Technische Universität Braunschweig

Institut für Betriebssysteme und Rechnerverbund

Labyrinth Games

Haunted Edition

Software-Entwicklungspraktikum (SEP)
Sommersemester 2016

Technischer Entwurf

Auftraggeber
Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund
Abteilung Algorithmik
Prof. Dr. Sándor P. Fekete
Mühlenpfordtstraße 23
38106 Braunschweig

Betreuer: Phillip Keldenich

Auftragnehmer:

Name	E-Mail-Adresse
Eva Vanessa Bolle	eva.bolle@tu-braunschweig.de
Florian Bahr	f.bahr@tu-braunschweig.de
Frauke Pommerehne	f.pommerehne@tu-braunschweig.de
Frederik Hinrichs	f.hinrichs@tu-braunschweig.de
Henning Pohl	henning.pohl@tu-braunschweig.de
Marvin Buhr	m.buhr@tu-braunschweig.de
Rafael Fornal	r.fornal@tu-braunschweig.de
Sandro Heuer	sandro.heuer@tu-braunschweig.de

Braunschweig, 29. Juni 2016

Bearbeiterübersicht

Kapitel	Autoren	Kommentare
1	Marvin Buhr Frederik Hinrichs Sandro Heuer Rafael Fornal Florian Bahr Eva V. Bolle	 Aktivitätsdiagramm (Abb. 1.1) Aktivitätsdiagramm (Abb. 1.1)
1.1	Frederik Hinrichs Sandro Heuer	
1.1.1	Eva V. Bolle	
1.1.2	Frederik Hinrichs Florian Bahr Rafael Fornal Eva V. Bolle	Aktivitätsdiagramm (Abb. 1.2)
1.1.3	Marvin Buhr Florian Bahr	
2	Florian Bahr Frauke Pommerehne Marvin Buhr	
2.1	Frauke Pommerehne Marvin Buhr Florian Bahr Frederik Hinrichs	 Sequenzdiagramm ⟨F010⟩
2.2	Frauke Pommerehne Marvin Buhr Florian Bahr Frederik Hinrichs	 Sequenzdiagramm ⟨F020⟩
2.3	Frauke Pommerehne Marvin Buhr Florian Bahr Frederik Hinrichs	 Sequenzdiagramm ⟨F030/040/050⟩
2.4	Frauke Pommerehne Marvin Buhr Florian Bahr Sandro Heuer	 Sequenzdiagramm ⟨F060/100/110⟩

Kapitel	**Autoren**	**Kommentare**
	Frederik Hinrichs Rafael Fornal Eva V. Bolle	
2.5	Frauke Pommerehne Marvin Buhr Florian Bahr	 Sequenzdiagramm ⟨F070⟩
2.6	Frauke Pommerehne Marvin Buhr Florian Bahr	 Sequenzdiagramm ⟨F080/090⟩
2.7	Marvin Buhr Sandro Heuer Florian Bahr	Sequenzdiagramm ⟨F120⟩ Sequenzdiagramm ⟨F120⟩ Sequenzdiagramm ⟨F120⟩
2.8	Frauke Pommerehne Sandro Heuer Marvin Buhr Florian Bahr Rafael Fornal	 Sequenzdiagramm ⟨F130⟩ Sequenzdiagramm ⟨F130⟩
2.9	Florian Bahr Marvin Buhr Frauke Pommerehne Sandro Heuer	Sequenzdiagramm ⟨F140⟩ Sequenzdiagramm ⟨F140⟩ Sequenzdiagramm ⟨F140⟩
3	Florian Bahr	
3.1	Florian Bahr Marvin Buhr Rafael Fornal Eva V. Bolle	 Komponentendiagramm (Abb. 3.1)
3.2	Marvin Buhr Eva V. Bolle	
3.3	Marvin Buhr Eva V. Bolle	
4	Marvin Buhr	
5	Henning Pohl Marvin Buhr Florian Bahr Frauke Pommerehne	
6	Florian Bahr	

Kapitel	**Autoren**	**Kommentare**
6.1	Florian Bahr Marvin Buhr Frederik Hinrichs Henning Pohl	Klassendiagramm ⟨D10⟩ Klassendiagramm ⟨D10⟩ Klassendiagramm ⟨D10⟩ Physisches Schema ⟨D30⟩
6.2	Florian Bahr Henning Pohl Rafael Fornal	Klassendiagramm ⟨D20⟩ Physisches Schema ⟨D30⟩
6.3	Florian Bahr Henning Pohl	Klassendiagramm ⟨D30⟩ Physisches Schema ⟨D30⟩
7	Frauke Pommerehne Frederik Hinrichs	
8	Sandro Heuer Frauke Pommerehne Frederik Hinrichs	
9	Rafael Fornal	
9.1	Florian Bahr Sandro Heuer	
9.2	Florian Bahr Sandro Heuer	
9.3	Marvin Buhr Sandro Heuer Frederik Hinrichs	
10	Florian Bahr Sandro Heuer Eva V. Bolle	

Inhaltsverzeichnis

1 Einleitung **8**
1.1 Projektdetails . . . 9
1.1.1 Spielregeln . . . 9
1.1.2 Menüführung . . . 10
1.1.3 Visibility . . . 11

2 Analyse der Produktfunktionen **12**
2.1 Analyse von Funktionalität ⟨F010⟩ – Grafik-Einstellungen . . . 12
2.2 Analyse von Funktionalität ⟨F020⟩ – Sound-Einstellungen . . . 13
2.3 Analyse von Funktionalitäten ⟨F030⟩, ⟨F040⟩, ⟨F050⟩ – Levelauswahl, Level laden und Darstellung des Spielgeschehens . . . 14
2.4 Analyse von Funktionalitäten ⟨F060⟩, ⟨F100⟩, ⟨F110⟩ – Spielfigur bewegen, Verlassen des begehbaren Bereichs (Level verlieren) und Betreten des Zielbereichs (Level gewinnen) . . . 15
2.5 Analyse von Funktionalität ⟨F070⟩ – Relaisstation aufheben . . . 17
2.6 Analyse von Funktionalitäten ⟨F080⟩, ⟨F090⟩ – Sichtbarkeitsbereich berechnen und Relaisstation setzen . . . 18
2.7 Analyse von Funktionalität ⟨F120⟩ – Spiel pausieren . . . 19
2.8 Analyse von Funktionalität ⟨F130⟩ – Highscore-System . . . 20
2.9 Analyse von Funktionalität ⟨F140⟩ – Spiel beenden . . . 21

3 Resultierende Softwarearchitektur **22**
3.1 Komponentenspezifikation . . . 22
3.2 Schnittstellenspezifikation . . . 25
3.3 Protokolle für die Benutzung der Komponenten . . . 27

4 Verteilungsentwurf **28**

5 Implementierungsentwurf **29**

6 Datenmodell **30**
6.1 Spielleveldaten ⟨D10⟩ . . . 30
6.1.1 Logisches Schema . . . 30
6.1.2 Physisches Schema . . . 33

6.2 Spielfortschrittsdaten ⟨D20⟩ 34
6.2.1 Logisches Schema 34
6.2.2 Physisches Schema 35
6.3 Einstellungsdaten ⟨D30⟩ 35
6.3.1 Logisches Schema 35
6.3.2 Logisches Schema 36

7 Konfiguration **37**

8 Änderungen gegenüber Fachentwurf **38**

9 Erfüllung der Kriterien **39**
9.1 Musskriterien 39
9.2 Sollkriterien 40
9.3 Kannkriterien 40

10 Glossar **41**

Abbildungsverzeichnis

1.1 Aktivitätsdiagramm des Spielablaufs 8
1.2 Aktivitätsdiagramm der Menüführung 10

2.1 Sequenzdiagramm zu Produktfunktion ⟨F010⟩ 12
2.2 Sequenzdiagramm zu Produktfunktion ⟨F020⟩ 13
2.3 Sequenzdiagramm zu Produktfunktionen ⟨F030⟩, ⟨F040⟩ und ⟨F050⟩ 14
2.4 Sequenzdiagramme zu Produktfunktionen ⟨F060⟩, ⟨F100⟩ und ⟨F110⟩ 15
2.5 Sequenzdiagramm zu Produktfunktion ⟨F070⟩ 17
2.6 Sequenzdiagramm zu Produktfunktionen ⟨F080⟩ und ⟨F090⟩ 18
2.7 Sequenzdiagramm zu Produktfunktion ⟨F120⟩ 19
2.8 Sequenzdiagramm zu Produktfunktion ⟨F130⟩ 20
2.9 Sequenzdiagramm von ⟨F140⟩ 21

3.1 Diagramm der Komponenten von Labyrinth Games 23

6.1 Logisches Schema der Spielleveldaten ⟨D10⟩ 30
6.2 Logisches Schema der Spielfortschrittsdaten ⟨D20⟩ 34
6.3 Logisches Schema der Einstellungsdaten ⟨D30⟩ 35

1 Einleitung

Das vorliegende Dokument bildet die technische Entwurfsdokumentation des Softwareprodukts Labyrinth Games: Haunted Edition.

Zunächst wird ein Überblick des generellen Spielablaufs auf Level-Ebene sowie der Menüführung durch Aktivitätsdiagramme gegeben. In den folgenden Abschnitten werden die Produktfunktionen, die Softwarearchitektur der Anwendung und die Verteilung ihrer Komponenten dokumentiert. Anschließend folgen ein detaillierter Implementierungsentwurf, sowie eine Modellierung der dauerhaft gespeicherten Daten mittels Klassendiagrammen der Anwendung. Dieses Dokument wird abgeschlossen durch eine Dokumentation der gegenüber dem Fachentwurf vorgenommenen Änderungen, sowie einer Überprüfung hinsichtlich Erfüllung der im Pflichtenheft für das Projekt Labyrinth Games: Haunted Edition aufgeführten Kriterien.

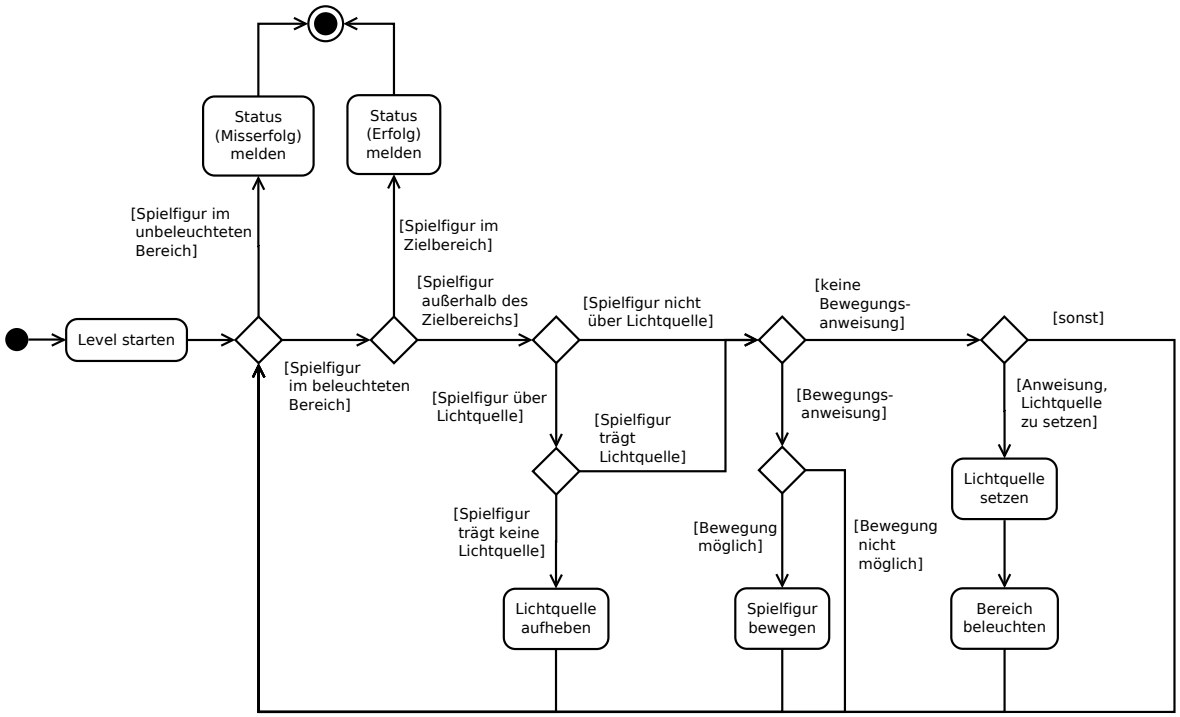

Abbildung 1.1: Aktivitätsdiagramm des Spielablaufs

Das bereits im Pflichtenheft eingeführte Aktivitätsdiagramm der Spiellogik in Abbildung 1.1 zeigt den Ablauf eines Levels aus Sicht des Spielers. Zunächst wird überprüft, ob sich der Spieler

in einem von einer Lichtquelle ausgeleuchteten Bereich befindet. Ist dies nicht der Fall, verliert der Spieler, ansonsten läuft das Spiel weiter. Wenn sich die Spielfigur in dem definierten Zielbereich befindet, gewinnt der Spieler das Level. Befindet sich die Spielfigur über einer Lichtquelle und trägt nicht bereits eine solche, wird diese automatisch aufgehoben. So wie der Spieler die Spielfigur mittels Bewegungsanweisungen durch das Level manövriert, hat er auch die Möglichkeit, die Spielfigur anzuweisen, eine getragene Lichtquelle zu platzieren. Sobald die Spielfigur die Lichtquelle ablegt, wird der Bereich um die Lichtquelle beleuchtet.

Diese grundsätzliche Spielidee wird im Folgenden im Hinblick auf die Ausgestaltung eines konkreten Spielkonzepts dahingehend präzisiert, dass die Spielfigur als IR-gesteuerter Roboter, Lichtquellen als Relaisstationen und beleuchtete Gebiete als von Relaisstationen abgedeckte Bereiche interpretiert werden.

1.1 Projektdetails

Folgend wird genauer auf die Besonderheiten des Projekts eingegangen.

1.1.1 Spielregeln

Dieser Abschnitt dient der genaueren Erläuterung der Spielregeln.

- Das Spiel wird in Echtzeit gespielt; der Spieler besitzt kein Zeitlimit um ein Level abzuschließen.
- Die Spielfigur kann frei über die Maus im Spielfeld bewegt werden, solange der Bereich nicht als unpassierbar festgelegt ist.
- Relaisstationen können an der aktuellen Position der Spielfigur platziert werden, solange der Bereich nicht als ungültig vorgegeben ist.
- Die Spielfigur kann zu jeder Zeit nur maximal eine Relaisstation im Inventar besitzen.
- Bereits gesetzte Relaisstationen können nicht wieder aufgenommen und neu platziert werden.
- Der Spieler verliert das Level, sobald sich die Spielfigur innerhalb eines Bereichs ohne IR-Verbindung befindet.
- Der Spieler gewinnt das Level, sobald sich die Spielfigur innerhalb des Zielbereichs befindet.

1.1.2 Menüführung

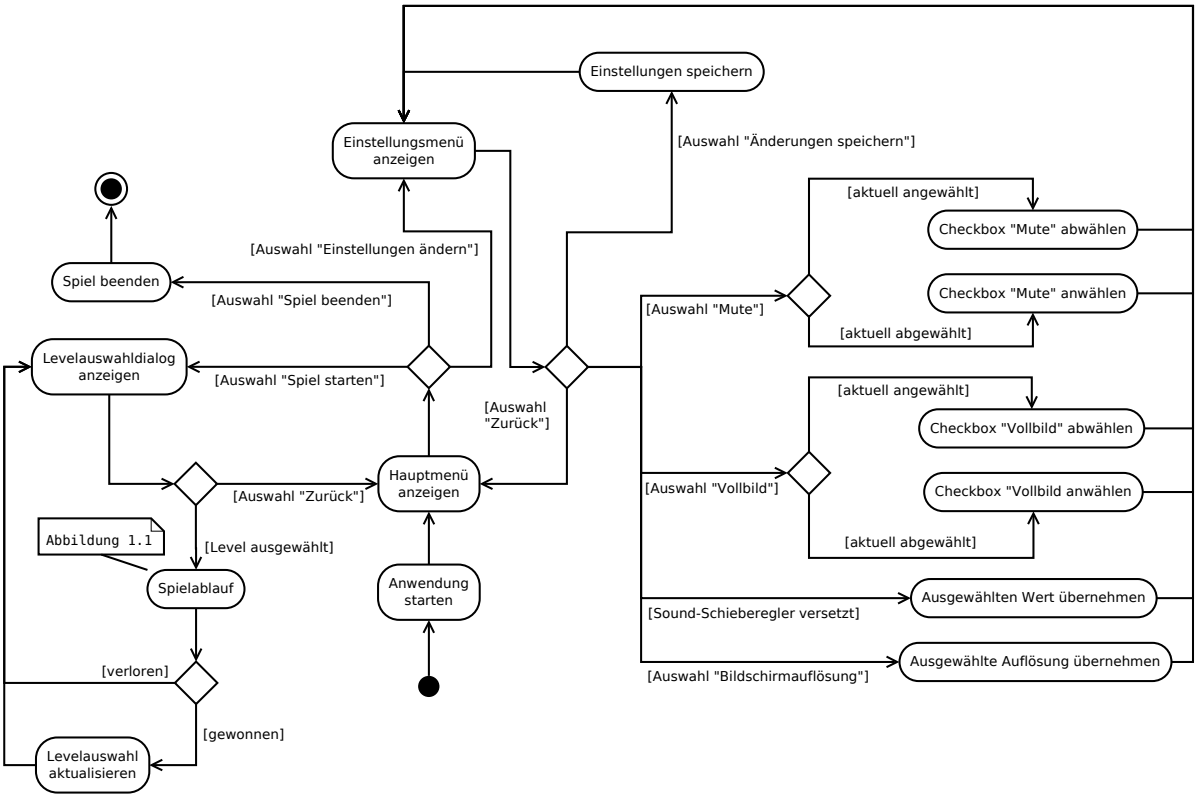

Abbildung 1.2: Aktivitätsdiagramm der Menüführung

Abbildung 1.2 erläutert die Menüführung innerhalb des Softwareprodukts. Vor Beginn des Spiels in einem Level können Produkteinstellungen geändert werden. Wählt der Spieler im Hauptmenü die Option „Einstellungen“, gelangt er in das Einstellungsmenü, in dem Grafik- und Soundoptionen angepasst werden können. Wurden die Einstellungen nach den Wünschen des Spielers festgelegt, kann aus dem Hauptmenü über Anwahl der Option „Spiel starten“ ein Level gestartet werden. Hierfür gelangt der Anwender zunächst in einen Levelauswahl-Dialog, der eine Auswahl unter den jeweils aktuell spielbaren Level ermöglicht. Nach Festlegung eines Levels wird das Spiel in dem gewählten Level aufgenommen. Der eigentliche Spielablauf wurde in Kapitel 1 detailliert. Vor Aufnahme des Spielgeschehens oder nach Beendigung eines Levels kann die Anwendung ferner aus dem Hauptmenü über die Option „Spiel beenden“ verlassen werden.

1.1.3 Visibility

Die Berechnung von Abdeckungsbereichen von Relaisstationen, d. h. der Bereiche eines Levels, die von IR-Signalen der Relaisstationen an ihren gegebenen Positionen abgedeckt sind, ist ein zentraler Bestandteil der internen Logik des Spiels.

In Labyrinth Games wird das Konzept der Abdeckungsbereiche mittels n-seitiger Polygone realisiert, die Bereiche eines Levels umschließen, welche von den Standorten aktivierter Relaisstationen aus „sichtbar“ sind. Die einzelnen Polygone werden dabei in ihrem Durchmesser durch die Sendeleistung der umschlossenen Relaisstationen begrenzt. Ihrer Konzeption folgend werden diese Konstruktionen im Weiteren auch als *Visibility-Polygone* bezeichnet.

Durch Überlappung von Level-Geometrie und Visibility-Polygonen ergeben sich die Bereiche des Spielfelds, in denen der Spieler seine Spielfigur bewegen kann, ohne das Spiel zu verlieren.

2 Analyse der Produktfunktionen

In diesem Kapitel wird das Verhalten der im Pflichtenheft dokumentierten Produktfunktionen ⟨F010⟩ bis ⟨F140⟩ unter Verwendung von Sequenzdiagrammen analysiert. Dabei werden, bedingt durch die inhärenten Abhängigkeiten zwischen den betreffenden Produktfunktionen, ⟨F030⟩, ⟨F040⟩ und ⟨F050⟩, ferner ⟨F060⟩, ⟨F100⟩ und ⟨F110⟩ sowie ⟨F080⟩ und ⟨F090⟩ in ihrem jeweiligen Zusammenwirken beschrieben. Für die übrigen Produktfunktionen, d. h. ⟨F010⟩, ⟨F020⟩, ⟨F070⟩, ⟨F120⟩, ⟨F130⟩ und ⟨F140⟩, erfolgt die Verhaltensmodellierung separat und in voneinander abgegrenzten Unterabschnitten.

2.1 Analyse von Funktionalität ⟨F010⟩ – Grafik-Einstellungen

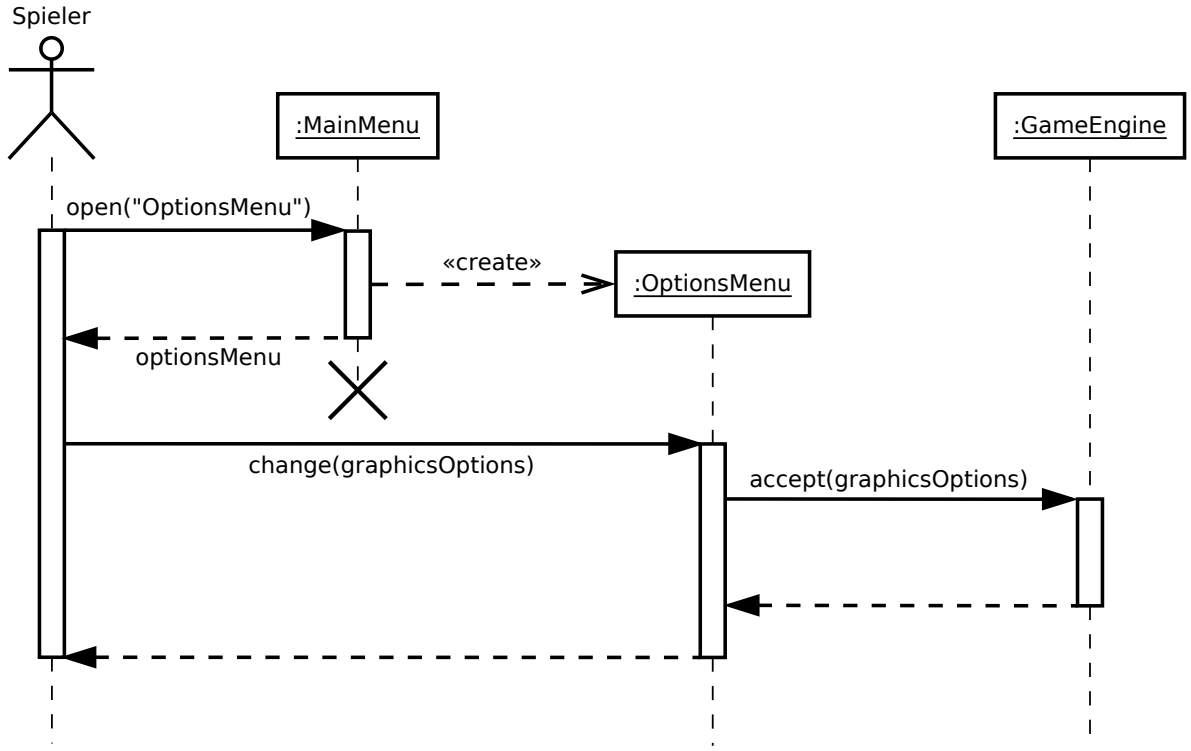

Abbildung 2.1: Sequenzdiagramm zu Produktfunktion ⟨F010⟩

Abbildung 2.1 beschreibt, wie Grafikeinstellungen der Anwendung modifiziert werden können.

Hierzu öffnet der Spieler über das Hauptmenü (realisiert durch das Absetzen einer `open("OptionsMenu")`-Anforderung) das Einstellungsmenü, welches dann die Möglichkeit bietet, Bildauflösung und Darstellungsmodus (Vollbild-, Fensteransicht) anzupassen.

Über Betätigung eines „Einstellungen speichern“-Buttons (veranschaulicht durch das Versenden einer `accept(graphicsOptions)`-Nachricht) werden die Änderungen dann von der Game-Engine übernommen.

2.2 Analyse von Funktionalität ⟨F020⟩ – Sound-Einstellungen

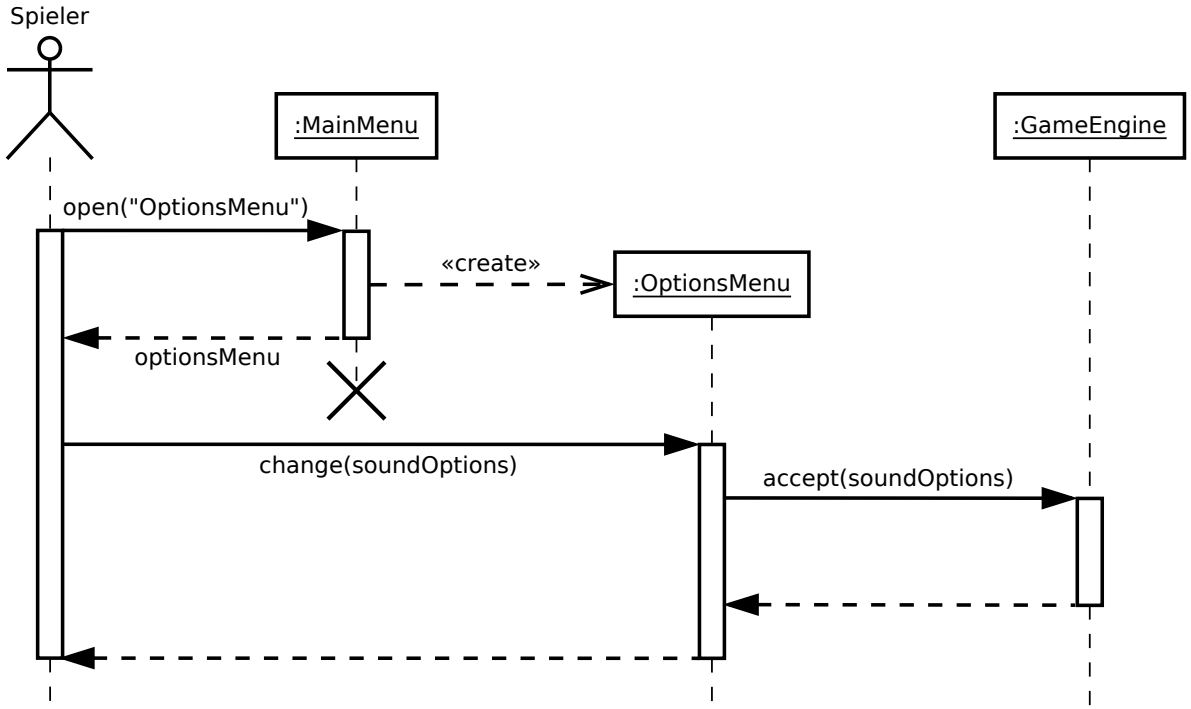

Abbildung 2.2: Sequenzdiagramm zu Produktfunktion ⟨F020⟩

Analog zu Abschnitt 2.1 stellt Abbildung 2.2 dar, wie der Spieler Änderungen an den Soundeinstellungen vornehmen kann.

Dazu öffnet er, wenn er sich im Hauptmenü befindet, das Einstellungsmenü. In diesem werden die gewünschten Änderungen an den Soundeinstellungen vorgenommen und über Betätigung eines „Einstellungen speichern“-Buttons (und Versenden einer `accept(soundOptions)`-Nachricht) von der Game-Engine übernommen.

2.3 Analyse von Funktionalitäten ⟨F030⟩, ⟨F040⟩, ⟨F050⟩ – Levelauswahl, Level laden und Darstellung des Spielgeschehens

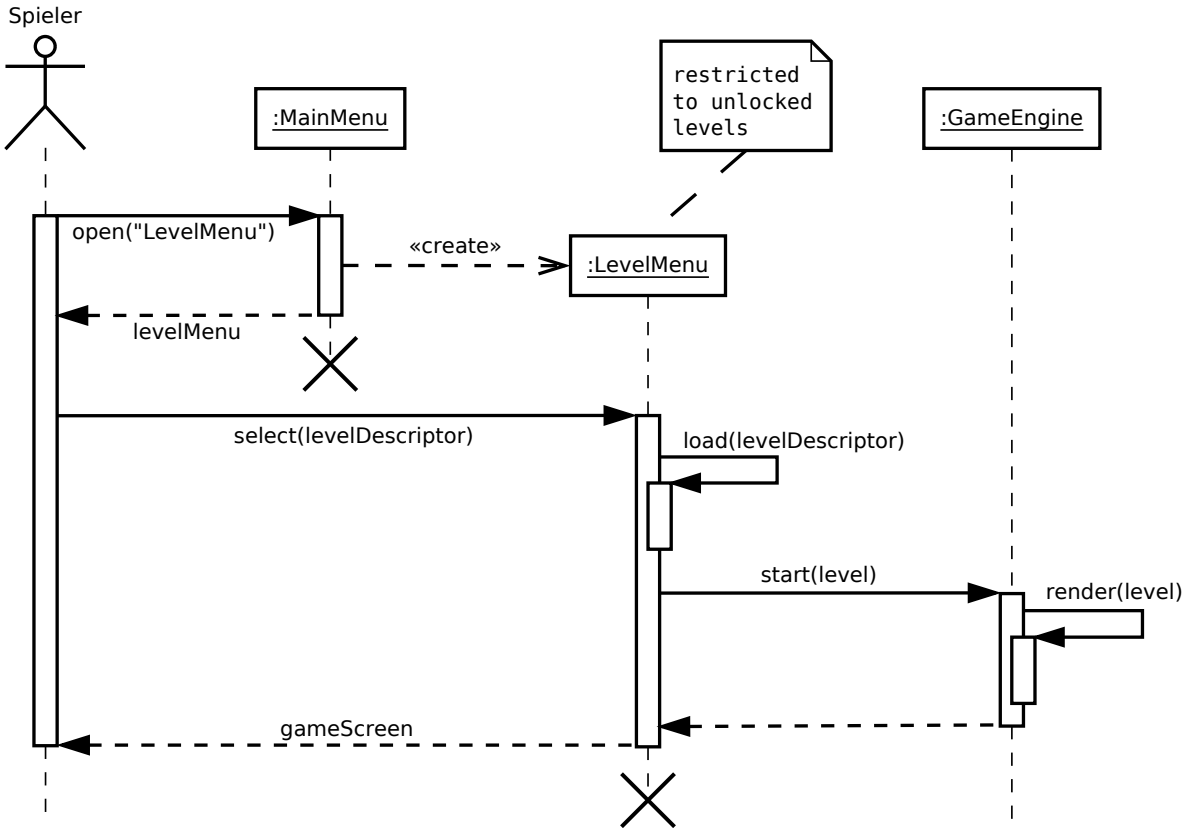

Abbildung 2.3: Sequenzdiagramm zu Produktfunktionen ⟨F030⟩, ⟨F040⟩ und ⟨F050⟩

Das Sequenzdiagramm in Abbildung 2.3 veranschaulicht die Prozesse der Levelauswahl, des Ladens eines Levels, sowie der Bereitstellung einer Visualisierung des Spielgeschehens.

Durch Absetzen einer `open("LevelMenu")`-Anforderung (über Anwahl der Option „Spiel starten“ im Hauptmenü der Anwendung) wird ein Levelauswahl-Dialog erzeugt, der eine Übersicht über die aktuell freigespielten Level bietet. Nach Festlegung des zu spielenden Levels (via `select(levelDescriptor)`), wird das betreffende Level geladen und an die Game-Engine übergeben. Diese realisiert durch Aufruf einer `render`-Funktion eine 3D-Visualisierung der Startkonfiguration eines Levels (d. h., der Wände und Gänge, Relaisstationen und Spielfigur in ihren definierten Konstellationen) und stellt die grafische Aufbereitung des Levels über einen Spielbildschirm dar.

Im weiteren Spielverlauf ruft die Game-Engine die Funktion `render` nach jeder Interaktion mit dem Anwender und Veränderung der jeweils aktuellen Konfiguration der Spielelemente erneut auf und realisiert so eine Echtzeit-Animation des Spielablaufs.

2.4 Analyse von Funktionalitäten ⟨F060⟩, ⟨F100⟩, ⟨F110⟩ – Spielfigur bewegen, Verlassen des begehbaren Bereichs (Level verlieren) und Betreten des Zielbereichs (Level gewinnen)

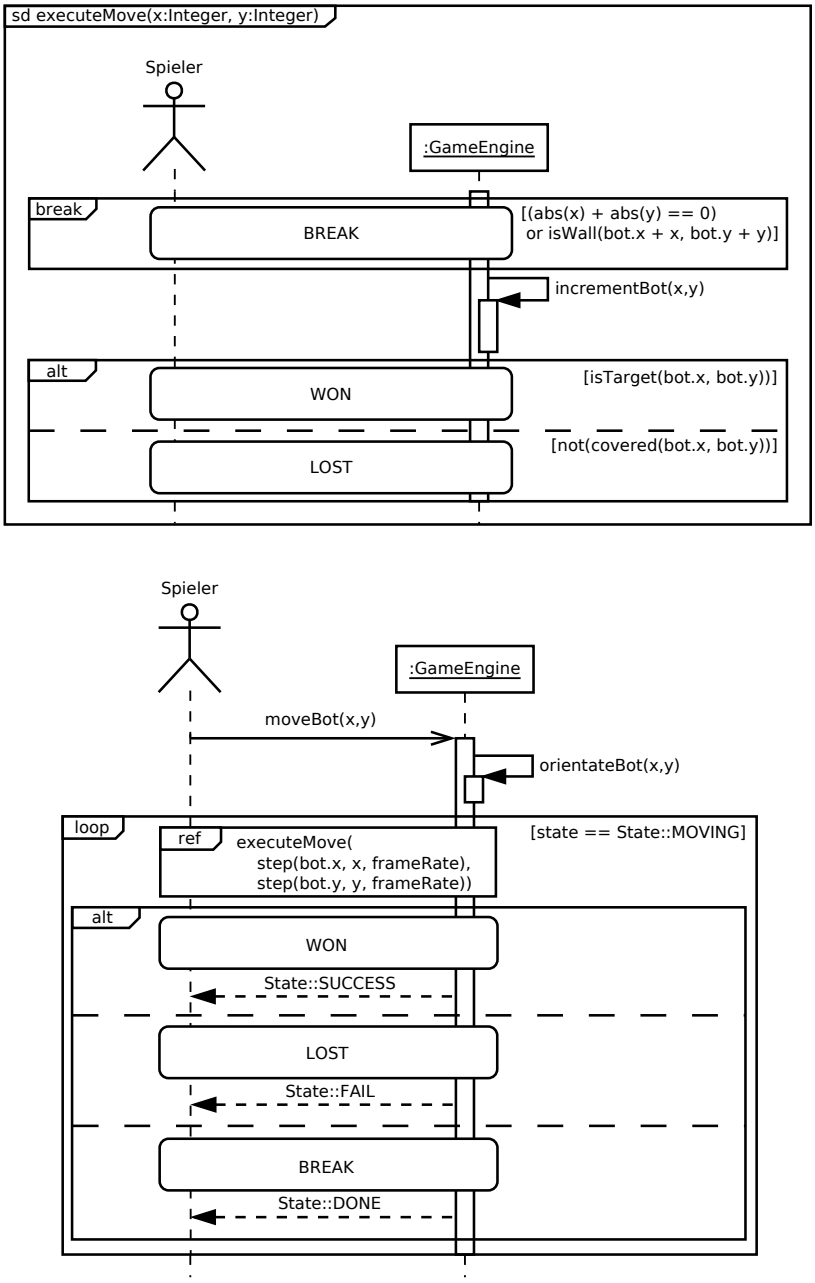

Abbildung 2.4: Sequenzdiagramme zu Produktfunktionen ⟨F060⟩, ⟨F100⟩ und ⟨F110⟩

Abbildung 2.4 beschreibt die Realisierung der Navigation der Spielfigur im Spielgeschehen, sowie die mit dieser Funktionalität unmittelbar verknüpften Produktfunktionen des erfolgreichen Absolvierens, ⟨F110⟩, respektive Verlierens eines Levels, ⟨F100⟩.

Durch Übermittlung einer `moveBot`-Nachricht (ausgelöst z. B. durch Betätigen der linken Maustaste über einer Zielposition im Spielbildschirm) weist der Spieler die Game-Engine an, die Spielfigur ausgehend von ihrem aktuellen Standort zu der gewünschten Position (gegebenen als (x, y)-Koordinatentupel) zu bewegen.

Infolge dessen wird die Spielfigur zunächst in Richtung der Zielkoordinaten ausgerichtet und anschließend in einer Sequenz von „Einzelschritten“[1] entlang des Richtungsvektors von Start- zu Zielposition verschoben. Dabei wird vor jedem Schritt überprüft, ob dieser zu einer Kollision mit einer Labyrinthwand führen würde. Ist dieses nicht der Fall, und wurde die Endposition der Bewegungssequenz noch nicht erreicht, wird die Position der Spielfigur aktualisiert; ansonsten verbleibt sie an der zuletzt eingenommenen Position und die Bewegungssequenz wird beendet.

Verlässt die Spielfigur in einer Bewegungssequenz den Abdeckungsbereich aller aktivierten IR-Relaisstationen, gilt das Level als verloren. Erreicht sie den Zielbereich, ist das Level gewonnen.

Da die Verarbeitung von `moveBot`-Direktiven nicht-blockierend erfolgt, kann der Spieler der Game-Engine während der Umsetzung einer Bewegungsanweisung neue Zielkoordinaten übermitteln. In diesem Fall wird die aktuelle Bewegungssequenz unterbrochen und von einer neuen, auf die aktualisierte Zielposition hin ausgerichteten Sequenz abgelöst.

Beendet eine Bewegungssequenz ein Level, wird der Spieler mittels eines Statusdialogs über seinen Erfolg bzw. Misserfolg (vgl. auch Abschnitt 2.8) informiert, der Spielbildschirm geschlossen und durch einen Levelauswahl-Dialog mit gegebenenfalls aktualisierten Einträgen ersetzt.

[1] Um den Bewegungsablauf in heterogenen Produktumgebungen einheitlich flüssig darzustellen, ist die Schrittweite als Funktion der Bildwiederholrate, in der das Spiel dargestellt wird, realisiert.

2.5 Analyse von Funktionalität ⟨F070⟩ – Relaisstation aufheben

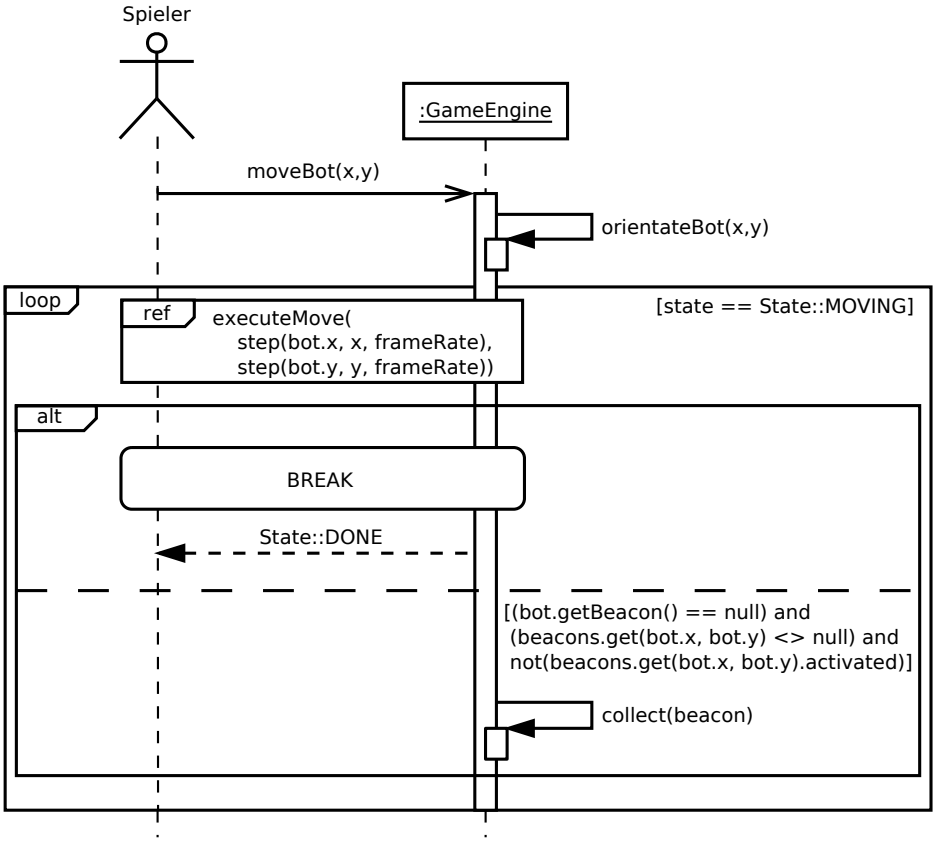

Abbildung 2.5: Sequenzdiagramm zu Produktfunktion ⟨F070⟩

Abbildung 2.5 veranschaulicht den Prozess der Aufnahme einer Relaisstation durch die Spielfigur.

Erreicht oder überschreitet die Spielfigur den Standort einer noch nicht aktivierten Relaisstation und führt nicht bereits eine andere Relaisstation mit sich, so wird die Relaisstation aufgenommen.

Transportiert die Spielfigur bereits eine Relaisstation oder ist die Relaisstation an dem gegebenen Standort bereits aktiviert, so wird der Standort passiert, ohne eine Folgeaktion auszulösen.

2.6 Analyse von Funktionalitäten ⟨F080⟩, ⟨F090⟩ – Sichtbarkeitsbereich berechnen und Relaisstation setzen

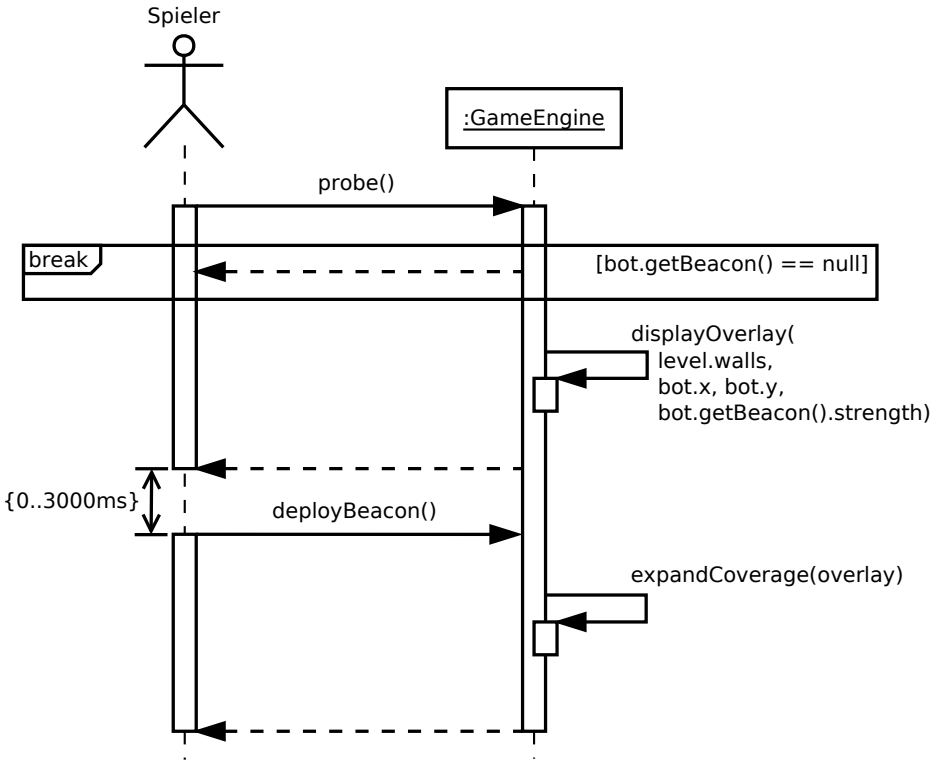

Abbildung 2.6: Sequenzdiagramm zu Produktfunktionen ⟨F080⟩ und ⟨F090⟩

Das Sequenzdiagramm in Abbildung 2.6 beschreibt das Vorgehen beim Platzieren einer Relaisstation.

Führt die Spielfigur eine Relaisstation mit sich, kann der Spieler die Game-Engine über eine `probe`-Anforderung (ausgelöst z. B. durch einmaliges Betätigen der rechten Maustaste) veranlassen, den Abdeckungsbereich[2] dieses Relais bei Platzierung am aktuellen Standort der Spielfigur als Overlay in die Spieldarstellung einzublenden.

Wiederholtes Betätigen der rechten Maustaste – bzw. eine äquivalente Eingabe über verwendetes Zeige-Eingabegerät – innerhalb eines Zeitintervalls von drei Sekunden nach Einblendung des Visibility-Polygons resultiert dann im Ablegen der Relaisstation an der eingenommenen Position und Aktualisierung des Abdeckungsbereichs des Netzwerks von IR-Relaisstationen des Levels.

Andernfalls, d. h. wenn diese Zeitspanne ohne eine Platzierungsanforderung verstreicht oder der Spieler eine andere als eine `deployBeacon`-Anweisung an die Game-Engine übermittelt, wird das Overlay wieder ausgeblendet.

[2] vgl. Abschnitt 1.1.3

2.7 Analyse von Funktionalität ⟨F120⟩ – Spiel pausieren

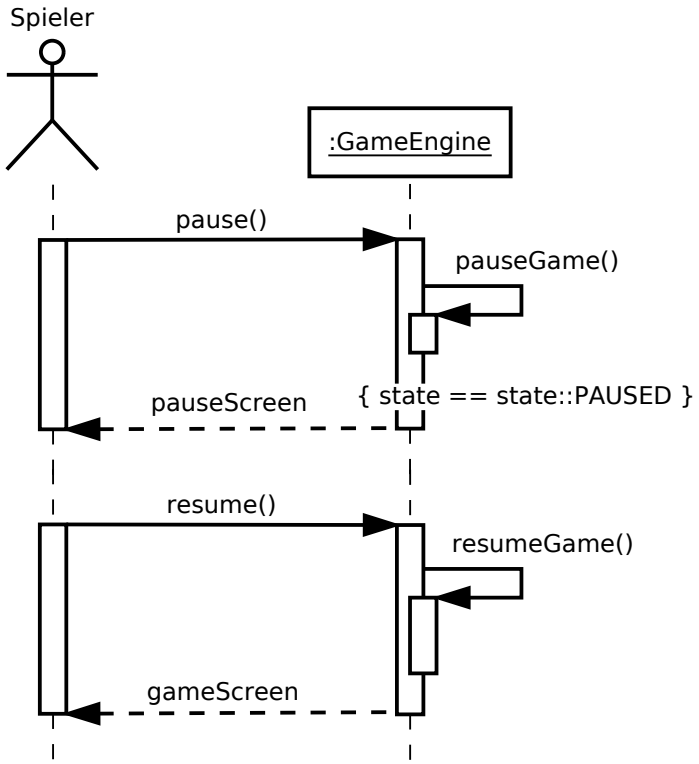

Abbildung 2.7: Sequenzdiagramm zu Produktfunktion ⟨F120⟩

Abbildung 2.7 beschreibt die Pausefunktion.

Betätigt der Spieler die definierte Pausetaste (`Escape`), wird das Spielgeschehen angehalten und der Spielbildschirm durch einen Pausebildschirm ersetzt.

Der Pausezustand wird erst durch wiederholtes Drücken der Pausetaste aufgehoben; andere Nutzereingaben, die während des Pausezustands erfolgen, werden von der Game-Engine ignoriert.

Nach Aufheben des Pausezustands wird der Pausebildschirm wieder durch den normalen Spielbildschirm ersetzt und das Spielgeschehen fortgesetzt.

2.8 Analyse von Funktionalität ⟨F130⟩ – Highscore-System

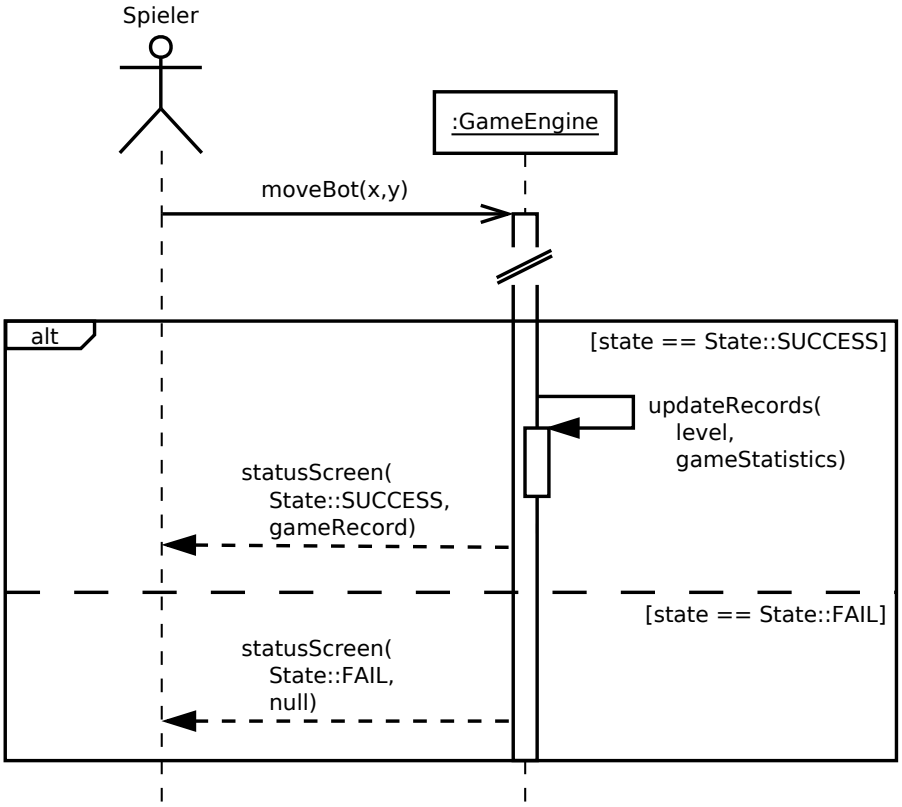

Abbildung 2.8: Sequenzdiagramm zu Produktfunktion ⟨F130⟩

Das Sequenzdiagramm in Abbildung 2.8 ergänzt die Ausführungen aus Abschnitt 2.4.

Beendet eine Bewegungssequenz ein Level, wird dem Spieler im Spielbildschirm ein Statusdialog präsentiert, der ein erfolgreiches Absolvieren des Levels bzw. das Scheitern dieses Versuchs kommuniziert.

Im Erfolgsfall werden zunächst die Spielfortschrittsdaten aktualisiert (vgl. Abschnitt 6.2) und anschließend dem Spieler das erzielte Ergebnis, nach Maßgabe einer Abstufung zwischen Anfänger, Fortgeschrittener und Experte (bemessen an der Anzahl verwendeter IR-Relaisstationen) und unter Bezugnahme auf bisherige Spielrekorde für das Level angezeigt.

Im Fall des Scheiterns wird der Spieler nur darüber benachrichtigt, dass die Spielfigur den Abdeckungsbereich der IR-Relaisstationen des Levels verlassen hat.

In beiden Fällen kann der Spieler abschließend über den Statusdialog zum Levelauswahl-Dialog zurückkehren oder – alternativ – das Spiel in gewähltem Level wiederholen.

2.9 Analyse von Funktionalität ⟨F140⟩ – Spiel beenden

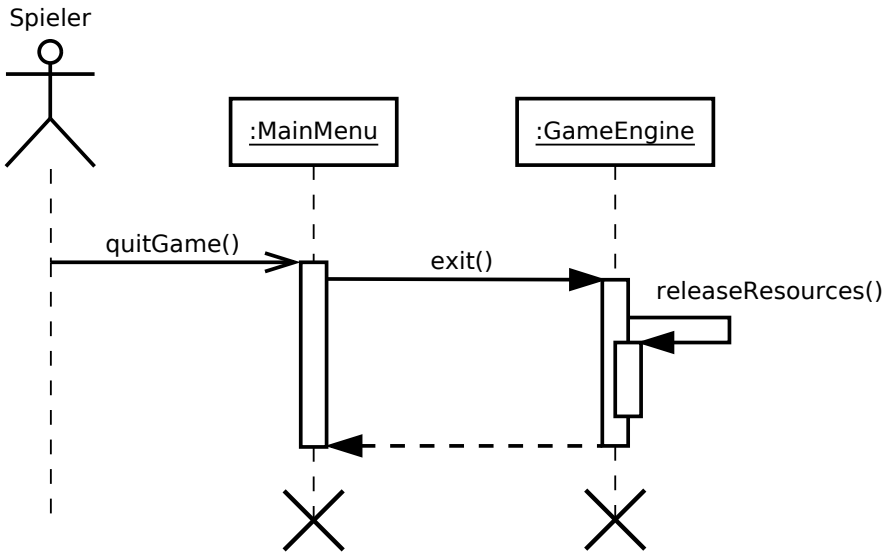

Abbildung 2.9: Sequenzdiagramm von ⟨F140⟩

Abbildung 2.9 beschreibt den Prozess des Beendens der Anwendung. Auf Anforderung des Spielers über Aufruf der `quitGame`-Methode (durch Anwahl der Option „Spiel beenden“ im Hauptmenü) gibt die Game-Engine die von ihr reservierten Systemressourcen frei und wird anschließend zusammen mit der Anwendung beendet.

3 Resultierende Softwarearchitektur

In diesem Kapitel werden die zu entwickelnden Komponenten und Subsysteme der Anwendung Labyrinth Games dargestellt und beschrieben. Die Softwarearchitektur der Anwendung ist eine typische Single-Computer-Anwendung und setzt keinen Server zur Verwendung voraus. Somit werden sämtliche Operationen auf dem System des Nutzers ausgeführt.

3.1 Komponentenspezifikation

Die Anwendung Labyrinth Games wird durch keine übliche Komponenten-Architektur realisiert, sondern ist eine Klassen- und State-basierte C++-Anwendung. Hierbei wird eine Backbone-Subsystem-Struktur verwendet, die um Komponenten und Teilkomponenten erweitert wurde.
Wie bereits erwähnt besitzt die Anwendung, abgesehen von den verwendeten Programmierbibliotheken, keine typischen Komponenten, sondern wird über sogenannte Teilkomponenten und States dargestellt.
Die Abbildung 3.1 zeigt ein abgewandeltes Komponentendiagramm mit den in unserer Anwendung vorhandenen Teilkomponenten und sonstigen Bestandteilen dieser, welche im folgenden genauer erläutert werden. Die Subkomponenten werden zusammen mit der Klasse `Laby`, die das sogenannte „Backbone“ bildet, zu Beginn der Anwendung ausgeführt und stellen Schnittstellen bereit, die jedoch genau definiert wurden und nur beim Anwendungsstart ausgeführt werden und eher einem ausgelagerten Methodenaufruf ähneln.
Besonders zu beachten ist die Rolle der Teilkomponente `State-Manger`, da dieser die bereits erwähnten States verwaltet. Von diesem wird, wenn aufgerufen, der gewünschte State über den Konstruktor jenes States neu erzeugt und dem Unique-Pointer in `Laby`, welcher den momentanen State der Anwendung verwaltet, übergeben. Der zuvor verwendete State wird aus dem Speicher gelöscht und muss für eine erneute Verwendung über den `State-Manager` neu erzeugt werden.
Diese States bilden die verschiedenen Funktionalitäten der Anwendung ab (Hauptmenü, Levelauswahl, Optionsmenü, Level) und können als Subsysteme betrachtet werden, welche auf dem „`Laby`-Backbone“ „aufgesteckt“ werden, wobei immer nur ein solches Subsystem auf dem „Backbone“ vorhanden sein kann. Neben den States existieren noch `Windows` ("Window gewonnen", "Window verloren") welche aufgerufen werden, wenn der Spieler ein Level gewinnt oder verliert. Sie werden als Zeigervariable im `Game` State gespeichert und dient der Visualisierung seiner Leistung.

Ein `Window` bildet somit keinen eigenen State, sondern ist eine ausgelagerte GUI-Darstellung mit eigenständig implementierten Event-Funktionen, welche, mit Ausnahme des Aufrufs, nicht von außerhalb des `Windows` aufgerufen werden können. Ein `Window` beinhaltet Funktionen, welche den State von `Laby` ändern, dies wird allerdings über `Laby` ausgeführt und nicht über einen direkten Zugriff auf den `State-Manager`.

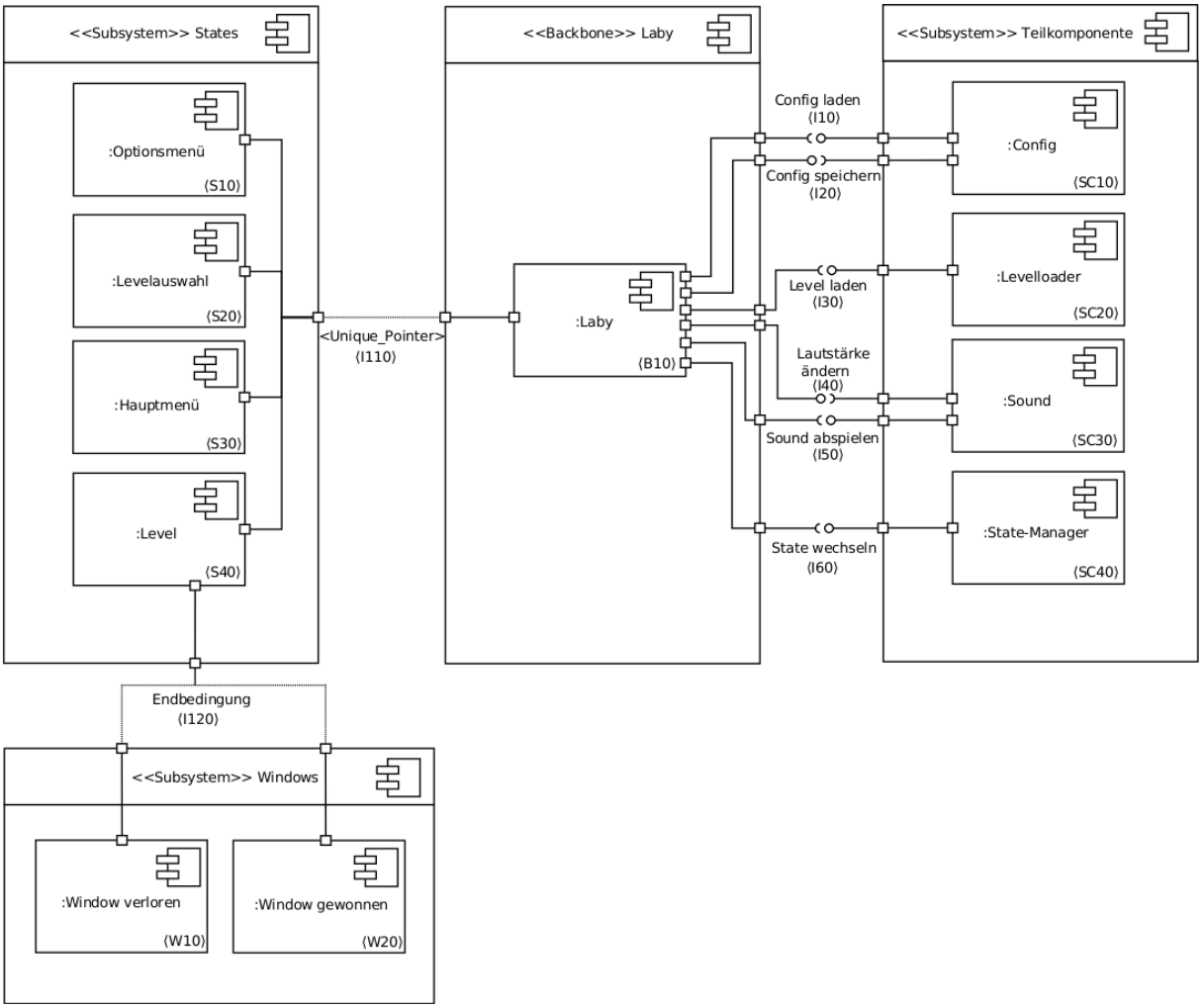

Abbildung 3.1: Diagramm der Komponenten von Labyrinth Games

States und Windows bilden also eigenständig funktionierende Teile der Anwendung, welche sich aber nicht als Komponente qualifizieren und keine oder nur vereinzelte Werte der Hauptklasse übernehmen.

Beschreibung der Komponenten:

Backbone:

Backbone⟨B10⟩: <**Laby**> **(Laby)**

Laby ist die „Hauptklasse“ der Anwendung und bildet das sogenannte „Backbone“ der Softwarestruktur. Als Backbone hat sie die Aufgabe, die einzelnen Unterklassen zu verwalten und die bereitgestellten Schnittstellen der anderen Komponenten (externe Bibliotheken vgl. Pflichtenheft) und Teilkomponenten (s.o.) zu verwenden und dem restlichen Programm zur Verfügung zu stellen. Sie stellt mit dem <Unique_Pointer> eine Art „Steckplatz“ für einen State bereit.

Teilkomponenten:

Teilkomponente⟨SC10⟩: <**Config**> **(Teilkomponente)**

Config ist eine C++ Klasse und stellt eine sogenannte „Teilkomponente“ dar. Sie dient dem Einlesen und Verändern der Konfigurationsdatei (config) und verwaltet die eingelesenen Daten.

Teilkomponente⟨SC20⟩: <**Levelloader**> **(Teilkomponente)**

Die Teilkomponente Levelloader hat die Aufgabe, beim Starten der Anwendung alle Leveldateien aus dem entsprechenden Ordner zu laden und in einem einheitlichen Format zwischenzuspeichern. Außerdem verwaltet sie die Level, um sie später für den Spieler bereitzustellen und zu laden.

Teilkomponente⟨SC30⟩: <**Sound**> **(Teilkomponente)**

Um jegliche Art von Klanguntermalung zu ermöglichen gibt es die Teilkomponente Sound. Ihre Aufgabe ist die Bereitstellung von Möglichkeiten Audiodateien einzulesen, abzuspielen und ihre Lautstärke anzupassen.

Teilkomponente⟨SC40⟩: <**State-Manager**> **(Teilkomponente)**

Der State-Manager bildet einen Zentralen Teil der Anwendung. Er hat die Aufgabe neue 'States' zu erstellen und diese dem Backbone zuzuweisen. Hierbei werden alte, nicht mehr benötigte 'States' aus dem Speicher gelöscht. Er ist somit eine der zentralen Steuereinheiten der Anwendung.

States:

State⟨S10⟩: <**Optionsmenü**> **(States)**

Das Hauptmenü bildet die zentrale Navigationseinheit unserer Anwendung und gibt dem Anwender die Möglichkeit die verschiedenen Bereiche der Anwendung zu nutzen.

State⟨S20⟩: <**Levelauswahl**> **(States)**

Der State Optionsmenü bietet dem Anwender die Möglichkeit, die Grafik- und Audioeinstellungen der Anwendung nach seinen Wünschen anzupassen.

State⟨S30⟩: <**Hauptmenü**> **(States)**

In der Levelauswahl werden dem Anwender alle freigespielten Level angezeigt. Aus diesen kann er eines auswählen, um es zu spielen.

State⟨S40⟩: <**Level**> **(States)**
Der Level-State stellt die eigentliche Anwendung dar. In dem Level werden dem Anwender Level und Spielfigur grafisch dargestellt und ihm ermöglicht das Spiel zu spielen.

Windows:
Window⟨W10⟩: <**Window verloren**> **(Windows)**
Dieses Window wird aufgerufen, wenn der Spieler das Level verloren hat. Es gibt dem ihm die Möglichkeit, das Level erneut zu starten oder zur Levelauswahl zurückzukehren.

Window⟨W20⟩: <**Window gewonnen**> **(Windows)**
Dieses Window wird aufgerufen, wenn der Spieler das Level gewonnen hat. Es zeigt seinen erreichten Highscore an und gibt ihm die Möglichkeit, das nächste freigeschaltete Level zu spielen oder zur Levelauswahl zurückzukehren.

3.2 Schnittstellenspezifikation

Wie aus dem Komponentendiagramm zu erkennen ist, ergeben sich aus der Komponentenstruktur mehrere Schnittstellen zwischen den Komponenten. Im Folgenden werden die einzelnen Komponenten beschrieben und die nötigen Operationen zum Datenaustausch erläutert.

Schnittstelle ⟨I10⟩: <Config laden>

Operation	**Beschreibung**
Config()	Der Konstruktor erzeugt die Klasse Config und liest automatisch die Werte aus der Konfigurationsdatei ein.

Schnittstelle ⟨I20⟩: <Config speichern>

Operation	**Beschreibung**
save()	Schreibt alle Variablenwerte der Klasse Config in die Konfigurationsdatei.

Schnittstelle ⟨I30⟩: <Level laden>

Operation	**Beschreibung**
readLevel(json: const json&)	Liest die Leveldaten aus der übergebenen .json-Datei.

Schnittstelle ⟨I40⟩: <Lautstärke ändern>

Operation	**Beschreibung**
setVolume(volume: float)	Ändert die Lautstärke auf den übergebenen Wert.

Schnittstelle ⟨I50⟩: <Sound abspielen>

Operation	Beschreibung
play(name: const String)	Spielt den Sound ab, der den übergebenen Namen besitzt.

Schnittstelle ⟨I60⟩: <State wechseln>

Operation	Beschreibung
switchState(std::move(state))	Wechselt den State in den angegebenen State.

Abstrakte Schnittstellen (Pointer):
Folgender Absatz dient der genaueren Beschreibung der `State` und `Window` Subsysteme, welche jedoch keine Interfaces bereitstellen.

Schnittstelle ⟨I110⟩: <Unique_Pointer>

Operation	Beschreibung
switchState(std::move(state))	Ändert den <Unique_Pointer> von Laby zu dem angegebenen State.

Schnittstelle ⟨I120⟩: <Endbedingung>

Operation	Beschreibung
setVisible(true)	Macht das zuvor erzeugte Window basierend auf der Endbedingung sichtbar. (Verloren: Window verloren; Gewonnen: Window gewonnen)

3.3 Protokolle für die Benutzung der Komponenten

In diesem Abschnitt wird die Wiederverwendbarkeit der Komponenten und mögliche Darstellung mittels Statechart-Diagrammen diskutiert.
Die Softwarekomponenten dieses Projekt sind grundsätzlich nicht für eine Wiederverwendung geeignet, da sie zu spezifisch für dieses Softwareprojekt entwickelt worden sind. Da es sich außerdem bei den verwendeten Softwarekomponenten, wie bereits beschrieben, eher um ausgelagerte Funktionalitäten als um Komponenten handelt, wäre eine Verwendung dieser in anderen Anwendungsfällen nur bedingt oder gar nicht möglich.
Eine Beschreibung der Komponenten mittels Statechart-Diagrammen ist aus den oben genannten Gründen deshalb nicht möglich und erübrigt sich auch für die „Teilkomponenten“ sowie `States` und `Windows`.
Zu sagen ist, dass eine ähnliche Anwendung basierend auf dem angewandten Backbone-Prinzip entwickelt werden könnte. Hier wäre allerdings eine entsprechende Abänderung von `Laby` notwendig, sofern es sich nicht um eine zu diesem Softwareprojekt äquivalente Anwendung handelt.

4 Verteilungsentwurf

Dieses Kapitel beschreibt die physische Verteilung der einzelnen Komponenten des Softwareprodukts. Es wird auf ein Verteilungsdiagramm verzichtet, da es sich bei der Einbettung der Anwendung auf dem System des Nutzers um eine für solche Anwendungen typische Verteilung handelt. Da es sich bei der Software, wie bereits in Kapitel 3 erwähnt, um eine nicht-verteilte Anwendung handelt, ist die Anwendung lediglich im Betriebssystem des Endgeräts eingebettet und setzt keine Client-Server Kommunikation oder Komponenten voraus.

5 Implementierungsentwurf

In diesem Abschnitt werden die verwendeten Programmierbibliotheken, sowie die in Kapitel 3 aufgeführten Komponenten (Teilkomponenten, States, etc.) detailliert beschrieben. Hierfür wird mittels Doxygen aus dem Code inklusive Entwicklerkommentaren eine Vollständige Doxygen Dokumentation erstellt und zusätzlich zu diesem Dokument
in PDF (ibr_alg_g1_TechnischerEntwurf_Doxygen.pdf)
und HTML Format (ibr_alg_g1_TechnischerEntwurf_Doxygen_HTML.zip) bereitgestellt. Aus dieser Dokumentation können alle Klassen, Methoden und sonstige Funktionalitäten entnommen werden.

6 Datenmodell

Daten, die das Softwareprodukt dauerhaft speichert, umfassen Leveldefinitionen, Spielfortschrittsdaten und Konfigurationseinstellungen für Grafik- und Soundoptionen. Vorgehalten werden die Daten in separaten `.json`-Dateien.

In den folgenden Abschnitten werden die einzelnen Produktdatenkategorien zunächst jeweils unter Verwendung von Klassendiagrammen logisch modelliert und anschließend auf das gewählte physische Datenmodell abgebildet.

6.1 Spielleveldaten ⟨D10⟩

6.1.1 Logisches Schema

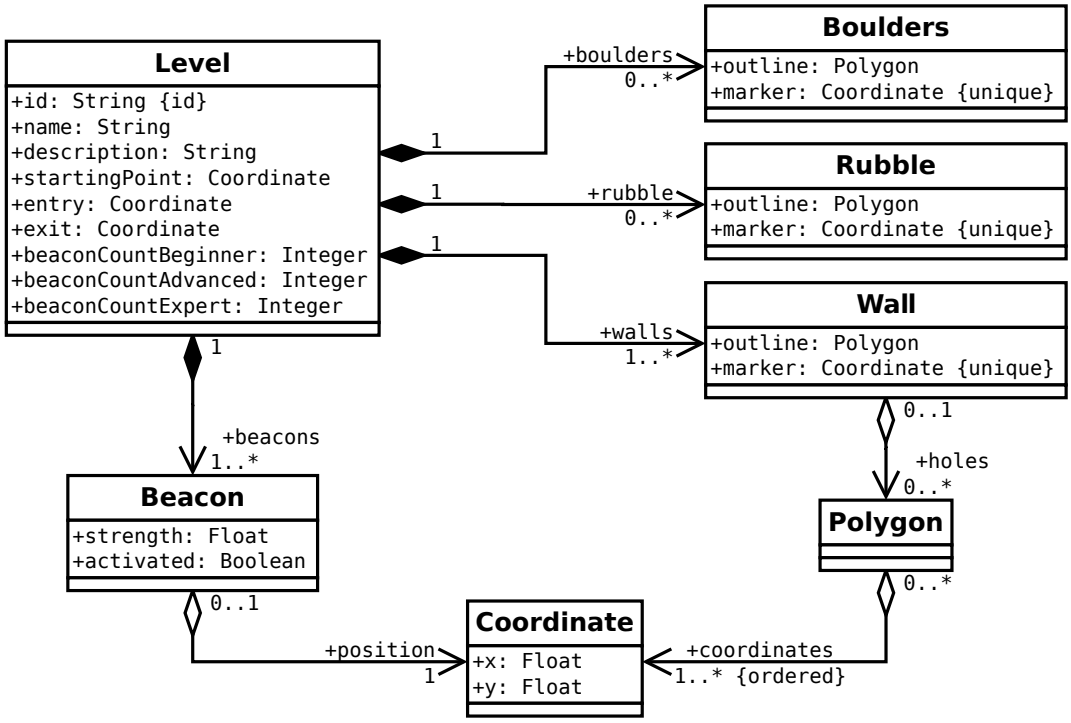

Abbildung 6.1: Logisches Schema der Spielleveldaten ⟨D10⟩

Erläuterung

Jedes Spiellevel ist eine Instanz des in Abbildung 6.1 gegebenen Datenmodells. Per Konvention besitzt jedes Level eine `Wall`, deren `outline` zugleich den Umriss des Levels beschreibt. Begehbare Bereiche sind `holes` dieser `Wall` und unpassierbare Bereiche werden durch `Wall`-Instanzen, die in `holes` platziert sind, konstruiert.

Optional kann ein Level Bereiche enthalten, die nicht passierbar sind, durch die aber die Visibility der Beacons nicht eingeschränkt wird (`Boulder`) und Bereiche, die zwar passierbar sind, in denen aber keine Beacons gesetzt werden können (`Rubble`).

Detailliertere Informationen zu Eigenschaften und Beziehungen der in dem Datenmodell definierten Entitites können den nachfolgenden Aufstellungen entnommen werden:

Level ⟨E10⟩

Attribut	Datentyp	Beschreibung
`id`	`String`	Eindeutiger Kennzeichner des Levels
`name`	`String`	Name des Levels
`description`	`String`	Beschreibung (Story) des Levels
`startingPoint`	`Coordinate`	Initialposition der Spielfigur
`entry`	`Coordinate`	Position des Leveleingangs
`exit`	`Coordinate`	Position des Levelausgangs
`beaconCountBeginner`	`Integer`	Maximal zulässige Anzahl zu verwendender Beacons, die nach Absolvieren eines Levels in der Verleihung des Prädikats „Anfänger“ resultieren
`beaconCountAdvanced`	`Integer`	Maximal zulässige Anzahl zu verwendender Beacons, die nach Absolvieren eines Levels in der Verleihung des Prädikats „Fortgeschrittener“ resultieren
`beaconCountExpert`	`Integer`	Maximal zulässige Anzahl zu verwendender Beacons, die nach Absolvieren eines Levels in der Verleihung des Prädikats „Experte“ resultieren

Beziehung	Kardinalität	Erwartete Datenmenge	Beschreibung
`walls`	1..*	Min: 1, Max: 99	Wände je Level
`boulders`	0..*	Min: 0, Max: 99	Boulder-Bereiche je Level
`rubble`	0..*	Min: 0, Max: 99	Rubble-Bereiche je Level
`beacons`	1..*	Min: 1, Max: 99	Relaisstationen je Level

Wall ⟨E20⟩

Attribut	Datentyp	Beschreibung
`outline`	`Polygon`	Umriss (2D) der Wand
`marker`	`Coordinate`	Koordinaten eines von `outline` eingeschlossen Punkts, der eindeutig einer `Wall`-Instanz zugeordnet werden kann und disjunkt zu allen Löchern (s. u.) der Wand ist

Beziehung	Kardinalität	Erwartete Datenmenge	Beschreibung
`holes`	0..*	Min: 0, Max: 99	Löcher (bzw. begehbare Bereiche) je Wand

Boulder ⟨E30⟩

Attribut	Datentyp	Beschreibung
`outline`	`Polygon`	Umriss des Boulder-Bereichs
`marker`	`Coordinate`	Koordinaten eines von `outline` eingeschlossen Punkts, der eindeutig einer `Boulder`-Instanz zugeordnet werden kann

Rubble ⟨E40⟩

Attribut	Datentyp	Beschreibung
`outline`	`Polygon`	Umriss des Rubble-Bereichs
`marker`	`Coordinate`	Koordinaten eines von `outline` eingeschlossen Punkts, der eindeutig einer `Rubble`-Instanz zugeordnet werden kann

Beacon ⟨E50⟩

Attribut	Datentyp	Beschreibung
`strength`	`Float`	Sendeleistung (Radius) der Relaisstation
`activated`	`Boolean`	Indikator, der beschreibt, ob die Relaisstation initial aktiviert ist

Beziehung	Kardinalität	Erwartete Datenmenge	Beschreibung
`position`	0..*	Min: 1, Max: 1	Position der Relaisstation

Polygon ⟨E60⟩

Beziehung	Kardinalität	Erwartete Datenmenge	Beschreibung
`coordinates`	0..*	Min: 3, Max: 999	Definierende Koordinaten eines geschlossenen Polygonzugs[1]

6.1.2 Physisches Schema

Repräsentation der Spielleveldaten als `.json`-Datei:

```
{
        "id": String,
        "name": String,
        "description": String,
        "startingPoint": [Numeric, Numeric],
        "entry": {
                "leftAnchor": [Numeric, Numeric],
                "rightAnchor": [Numeric, Numeric]
        },
        "exit": {
                "leftAnchor": [Numeric, Numeric],
                "rightAnchor": [Numeric, Numeric]
        },
        "walls": [{
                "marker": [Numeric, Numeric],
                "outline": [[Numeric, Numeric]],
                "holes": [[[Numeric, Numeric]]]
        }],
        "boulders": [{
                "marker": [Numeric, Numeric],
                "outline": [[Numeric, Numeric]]
        ]},
        "rubble": [{
                "marker": [Numeric, Numeric],
                "outline": [[Numeric, Numeric]]
        ]},
```

[1] Leere Polygone sind zulässig, sinnvolle Konfigurationen sollten jedoch drei (oder mehr) Koordinatentupel umfassen.

```
        beacons: [{
                "x": Numeric,
                "y": Numeric,
                "strength": Numeric,
                "activated": Boolean
        }],
        "beaconCountBeginner": Numeric,
        "beaconCountAdvanced": Numeric,
        "beaconCountExpert": Numeric
}
```

6.2 Spielfortschrittsdaten ⟨D20⟩

6.2.1 Logisches Schema

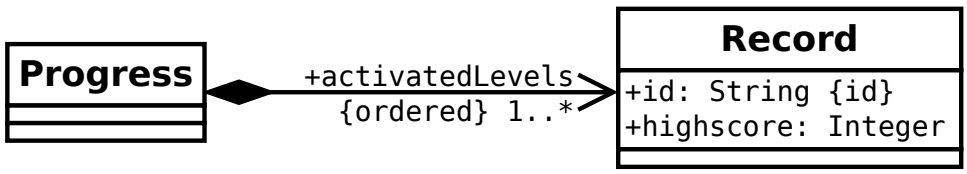

Abbildung 6.2: Logisches Schema der Spielfortschrittsdaten ⟨D20⟩

Erläuterung

Der Spielfortschritt wird über eine Liste von Rekorden freigespielter (aktivierter) Level dokumentiert, wobei initial nur ein Level aktiviert ist und die Eigenschaft `highscores` der zugeordneten `Record`-Instanz nicht gesetzt sind.

Weitere Details zu den Attributen der Entitäten `Progress` und `Record` sowie zu der Beziehung, in der die Entitäten zueinander stehen, sind den nachstehenden tabellarischen Auflistungen zu entnehmen:

Progress ⟨E50⟩

Beziehung	Kardinalität	Erwartete Datenmenge	Beschreibung
`activated-Levels`	1..*	Min: 1, Max: 999	Liste der `Record`-Instanzen aktivierter Level

Record ⟨E60⟩

Attribut	Datentyp	Beschreibung
`id`	`String`	Eindeutiger Kennzeichner eines Levels
`highscore`	`Integer`	Minimale Anzahl verwendeter Beacons, mit denen das bezeichnete Level absolviert wurde

6.2.2 Physisches Schema

Repräsentation der Spielfortschrittsdaten als `.json`-Datei:

```
{
        "activatedLevels": [{
                "id": String,
                "highscore": Numeric
        }]
}
```

6.3 Einstellungsdaten ⟨D30⟩

6.3.1 Logisches Schema

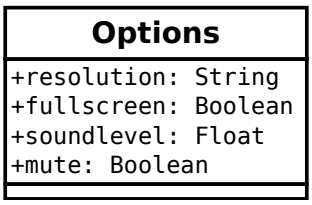

Abbildung 6.3: Logisches Schema der Einstellungsdaten ⟨D30⟩

Erläuterung

Als Einstellungsparameter der Anwendung werden die gewählte Bildauflösung, der Darstellungsmodus (Vollbild, Fenster) sowie der Volumenpegel der Audio-Wiedergabe (mit der Option einer Stummschaltung) gespeichert.

Options ⟨E70⟩

Attribut	Datentyp	Beschreibung
`resolution`	`String`	Bildauflösung (erwartete Werte: "800x600", "1024x768" oder "1280x1024")
`fullscreen`	`Boolean`	Indikator, der definiert, ob die Anwendung im Vollbildmodus betrieben wird
`soundlevel`	`Float`	Volumenpegel der Audio-Wiedergabe
`mute`	`Boolean`	Indikator für Stummschaltung der Sound-Wiedergabe

6.3.2 Logisches Schema

Repräsentation der Einstellungsdaten als `.json`-Datei:

```
{
        "resolution": String,
        "fullscreen": Boolean,
        "soundlevel": Numeric,
        "mute": Boolean
}
```

7 Konfiguration

Eine besondere Konfiguration der Produktumgebung über die Bereitstellung ist nicht erforderlich.

Unser Projekt verwendet keinen Server Konformität mit den im Pflichtenheft beschriebenen Anforderungen an die Zielplattform, d. h. insbesondere, keine Anpassungen des Betriebssystems, von Start-Up-Skripten, Umgebungsvariablen o. ä. sind erforderlich. Gegebenenfalls sind bestimmte Third-Party-Libraries manuell zu installieren dies sind jedoch Abhängigkeiten und keine Konfigurationen.

Konfigurationsdateien gibt es. Allerdings sind diese Bestandteil des Lieferumfangs des Produkts (speziell: Produktdaten) und insofern nicht erforderlich (im Sinne von Voraussetzung) für Installation und Inbetriebnahme des Produkts.

Es sollte beschrieben werden, wo diese zu finden sind und welchen Zweck sie erfüllen. Gibt es Parameter die angepasst werden können, so sollte auf diese eingegangen werden.

Diese sind in den Produktdaten (und somit bereits in Kapitel 6 „Datenmodell“) abgehandelt.

8 Änderungen gegenüber Fachentwurf

Im Datenmodell Spielfortschrittsdaten (Abschnitt 6.2) wurde die Zeit als definierendes Highscore-Kriterium entfernt, da bei genauerer Betrachtung die Zeit bei einem Logikspiel als primärer Maßstab nicht sinnvoll ist.

Die Spielfigur wird nun als IR-gesteuerter Roboter, Lichtquellen als Relaisstationen und beleuchtete Gebiete als von Relaisstationen abgedeckte Bereiche interpretiert.

Zum Datenmodell der Spielleveldaten wurden optionale Bereiche hinzugefügt, die in einem Level enthalten sein können: Boulder-Bereiche (nicht passierbar, schränkt aber nicht die Visibility der Beacons ein) und Rubble-Bereiche (passierbar, aber es können keine Beacons gesetzt werden).

9 Erfüllung der Kriterien

In diesem Kapitel wird beschrieben, wie die einzelnen Kriterien des Pflichtenhefts von dem realisierten Softwareprodukt erfüllt werden und welche Komponenten diese erfüllen.

9.1 Musskriterien

⟨RM10⟩ Spiellevel
Der Lieferumfang des Produkts umfasst mehr als 4 Spiellevel.

⟨RM20⟩ 2D-Logik
Die Komponente `GameEngine` realisiert eine 2D-Spiellogik, d. h. die Navigation der Spielfigur, Kollisionsdetektion und Berechnung von Abdeckungsbereichen erfolgen ausschließlich nach Maßgabe von Koordinaten in der (zweidimensionalen) Ebene.

⟨RM30⟩ 3D-Grafik
Die 2D-Spiellogik wird von der Game-Engine unter Verwendung der Bibliothek `Ogre` dreidimensional aufbereitet dargestellt.

⟨RM40⟩ Maussteuerung
Die Navigation der Spielfigur durch das Spielgeschehen erfolgt über Benutzereingaben mittels Zeiger-Eingabegerät; diese werden von der Game-Engine mithilfe der Bibliothek `OIS` abgefangen und verarbeitet.

⟨RM50⟩ Aufheben von Relaisstationen
Noch nicht aktivierte Relaisstationen, die sich im gleichen Abdeckungsbereich aktivierter Relaisstationen befinden wie die Spielfigur, können eingesammelt und transportiert werden. Als elementarer Aspekt der Spiellogik wird dieses Kriterium von der Komponente `GameEngine` realisiert.

⟨RM60⟩ Setzen von Relaisstationen
Sofern eine Relaisstation aufgehoben wurde, kann sie im Abdeckungsbereich abgesetzt werden. Die Game-Engine stellt dann den neuen Abdeckungsbereich dar.

9.2 Sollkriterien

⟨RS10⟩ Level-Loader
Wird im Level-Menü ein wählbares Spiellevel ausgewählt, erzeugt das Level-Menü einen neuen Level-State mit dem gewählten Level.

⟨RS20⟩ Spiellevel-Auswahl
Die Komponente `LevelMenu` ermöglicht den wahlfreien Zugriff auf aktivierte Level.

⟨RS30⟩ Grafik-Einstellungen
Modi der Auflösung und Darstellung (Fenster, Vollbild) der Spielgrafik sind unter Verwendung der Komponente `OptionMenu` anpassbar.

⟨RS40⟩ GUI
Die Komponenten `MainMenu`, `OptionMenu` und `LevelMenu` bilden im Verbund eine grafische Benutzerschnittstelle, über die der Spieler Grafik- und Sound-Einstellungen verändern sowie die Spiellevel-Auswahl über eine dedizierte GUI tätigen kann.

⟨RS50⟩ Audio
Bei erfolgreichem Abschluss sowie beim Verlieren eines Levels wird mittels der Programmierbibliothek `OpenAL` ein entsprechender Sound abgespielt.

9.3 Kannkriterien

⟨RC10⟩ Story-System
Ein Story-System wurde bislang nicht umgesetzt.

⟨RC20⟩ Highscore-System
Ein Highscore-System wird mittels Drei-Sterne-Bewertung umgesetzt. Hierbei besitzt jedes Level eine festgelegte Anzahl an Beacons, die die Schwellenwerte für die entsprechende Bewertung festlegen. Der Spieler erhält, je nach Anzahl der verwendeten Beacons, bei erfolgreichem Abschluss eines Levels seine entsprechende Wertung ausgegeben.

⟨RC30⟩ Level-Editor
Ein Level-Editor wurde bislang nicht umgesetzt.

⟨RC40⟩ Tastatursteuerung
Eine alternative Möglichkeit der Navigation der Spielfigur bietet die Steuerung mittels Tastendruck der entsprechenden Tastatur-Tasten. Diese Eingaben werden äquivalent zu der Maussteuerung über die Programmierbibliothek `OIS` abgefangen und von der Game-Engine verarbeitet.

⟨RC50⟩ Ambientes Audio
Ambientes Audio wurde bislang nicht umgesetzt.

10 Glossar

Game Engine: Bildet die Basis einer Spielanwendung und ist für die logische und grafische Darstellung dieser verantwortlich.

IR: Abkürzung für Infrarot.

IrDA: Protokoll zur Datenübertragung mittels Infrarotlicht.

IR-Relais (Infrarot-Relaisstation): Empfangs- und Sendeeinheit, die empfangene IrDA-Signale automatisch weitervermittelt.

`Ogre` (**Object-Oriented Graphics Rendering Engine**, `http://www.ogre3d.org/`): Eine Open-Source-Programmbibliothek für `C++` zur Darstellung von zwei- und dreidimensionalen Grafiken.

`OIS` (**Object-Oriented Input System**,
`https://github.com/miguelbernadi/Object-oriented-Input-System--OIS-`):
Eine für C++ Anwendungen geschriebene Open-Source-Programmierbibliothek zur Verarbeitung von Eingaben verschiedener Peripheriegeräte. Mögliche Eingabegeräte sind neben Maus und Tastatur auch Joysticks, Gamepads und andere Spielecontroller.

Visibility (deutsch: *Sichtbarkeit*): Konzept, welches ein geometrisch-mathematisches Verfahren verwendet, um zu einem bestimmten Zeitpunkt nur ausgewählte Bereiche hervorzuheben. Im Zusammenhang dieser Anwendung bezieht sich dies auf den Bereich, welchen die Spielfigur betreten darf.

`OpenAL` (**Open Audio Library**), Eine Open-Source-Programmbibliothek zum Erzeugen von Sounds.