, sodass diese auf das Event (z.B. durch Fortbewegung) reagieren kann:

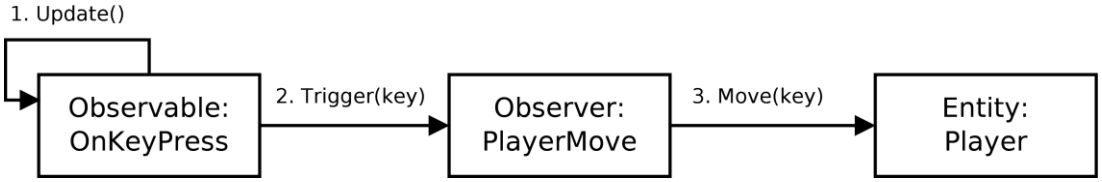

Abbildung 9 Beispiel Event Workflow

Die Spezialisierungen «Always» und «OnKeyPress» sind vordefinierte Observables die von der Game SDK zur Verfügung gestellt werden. Die «EventController» Klasse verwaltet die Events bzw. Observables, sie ist dafür zuständig, dass die Observables immer zum gleichen Zeitpunkt im Game Loop ausgelöst werden:

4.4.5 UML Sequenzdiagramm: Initialisierung

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

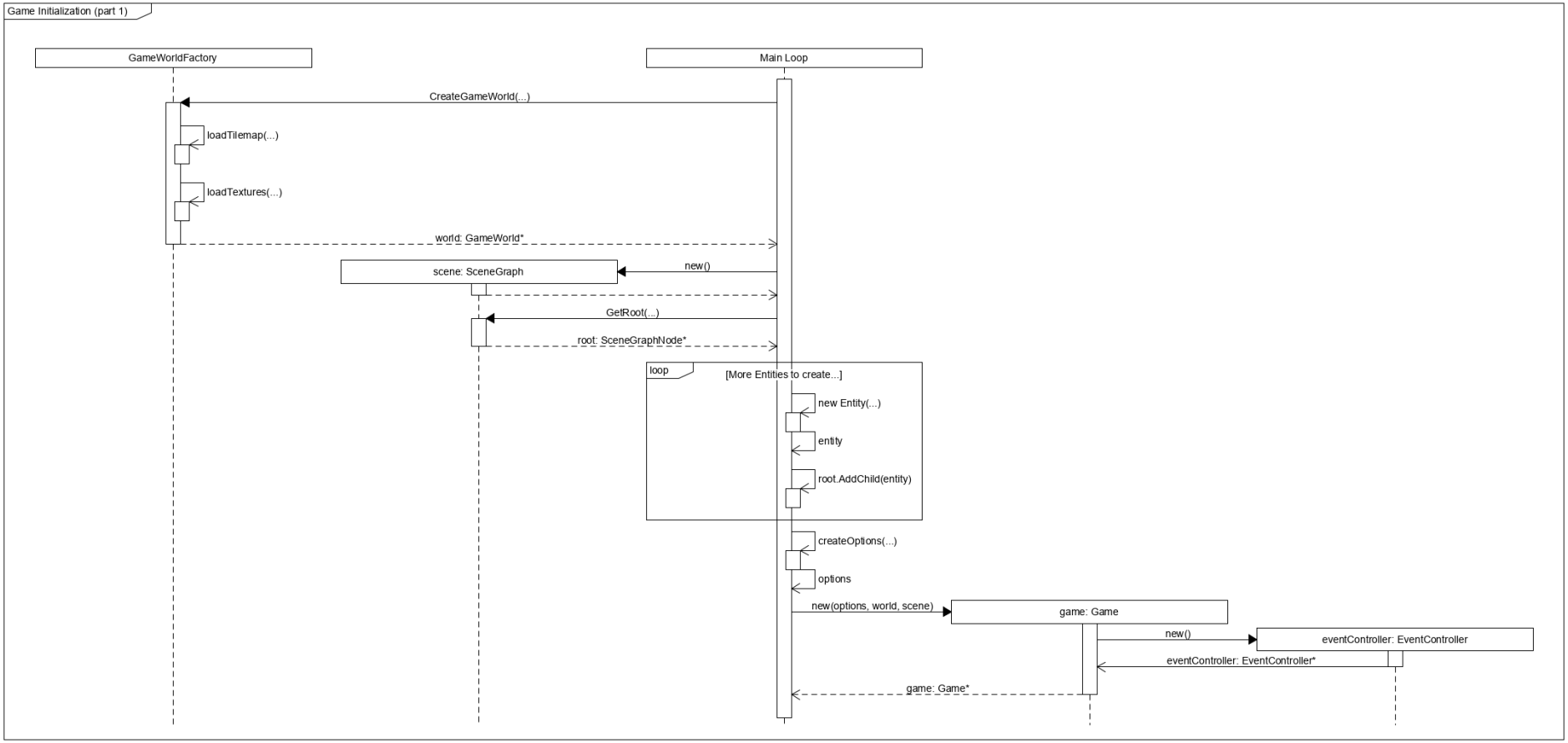

Abbildung 10 UML Sequenzdiagramm: Initialisierung Phase 1

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

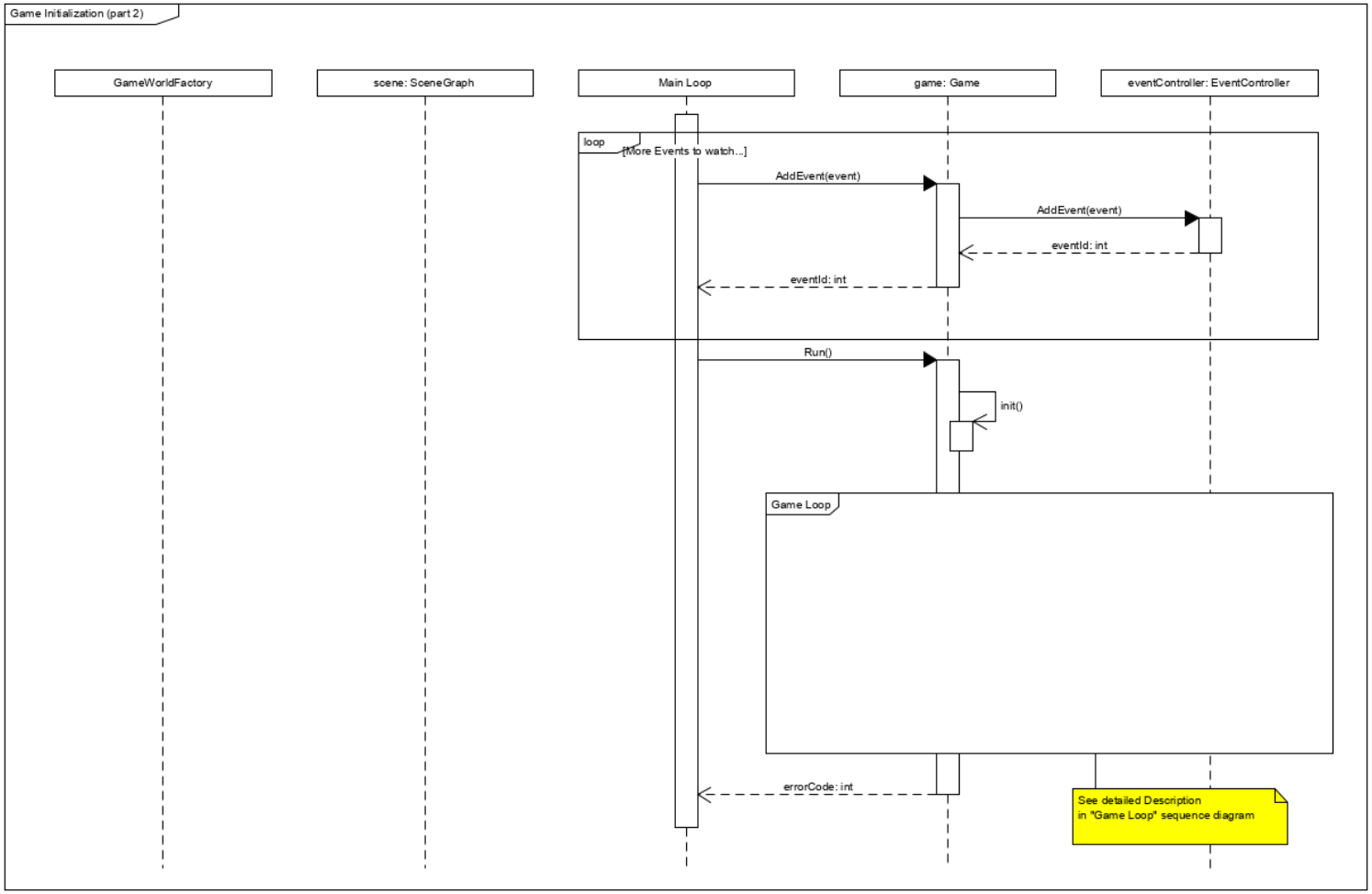

Abbildung 11 UML Sequenzdiagramm: Initialisierung Phase 2

4.4.6 UML Sequenzdiagramm: Game Loop

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

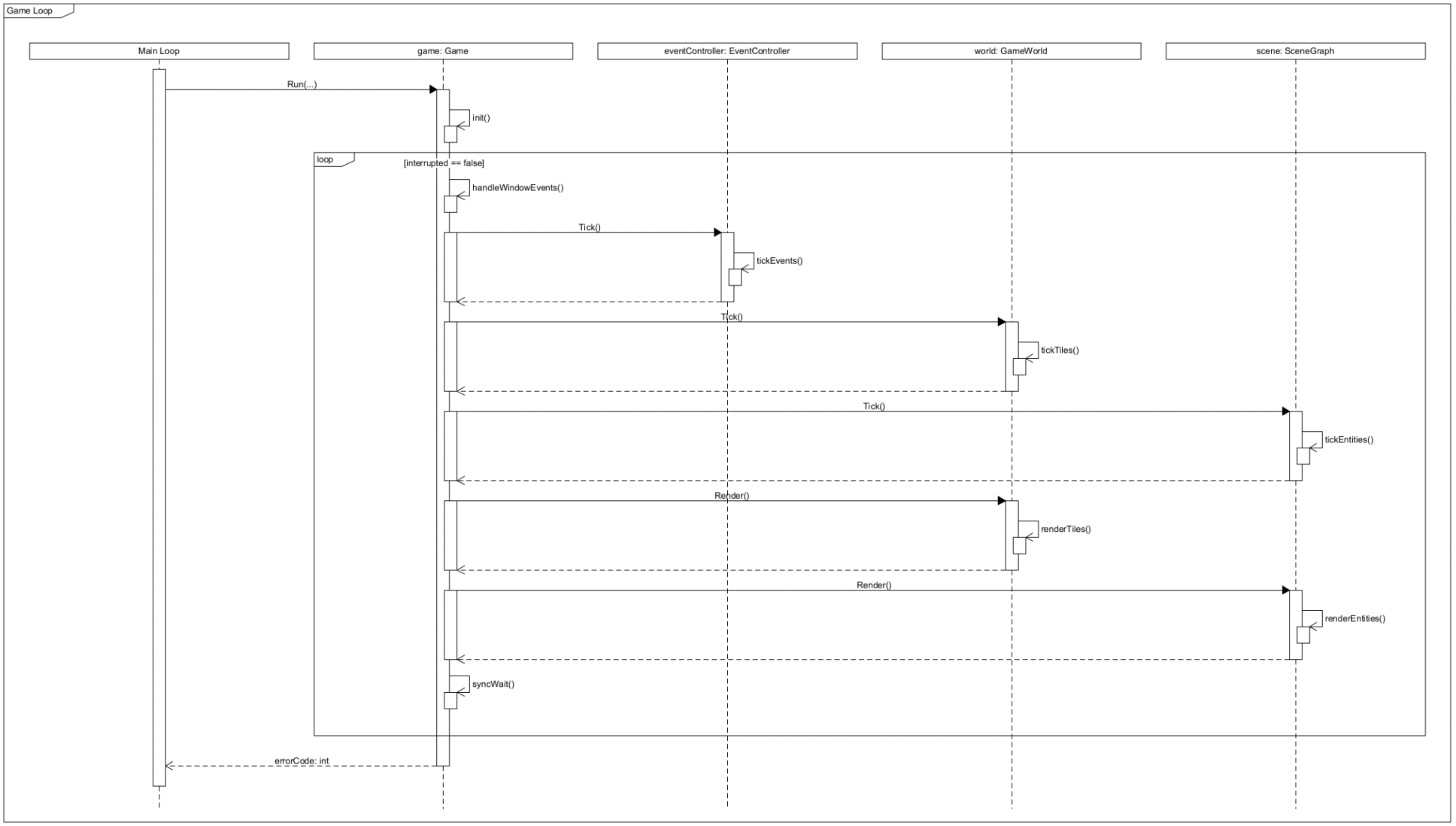

Abbildung 12 UML Sequenzdiagramm: Game Loop

4.5 Testing

4.5.1 Konzept

Zum Testen der 2DGameSDK wurden im Rahmen des Projekts mehrere unterschiedliche Prototypen erstellt, welche die einzelnen Komponenten der 2DGameSDK verwenden und illustrieren. Die Prototypen demonstrieren auch die Resultate des Variantenstudiums aus dem Pflichtenheft, [5] bei dem es darum geht die unterschiedlichen Methoden mit denen Animationen dargestellt werden können mit einander zu vergleichen.

4.5.2 Prototyp 1: Scene Graph

Der erste Prototyp besteht aus einer Graslandschaft aus Kacheln (Tiles) mit einer Grösse von 16x16 Pixel. Es wurden nur zwei unterschiedliche Kacheln für die Darstellung der Spielwelt verwendet. Der Spieler hat die Kontrolle über einen Hubschrauber. Die Tasten «Links» und «Rechts» rotieren den Hubschrauber um die eigene Achse, die Tasten «Oben» und «Unten» bewegen den Hubschrauber nach vorne oder nach hinten entlang der aktuellen Ausrichtung. Auf dem Hubschrauber ist ein Rotor befestigt, dieser ist auf dem Scene Graph ein hierarchisches Kind des Helikopters und dreht sich um die eigene Achse:

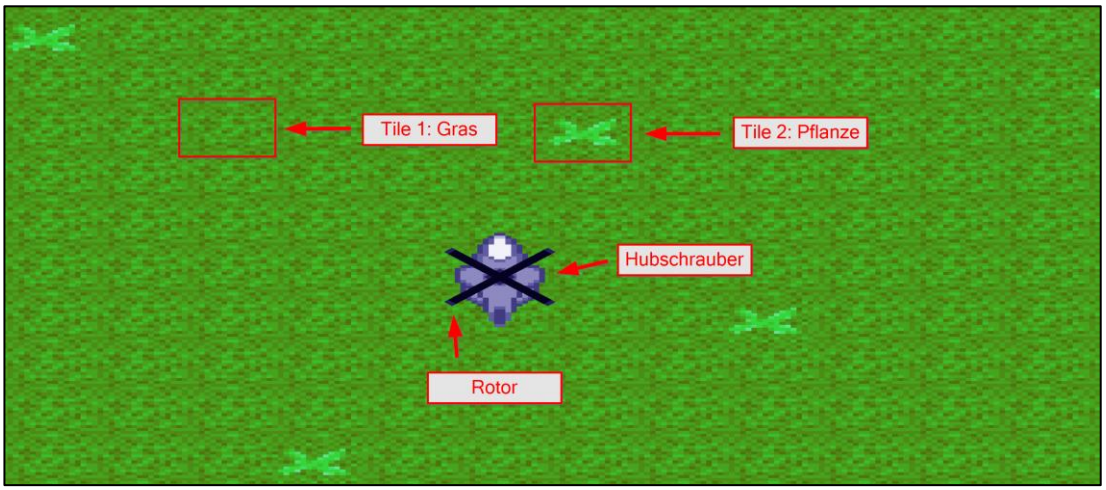

Abbildung 13 Demonstration Prototyp 1

Der Prototyp zeigt sehr gut wie der Scene Graph funktioniert, dadurch dass der Rotor dem Hubschrauber hierarchisch untergeordnet ist, bewegt er sich mit dem Hubschrauber mit. Durch die Rotation des Rotors um die eigene Achse, kann demonstriert werden, wie anhand des Scene Graphs Animationen dargestellt werden können, wenn auch mit gewissen Einschränkungen.

4.5.3 Prototyp 2: Animationen

Der zweite Protoyp enthält wieder eine Landschaft aus Kacheln (Tiles) mit einer Grösse von 16x16 Pixel. Diese Landschaft ist aber deutlich komplexer und besteht aus 32 unterschiedlichen Kacheln:

Abbildung 14 Demonstration Prototyp 2

Der Prototyp demonstriert eine zweite Methode, wie auf Spielobjekten Animationen dargestellt werden können. Der Spielercharakter (eine Erweiterung der Klasse «AnimatedTransformableEntity») hat 5 unterschiedliche Texturen gespeichert und kann die aktive Textur nach Bedarf ändern. Nur die aktive Textur wird beim nächsten Zyklus gerendert, dadurch kann mittels einfachen Ergänzungen der Spiellogik eine Laufanimation erzeugt werden:

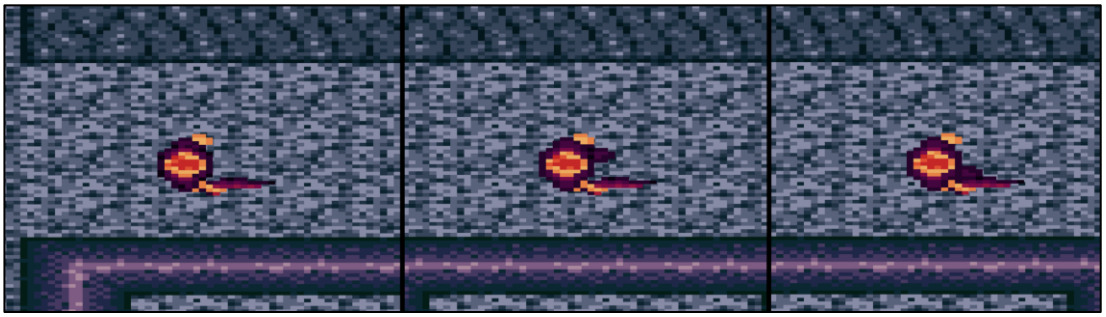

Abbildung 15 Prototyp 2: Laufanimation

4.5.4 Prototyp 3: OpenGL Test

Der letzte Prototyp ist ein experimenteller Prototyp, der im Rahmen einer Machbarkeitsstudie erstellt wurde. Er beinhaltet daher keine Spielwelt. Das Ziel des Prototyps war es, durch Integration von OpenGL eine weitere Methode zur Darstellung von Animationen zu erforschen: Animationen durch Transformation von 3D Objekten. Ähnlich wie beim Beispiel mit dem Helikopter in Prototyp 1, könnte man ein Objekt im 3-dimensionalen Raum rotieren lassen und so ein bewegendes Bild erzeugen. Im vorliegenden Prototyp konnte gezeigt werden, dass dies grundsätzlich möglich ist. Der Prototyp beinhaltet ein rotierendes mit OpenGL gezeichnetes 3D Volumen und ein mit SFML gezeichnetes 2D Sprite:

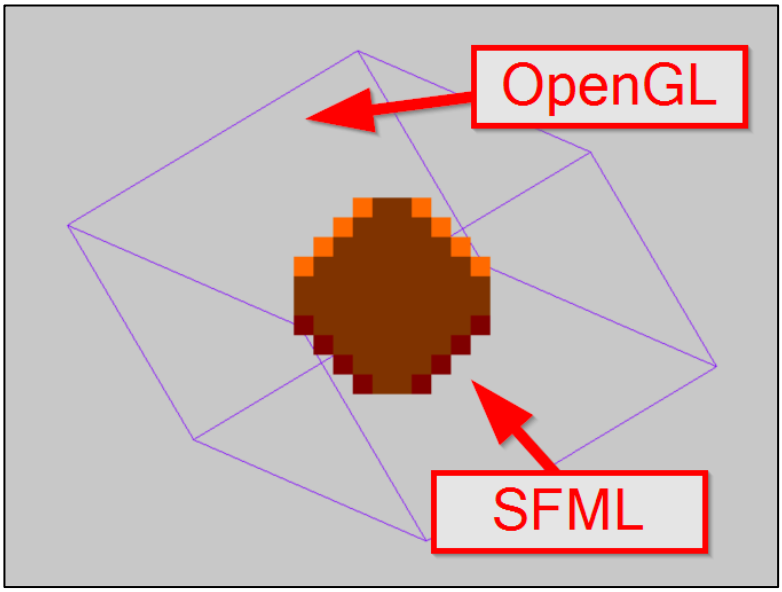

Abbildung 16 Demonstration Prototyp 3

Die Resultate dieses Prototyps sind im Github Repository in einem separaten Branch mit dem Namen «opengl_poc» gespeichert. Der Prototyp zeigt, dass es zwar möglich ist OpenGL und SFML mit einander zu verbinden, jedoch auch dass es bestimmte Probleme gibt, die noch gelöst werden müssen. Eines davon ist zum Beispiel die richtige Umwandlung der OpenGL Koordinaten in SFML Koordinaten bzw. in Koordinaten auf dem Scene Graph der 2DGameSDK.

5 Retrospektive

5.1 Zeitplan

Zeitplan gemäss Pflichtenheft (SOLL):

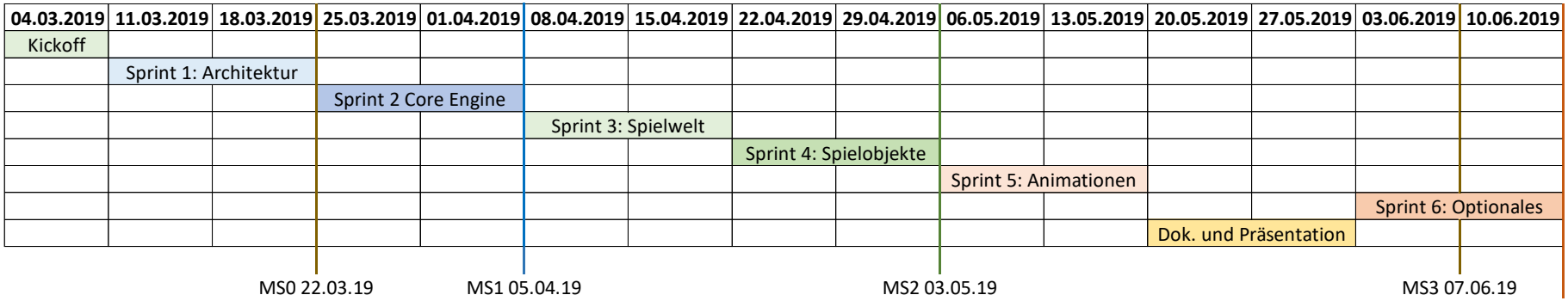

Abbildung 17 Zeitplan gemäss Pflichtenheft

Zeitplan tatsächlich umgesetzt (IST):

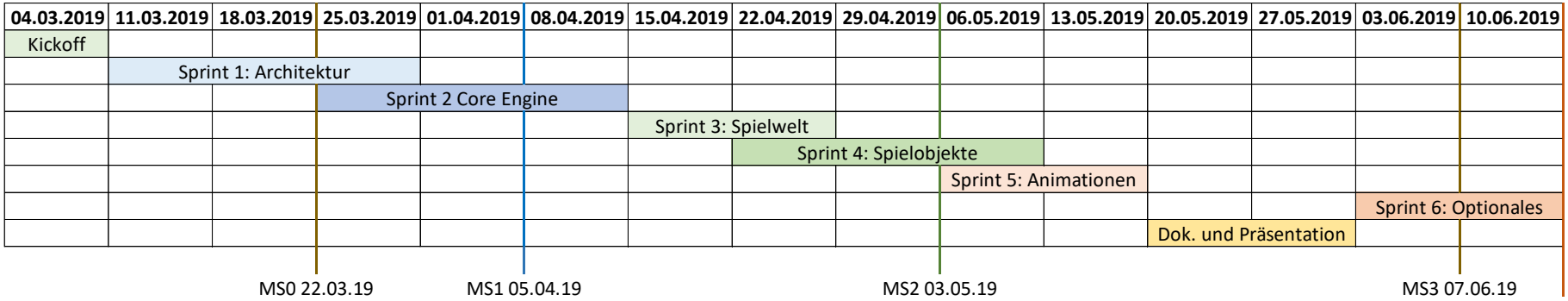

Abbildung 18 Zeitplan tatsächlich umgesetzt

Der Zeitplan konnte aufgrund von nicht vorhergesehenen Schwierigkeiten nicht ganz eingehalten werden: Zuerst gab es Probleme mit der Konfiguration eines gut funktionierenden Build Scripts, (siehe auch Kapitel 5.2) und zeitweise wurde gleichzeitig an der Core Engine und an der Architektur bzw. dem Projekt Setup gearbeitet. Das Event Controlling System das auch Teil der Core Engine ist, wurde unterschätzt, sodass sich die Implementation der Core Engine auch im eine Woche verzögert hat. Aufgrund dieser Verzögerung konnte erst etwas später mit der Implementation der Spielobjekte begonnen werden. Auch die Implementation der Spielobjekte war mit ungeahnten Komplikationen verbunden, denn es musste zuerst eine gute Scene Graph Implementation gefunden werden. Leider musste bei den Spielobjekten vorläufig auf die Implementation der Kollisionsdetektion verzichtet werden. Letztendlich wurden aber dafür die Animationen relativ schnell umgesetzt und so konnte die Zeit wieder aufgeholt werden. Im letzten Sprint («Optionales») wurde letztendlich die Doxygen Dokumentation erstellt und das POC zur OpenGL Integration.

5.2 Anforderungen

5.2.1 Funktionale Anforderungen

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

Game Controller

Tabelle 1 Funktionale Anforderungen: Game Controller Subsystem

ID	Beschreibung	Erfüllt?
FG1	Der Spieleentwickler möchte durch Festlegung der Ausgabeparameter (Fenstergrösse, Framerate, Skalierung) eine **Spielinstanz erstellen** können, sodass nach Start der Spielinstanz der **Game Loop** (Tick-Render Zyklus) **aktiviert** wird.	Ja
FG2	Der Spieleentwickler möchte unterschiedliche **Arten von Spielobjekten** festlegen können, sodass einzelne Instanzen dieser Spielobjekte die **Eigenschaften** und das **Verhalten** der eigenen Art **erben**.	Ja

Input / Event Handler

Tabelle 2 Funktionale Anforderungen: Input / Event Handler Subsystem

ID	Beschreibung	Erfüllt?
FI1	Der Spieleentwickler möchte die **Benutzereingaben** mit möglichen **Aktionen im Spiel verknüpfen** können, sodass anhand der Benutzereingaben eine Spiel-/Spielersteuerung realisiert werden kann.	Ja
FI2	Der Spieleentwickler möchte in der Spielinstanz **Events und Callbacks registrieren** können, sodass das **Verhalten** des Spiels gezielt beeinflusst werden kann.	Ja

World Loader

Tabelle 3 Funktionale Anforderungen: World Loader Subsystem

ID	Beschreibung	Erfüllt?
FW1	Der Spieleentwickler möchte eine **2D Spielwelt** als **Anordnung von Texturflächen** (Tiles) lesen können, sodass diese Spielwelt im Spiel dargestellt werden kann.	Ja
FW2	Der Spieleentwickler möchte die Möglichkeit haben für Texturflächen der Spielewelt (Tiles) **Materialeigenschaften** zu definieren, sodass diese Eigenschaften das Verhalten mit anderen Spielobjekten festlegen. Dies umfasst bspw. ob das Material **fest** oder **durchlässig** ist oder den **Z-Index** beim Rendering.	Nein

Game Object Loader

Tabelle 4 Funktionale Anforderungen: Game Object Loader Subsystem

ID	Beschreibung	Erfüllt?
FO1	Der Spieleentwickler möchte unterschiedliche **Arten von Spielobjekten** festlegen können, sodass einzelne Instanzen dieser Spielobjekte die **Eigenschaften** und das **Verhalten** der eigenen Art **erben**.	Ja
FO2	Der Spieleentwickler möchte **Animationen als Abfolge von Bildern** definieren können, sodass Spielobjekte mit dieser Methode animiert werden können.	Ja
FO3	(Variante) Der Spieleentwickler möchte **Animationen als Bewegung eines 3D Objekts** auf einem 2D Sichtfeld definieren können, sodass Spielobjekte mit dieser Methode animiert werden können.	Teilweise (POC erstellt)

State / Object Manager

Tabelle 5 Funktionale Anforderungen: State / Object Manager Subsystem

ID	Beschreibung	Erfüllt?
FS1	Der Spieleentwickler möchte die Möglichkeit haben einzelne **Objekte** im Spiel **effizient ansprechen** zu können sodass komplexe Interaktionen effizient innerhalb eines Frame Zyklus' realisiert werden können. Es bietet sich an hier auf einen **Scene Graph** oder auf eine **ähnliche Struktur** zurückzugreifen.	Ja

Zusatzfunktionen

Tabelle 6 Funktionale Anforderungen: Zusatzfunktionen

ID	Beschreibung	Erfüllt?
FA1	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.	Nein
FA2	Der Spieleentwickler möchte ein geeignetes Werkzeug haben, um **Künstliche Intelligenz auf Basis einfacher Regeln** zu realisieren, sodass ich das Verhalten von nicht-Spieler Charakter effizient definieren kann.	Nein

5.2.2 Technische Anforderungen

Im folgenden Teil sind die technischen Anforderungen gemäss Pflichtenheft [5] beschrieben, sowie die Information, ob diese Anforderungen umgesetzt werden konnten oder nicht:

Tabelle 7 Technische Anforderungen

ID	Beschreibung	Erfüllt?
T1	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.	Teilweise (zurück-gestellt)
T2	Die **Prozesse** sollen im sinnvollen Masse **parallel** ablaufen, sodass moderne Multi-Core Prozessoren möglichst gut ausgelastet werden können.	Ja
T3	Die **Frame Zyklen** müssen **präzise getaktet** sein, sodass ein flüssiges Spielerlebnis realisiert werden kann.	Ja
T4	Bei zu hoher **Zeitüberschreitung** soll ein **Frame übersprungen** werden sodass die Darstellung mit der Spiellogik wieder synchron ist.	Nein

5.2.3 Anforderungen rendering Qualität

Im folgenden Teil sind die Qualitätsanforderungen gemäss Pflichtenheft [5] beschrieben, sowie die Information, ob diese Anforderungen umgesetzt werden konnten oder nicht:

Tabelle 8 Anforderungen rendering Qualität

ID	Beschreibung	Erfüllt?
Q1	Das Rendering soll eine **Double- oder Triple Buffering** Strategie verfolgen sodass die Darstellung nahtlos und flüssig erscheint.	Ja
Q2	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).	Nein (zurück-gestellt)
Q3	Die Framerate des Spiels soll **stabil** bei 50Hz sein, wenn **1'000 kollisionsfähige Spielobjekte gleichzeitig im Spiel** sind (Wunschkriterium **10'000** Spielobjekte).	Nein (zurück-gestellt)

5.3 Herausforderungen

Die erste grosse Herausforderung, die mit dem Projekt einher ging, war die Erstellung eines funktionsfähigen Build Scripts, das zuerst das SDK Projekt erstellt und dieses dann mit dem Prototyp Projekt verbindet und zuletzt das Prototyp Projekt erstellt. Eine frühe Projektkonfiguration basierte allein auf einem Befehl für den GCC Compiler, es wurde aber schnell klar, dass die Anforderungen des Projekts zu komplex sind für eine solche Konfiguration. In einem zweiten Schritt wurde deshalb das Build Tool CMake eingesetzt, dieses bietet deutlich mehr Flexibilität durch die eigene Script Sprache und ermöglicht die Verwendung von verschiedenen Entwicklungsumgebungen wie zum Beispiel Codeblocks, Visual Studio oder VS Code. Die Erstellung eines funktionsfähigen CMake Scripts war letztendlich sehr zeitintensiv.

Eine weitere Herausforderung war der richtige Umgang mit C++ Templates, diese werden an mehreren Stellen des Projektes verwendet (Events, Asset Management). Jemand ohne allzu viel C++ Erfahrung wird irgendwann vor einem Linker Problem stehen, weil die Implementationen der Templates die aus der Library exportiert werden nicht gefunden werden kann. Die Lösung dieses Problem war es, die gesamte Implementation der Templates in der Header Datei zu definieren. Durch dieses Problem wurde die Implementation der Core Engine mit dem Event Management System leicht verzögert.

Die Implementation des Scene Graphs war auch mit ungeahnten Konsequenzen verbunden, denn es musste zuerst eine Lösung gefunden werden, die gut mit der Funktionalität des Frameworks SFML verbunden werden kann. Letztendlich konnte aber eine gute Lösung auf Basis der SFML RenderStates gefunden werden.

Die letzte Herausforderung stellte die Machbarkeitsstudie der OpenGL Integration dar. Es war zuerst schwierig die richtigen OpenGL Libraries zu finden damit diese mit dem Framework SFML zusammenarbeiten können. Die Lösung war zuletzt relativ einfach, denn SFML braucht weder Glut, noch GLEW um mit OpenGL zu arbeiten, es fehlte nur eine Referenz zu den OpenGL Libraries im CMake build Script.

5.4 Ausblick

Im Rahmen dieses Projekts konnte im Ansatz bereits eine stabile Game Engine mit einigen nützlichen Features erstellt werden. Damit aber die Game Engine komplett ist, müssen noch weitere Komponenten und Features dazu implementiert werden:

- **Kollisionsdetektion:** Spielobjekte müssen erkennen, wenn sie mit einander oder mit der Spielwelt kollidieren.
- **In-Game Overlay Display:** Eine Anzeige (Head Up Display) soll den Spieler über den Zustand des Spiels informieren.
- **OpenGL Integration:** Die Resultate Machbarkeitsstudie mit 3D Objekten sollen für die Realisierung von Animationen verwendet werden.
- **Physik Engine:** Idealerweise soll eine Physik Engine implementiert werden.
- **Behaviours und Künstliche Intelligenz:** Für die Umsetzung von nicht-Spieler Charakter sollen von der Engine typische Verhaltensarten der Spieltheorie (z.B. Steering oder Flock Behaviour) bereitgestellt werden.

Darüber hinaus sollen in einem späteren Projekt mehr Prototypen erstellt werden, diese sollen zeigen, dass mit der Engine unterschiedliche Genres von Spielen realisiert werden können. Zuletzt soll im Rahmen eines späteren Projekts eine Betriebsanleitung zur Verwendung der Game Engine erstellt werden.

6 Schlussfolgerungen/Fazit

Viele der Ziele, die an dieses Projekt gestellt wurden, konnten erreicht werden. Einige der Ziele mussten aber zurückgestellt werden, weil sie den zeitliche Rahmen des Projektes überschreiten würden. Letztendlich konnte aber eine funktionsfähige Game Engine erstellt werden, sowie einige Prototypen, welche die Funktionalität der Engine demonstrieren. Das Ziel ist somit weitgehend erreicht.

7 Referenzen

[1] L. Gomila, «SFML Home Page,» [Online]. Available: https://www.sfml-dev.org/. [Zugriff am 15 März 2019].

[2] Khronos Group Inc., «OpenGL Home Page,» [Online]. Available: https://www.opengl.org/. [Zugriff am 20 März 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] A. Markóczy, Pflichtenheft 2DGameSDK, Bern: Berner Fachhochschule, 2019.

[6] Free Software Foundation, Inc., «GCC, the GNU Compiler Collection,» 22 Februar 2019. [Online]. Available: https://gcc.gnu.org/. [Zugriff am 20 März 2019].

[7] N. Lohmann, «JSON for Modern C++,» 24 Mai 2019. [Online]. Available: https://nlohmann.github.io/json/. [Zugriff am 2 Juni 2019].

[8] Unity Technologies, «Unity Home Page,» [Online]. Available: https://unity.com/. [Zugriff am 20 März 2019].

8 Glossar

Begriff	Erläuterung
JSON	Die JavaScript Object Notation, kurz JSON, ist ein kompaktes Datenformat in einer einfach lesbaren Textform zum Zweck des Datenaustauschs zwischen Anwendungen.
CMake	CMake ist eine plattformübergreifende Gruppe von Open-Source Tools zum Erstellen, Testen und Bereitstellen von Software.
GCC (GNU Compiler Collection)	Die GNU Compiler Collection ist eine Sammlung von Open Source Compiler für Sprachen wie C, C++, Go oder Fortran
Library (Software-bibliothek)	Eine Softwarebibliothek ist eine Sammlung von Ressourcen, die für Computerprogramme bereitgestellt werden. Dazu gehören unter anderem Klassen, Typdefinitionen und Konfigurations-daten.
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].
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.

9 Abbildungsverzeichnis

10 Tabellenverzeichnis