UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN
ABTEILUNG INFORMATIK UND ANGEWANDTE KOGNITIONSWISSENSCHAFT

Bachelorarbeit

Erweiterung des CT Game Studios um einen „Open Stage“ Modus

Daniel Rose
Matrikelnummer: 2270435

Abteilung Informatik und angewandte Kognitionswissenschaft
Fakultät für Ingenieurwissenschaften
Universität Duisburg-Essen

12. September 2018

Betreuer:
Prof. Dr. H. U. Hoppe
Sven Manske
Sören Werneburg

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

TODO

1.1 Motivation

1.2 Aufgabenstellung

1.3 Aufbau der Arbeit

2 Grundlagen

2.1 Computational Thinking

Unter Computational Thinking (CT) wird Kognitionsprozess oder Gedankenprozess verstanden, der durch die Fähigkeit, in Form von Dekomposition, abstrahierend, evaluierend, algorithmisch und generalisierend zu denken, reflektiert wird [Selby et al., 2011]. Nach [Wing, 2011] ist Ziel des Prozesses, ein Problem so darzustellen, dass es von einem Computer gelöst werden kann. Dazu werden zur Entwicklung von Problemlösungen logische Artefakte erstellt, die auf dem Computer ausgeführt werden. Dies kann zum Beispiel Quelltext in einer Programmiersprache sein.

Diese Arbeit fokussiert sich primär auf algorithmisches Denken, Abstraktionen und Evaluation. Algorithmisches Denken beschreibt die Fähigkeit, ein Problem sequentiell abzuarbeiten. Dazu werden Kontrollstrukturen als Teil eines Abstraktionsprozesses genutzt. Um zum Beispiel eine Wiederholung eines Befehls umsetzen, könnte der Befehl mehrfach hintereinander im Quellcode stehen. Zur Komprimierung des Codes kann man die Wiederholung mit einer Schleife abstrahiert werden. Dazu muss ein zusätzliches Vokabular verwendet werden, dass dem Computer klar macht, dass dieser Befehl mehrfach wiederholt werden soll. Eine tiefere Abstraktion kann an diesem Beispiel erfolgen, wenn durch die Nutzung von Variablen innerhalb der Schleife mit jeder Wiederholung unterschiedliche Werte benutzt werden. Ein weiteres abstrahierendes Konzept besteht darin Prozeduren und Funktionen zu bilden. Dabei werden mehrere Kommandos werden unter einem Namen gebündelt und parametrisiert, und im Fall von Funktionen ein Rückgabewert generiert. Prozeduren und Funktionen können mehrmals an verschiedenen Stellen im Programm aufgerufen werden zu können, ohne die Kommandos einzeln zu wiederholen. Abstraktionen erlauben uns Konzepte auf einer Metaebene zu nutzen und diese dann umzusetzen.

Zur Entwicklung der logischen Artefakte gehört neben der Programmierung selbst auch die Evaluation des Artefakts. Dazu wird es ausgeführt und anschließend mithilfe von visuellen oder textuellen Feedback analysiert. Aufgrund des Feedbacks werden Rückschlüsse darüber gezogen, ob das zugrunde liegende Problem gelöst wurde, und wenn nicht, wie das Artefakt verändert werden muss, um das Problem adäquat zu lösen.

In einer Richtlinie zur Etablierung von Computational Thinking in der K-12-Bildung ([ISTE, CSTE, 2011]) der International Society for Technology in Education und Computer Science Teacher Association wird die Wichtigkeit von CT beschrieben: *„Because we can expect that every student will rely on computing in some way to amplify his or her skills, we must ensure that all students have the opportunity to learn the basics of CT during their K–12 education.“* .

(TODO mögliche Erweiterungen: Use-modify-create, Kreislauf nach Repenning, Überleitung zu Werkzeugen, die CT anwenden)

2.2 Turtle Geometrie

Den Computer als Werkzeug zum Lernen zu benutzen ist der Fokus von Seymour Paperts Forschung zu Logo und der Turtle-Umgebung [Papert, 1980]. Logo ist eine Programmiersprache die dazu gestaltet ist, möglichst einfach zu erlernen zu sein. Dazu besitzt sie eine minimale Syntax und ein minimales Kontingent an Kontrollstrukturen.

Papert konzipiert die Turtle-Umgebung zur Exploration mathematischer Konzepte. Sie ist ein Beispiel einer Mikrowelt, eine Anwendung in der Objekte programmatisch manipuliert werden können, um dem Lerner ein Medium zu bieten, dass dem natürlichen Erlernen eines Konzepts dient. Die Turtle ist ein Objekt, das mithilfe von einfachen Kommandos bewegt werden kann. Durch Aktivierung eines Stifts wird die Ausgabe von Linien erzeugt, die der Bewegung folgen. Die Bewegung der Turtle erfolgt mittels Logo-Programmen.

Papert argumentiert, dass eine Mikrowelt dann zum einem wertvollen Lerninstrument wird, wenn sie dem eine natürliche Interaktion erlaubt. Die Bewegung der Turtle soll dem Sprechen mit der Turtle gleichen. Die Definition von Prozeduren gleicht dem Beibringen eines neuen Vokabulars. Das Erlenen eines Konzepts wird dann durch iteratives Vorgehen ermöglicht, dass der Exploration und dem Lernen einer der menschlichen Sprache im Kindesalter gleicht. Das Objekt, dass in der Mikrowelt manipuliert wird, sollte dabei menschliche Charakteristiken besitzen. Die Turtle ist „body-syntonic“, ihre Bewegung und Orientierung lässt sich konkret auf die Bewegung und Orientierung des Menschen übertragen. Um das Verhalten der Turtle nachzuvollziehen, kann der Lerner den eigenen Körper als Referenz nutzen.

2.3 Game-based learning von Computational Thinking

Spiele werden häufig dazu genutzt, als Lernumgebungen zum Erlenen von Computational Thinking-Konzepten zu dienen (z.B. „Program your Robot“ [Kazimoglu et al., 2012], „Lightbot“ [Gouws et al., 2013] oder AgentSheets [Repenning et al., 2010]). Spiele können aufgrund der Herausforderung, die sie stellen, einem Fantasieaspekt der den Lerner in eine imaginäre Welt versetzt, dem Ausnutzen der natürlichen menschlichen Neugierde, und der Kontrolle, die der Lernen über den Spielverlauf zu geben, Mikrowelten in einer Weise bereit stellen, in der Lerner internal motiviert sind Spielziele zu verfolgen und damit Lernerfolge zu erzielen ([Rieber, 1996]).

2.3.1 Program Your Robot

„Program your Robot“ [Kazimoglu et al., 2012]

Program your Robot als Spiel das Computational Thinking skills direkt auf Spielelemente mappt.

2.3.2 RoboCode

RoboCode[1] ist ein Spiel, dass zum Erlernen der Programmiersprache Java konzipiert wurde. In einer Arena (Abb. 2.1) versuchen zwei oder mehrere Roboter sich gegenseitig zu zerstören. Dabei führen die Roboter Kampfstrategien in Form von Java-Programmen aus. RoboCode integriert einen Quellcode-Editor, in dem diese Strategien gebaut werden können. Mitgelieferte Beispielstrategien können als Orientierung dienen und als Gegnerstrategien festgelegt werden.

Um RoboCode hat sich eine Community gebildet. Im RoboCode-Wiki[2] enthält eine Dokumentation des Spiels, Anleitungen zum Erstellen eigener Strategien, und weitere Kampfstrategien zum Download. RoboRumble[3] ist eine Sammlung von dauerhaft aktualisierenden Ranglisten, in denen von Spielern hochgeladenene Kampfstrategien aufgrund von Kämpfen gegeneinander eingestuft werden. Die Strategien können heruntergeladen, eingesehen und für eigene Kämpfe benutzt werden.

(TODO mögliche Erweiterung Referenz RoboCode zum Lernen von Problem Based Learning und RoboBuilder)

2.3.3 ctGameStudio

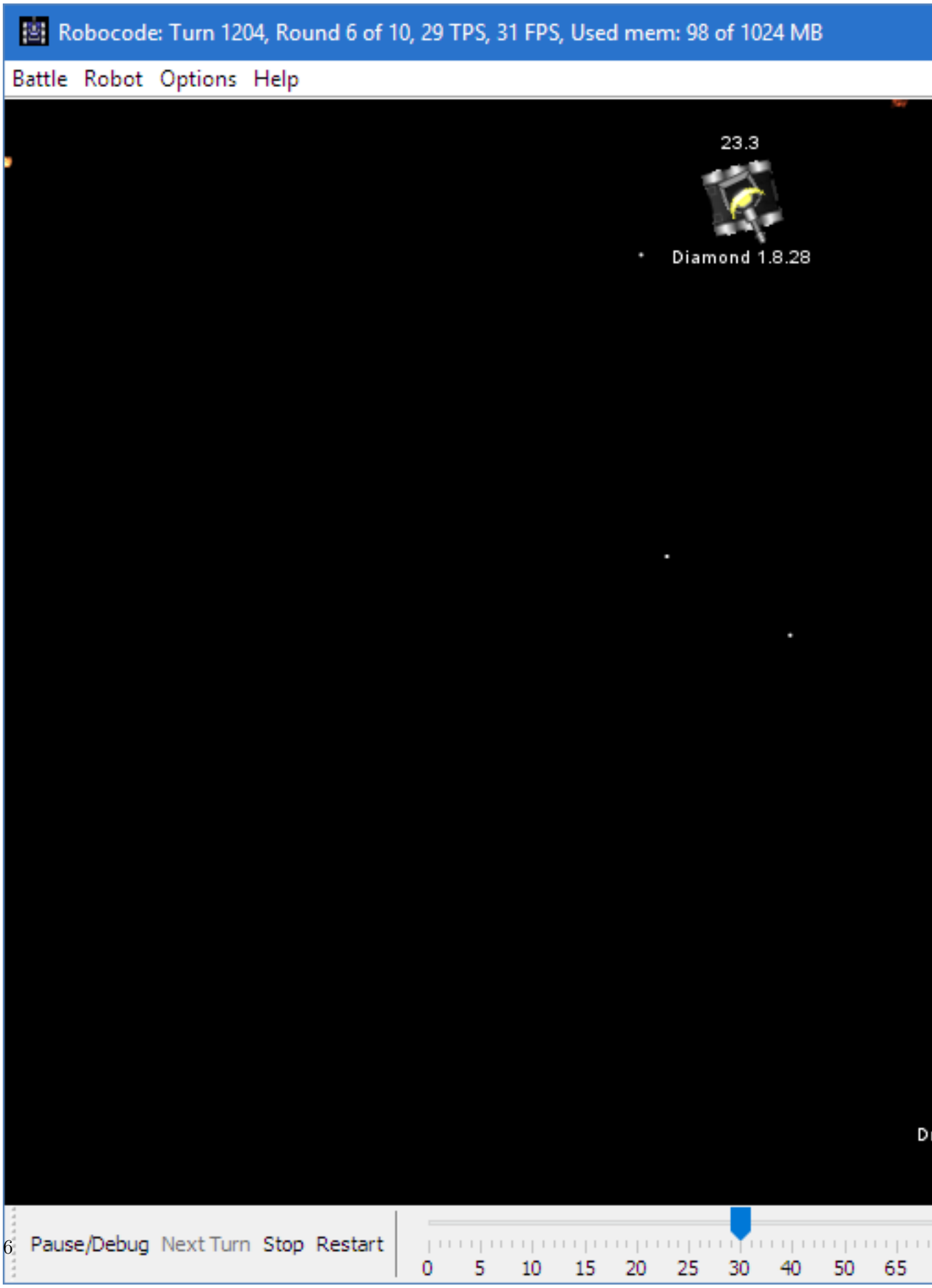
CTGameStudio ([WERNEBURG et al., 2018]) ist eine webbasierte Spielumgebung zum Erlernen von CT-Fähigkeiten und Programmierung. Mithilfe eines Editors entwickeln Schüler ein Programm zur Steuerung eines virtuellen Roboters in einer Mikrowelt.

Der Editor ist ein visuelles, blockbasiertes Programmierwerkzeug basierend auf der Blockly-Bibliothek[4]. Visuelle blockbasierte Editoren haben sich als einsteigerfreundliche Programmierwerkzeuge erwiesen ([Weintrop and Wilensky, 2015]). Programmkonstrukte werden durch Blöcke repräsentiert, die mittels Drag-and-Drop ineinander verschachtelt und kombiniert werden, um ein Programm zu bilden. Eine Blockbibliothek gibt eine eingebaute Übersicht über die verfügbaren Programmkonstrukte, so dass auf einen Blick die Möglichkeiten der Programmierung eingesehen werden können. Durch die strukturierte Komposition von Blöcken können keine Syntaxfehler auftreten, was in textbasierten Programmen eine häufige, den Spielfluss störende Fehlerquelle ist.

Zur Einführung in die Programmierung und Vermittlung von CT-Fähigkeiten, sowie zur Einführung in die Mechaniken des Spiels enthält das ctGameStudio einen Storymodus. Der Storymodus enthält mehrere Level, die nacheinander abgearbeitet werden, und jeweils auf das Erlenen einer oder mehreren CT-Fähigkeiten, einer neuen Abstraktion oder Roboterfähigkeit abzielen. Dazu enthält das Level eine Aufgabenbeschreibung, Anleitungen und optionale Hilfestellungen. Zur Lösung eines Levels baut der Schüler ein Programm welches einen Algorithmus darstellt, der den Roboter so steuert, dass die Herausforderung gelöst wrid. Diese Problemlösungstrategie kann immer wieder neu ausgeführt werden, bis das gewünschte Ziel erreicht wurde. Ist das der Fall, wird zum nächsten Level fortgeschritten. Mit steigenden Level erhöht sich die Komplexität der benötigten Lösungstrategien. So reicht im ersten Level ein einfacher Befehlsaufruf, während

[1] https://sourceforge.net/projects/robocode/
[2] http://www.robowiki.net/wiki/Robocode
[3] http://literumble.appspot.com/
[4] https://developers.google.com/blockly/

Abbildung 2.1: Ein Kampf zwischen zwei Robotern in RoboCode.



im letzen Level durch gezielte Bewegung, Zielen und Schießen ein Gegner besiegt werden, bevor dieser einem zu viel Schaden zufügt.

Angelehnt an der Turtle-Geometrie nach Papert wurde eine Bewegungssemantik für den Roboter entwickelt, und um Fähigkeiten erweitert, die für die das Spielen der Story relevant sind.

- Positionierung auf dem Spielfeld durch die Bewegung um eine Distanz oder bis zu einem Punkt auf der x- und y-Achse, der Bewegung bis die Spielfeldgrenze, ein Gegenstand oder der Gegner berührt wurde, dem Drehen um einen Winkel, und dem Drehen, um einen Punkt anzupeilen.
- Angriff durch Abgeben eines Schusses in Blickrichtung.
- Das Ausfinden machen des nächstgelegenden Objektes oder Gegners durch einen Scanvorgang in eine Richtung. Wurde ein Ergebnis gefunden, kann die Distanz und Winkel zum Ergebnis und seine Position gelesen werden.
- Feststellen von Ereignissen, wie der Fall ob man getroffen wurde, oder ob man mit der Wand, einem Gegner oder einem Objekt kollidiert ist.
- Lesen der eigenen Attribute, darunter die Position auf dem Spielfeld und die Anzahl an verbleibenden Lebenspunkten.

Nach Beenden des letzten Levels sind die Inhalte des ctGameStudio im aktuellen Stand ausgeschöpft. Dabei wäre es sinnvoll, Schülern eine Möglichkeit zu geben, ihre gelernten Fähigkeiten anzuwenden. Während Schüler im Storymodus konkrete Anweisungen zu der Lösung eines Problems bekommen, sollte das Spiel Herausforderungen enthalten, die eigenständig vom Schüler gelöst werden müssen, um besonders die Evaluationsfähigkeit stärker auszubilden. Spiele wie RoboCode zeigen uns, dass ein Spiel durch einen offenen Spielmodus einen hoher Wiederspielwert erreicht werden kann.

3 Ansatz

Im Storymodus des ctGameStudios lernen Schüler grundlegende Konzepte der Programmierung wie Schleifen, Verzweigungen, Ereignisse, Prozeduren und Funktionen kennen. Der Storymodus ist in sich geschlossen, da er einen festen Anfang und Ende hat, und im vornherein festgelegte, spezifische Herausforderungen stellt.

Wir wollen das ctGameStudio erweitern, um Spielern die Möglichkeit und die Motivation dazu zu geben, die im Storymodus gelernten Fähigkeiten anzuwenden und zu trainieren. Im Gegensatz zum Storymodus soll dieser Spielmodus offen sein, so dass der Spieler seine Herausforderungen und Lösungsansätze selbst bestimmen kann. Damit wiederholtes Spielen motiviert wird und die eigene Kreativiät gefördert wird, sollen Herausforderungen durch sich selbst oder andere generiert werden können.

Ein offener Spielmodus stellt die Fähigkeit in den Vordergrund, Problemstellungen zu analysieren, und adäquate Problemlösestrategien zu entwickeln. Dadurch werden speziell die CT-Fähigkeiten der Abstraktion und der Evaluation gefördert.

Inspiriert vom RoboCode und angelehnt an das letzte Level des Storymodus soll dieser Spielmodus aus einem Roboterkampf bestehen. Es gilt, eine Strategie zu entwickeln, um die Strategie des Gegners zu überwinden.

3.1 Ein Lernszenario

Das Spiel soll solche Schüler unterstützen, die die Grundlagen der Programmierung kennen lernen und ausbauen sollen. Der Kernlehrplan Informatik für Gymnasien und Gesamtschule in der Sekundarstufe II ([Ministerium für Schule und Weiterbildung des Landes Nordrhein-Westf
definiert als Inhaltsfeld die Entwicklung von Algorithmen, welche als "genaue Beschreibung zur Lösung eines Problems"(S. 17) definiert werden. Dies entspricht der Entwicklung von Problemlösestrategien in der Roboter-Mikrowelt des ctGameStudio, und zeigt, dass das dieses Spiel die Anwendung in der Sekundarstufe II unterstützen sollte.

Folgendes Lernszenario soll darstellen, wie das Spiel die Ziele des Lehrplans unterstützt und in den Unterricht eingebunden werden kann.

Um die Grundlagen des Spiels sowie die zur Entwicklung von Problemlösestrategien nötigen Programmierkonzepte kennen zu lernen, spielen Schüler zunächst den Storymodus des ctGameStudio. Durch Bearbeiten der Level des Storymodus werden die im Kernlehrplan definierten inhaltlichen Schwerpunkte der Analyse, Entwurf und Implementierung einfacher Algorithmen (S. 23) unterstützt, und anhand dessen die Kompetenzen des Argumentierens, der Modellierung, der Implementation, des Darstellen und Interpretieren und Kommunizieren und Kooperieren gefördert.

Mit dem offenen Spielmodus werden die erlenten Kompetenzen vertieft. In Kämpfen zwischen dem eigenen gegen einen Gegnerroboter entwickeln die Schüler eigene Kampfstrategien. Dabei können sie die Gegnerstrategie aus eigenen oder vorgefertigten Strategien festlegen, um eine generell anwendbare, gegen viele Herausforderungen effektive Strategie zu entwickeln. Nachdem die Schüler eine oder mehrere Strategien entwickelt haben, kann der Lehrer ein Turnier veranstalten, in dem die Strategien gegeneinander ausgespielt werden und eine Rangliste entsteht. Das Turnier wird an einem gemeinesamen Bildschirm oder Projektion verfolgt, und liefert eine Diskussionsgrundlage um Strategien gemeinsam zu evaluieren. In weiteren Durchgängen können Schüler ihre Strategie verbessern, und weitere Turniere veranstaltet werden.

3.2 Offener Spielmodus: RoboStrategist

Kern des offenen Spielmodus ist der Kampf zwischen zwei Robotern, die vorprogrammierte Strategien ausführen. Die Strategien sind Algorithmen zur Koordination der Fähigkeiten des Roboters, um den Gegner Schaden zuzufügen, und eigenen Schaden zu vermeiden. Eine erfolgreiche Strategie besteht aus effektiver Positionierung des Roboters auf dem Spielfeld, dem Ausweichen von Schüssen nachdem man getroffen wurde, dem Ausfinding machen und Verfolgen des Gegners, und das Zielen und Schießen auf den Gegner.

Im Training kann der Spieler eigene Strategien entwickeln. Verschiedene Strategien können im Turniermodus gegeneinander antreten.

ToDo: Unterbringen, dass die Startpostiionen von Robotern zufällig bestimmt werden.

3.2.1 Training

Der Training fördert eigenständiges, selbst-geleitetes Lösen von Problemen, und bietet damit eine Plattform, seine Programmierfähigkeiten zu vertiefen. Zum Einen werden komplexe Herausforderungen in Form von unterschiedlichen Gegnerstrategien gestellt. Der Spieler wendet Abstraktionen wie Schleifen, Verzweigungen, Variablen, Prozeduren und Funktionen an, um einen Algorithmus zu entwickeln. Der Algorithmus ist das Code-Artefakt, das gegen das Problem in Form der Gegnerstrategie evaluiert wird.

Zum Anderen kann der Spieler eigenständig wählen, wie er bei der Entwicklung der Problemlösestrategie vorgeht, in dem er selbst wählt, gegen welche Strategien er seine Strategie testet. Im Rahmen dieser Arbeit sollen dafür dafür drei vorgefertigte Gegnerstrategien verfügbar sein. Die Strategien unterscheiden sich in ihren Ansätzen und ihrer Komplexität.

Die erste Strategie, „Verwirrt“, (Abb. **??**) ist dazu gestaltet, einen einfachen Einstieg in die Strategieentwicklung zu geben. Der Roboter bewegt sich in zufälligen Zeitabständigen auf zufällig gewählte Positionen, und gibt zwischendurch Schüsse in zufällige Richtungen ab. Um den Roboter zu besiegen, muss der Spieler eine Strategie entwickeln, die den Gegner immer wieder auffindet, und Schüsse in seine Richtung feuert. Dadurch, dass kein Zielen auf den Roboter des Spielers besteht, muss die Lösungsstrategie nicht besonders effizient sein. Auch um die eigene Positionierung muss sich der Spieler keine Gedanken machen.

Die zweite Strategie, „Wandkrabbler“ (Abb. **??**), agiert in einem vorsehbaren Muster. Der Roboter bewegt sich an am Spielfeldrand entlang und scannt in kleinen Abständen voneinander entgegengesetzt der Wand nach dem Gegner. Wurde dieser entdeckt, schießt er solange, bis der Gegner seine Position ändert. Diese Strategie erfordert vom Spieler eine Strategie zu entwickeln, in der unterschiedliche Postionen eingenommen werden, um die Entdeckung durch den Gegner zu verzögern und im Fall eines Treffers weiteren Schüssen auszweichen. Die erhöhte Gefahr durch den Wandkrabbler erfordert auch einen effizienten Scanvorgang, so dass man den Gegner öfter und schneller auffindet und Schaden zufügt, als er es tut.

Die dritte Strategie, „Eckenschütze“ (Abb. **??**), ist die gefährlichste aller Strategien. Zunächst hat sie den effizientesten Scanmechanismus. Dazu bewegt der Roboter sich zwischen den Ecken des Spielfelds, und scannt von den Ecken aus in kleinen Schritten das gesamte Spielfeld ab. So ist die Chance groß, innerhalb eines kurzen Zeitraums den Gegner aufzufinden. Wurde der Gegner entdeckt, werden so lange Schüsse abgegeben, bis der Gegner seine Position ändert. Die Strategie implementiert zudem einen Ausweichmechanismus, so dass der Roboter in die nächste Ecke wechselt, wenn er getroffen wurde. Um diese Strategie zu besiegen erfordert ständige Bewegung auf dem Spielfeld, um den Scanvorgängen zu entgehen. Wahrscheinlich sollte sie ein eigenen Ausweichmechanimus implementieren, und einen effiziente Scanvorgang beinhalten.

Wie vorher gezeigt (? Bezug auf Papert), sollte eine Mikrowelt iteratives, selbst gesteuertes Vorgehen erlauben, um die inherente Kreativität und Explorationswillen des Menschen zu nutzen. In diesem Sinne wollen wir dem Spieler die Möglichkeit geben, als Gegnerstrategien neben den vorgefertigten auch selbst erstellte Strategien zu wählen. Um seine Strategie zu verbessern, könnte der Spieler durch Analyse Schwachpunkte seiner Strategie feststellen, und versuchen diese mit einer neuen Strategie konkret auszunutzen.

Das eigenständige, selbst-geleitete Lösen von Problemen soll einen kreativen Lösungsprozess des Spielers fordern, ein exploratives Vorgehen motivieren, und damit verglichem mit dem geleiteten Prozess des Storymodus einen höheren Lerneffekt erzielen.

Folgendes Schema stellt eine neue Strategieentwicklung dar und zeigt, dass die Entwicklung einer Strategie im Training den drei Stufen des Compuational Thinking Prozesses **??**.

1. Die initial geladenene Gegnerstrategie wird ausgeführt.
2. Man analysiert die Gegnerstrategie und bildet erste Lösungsansätze (Problem Formulation).
3. Man formuliert und implementiert erste Lösungsansätze (Solution Expression). Man kann auf das Wissen aus den letzten Leveln des Storymodus zurückgreifen, um einen ersten Anhaltspunkt dafür zu haben.
4. Man führt den Kampf aus und analysiert seinen Verlauf (Solution Execution & Evaluation). Die Evaluationskriterien sind, ob der Roboter das macht, was man mit dem Programm erzielen wollte, und ob der Problemansatz zum Erfolg führt.
5. Aufgrund der Analyse verbessert der Spieler seine Strategie.

6. Schritt 4 und 5 werden wiederholt, bis der Gegner wiederholt geschlagen werden kann.
7. Der Spieler wählt eine neue Gegnerstrategie.
8. Schritt 4 bis 7 werden wiederholt, bis der Spieler alle Gegnerstrategien hat. Der Spieler kann außerdem versuchen, seine eigene Strategie zu besiegen, in dem er sie als Gegnerstrategie festlegt.

Um verschiedene Strategieansätze ausprobieren zu können, und in mehreren Sitzungen an den Strategien arbeiten zu können, kann der Spieler seine Strategien speichern, kopieren, neue anlegen, und zwischen diesen wechseln.

3.2.2 Turniere

Turniere sind spannend und erhöhen Spielspaß und damit die Motivation, das Spiel zu spielen. Der kompetitive Aspekt wird Spieler dazu motivieren, gute Strategien zu entwickeln. Die Turnierdurchführung selber bietet einen Vergleich zwischen Strategien und ist so Diskussionsgrundlage für verschiedene Strategieansätze und Verbesserungen.

Da es verschiedene Möglichkeiten gibt, ein Turnier durchzuführen, und diese ihre eigenen Vor- und Nachteile haben, wollen wir dem Turnierveranstalter die Wahl eines Turniersystems geben. In dieser ersten Version des Open Stage-Modus wollen wir die zwei gebräuchlisten Turniersysteme zur Wahl stellen.

Das Jeder-gegen-Jeden-System ist die ausführlichste Weise, ein Turnier durchzuführen. Ein Beispiel für diese Turnierart ist die deutsche Fußballbundesliga. Um in diesem System eine Rangfolge zwischen Teilnehmern zu erzielen, tritt jeder Teilnehmer gegen jeden anderen Teilnehmer an und zählt die Siege, die erreicht wurden. Vorteil ist, dass diese Rangfolge erschöpfend ist. Nachteil dieses Modus ist die quadratisch steigende Anzahl von Kämpfen (n x n Kämpfe bei n = Anzahl der Teilnehmer), die durchgeführt werden müssen. Für die Evaluation von Strategien im Open Stage-Modus bietet sich das Jeder-gegen-Jeden-System an, wenn eine geringe Anzahl von Teilnehmern besteht.

Das KO-System ist eine effizientere Weise, ein Turnier durchzuführen. Ein Beispiel für diese Turnierart ist die KO-Phase eines Fußball-Weltmeisterschaft. Das Turnier verläuft in Runden, wobei in der ersten Runde zufällig ausgewählte Pärchen gegeneinander antreten. Während der Verlierer eines Kampfes vom Turnier ausscheidet, geht der Sieger in die nächste Runde. Vorteil dieses Turnieres ist, dass bei n Teilnehmern lediglich n - 1 Kämpfe ausgeführt werden. Außerdem hat dieser Modus einen interessanteren Spannungsbogen, da bei jedem Kampf ein Spieler ausscheidet, und nach jeder Runde die Anzahl der Spieler durch zwei geteilt wird, bis es am Ende ein spannendes Finale gibt. Nachteil dieses Systems ist, dass die aus dem Turnier resultierende Rangfolge nicht zwangsläufig repräsentativ für die tatsächliche Rangfolge der Strategien ist. So mag es beispielsweise sein, dass die Gewinnerstrategie bei einem Turnier mit acht Teilnehmern eigentlich gegen vier der sieben Gegner verlieren würde, durch die zufällige initale Paarung jedoch auf die drei übrigen Strategien getroffen ist. Ein weiterer Nachteil ist, dass ein faires Turnier im KO-Modus mit mehr als zwei Teilnehmern eine Teilnehmeranzahl haben muss, die durch vier teilbar ist. Da dies nicht immer möglich ist, wollen wir dem Veranstalter die Möglichkeit geben, fehlende Teilnahmen durch selbst festgelegte Strategien aufzufüllen. Für die

Evaluation von Strategien im Open Stage-Modus bietet sich bei einer hohen erwarteten Anzahl von Teilnehmern an, und der Unterhaltungswert des Turniers eine große Rolle spielt.

Wenn Strategien ineffektiv sind, kann es sein, das in einem Kampf innerhalb eines vertretbaren Zeitraums kein Sieger festegestellt werden kann. Deshalb soll bei Erstellung des Turniers eine maximale Rundendauer festgelegt werden können. Im Jeder-gegen-Jeden-Modus geht der Kampf nach Ablauf der Zeit unentschieden aus. Im KO-Modus muss ein Sieger festgestellt werden. Dauzu gewinnt in diesem Modus nach Ablauf dieser Zeit der Spieler, dessen Roboter weniger Schaden genommen hat. Sind hier beide Spieler gleich, kann der Turnierleiter selbst einen Sieger festlegen.

Da bereits zu vor Ablauf der Zeit absehbar sein kann, dass es zu keinem Sieger kommen wird, soll es außerdem es die Möglichkeit geben, eine Runde abzubrechen. Der Turnierleiter kann den Kampf wiederholen, oder das Ergebnis selbst bestimmen.

Das Resultat mehrerer Kämpfe mit den selben Strategien kann aufgrund der zufälligen Startposition der Roboter unterschiedlich sein. Um die tatsächliche bessere Strategie zu ermitteln, könnte es nötig sein, einen Kampf aus mehreren Runden bestehen zu lassen. Ein Spieler würde dann den Kampf gewinnen, der zuerst zwei oder drei Runden gewonnen hat. Auch hierfür soll es bei Erstellung des Turnieres eine Option geben.

Die Turnierfunktion ist so gestaltet, dass die Teilnehmer nicht bei Erstellung des Turniers bekannt sein müssen. Dazu wird bei Erstellung einen Zugangscode generiert. Spieler, die diesen Code kennen, können sich mit einer ihrer im Training erstellten Strategien am Turnier anmelden.

In einer Turnierliste kann der Turniererstellter die Namen und Strategienamen der Teilnehmer einsehen, und das Turnier starten. Um die Durchführung des Turniers zu verfolgen muss der Bildschirm des Turniererstellers betrachtet werden.

Eine Turnierübersicht zeigt den Verlauf und Ranglisten des Turniers. Die Ansichten unterscheiden sich dabei je nach Turniersystem. Im Jeder-gegen-Jeden-Modus werden vergangene Kämpfe, noch anstehende Kämpfe, und eine Tabelle mit der Anzahl von Siegen pro Spieler angezeigt. Im KO-Modus wird der Turnierbaum angezeigt. Der Turniererstellter kann den jeweils nächsten Kampf über einen Button starten. Das Spielfeld wird geladen und der Kampf ausgeführt. Nach Ende des Kampfes wird die aktualisierte Turnierübersicht dargestellt.

3.3 Anforderungsanalyse

Die Mikrowelt des Storymodus des ctGameStudio bietet die Grundlage für den - Training - Gegnerstrategien auswählen (vorgegebene und eigene) - Strategiemanagement: Speichern, Laden, Bearbeiten - Turniere - Verschiedene Modi (warum?) - Verschiedene Optionen - Teilnahme - Gameplay - Ziel: Bessere/überlegtere Strategien sollen sich auch im Matchergebnis niederschlagen, ein intelligenter Einsatz der Programmierfähigkeiten führt zu größerem Erfolg führt - Verschiedene Ansätze sollen zum Erfolg führen können, z.B. effizientes Ausweichen/Positionieren, oder besonders raffiniertes Verfolgen - Vereinfachte Evaluation durch langsames Tempo - Design

3.4 Rundgang durch den Open Stage-Modus

4 Implementierung

Die Architektur des CTGameStudio in der Erstversion musste angepasst werden, um den Anforderungen des neuen Spielmodus zu erfüllen. Das Hinzufügen eines zweiten Roboters erfordert, mehrere Instanzen der blockly-Komponente und des Strategie-Interpreters zu erzeugen. Die gleichzeitige Ausführung mehrerer Strategien musste ermöglicht werden. Die Gameloop und Roboterspielobjekte mussten so angepasst werden, dass die Roboter sinnvoll miteinander interagieren und sich so verhalten, dass ein gut nachverfolgbares Gameplay möglich ist und unterschiedliche Strategieansätze zum Erfolg führen. Das Spiel musste um Dialoge mit Formularelementen und Turniervisualisationen erweitert werden. Zur Persistenz und Interaktion zwischen verschiednen Benutzern des Systems muss das System das Speichern und Laden von Daten über HTTP-Schnittstellen des Servers unterstützen.

4.1 Architektur der Erstversion

Das ctGameStudio ist ein Webanwendung. Der Server liefert Seiten an den Browser aus und bietet Login-Management. Unterstützt wird dies durch das auf Node.JS und MongoDB basiertem KeystoneJS-Framework [1]. Die Darstellung und das dynamische Verhalten des Spiels basiert auf browser-seitig ausgeführtem HTML und CSS, und Javascript (Abb. 4.1). Neben Standard-HTML-Inputs wird Phaser [2] genutzt, um mittels WebGL Spielelemente zu rendern.

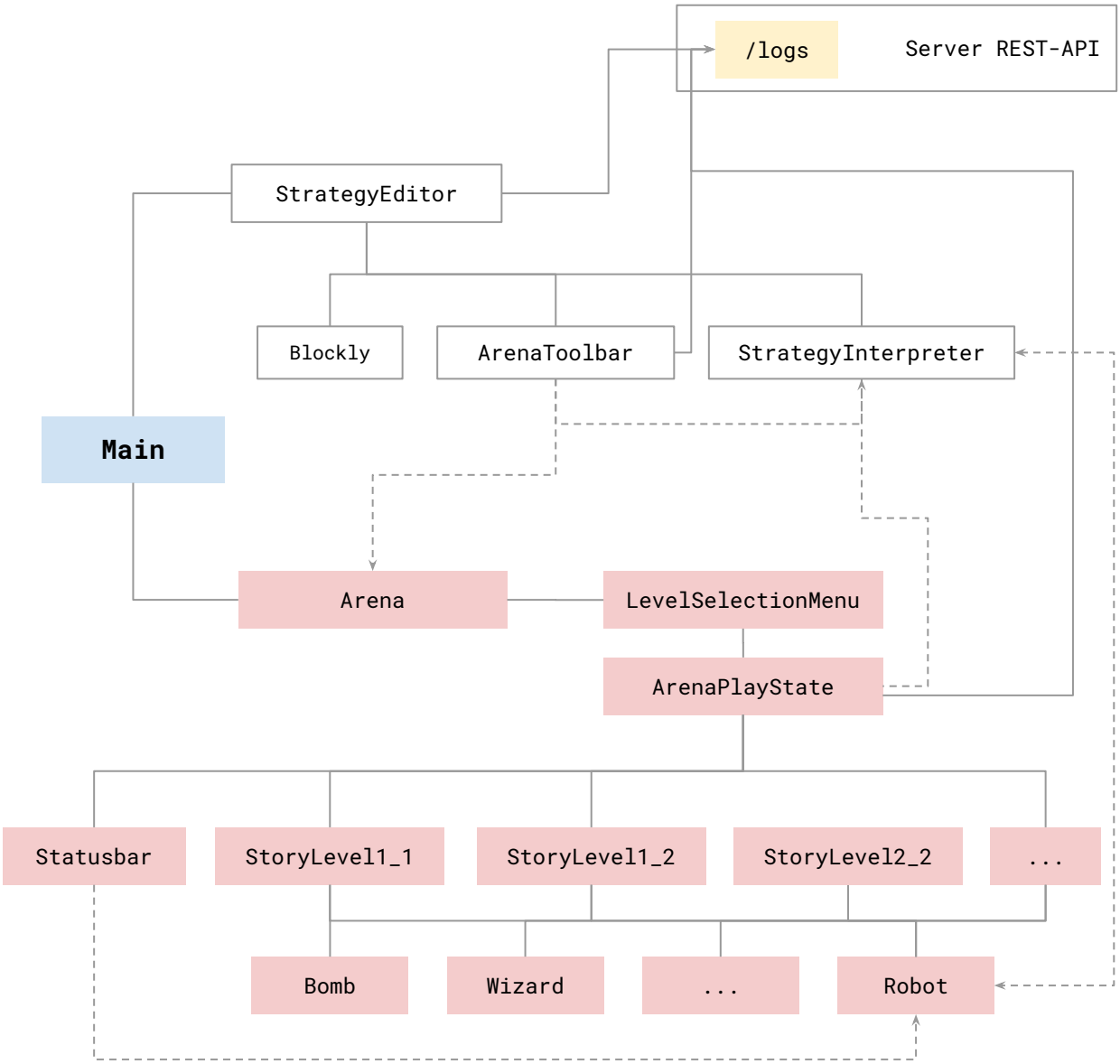
Komponenten sind als als Javascript-Funktionen, Objekte oder Klassen definiert. Während eine Strukturierung dadurch erfolgt, dass Komponenten auf mehrere Dateien aufgeteilt sind, liegen die Komponenten, Instanzen der Komponenten als auch interne Zustandsvariablen in einem globalen Namensraum. Dadurch ist direkter Zugriff von eigentlich unabhängigen Komponenten aufeinander möglich und führt zu einer hohen Kopplung zwischen Komponenten.

4.1.1 Main

Die Hauptfunktion wird nach Laden der Webseite aufgerufen und inialisiert das Phaser-Spiel, welches das Spielmenü und die Roboter-Umgebung enthält, sowie den Strategieeditor.

[1] https://keystonejs.com
[2] https://phaser.io

Abbildung 4.1: Architektur des ctGameStudio in der Erstversion. Phaser-Spielobjekte sind rot hervorgehoben. Durchgezogene Linien zeigen Beziehungen zwischen Komponenten und Subkomponenten, wobei die Komponente die Subkomponente erstellt, besitzt, oder einbindet. Gestrichelte Linien zwischen Komponenten zeigen Abhängigkeiten, so dass eine Komponente die andere referenziert und den internenen Zustand der Komponente verändert.



4.1.2 Strategieeditor

Der Strategieeditor enthält den Blockly-Oberfläche zur Erstellung eines Programms, den Strategieinterpreter der für die Ausführung der Strategie zuständig ist, und eine Toolbar mit Steuerungselemente zur Ausführung der Strategien und Aufruf des Blocklexikons.

4.1.3 Blockly

Die Blockly[3]-Bibliothek stellt einen Editor zur blockbasierten Programmierung bereit. Zur Konfiguration des Editors werden die Blöcke definiert, aus denen der Anwender sein Programm zusammenstellen kann. Die Konfiguration besteht aus einer Beschreibung der Blöcke sowie aus Funktionen, die beschreiben, welcher Code aus einem Block generiert werden soll (Abb. 4.2). Die Blockly-Developer-Tools[4] können genutzt werden, um die Block-Konfiguration zu generieren und zu verwalten.

Für das ctGameStudio wurden Blöcke definiert, die Aktionen des Roboters entsprechen. Der aus einem Block generierte Code ist ein Funktionsaufruf, der im Aufruf einer Methode des Roboter-Spielobjekts resultiert (z.B. Abb. 4.2, Zeile 18).

Die Programme, die mit dem Editor erstellt wurden, können als XML exportiert und importiert werden. Auch die Leiste mit verfügbaren Blöcken wird über eine XML-Struktur konfiguriert. Dies wird im ctGameStudio genutzt, um beim Laden eines Storylevels den Editor an die Anforderungen des Levels anzupassen. So werden nur die Blöcke verfügbar gemacht, die in dem Level benutzt werden sollen, und ein leeres oder teils vordefiniertes Programm in den Editor geladen.

Zur Ausführung der Roboterstrategie wird eine Javascript-Version des Blockprogramms generiert (Abb. 4.3), und mit dem Strategieinterpreter ausgeführt.

4.1.4 Strategieinterpreter

Der Strategieinterpreter ist dazu da, den aus Blockly generierten Javascript-Code auszuführen, und so den Roboter auf dem Spielfeld zu steuern. Er basiert auf dem JS-Interpreter[5] von Neil Fraser, der eine Sandbox-Umgebung zur sicheren Ausführung von Javscript bereit stellt. Das Sandboxing garantiert, dass der Code isoliert von der Host-Umgebung, also der Javascript-Umgebung in der das Spiel läuft, ausgeführt werden kann. Der durch den Anwender erstellten Code kann durch seine Ausführung keine Crashes oder Endlosschleifen verursachen. Der Code bekommt keinen Zugang auf das DOM. Außerdem wird verhindert, dass der Code übermäßig Speicher belegt.

Neben nativer Javascript-Funktionalität (z.B. Rechenoperationen, Schleifen, und dem Definieren und Ausführung von Funktionen) gibt der Interpreter die Möglichkeit, Funktionen zu definieren, die aus der Sandbox heraus aufgerufen werden können. Im ctGameStudio wird dies genutzt, um Befehle zur Steuerung des Roboters bereit zu stellen. Dazu wird für jede Fähigkeit des Roboters eine Funktion definiert, die eine zugehörige Methode auf dem Roboter-Spielobjekt ausführt. (Abb. 4.4).

[3] https://developers.google.com/blockly/
[4] https://blockly-demo.appspot.com/static/demos/blockfactory/index.html
[5] https://neil.fraser.name/software/JS-Interpreter/docs.html

Abbildung 4.2: Konfiguration eines Blocks

```
// Beschreibung des Blocks
Blockly.Blocks.turn = {
  init() {
    this
      .appendField('drehen nach')
      .appendField(new Blockly.FieldDropdown([
        ['rechts', 1], ['links', -1]
      ]), 'direction')
      .appendField('um')
      .appendField('rechts')
      .appendValueInput('angle')
    this.setTooltip(
     `Roboter dreht sich um einen bestimmten Winkel`
     `in angegebener Richtung`
    );
  }
}

// Generierung des Codes
Blockly.JavaScript.turn = function (block) {
  const direction = block.getFieldValue('direction');
  const angle = Blockly.JavaScript.valueToCode(
    block, 'angle', Blockly.JavaScript.ORDER_ATOMIC
  );
  return `turn(${direction},${angle});\n`;
};

// jeder aus einem Block resultierende Code wird mit diesem
// Funktionsaufruf kombiniert, um bei Ausführung des Codes
// den aktuell ausgeführten Block im Editor hervorheben zu können
Blockly.JavaScript.STATEMENT_PREFIX = 'highlightBlock(%1);\n';
```

Abbildung 4.3: Eine Strategie als Blockly-Programm und der resultierende Code. (TODO Bild des Programms einfügen)

```
while (true) {
  highlightBlock('x12%AV$');
  turn(15, -1);
  highlightBlock('ba+AV5i');
  forward(100);
}
```

Abbildung 4.4: Initalisierung des JS-Interpreters mit Strategie-Code, und der API zur Steuerung des Roboters.

```
const strategyCode = Blockly.JavaScript.workspaceToCode(workspace);
const interpreter = new JSInterpreter(
  strategyCode,
  function (interpreter, scope) {
    const turn = interpreter.createNativeFunction(
      (direction, angle) => robot.turn(direction, angle)
    );
    const forward = interpreter.createNativeFunction(
      (distance) => robot.foward(distance);
    )
    ...
    interpreter.setProperty(scope, 'turn', turn);
    interpreter.setProperty(scope, 'forward', forward);
    ...
  }
);
```

Bei Ausführung der Strategie wird der Strategiecode in den Interpreter geladen. Dieser stellt dann eine Funktion bereit, mit der Code schrittweise ausgeführt werden kann. Eine eigens definierte Schritt-Funktion ist dafür da, den Code so lange auszuführen, bis jeder Befehl abgearbeitet wurde, bis der Roboter zerstört wurde, oder bis der Roboter als „busy“ bzw. „nicht idle“ markiert wurde. Letzteres ist immer dann der Fall, wenn der Roboter einen Befehl ausführt. Die Die wiederholte Ausführung wird durch einen rekursiven Aufruf der Schritt-Funktion erreicht. Wenn der Interpreter viele Befehle ohne Unterbrechung abarbeitet, kann dies einen Stack Overflow hervorrufen, was Fehlfunktionen der Strategieausführung nach sich zieht.

Die Schritt-Funktion wird initial beim Start der Ausführung durch den Nutzer aufgerufen. Daraufhin wird die Funktion immer nach Abschluss einer Aktion des Roboters oder wenn durch ein Spielereignis (z.B. Kollision mit dem Rand der Spielwelt) die gerade ausgeführte Aktion des Roboters unterbrochen wurde.

4.1.5 Arena

Die Levelauswahl und die Level selber sind in Phaser implementiert. Phaser bietet verschiedene Primitive um ein hoch-interaktives und mit 60fps aktualisierendes Programm mittels WebGL auf einem Canvas-Element abzubilden. So gibt es z.B. Klassen für Rechtecke, Kreise und Linien, Sprites, Textboxen, Buttons, Shader-Effekte, etc, sowie die Fähigkeit, Objekte zu animieren, und Kollisionen von Objekte festzustellen.

Die Arena ist ein Instanz der Game-Klasse, welche das Spiel initialisiert. Alle Spielobjekte werden dieser Instanz hinzugefügt, um von Phaser verwaltet und gerendert zu werden. Um die Darstellung eines Objekts zu verändern, können Methoden und Attribute der

Abbildung 4.5: Vereinfachter Auszug aus der Schritt-Funktion zum Ausführen des Strategie-Codes, sowie beispielhafte Ausschnitte aus der Roboter-Klasse und dem Gameloop, die Aufrufe der Schritt-Funktion zeigen. Durch den Aufruf von interpreter.step() wird eine Methode des Roboters aufgerufen. Dieser setzt isIdle auf false, wodurch die Ausführung der Strategie unterbrochen wird. Am Ende der Aktion wird isIdle wieder auf true gesetzt, und die Strategieausführung fortgesetzt.

```
// StrategyInterpreter
function step() {
  const hasMoreCode = interpreter.step();
  if (hasMoreCode && robot.alive && robot.isIdle) {
    step();
  }
}

// Robot
Robot.protoype.turn = function (angle, direction) {
  this.isIdle = false;
  const tween = game.add.tween(this)
    .to({ angle: this.angle + direction * angle }, TURN_TIME)
    .start();
  tween.onComplete(() => {
    this.isIdle = true;
    step();
  });
}

// Gameloop, wird von Phaser kontinuierlich aufgerufen
function update() {
  ...
  if (robotHitBounds) {
    robot.stop();
    step();
  }
  ...
}
```

Spielobjekte verändert werden. Phaser rendert die Spielobjekte kontinuierlich in einer festen Framerate.

Ein Phaser-Spiel wird durch sogenannte „States“ strukturiert. Sie stellen unabhängig voneinander dargestellte Sichten dar. Jede Sicht definiert Funktionen, die von Phaser aufgerufen werden. In der preload-Funktion werden die Assets, also z.B. Sounds oder Bilder geladen, die in dem State benutzt werden sollen. Wurde der Ladevorgang abgeschlossen, wird die create-Funktion aufgerufen. Hier werden dem Phaser-Spiel die Spielobjekte hinzugefügt, die bei der ersten Darstellung des States verfügbar sein sollen. Nach dem initialen Rendering wird kontinuierlich die Update-Funktion aufgerufen. Der Programmierer implementiert hier die Logik, die das Verhalten der Spielobjekte aufgrund ihrer Interaktionen miteinander, aufgrund der Zeit die vergangen ist, oder aufgrund der Eingaben durch den Nutzer, steuert.

Das ctGameStudio hat mehrere States zur Darstellung der Menüs, darunter die Level- und Charakterauswahl. Sie bestehen lediglich aus der Darstellung des Menühintergrunds und einigen Buttons, und sind bis auf das Abarbeiten von Button-Klicks komplett statisch.

Im ArenaPlayState wird das ausgewählte Level gerendert. Dazu gibt es für jedes Level eine Create- und Update-Funktion, die Level-spezifische Spielobjekte generiert und aktualisiert, und aus der Create- bzw. Update-Funktion des PlayState heraus aufgerufen werden. Zusätzlich werden in der Update-Funktion des PlayStates die Kollisionen und Interaktionen abgehandelt, die unabhängig vom ausgewählten Level gelten. Darunter fällt beispielsweise die Kollision des Roboters mit den Weltgrenzen, die Kollision zwischen dem Mentor und und dem Roboter, oder die Kollision der Schüsse zwischen Gegner und Roboter. Am Ende jedes Update-Durchlaufs wird über eine Methode-des Levels getestet, ob das Level erfolgreich beendet wurde. Ist dies der Fall, wird der PlayState mit dem nächsten Level neu gestartet.

Die Statusleiste enthält die Lebensanzeige des Roboters und den Pausebutton und das Pausemenü, und ist ebenfalls als Objekt mit eigener Create- und Update-Funktion implementiert, die vom PlayState aufgerufen werden.

Zentral für das ctGameStudio ist das Roboter-Spielobjekt, repräsentiert als eine eigene Klasse mit Create- und Update-Funktion, die vom GameState aufgerufen werden. Die Update-Funktion beschäftigt sich mit der Bewegung des Roboters. Die Methoden der Roboter-Klasse repräsentieren die Aktionen, die der Nutzer durch das Erstellen der Strategie ausführen kann. Wie im vorigen Abschnitt beschrieben, werden diese vom Strategie-Interpreter aufgerufen.

4.2 Anpassung der Architektur für den Open Stage-Modus

Zur Implementation des Open Stage-Modus mussten einige Anpassungen an der Architektur vorgenommen werden (Abb. 4.2). Um die Erstellung und Gestaltung der erweiterten Statusleiste, Toolbar und besonders der vielzäligen neuen Menüs zur Auswahl des Spielmodus und der Erstellung von, Verwaltung von, Partizipation an und Ausführung von Turnieren zu vereinfachen, wurden diese in HTML anstatt in Phaser implementiert. Um die stark gesteigerte Komplexität des Programms im Zuge der gesteigerten Anzahl

der Dialoge, des neuen Spielmodus, des erweiterten Strategieeditors, der erweiterten Statusleiste und dem dynamischen Laden und Speichern von Daten über den Server auf wartbare Weise zu unterstützen, wurde ein Zustandsmanagement sowie ein System zur effizienten Darstellung von HTML-Komponenten eingeführt. Bestehende Komponenten wurden umstrukturiert und voneinander entkoppelt, um die Implementation neuer Features zu vereinfachen.

Einzelne Komponenten wurden in native Javascript-Module übersetzt. Nun werden Abhängigkeiten zwischen Modulen über import-Statements explizit ausgedrückt. Anstatt alles auf einen globalen Namespace zu legen, definieren Module über export-Statements explizit, welche Variablen, Funktionen oder Klassen von anderen Modulen importiert werden können. Dies erhöht die Sichtbarkeit von Abhängkeiten und damit die Wartbarkeit des Programms beträchtlich. Es verhindert zudem, dass interne Variablen einer Komponente von anderen Komponenten benutzt oder verändert werden.

Das ctGameStudio mit Open Stage-Modus braucht die Möglichkeit, aufgrund des Programmzustands HTML-Komponenten dynamisch darzustellen. Mithilfe der hyperHTML[6]-Bibliothek wurden Komponenten konzipiert, die ineinander verschachtelt werden können, deren HTML und Event-Listener deklarativ ausgedrückt werden, und effizient gerendert und aktualisiert werden können.

4.2.1 Main

Der Hauptfunktion kommt nun eine größere Rolle zu. Während in der Erstversion lediglich die Initialisiation des Strategieeditors und der Arena stattfand, und die Komponenten direkt aufeinander zugriffen, findet in diesem Top-Level-Modul nun die Koordination aller Komponenten statt. Kern dieser Koordination ein Modell des Programmzustands.

Als Programmzustand werden hier die Parameter bezeichnet, die bestimmen, welche Komponenten gerendert werden und welche Daten diese darstellen. In der App wird er durch ein unveränderliches Javascript-Objekt modelliert (Abb. 4.7 zeigt einen beispielhaften Ausschnitt aus dem Zustandsobjekt, das Zustande kommt, wenn der Nutzer zu der Liste von seinen Turnieren navigiert). Das Zustandsobjekt ist unveränderlich („Immutable“), so dass sein Inhalt nie direkt im Speicher geändert werden kann. Soll der Zustand geändert werden, muss eine Kopie dessen angelegt werden. Dieser Schutz sorgt dafür, dass keine Komponente, die eine Referenz auf den Zustand hat, diesen direkt für andere Komponenten ändern kann.

Änderungen des Zustands finden ausschließlich im Main-Modul statt, und müssen von Komponenten explizit über das versenden von „Actions“ herbeigeführt werden (Abb. 4.8). Die Update-Funktion (Abb. 4.9) ist dafür zuständig, aufgrund einer Action einen neuen Zustand zu bauen. Auf die Aktualisierung des Zustands folgt die Darstellung des neuen Zustands mithilfe der Render-Funktion. Sie erhält den Programmzustand und gibt einr Struktur aus hyperHTML-Elementen zurück. Mithilfe von hyperHTML wird diese in das DOM eingefügt. Bei wiederholter Darstellung der Struktur aufgrund von Veränderungen des Zustands nimmt die Bibliothek nur die minimalen Änderungen am DOM vor, die nötig sind, um die veränderte Struktur darzustellen. So ist es möglich, die Render-Funktion bei jeder Zustandsaktualisierung komplett auszuführen, ohne das

[6] https://viperhtml.js.org/

Abbildung 4.6: Die Architektur des ctGameStudio nach Einführung des Open Stage-Modus. Phaser-Spielobjekte sind rot und HTML-Komponenten grün hervorgehoben. Durchgezogene Linien zeigen Beziehungen zwischen Komponenten und Subkomponenten, wobei die Komponente die Subkomponente erstellt, besitzt, oder einbindet. Gestrichelte Linien zwischen Komponenten zeigen Abhängigkeiten, so dass eine Komponente die andere referenziert und Funktionen der Komponente aufruft.

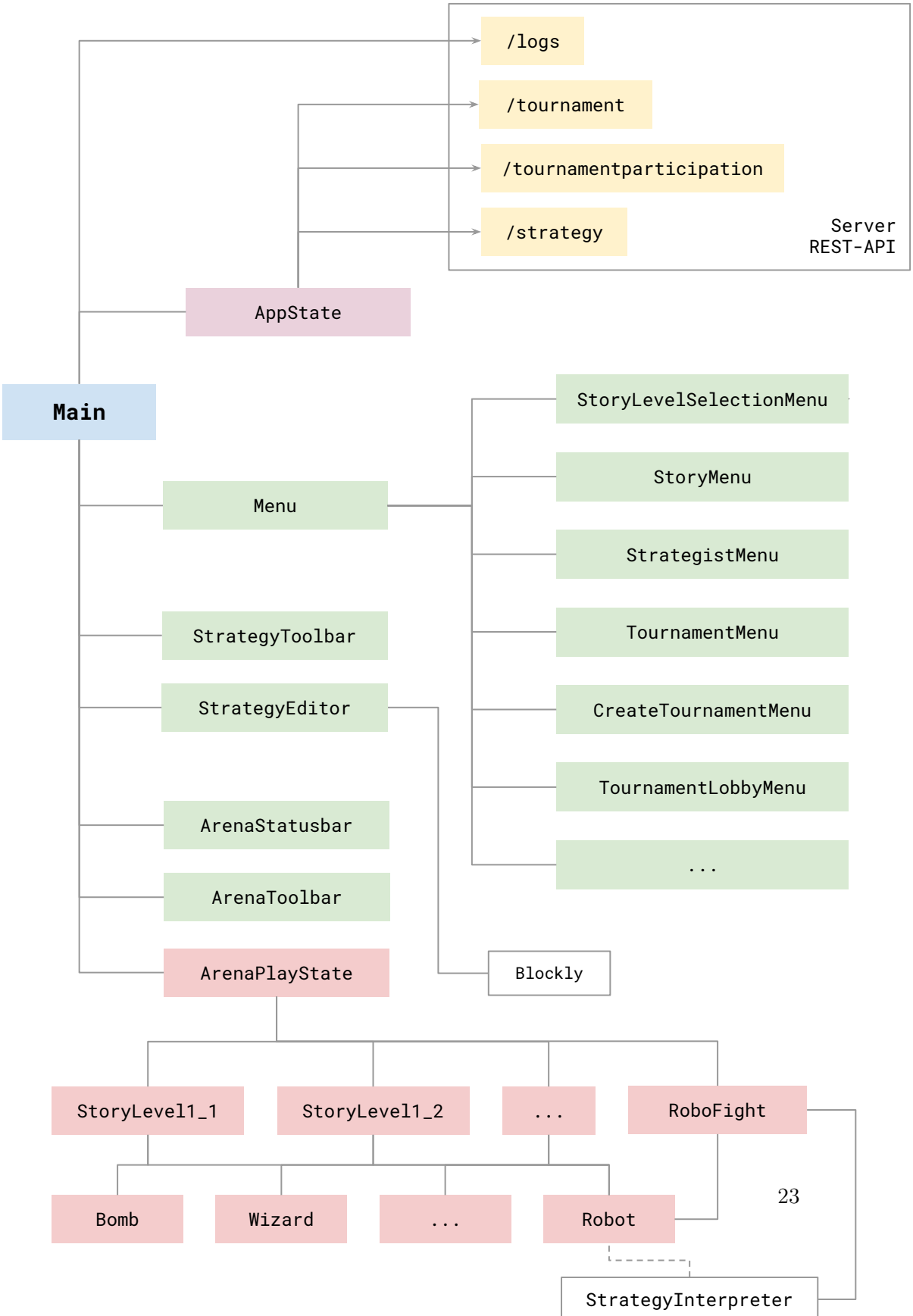


Abbildung 4.7: Ausschnitt aus dem Zustandsmodell.

```
{
  view: VIEW_MENU,
  currentMenu: MENU_TOURNAMENT_LIST,
  tournamentsList: {
    isLoading: false,
    success: true,
    tournaments: [
      {
        "_id":"xxxxx",
        "accessCode": "1OVkM",
        "createdBy": "5acf147740f1e526a82c21ea",
        "publishedDate": "2018-07-29T18:55:16.995Z",
        "maxMatchDuration": 120,
        "maxRoundCount": 1,
        "mode":"TOURNAMENT_MODE_ROUND_ROBIN",
      },
      ...
    ]
  },
  ...
}
```

DOM dabei komplett neu zu konstruieren und durch den Browser darzustellen. Dadurch wird vermieden, in der Darstellungsebene komplizierte Logik zu brauchen, die bestimmt, wann und wie das DOM geändert wird.

Dieses System ist dazu konzipiert, keine unvorgesehenen oder schlecht nachvollziehbaren Änderungen des Programmzustands zu haben. Durch die explizite Änderung durch Actions ist immer sichtbar, wann und wie sich der Zustand verändert hat. Die Business- und Darstellungs-Logik ist in zustandslosen und damit einfach zu testenden Funktionen hinterlegt (Update- und Render-Funktion). Bei Programmfehlern ist leicht nachvollziehbar, ob der Fehler in der Darstellung liegt, oder ob ein unkorrekter Zustand gebildet wurde. Durch die explizite Modellierung des Zustands, der Actions und der Aktualisierung des Zustands in Form der Update-Funktion ist übersichtlich zu sehen, welche Formen der Zustand annehmen kann.

Das beschriebene Zustandsmanagement lässt sich nicht auf Phaser übertragen. Wie in Abschnitt 4.1.5 beschrieben, arbeitet Phaser mit regulären, veränderbaren Javascript-Objekten und einem eigenen Update/Render-Loop. Daher wird die Arena-Komponente nicht bei jeder Aktion neu dargestellt, sondern bei Start des Trainings oder Start eines Turnierkampfes einmalig initialisiert. Zustand, der sowohl die Arena als auch den Rest des Spiels betrifft, ist doppelt im System repräsentiert; einmal im Programmzustand und einmal in den Spielobjekten der Arena. Über ein Event-Interface der Arena-Komponente werden die beiden Zustände synchronisiert. Ein Beispiel soll anhand der Lebenspunktzahl der Roboter gegeben werden. Einerseits wird sie im Roboter-Spielobjekt des Roboters gespeichert und bei Treffern verändert. Andererseits wird auch im Programmzustand dieser

Abbildung 4.8: Darstellung und Aktualisierung des Spiels.

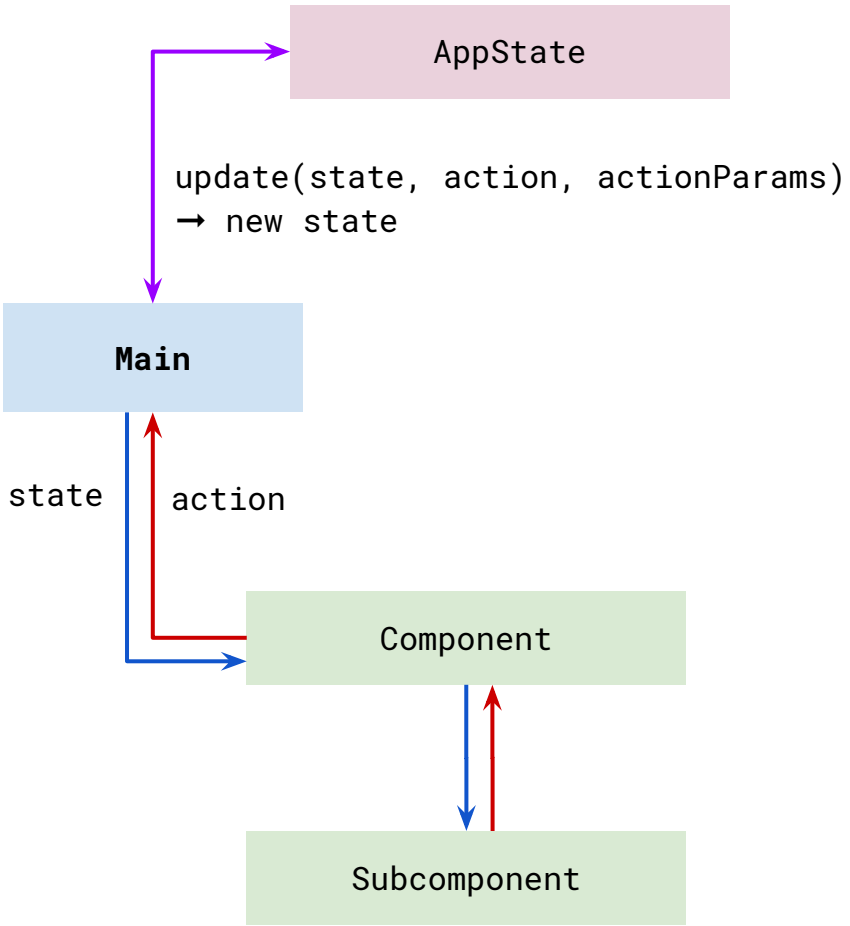


Abbildung 4.9: Vereinfachter Auszug des Codes zur Darstellung und Aktualisierung des Spiels.

```
let state;
let root = document.querySelector('#app');
let menuComponent = new MenuComponent();
let arenaComponent = new ArenaComponent();
let editorComponent = new StrategyEditorComponent();

root.addEventListener(
  'update', // custom event
  (event) => handleAction(event.detail.name, event.detail.params)
);

handleAction(ACTION_INITIALIZE);

function handleAction (action, params) {
  state = update(state, action, params);
  hyperHTML.bind(root, render(state));
}

function render (state) {
  if (state.viewMode === VIEW_MENU) {
    return hyperHTML.wire()`
      <div class="menu">
        ${menuComponent.render(state)}</div>
        <button onclick=${() =>
          this.dispatch('action', { type: ACTION_MENU_BACK })
        }>Back</button>
      `;
  } else if (state.viewMode === VIEW_TRAINING) {
    return hyperHTML.wire()`
      <div class="editor">${editorComponent.render(state)}</div>
      <div class="arena">${arenaComponent.render(state)}</div>`;
  } else if (state.viewMode === VIEW_TOURNAMENT) {
    ...
}

function update (state, action, params) {
  switch (action) {
    case ACTION_INITIALIZE: return initialState;
    case ACTION_OPEN_STORY_MENU:
      return state.set('currentMenu', MENU_STORY);
    ...
  }
}
```

Wert gespeichert, um die Lebenspunkte in der Statusbar, die eine HTML-Komponente ist, anzuzeigen. Um die Werte zu synchroniesieren, wird bei Treffer eines Roboters ein Ereignis ausgelöst. Dieses Ereignis wird in eine Aktion transformiert, die dann den Programmzustand ändert.

4.2.2 Datenpersistenz

Der ctGameStudio-Server unterstützt MongoDB für die Persistenz von Daten. Mithilfe von KeystoneJS werden JSON-Rest-APIs bereit gestellt, über die Daten gespeichert und geladen werden. Dazu können Schemas definiert werden, die die Struktur und Datentypen eines Modells festlegen, und zur Validation der Daten benutzt werden.

In der Erstversion des ctGameStudio bereits verfügbar war das Nutzermanagement, welches das Einloggen und Personalisieren der der Spielinhalte ermöglicht, und eine JSON-Rest-Endpunkt zum Loggen von Spielereignissen zur späteren Nachbereitung. Um den Open Stage-Modus zu unterstützen, mussten neue Endpunkte und Datenmodelle für Strategien, Turniere und Teilnahmen an Turnieren erstellt werden.

Im ctGameStudio mit Open Stage-Modus werden Server-Anfragen werden durch Actions angestoßen, und im AppState-Modul abgehandelt. Nach der Antwort durch den Server werden wiederrum Actions angestoßen, um das User Interface zu aktualisieren. Dabei wird das Anfrageergebnis und der Anfragestatus im Programmzustand gespeichert. Dazu gehört die Information, ob die Anfrage gestartet wurde, ob sie abgeschlossen wurde, und ob sie erfolgreich oder fehlerhaft war. Diese Informationen werden benutzt, um im Dialog entsprechendes Feedback an den Nutzer zu geben.

Abb. 4.10 stellt beispielhaft den Prozess dar, der beim Erstellen eines neuen Turniers abläuft. Aufgrund des initialen Zustands zeigt der Dialog das Turnierformular an (Schritt 1). Wenn der Nutzer den Button zur Erstellung des Turniers drückt, wird eine Aktion zum Starten der Anfrage ausgelöst (Schritt 2). Als Aktionsparamter werden die Turnieroptionen beigefügt, die der Nutzer eingestellt hat. Im Hintergrund wird die an den Turnier-Endpunkt mit den übergebenen Parametern gestartet. Der Dialog zeigt nun an, dass die Anfrage läuft. War das Speichern erfolgreich (Schritt 3a) wird der Programmzustand über eine entsprechende Action wird der Programmzustand aktualisiert, um dem Nutzer nun Feedback über das erfolgreiche Speichern zu geben, und den vom Server generierten Zugangscode anzeigen. Trat bei der Anfrage ein Fehler auf (Schritt 3b), wird über eine andere Action ausgelöst, welche den Programmzustand um die Fehlerinformation bereichert. Daraufhin kann im Dialog eine entsprechende Fehlermeldung angezeigt werden.

4.2.3 Strategieinterpreter

In der Erstversion des ctGameStudio wurde die Schritt-Funktion des Strategie-Interpreters aus drei Komponenten aufgerufen - bei erster Ausführung der Strategie durch die Toolbar, bei Beenden einer Aktion aus dem Roboter, und aus der Update-Funktion des PlayState. Dieser Aufbau stellte unnötige Komplexität dar, und machte es schwer zu identifieren, wann genau die Schrittfunktion aufgerufen wird. Die Einführung einer zweiten, parallel

Abbildung 4.10: Verlauf von Actions und dem Programmzustand beim Speichern eines Turnieres.

```
1) Öffnen des Dialogs

   -> Programmzustand: {
     requestStarted: false
   }

2) Action: CREATE_TOURNAMENT_REQUEST,
   Parameter: { maxRoundDuration: 3, maxRoundCount: 1, ... }

   -> Anfrage: POST /tournament

   -> Programmzustand: {
     requestStarted: true,
     isLoading: true
   }

3a) Action: CREATE_TOURNAMENT_SUCCESS
     Parameter: { accessCode: 'xxxxxx' }

   -> Programmzustand: {
     requestStarted: true,
     isLoading: false,
     success: true,
     accessCode: 'xxxxxx'
   }

3b) Action: CREATE_TOURNAMENT_FAILURE

   -> Programmzustand: {
     requestStarted: true,
     isLoading: false,
     success: false,
     failureReason: UNEXPECTED_SERVER_ERROR
   }
```

ausgeführten Strategie durch den Open Stage-Modus motivierte eine Vereinfachung der Strategieausführung.

Im ctGameStudio mit Open State-Modus wird der StrategieInterpreter nur aus dem PlayState heraus aufgerufen (Abb. 4.11). Die Schrittfunktionen der Interpreter der beiden Strategien werden kontinuierlich bei jedem Update ausgeführt. Dadurch wird auch der Aufruf aus der Roboter-Klasse hinfällig.

Um zu vermeiden, dass die Ausführung der Strategie bis zur nächsten Unterbrechung zu lange dauert, und es dazu zu Framedrops kommt, als auch die Ausführung der Aktionen des Gegner-Roboters verschoben wird, wird die Ausführung auf eine geringe Zeit begrenzt. Um Stack Overflows vorzubeugen, wird die wiederholte Ausführung der Schritt-Funktion des JS-Interpreters mit einer while-Schleife realisiert.

4.3 Anpassung des Roboterkampfes

Da sich nun zwei Roboter auf dem Spielfeld befinden, musste entschieden werden, was bei Kollisionen zwischen diesen Robotern passiert. Um die Strategien in diesem Fall möglichst uneingeschränkt weiter laufen zu lassen, stoßen sich die die Roboter bei einer Berührung voneinander ab, und setzen ihre Bewegung fort. Umgesetzt wurde dies mit der Standard-Kollisionsfunktion der Phaser-Arcade-Physik **??**.

Um die Evaluation seitens des Strategieentwicklers zu unterstützen, ist es sinnvoll, Roboteraktionen in einem langsamen, gut nachverfolgbaren Tempo auszuführen. Dabei soll das Gameplay spannend bleiben. Da die Geschwindigkeit der Roboteraktionen als zu schnell empfunden wurden, wurden relevante Konstanten und Timingberechnungen in der Roboter-Klasse angepasst, um die Bewegungs- und Drehgeschwindigkeit der Roboter sowie die Geschwindigkeit der Schussprojektile zu verringern.

Um Ausweichmanöver ermöglichen zu können, wurde die Scanfähigkeit eingeschränkt. Ein Ausweichmanöver besteht darin, seine Position zu verändern, nachdem man festgestellt hat, dass man getroffen wurde. Wenn der Gegner in einer Schleife dauerhaft auf einen Schuss direkt einen Scanvorgang folgen lässt, kann er jedoch jede Bewegung nachverfolgen. Ausweichen nach Treffern war so kaum möglich. Der Scanvorgang wurde daher so angepasst, dass er nicht mehr verzögerungsfrei über das gesamte Spielfeld reicht. Stattdessen muss sich der Scanstrahl nach einer Bewegung oder einem Schuss nun erst ausbreiten, bis er seine volle Reichweite erreicht hat. Ein Scan von der einen Spielfeldseite bis zur Anderen dauert nun wenige Sekunden. Um dem Strategieentwickler Kontrolle über die Dauer des Scans zu geben, wurde der Scanblock um einen Parameter erweitert, mit dem die maximale Reichweite angegeben werden kann, die erreicht werden soll, bis der nächste Block ausgeführt wird.

Zur Implementation dieser Änderung wurde die Roboter-Klasse und die Scanblöcke angepasst. Die Scan-Methode hatte zuvor den gesamten Scanvorgang behandelt, und hatte das Scanergebnis als Rückgabewert. Nun initialisiert sie den Scanvorgang lediglich, und gibt ein Objekt zurück, das erst nach Abschluss des Scans das Scanergebnis beinhaltet. Die Update-Funktion der Roboter-Klasse wurde erweitert, so dass der Scanstrahl aufgrund seiner Ausbreitung über die Zeit kontinuierlich neu dargestellet wird, kontinuierlich festgestellt wird, ob ein Gegner durch den Strahl erfasst wurde, und der Scanvorgang

Abbildung 4.11: Vereinfachter Auszug aus dem PlayState und dem StrategyInterpreter, der die vereinfachte Strategieausführung verdeutlicht.

```
// PlayState
function create () {
  ...
  player1Interpreter = StrategyInterpreter(
    player1StrategyCode, player1Robot
  );
  player2Interpreter = StrategyInterpreter(
    player2StrategyCode, player2Robot
  );
  ...
}

function update () {
  ...
  player1Interpreter.executeNextCommand();
  player2Interpreter.executeNextCommand();
  ...
}

// StrategyInterpreter
...
function executeNextCommand() {
  if (!robot.isIdle || !robot.sprite.alive) {
    return;
  }
  const lastReturnTimestamp = Date.now();
  let hasMoreCode = interpreter.step();
  while (
    hasMoreCode &&
    robot.isIdle &&
    robot.sprite.alive &&
    Date.now() - lastReturnTimestamp <= MAX_TIME_SPENT_IN_EXECUTION
  ) {
    hasMoreCode = interpreter.step();
  }
}
...
```

beendet wird, wenn ein Roboter erfasst wurde, oder der Rand des Spielfelds, bzw. die angegebene Scanreichweite erreicht wurde. Der Code, der aus den Blöcken generiert wird, die mit dem Scanergebnis arbeiten, wurde angepasst, um mit dem Objekt arbeiten zu können, dass von der Scan-Methode zurück gegeben wird.

5 Fazit und Ausblick

Literaturverzeichnis

[Gouws et al., 2013] Gouws, L. A., Bradshaw, K., and Wentworth, P. (2013). Computational thinking in educational activities: an evaluation of the educational game light-bot. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 10–15. ACM.

[ISTE, CSTE, 2011] ISTE, CSTE (2011). Computational thinking in k–12 education leadership toolkit.

[Kazimoglu et al., 2012] Kazimoglu, C., Kiernan, M., Bacon, L., and Mackinnon, L. (2012). A serious game for developing computational thinking and learning introductory computer programming. *Procedia-Social and Behavioral Sciences*, 47:1991–1999.

[Ministerium für Schule und Weiterbildung des Landes Nordrhein-Westfalen, 2014] Ministerium für Schule und Weiterbildung des Landes Nordrhein-Westfalen (2014). Informatik - Kernlehrplan, Gymnasium/Gesamtschule, Sek II-4725n.

[Papert, 1980] Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.

[Repenning et al., 2010] Repenning, A., Webb, D., and Ioannidou, A. (2010). Scalable game design and the development of a checklist for getting computational thinking into public schools. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 265–269. ACM.

[Rieber, 1996] Rieber, L. P. (1996). Seriously considering play: Designing interactive learning environments based on the blending of microworlds, simulations, and games. *Educational technology research and development*, 44(2):43–58.

[Selby et al., 2011] Selby, C. C., Selby, C., Woollard, J., and Woollard, J. (2011). Computational Thinking: The Developing Definition. page 6.

[Weintrop and Wilensky, 2015] Weintrop, D. and Wilensky, U. (2015). To block or not to block, that is the question: students' perceptions of blocks-based programming. pages 199–208. ACM Press.

[WERNEBURG et al., 2018] WERNEBURG, S., MANSKE, S., and HOPPE, H. U. (2018). ctGameStudio–A Game-Based Learning Environment to Foster Computational Thinking. In *Proceedings of the 2 6 th International Conference on Computers in Education*.

[Wing, 2011] Wing, J. (2011). Research notebook: Computational thinking—what and why? the link magazine, spring.