

Projektarbeit "Scramble"

Nico Kimmel, Marlon Zarnke

Sommersemester 2022

Inhaltsverzeichnis

1	Über Scramble	3
2	Planung	4
2.1	Entwicklungsumgebung	4
2.2	Anforderungen	4
2.3	Funktionalitäten	5
3	Implementierung	7
3.1	Physics	7
3.2	Leveldefinitionen	8
3.2.1	Levelmanager	8
3.2.2	Levelklasse	9
3.3	Controller	10
3.3.1	Initialisierung	10
3.3.2	Gameloop	10
3.4	View	11
3.5	Texturen	12
3.6	Modellklassen	14
3.6.1	Entity	14
3.6.2	Player	14
3.6.3	Explosion	14
3.7	Events	14
3.7.1	Fenstereingaben	14
3.7.2	EventManager	14
4	Test	16

1 Über Scramble

Scramble ist ein von Konami entwickeltes Arcade Spiel, welches 1981 erschien und zu den allerersten multi-Level-Shoot-'em-ups zählt. Es war so erfolgreich, dass es in den ersten zwei Monaten 20 Millionen Dollar einspielte.

Es ist ein horizontal-scrollender Shooter in dem der Spieler eine Rakete durch mehrere Level steuert, welche vom Aussehen und der Art der Gegner variieren. Zur Hilfe hat der Spieler Raketen, die er horizontal nach vorne schießt und dann langsam auf den Boden fallen, sowie eine Kanone, welche Schüsse horizontal nach vorne feuert.

Ein weiteres Hindernis ist der Treibstofftank, der kontinuierlich beim Spielen leerläuft und nachgefüllt werden muss. Dafür muss der Spieler Treibstoffbehälter, welche auf dem Boden stehen, abschießen um seinen Raketentank wieder ein wenig aufzufüllen.

Der Spieler hat drei Leben und verliert eines bei Kollision oder wenn der Tank leer ist und die Rakete abstürzt. Sind alle Leben aufgebraucht so ist das Spiel vorbei. Gewonnen kann das Spiel nur werden indem der Spieler die Basis am Ende des letzten Levels zerstört.

2 Planung

2.1 Entwicklungsumgebung

Entwickelt wurde mit Visual Studio Code und der C++ Erweiterung gemeinsam über ein GitHub Repository. Als Compiler wurde GCC in Kombination mit CMake verwendet. Die Kommentare/Dokumentation im Quellcode nutzen Doxygen.

2.2 Anforderungen

Das Spiel soll mit C++ und OpenGL implementiert werden. Außerdem soll es nach dem MVC-Paradigma aufgebaut sein, der Quellcode dokumentiert sein und die Modellklassen automatisiert getestet werden.

Jedes Teammitglied sollte ein Aspekt einer Game-Engine implementieren. Wir haben uns für eine Physics Engine sowie das Laden der Levels aus dem Dateisystem entschieden.

Außer GLFW, GLEW und der Standardbibliothek von C++ durften keine weiteren Bibliotheken genutzt werden. Eine Soundausgabe musste nicht implementiert werden.

2.3 Funktionalitäten

Nr	Name	Beschreibung	Relevanz
F-01	Spiel aufbauen	Nach Ausführen des Programmes wird das Spiel gestartet.	muss
F-02	Nach oben fliegen	Schiff des Spielers fliegt nach oben beim Betätigen der Pfeiltaste nach oben. Maximal bis zum oberen Rand des Fensters.	muss
F-03	Nach unten fliegen	Schiff fliegt nach unten beim Betätigen der Pfeiltaste nach unten. Maximal bis zum unteren Rand des Fensters.	muss
F-04	Nach links fliegen	Das Schiff fliegt nach links beim Betätigen der Pfeiltaste nach links. Maximal bis zum linken Rand des Fensters.	muss
F-05	Nach rechts fliegen	Das Schiff fliegt nach rechts beim Betätigen der Pfeiltaste nach rechts. Maximal bis zur Hälfte des Fensters.	muss
F-06	Rakete schießen	Der Spieler schießt eine Rakete horizontal ab welche dann langsam in Richtung Boden sinkt und Gebäude zerstört.	soll
F-07	Laser schießen	Der Spieler schießt einen Laser, welcher horizontal fliegt bis er den Bildschirmrand oder ein Objekt trifft.	soll
F-08	Spielerkollision	Falls der Spieler mit dem Level oder einem Objekt (außer Laser und Raketen, welche der Spieler selbst abschießt) kollidiert, so wird das Raumschiff des Spielers zerstört.	soll

F-09	Raketenkollision	Falls eine vom Spieler abgeschossene Rakete mit dem Level oder einer Entität im Spiel kollidiert, so explodiert sie.	soll
F-10	Laserkollision	Falls ein vom Spieler abgeschossener Laser mit dem Level, dem Rand des Fensters oder einer Entität im Spiel kollidiert, so wird das Projektil entfernt.	soll
F-11	Entitätenkollision	Wird eine Entität von einem Laserprojektil oder einer vom Spieler abgefeuerten Rakete getroffen, so explodiert sie und verschwindet	soll
F-12	Entität Rakete starten	Nähert sich der Spieler einer Raketenentität (nicht vom Spieler abgeschossene Raketen), so fängt diese an sich nach oben zu bewegen.	soll
F-13	Spieler Leben abziehen	Entsteht nachdem F-08 geschehen ist. Spieler verliert eins von seinen drei Leben.	kann
F-14	Reset des Spiels	Nach dem F-13 geschehen ist, Wird das Spiel resettet und startet erneut.	soll
F-15	Spiel beenden	Wenn der Spieler sein letztes Leben verliert, so beendet sich das Spiel automatisch.	kann
F-16	Treibstoff	Es gibt einen endlichen Treibstoff tank, welcher sich abhängig der von der Zeit leert. Ist er auf null so stürzt der Spieler ab.	kann
F-17	Tankstelle	Kollidieren Projektile des Spielers mit einer auf dem Boden stehenden Tankstelle, so füllt sich der Tank wieder ein wenig auf und die Tankstelle verschwindet.	kann

3 Implementierung

3.1 Physics

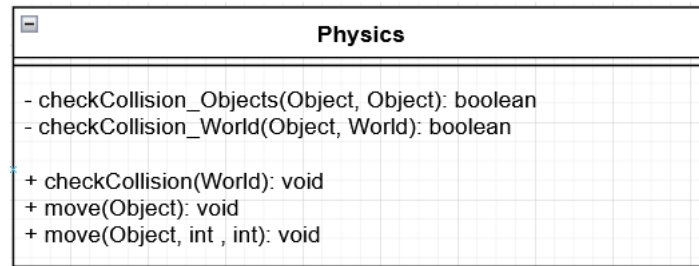


Abbildung 1: UML Diagramm der Klasse Physics

Die Physics Engine dient zur Simulation physikalischer Eigenschaften und besteht aus 4 Klassen sowie den dazugehörigen Headern.

Mit Hilfe der **Physics** Klasse werden Funktionalitäten wie Kollisionen zwischen zwei Objekten oder einem Objekt mit der Welt berechnet und überprüft. Hierfür iteriert eine verschachtelte Schleife über eine übergebene Liste aller Objekte und überprüft anhand der Eckpunkte zweier Objekte ob diese kollidieren oder anhand der Transparenzwerte des Levels an einem bestimmten Pixel (siehe Abbildung 6), ob ein Objekt mit der Welt kollidiert.

Des Weiteren stellt sie Funktionen bereit, welche Objekte bewegen in dem man der Funktion das Objekt das bewegt werden soll übergibt. Hierbei wird die Geschwindigkeit die das Objekt besitzt auf die alte Position gerechnet und erhält somit die neue Position.

Ein weiterer Bestandteil der **Physics Engine** ist die Klasse **Object**. In ihr werden alle physikalischen Parameter gespeichert welche benötigt werden, wie zum Beispiel Geschwindigkeit, Position oder die Ausdehnung des Objektes. Da die **Physics** Klasse die **Object** Klasse nutzt ist die Engine unabhängig vom Rest des Codes und der Programmierer der die Engine verwenden möchte muss von der Objekt Klasse erben um die Engine benutzen zu können, da die **Physics** Klasse Attribute vom Typ Objekt erwartet.

Genauso ist es auch mit der **World** Klasse. Von dieser muss das Level erben, da die Physics Klasse die Welt benötigt für Kollisionserkennung. Damit dies alles abgekapselt ist nutzt die Physics Klasse die World Klasse von welcher der Programmierer Gebrauch machen muss um die Engine nutzen zu können.

Da das Spiel in einer 2-dimensionalen Szene spielt benötigt natürlich die Objekte sowie die Engine 2-dimensionale Vektoren damit im Raum navigiert werden kann. Deshalb gibt es noch die Vektor2 Klasse von welcher die Physics Engine Gebrauch macht, da Werte wie Position oder Velocity jeweils aus einem x und y Parameter bestehen.

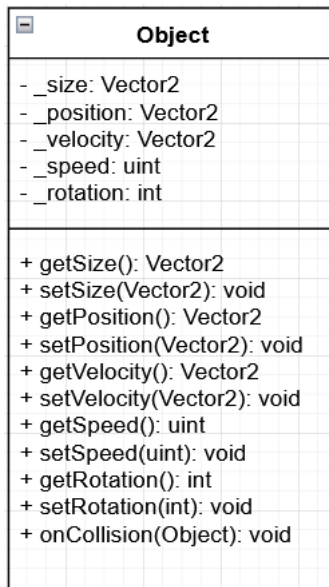


Abbildung 2: UML Diagramm der Klasse Object

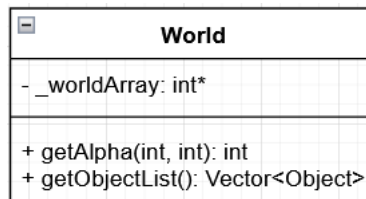


Abbildung 3: UML Diagramm der Klasse World

3.2 Leveldefinitionen

3.2.1 Levelmanager

Diese Klasse existiert um eine proprietäre Leveldefinition von der Festplatte einzulesen, welche extra für dieses Projekt erstellt wurde.

Die Leveldateien enden mit der `.sc` Endung und besitzen Informationen über das Aussehen des Levels in Form von Pixeldaten sowie Informationen über die Entitäten innerhalb des Levels. Außerdem enthält es Metadaten über das Level sowie die Kennung `SC` am Anfang. Ein genaueres Schaubild bietet Abbildung 6.

Der Levelmanager liest ein übergebenes Level ein und erstellt aus den oben genannten Informationen eine Liste mit Objekten aller Entitäten sowie eines Spielerobjektes.

Die Entitäten werden richtig positioniert und bekommen eine negative Velo-

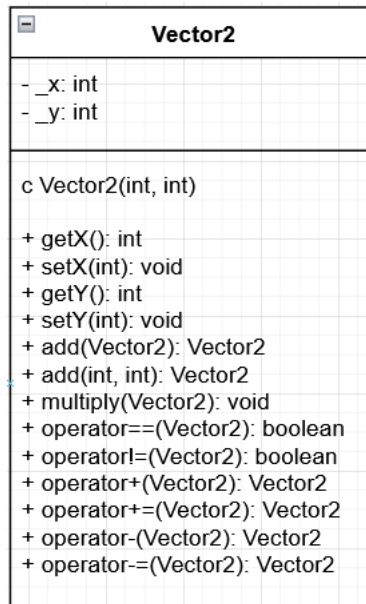


Abbildung 4: UML Diagramm der Klasse Vector2

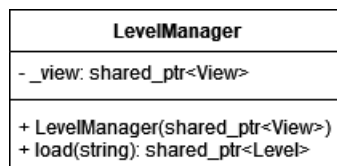


Abbildung 5: UML Diagramm der Klasse LevelManager

city, die der Scrollgeschwindigkeit des Levels entspricht, um sie mit dem Level mitzubewegen. Außerdem werden diese auch direkt von der **View** gebuffert, um die Texturen in den Arbeitsspeicher zu laden.

Ebenfalls erstellt wird eine Kollisionsmap, welches anschließend von der **Physics Engine** genutzt wird um Kollisionen mit der Welt festzustellen. Es ist ein (indirektes) zweidimensionales **Integer Array**, welches die Alphawert aller Pixel des Levels abbildet.

Am Ende der Funktion wird ein Objekt von **Level** erstellt und alle Listen und Objekte übergeben.

3.2.2 Levelklasse

Objekte der Klasse **Level** werden ausschließlich von **LevelManager** erstellt, nachdem dieser das Level von der Festplatte eingelesen hat.

00000000	53 43 BD 13 00 00 A9 01 00 00 02 00 00 00 40 00	SC @ .
00000010	00 00 01 00 00 00 1A 05 00 00 BE 01 00 00 01 00 J
00000020	00 00 88 05 00 00 BE 01 00 00 01 00 00 00 F4 05	. . ê J
00000030	00 00 BE 01 00 00 01 00 00 00 44 06 00 00 BE 01	. . J D
00000040	00 00 01 00 00 00 7A 06 00 00 BE 01 00 00 01 00 Z
00000050	00 00 CC 06 00 00 BE 01 00 00 03 00 00 00 28 07	. . f J
001B7220	E0 FF D3 00 E0 FF D3 00 E0 FF D3 00 E0 FF D3 00	α α α α α α α α
001B7230	E0 FF D3 00 E0 FF D3 00 E0 FF D3 00 E0 FF D3 00	α α α α α α α α
001B7240	E0 FF D3 00 E0 FF D3 00 E0 FF D3 00 E0 FF D3 00	α α α α α α α α
001B7250	E0 FF D3 00 E0 FF D3 00 E0 FF D3 00 E0 FF D3 00	α α α α α α α α
001B7260	E0 FF D3 00 E0 FF D3 00 E0 FF D3 00 E0 FF D3 00	α α α α α α α α

Levelinformationen	Entitäten	Pixeldaten
SC	Entitätentyp	Blau
Levelbreite	X-Koordinate	Grün
Levelhöhe	Y-Koordinate	Rot
Bewegungsgeschwindigkeit		Transparenz
Anzahl Entitäten		

Abbildung 6: Aufbau der Leveldatei

Sie hält Levelinformationen, erbt von den beiden Interfaces `Drawable` und `World` und implementiert deren Funktionen. Außerdem besitzt sie wie die Modellklassen auch eine `update()` Funktion, in der die Scrollgeschwindigkeit des Levels auf das Leveloffset addiert wird, um eine Linksbewegung zu simulieren.

Die Klasse stellt auch Funktionen bereit um die Entitätenliste zu beeinflussen. Sie kann neue Entitäten spawnen, despawnen (löschen) und explodieren lassen.

3.3 Controller

3.3.1 Initialisierung

Der Controller erstellt Objekte der Hilfsklassen `Physics`, `EventManager` und `LevelManager`. Er übergibt `Callbacks` für die Eingaben an `GLFW` und registriert einen Taktgeber für das Aktualisieren des Raketentanks sowie das Inkrementieren der Animationen durch die `View`. Zudem lädt er das Startlevel durch den `LevelManager`.

3.3.2 Gameloop

Die Hauptschleife besitzt einen inneren und einen äußeren Teil.

Der innere Teil wird genau **60-mal pro Sekunde** aufgerufen. Dies geschieht indem sich der Controller merkt, wann die Schleife zuletzt aufgerufen wurde, und ein ΔT errechnet. Dies hat den Vorteil, dass Bewegungen im Spiel auf jeder Hardware gleich schnell ablaufen und diese unabhängig von den FPS sind.

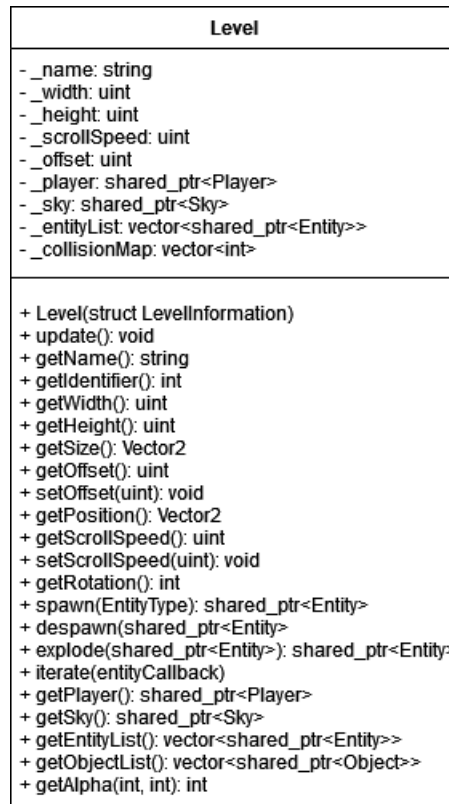


Abbildung 7: UML Diagramm der Klasse Level

In diesem Teil wird die `update()` Funktionen vom Level und den Entitäten aufgerufen, um deren Status und Position zu aktualisieren und bereits kollidierte Entitäten durch eine Explosion auszutauschen und anschließend zu löschen. An dieser Stelle führt die `Physics Engine` auch Berechnungen zu Kollisionen durch. Des Weiteren werden hier auch die Eingaben des Spielers verarbeitet.

Der äußere Teil der Schleife läuft so schnell wie für die Hardware möglich ist. Die `View` zeichnet hier alle Entitäten und das Level auf das Fenster.

3.4 View

Die `View` Klasse initialisiert bei Start `GLFW` und `GLEW` und erstellt damit ein Fenster, in welches anschließend gezeichnet wird.

Zu Beginn werden alle Bilder/Animationen mithilfe der `buffer()` Funktion in Texturobjekte geladen. Ein unabhängiger Taktgeber iteriert den Spritepointer mit `animate()` für alle Texturobjekte, um eine flüssige Animation abzuspielen, welche dann mit `render()` von OpenGL ins Fenster gezeichnet werden.

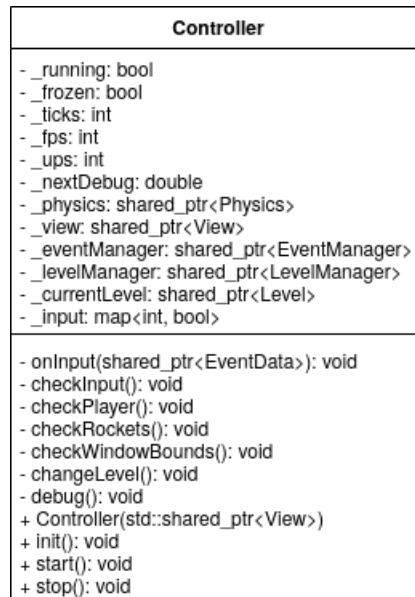


Abbildung 8: UML Diagramm der Klasse Controller

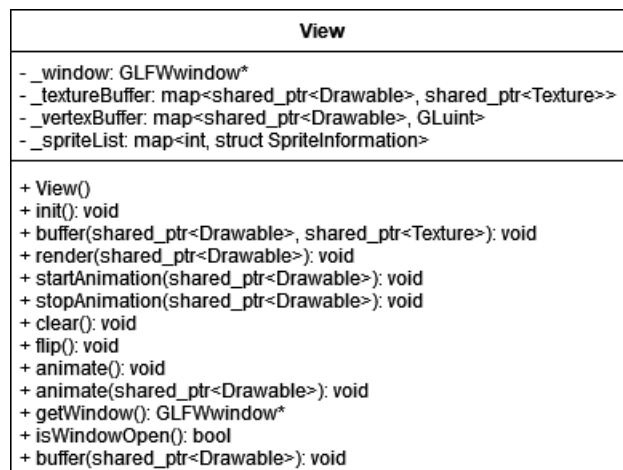


Abbildung 9: UML Diagramm der Klasse View

3.5 Texturen

Texturen werden durch die Klasse `Texture` bereitgestellt. Sie nimmt einen Pfad zu einer Datei auf der Festplatte und die Anzahl der darin befindlichen Sprites entgegen und buffert diese.

Voraussetzung der Texturen ist, dass diese im `Bitmap Format` vorliegen,

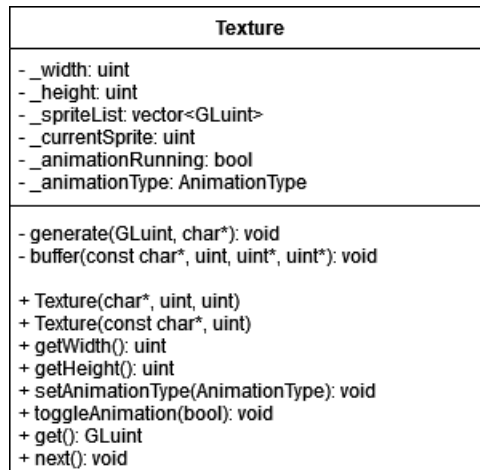


Abbildung 10: UML Diagramm der Klasse Texture

um ein einfaches byte-per-byte Einlesen (ohne Dekomprimierung) möglich zu machen. Alle Sprites einer Animation müssen wie in Abbildung 11 untereinander angeordnet sein.



Abbildung 11: Spritesheet des Spielers

Um die Textur einzulesen wird zuerst der Bitmap Header ausgelesen um Höhe, Breite und den Offset zu den Pixeldaten zwischenspeichern. Dann werden in einer Schleife die Abschnitte der Bitmap als einzelne Bilder ausgewertet, um als Endresultat eine Animation abspielen zu können.

Die Pixeldaten werden anschließend von OpenGL ausgewertet und einer ID zugewiesen. Diese IDs werden in einer Liste gespeichert – eine ID für jedes Bild der Animation.

Um eine Animation abzulaufen muss die `next()` Funktion jedes Texturobjekts aufgerufen werden. Dieses wird von `View` alle `100ms` (bzw für das Objekt des Himmels alle `800ms`) erledigt. So wird der Zeiger auf das aktuelle Bild je nach Animationstyp inkrementiert, falls die Animation gestartet ist.

3.6 Modellklassen

3.6.1 Entity

Die Klasse `Entity` ist die Oberklasse aller Modellklassen. Sie erbt von `Object` und `Drawable` und enthält alle Basisfunktionen und -attribute. Das `Object` Interface besitzt physikalische Attribute wie Größe und Geschwindigkeit, das `Drawable` Interface wird von der `View` zum Zeichnen genutzt. `Entity` besitzt zwei virtuelle (abstrakte) Funktionen, die von den Modellklassen implementiert werden: `update()`, welche die Position und eventuelle andere Attribute ändert und `onCollision()`, die entscheidet, mit welchen anderen Entitäten die Entität kollidieren darf und wie sie sich dann weiter verhält.

3.6.2 Player

Die `Player` Klasse verwaltet das Raumschiff des Spielers. Implementiert werden `update()` und `onCollision()` von `Entity`. Der Spieler kollidiert mit allen anderen Entitäten außer seiner eigenen Raketen und Lasern. Außerdem besitzt die Spielerklasse Funktionen um den Treibstofftank und die Spielerleben zu verwalten sowie die Rakete zu steuern.

3.6.3 Explosion

Explosionen werden ausschließlich vom Controller hinzugefügt, wenn eine Entität ihr Attribut `crashed` gesetzt hat. Jede Explosion hat auch einen `ExplosionType`, der zeigt, ob es sich um eine Spielerexplosion handelt oder nicht. Sie werden vom `Controller` automatisch nach `800ms` automatisch wieder gelöscht.

3.7 Events

3.7.1 Fenstereingaben

Eingaben des Spielers werden durch einen `Callback` von `GLFW` weitergeleitet, welcher mit `glfwSetKeyCallback()` gesetzt wird. Der Callback gibt einen Pointer auf das entsprechende `GLFWwindow` (Fenster), den `Key` (Taste), den `Scancode` (plattformspezifische Taste), die `Action` (gedrückt, losgelassen) und die `Mods` (ALT, STRG, SHIFT) zurück.

Die gedrückten Tasten werden in einer Hilfsliste zwischengespeichert und dann vom Controller in der Gameloop abgefragt.

Mit den Pfeiltasten steuert der Spieler sein Raumschiff, mit STRG schießt er Raketen und mit ALT Laser.

3.7.2 EventManager

Der `EventManager` verwaltet Ereignisse und ruft die passenden `Callback` Funktionen auf, sollten diese eintreten. Mit `registerEvent()` werden `Callbacks`

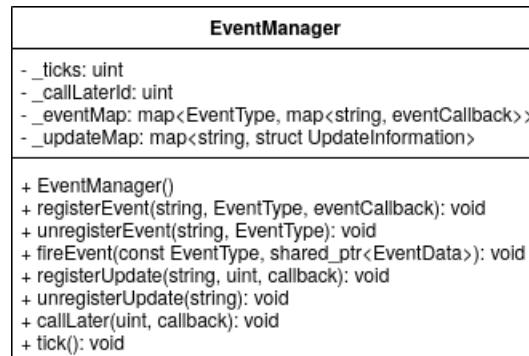
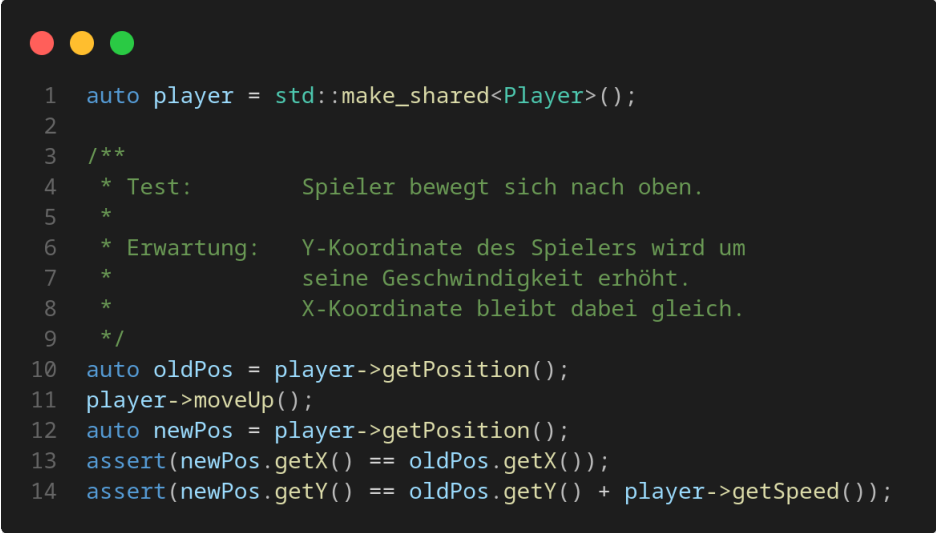


Abbildung 12: UML Diagramm der Klasse EventManager

registriert, die bei einem Aufruf von `fireEvent()` dann alle ausgeführt werden. Außerdem dient der `EventManager` mithilfe der `registerUpdate()` Funktion auch als Taktgeber. Der übergebene `Callback` wird dann zu einem übergebenen Intervall aufgerufen. Die Funktion `callLater()` nutzt dies um eine Funktion verspätet auszuführen.

4 Test

Die Modellklassen wurden automatisiert mit CTests getestet, wobei jede Klasse eine eigene Datei mit Tests besitzt. Die Tests nutzen die `assert()` Funktion und sind nach dem Schema in Abbildung 13 aufgebaut. Sie werden automatisch mitkompiliert und ausgeführt.

A screenshot of a code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in C++ and represents a unit test for a Player class. It includes a comment in German describing the test: 'Spieler bewegt sich nach oben.' (Player moves up). The test logic involves creating a shared Player object, calling moveUp(), and then using assert() to verify that the X-coordinate remains unchanged and the Y-coordinate increases by the player's speed.

```
1  auto player = std::make_shared<Player>();
2
3  /**
4   * Test:      Spieler bewegt sich nach oben.
5   *
6   * Erwartung: Y-Koordinate des Spielers wird um
7   *             seine Geschwindigkeit erhöht.
8   *             X-Koordinate bleibt dabei gleich.
9   */
10 auto oldPos = player->getPosition();
11 player->moveUp();
12 auto newPos = player->getPosition();
13 assert(newPos.getX() == oldPos.getX());
14 assert(newPos.getY() == oldPos.getY() + player->getSpeed());
```

Abbildung 13: Aufbau eines Tests der Player Klasse