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

UNIVERSITÄT DUISBURG ESSEN

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

3. September 2018

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

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Computer betreffen jedes Leben und haben großes Potential und Herausforderungen -> Mündige Bürger sollten Computer programmieren können -> CT Skills -> auch hilfreich im Alltag und Job

Spiele haben sich als effektive Lernumgebungen heraus gestellt -> CT in Spiel vermitteln

Welche Spiel-Aspekte sind wichtig um Lernfortschritt zu erzielen? Wie lässt sich das Spiel in den Unterrichtsalltag einbinden?

"Wie Facebook 4 Mio. Datensätze verloren hat"(Fakenews 2017). ÄI wird uns zerstören"(Fakemag, 2015). "Big Technology invests 500 M to bring CS into schools"(Mustermag, 2017). Die Informatik umgibt nahezu alle Menschen, sie transformiert unseren Alltag und unseren Arbeitsplatz. Dies erfordert eine Gesellschaft, die in Kontrolle der Technologie ist, und weiß, wie damit umzugehen ist. Dabei fangen wir gerade erst an, die Informatik auch in dieser Reichweite und angemessenen Umfang in die Schulen zu bringen.

Mit "Computational Thinking"hat Jeanette Wing einen einflussreichen Forschungsansatz gestellt zu der Frage "Welche Fähigkeiten werden in der Informatik gebraucht?". Unter diesem Begriff haben sich eine Reihe von Curricula und Werkzeugen gesammelt, die versuchen, diese Fähigkeiten Schülern und anderen Interessierten näher zu bringen.

Mit RoboPlanet wurde ein Spiel entwickelt, bei dem der Lernende mittels einem zugänglichen visuellen Programmiersprache einen Roboter programmiert, um verschiedene Spielziele zu erreichen. Das Spiel besitzt einen Storymodus, bei dem dem Spieler schrittweise neue Programmierkonzepte beigebracht werden. Zu Ende der Story kann der Roboter komplexe Probleme lösen.

Während der Lernende mit dem Storymodus eine geleitete

1.2 Aufgabenstellung

Im Rahmen dieser Arbeit wird RoboPlanet um einen offenen Spielmodus erweitert, bei dem der Spieler seinen Roboter gegen andere Roboter antreten lässt. Im Training entwickelt der Schüler Strategien, um die Gegner, die wiederrum verschiedene Kampfstrategien besitzen, zu besiegen. In einem klassenübergreifenden Wettbewerb werden die entwickelten Roboter gegeneinander antreten. Durch die Erweiterung soll motiviert werden, und fortgeschrittene, nahezu unbegrenzte Programmierung gefördert werden.

1.3 Aufbau der Arbeit

Im Folgenden wird zunächst dargestellt, wie Computational Thinking durch RoboPlanet gefördert wird. Dazu wird die Theorie und verwandte Arbeiten vorgestellt. Wir untersuchen die Bestandteile von Computational Thinking, und die Mechanismen mit denen Schüler diese Bestandteile lernen, bzw. Lernen dieser Bestandteile gefördert werden. Daraufhin wird exploriert, welchen Mehrwert RoboArena bringt, welche Konzeption daraus entsteht, und wie dieses technisch umgesetzt wird.

Test [Ikeda et al., 1997]

2 Grundlagen

2.1 Computational Thinking

Wing, 2006, 2008; Selby, 2011; Grover, 2011?

Unter Computational Thinking wird Kognitionsprozess oder Gedankenprozess verstanden, der durch die Fähigkeit, in Form von Dekomposition, abstrahierend, evaluierend, algorithmisch und generalisierend zu denken, reflektiert wird. Ziel des Prozesses ist es, ein Problem so dar zu stellen, dass es von einem Computer gelöst werden kann. Im Folgenden sollen diese Denkweisen im Einzelnen vorgestellt werden.

CT-Aktivitäten (z.B. Debugging)

- Algorithmisches Denken

... (Scratch als Kontextualisierung CT in Education)

CT hat seinen Weg in den Schulalltag gefunden.

Computational Thinking (CT)

Aktivitäten/Fähigkeiten Computational Artefacts

2.2 Turtle Geometrie

- Vereinfachte Programmiersprache, entwickeln eines eigenes Vokabulars ermöglicht - Mikrowelten - Body-syntonic - Iteratives Vorgehen - Bewegungssemantik

2.3 Game-based learning von Computational Thinking

- Spiele häufig genutzt als Lernumgebungen (Kara, Program you robot, etc.) - Rieber, 1996 -> Spiele können aufgrund verschiedener Aspekte (fantasy, control, competition, etc.) Mikrowelten und Simulationen in einer Weise bereit stellen, in der Lerner internal motiviert sind Spielziele und damit Lernerfolge zu erzielen - low floor, high-ceiling

2.3.1 Program Your Robot

[?]

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

2.3.2 RoboCode

- Paper referenzieren (problem based learning)

[?]

- Herkunft, RoboCode als Beispiel für ein Programmierspiel und offenen Spielmodus - Beschreibung des Gameplays - Community: Wiki, Austausch von Strategien, Turniere.

2.3.3 CT Game Studio

[?]

- Guidance - Microworld - block-based programming - Storymodus, Level mit Fokus auf spezifischen oder Kombination von CT Skills - Evaluation unterstützt durch direktes Ausführen des Codes/Abbildung des Verhaltens/wiederholtes Ausführen, Highlight von ausgeführten Codeabschnitten, Erreichen von Zielen - Kreislauf? (Lightbot?) - analytics component - Ausblick open stage -> Strategien - Man könnte auch hier scratch nennen - Phaser game framework, Keystone server web framework and database, blockly-Bibliothek

Generell soviel wie möglich von vorher aufgreifen um Connections zu ziehen

3 Ansatz

3.1 Ein Lernszenario

Siehe Folien. Kontext: Informatikunterricht.

1. Schüler spielen Storymodus um Grundlagen zu lernen, 2. Schüler spielen Strategie-Traningsmodus um Erlentes anzuwenden und zu verbessern, 3. Lehrer erstellt Turnier und Schüler melden sich mit ihrerer Strategie an, 4. Turnier wird ausgeführt und die Spiele analysiert 5. Schüler verbessern ihre Strategien (Gewinnstrategie im Training verfügbar?), erneutes Turnier

3.2 Anforderungsanalyse

Welche Aspekte von CT werden noch nicht vom CT Game Studio unterstützt,

z.B. Evaluation, Kreislauf, selber dem Gegnerroboter Strategie auswählen

Anforderungen aus Szenario und Litaratur extrahieren

3.3 Offener Spielmodus

- Was wollen wir mit einem offenen Spielmodus erreichen? - Herausforderungen bieten, die an die Fähigkeiten anknüpfen, die man im Storymodus lernt, also CT-Fähigkeiten fordert und fördert, so dass ein intelligenter Einsatz dieser Fähigkeiten zu größerem Erfolg führt - Fortgeführter Spielspaß nach Beenden des Storymodus (- Einbettung in Unterrichtsszenarios, so dass das Spiel einerseits Lernen im Einzelunterricht und Evaluation in der Gruppe unterstützt) - Inspiriert von RoboCode soll es ein Roboterkampf geben, so dass eine eigens entwickelte Strategie gegen eine gegnerische Antritt - Nutzen die Bewegungssemantik des Storymodus, um ein zugängliches und nachvollziehbares (computational model/medium/was ist es?) zu bieten, an dem sich Schüler ihre Fähigkeiten auf iterative Weise ausbilden können (->Papert bilden einer Sprache) - In der Mikrowelt können Abstraktionen entwickelt werden, die einen erfolgreichen Roboterkampf ausmachen, z.B. Ausfinden machen des Gegners, Zielen und Angreifen, strategisch wirksame Positionen einnehmen, den Gegner verfolgen, Ausweichen, Verhalten verändern aufgrund veränderter Spielsituationen - Das Entwickeln einer Strategie, bzw. die Umsetzung dieser Abstraktionen erfordert andere CT-Fähigkeiten wie Aufspaltung (z.B. Ausweichen = Treffer feststellen + neue Position einnehmen) in und Algorithmisches Denken - Gesamte Strategie bildet ein Algorithmus (Initiale Positionierung, Endlosschleife, usw.) mit - z.B. Verfolgen und Angreifen umsetzen in dem man ein einer Schleife scan + schießen + neu orientieren falls Roboter nicht mehr in Sicht + ...) - Der Gegner-Roboter ist zum einen ein

motivierender Faktor aufgrund des Wettbewerbs (Ïch muss überleben/will besser sein"), zum anderen ein Spiegelbild des eigenen Roboters und damit Evaluationssubjekt (Üm mich zu verteidigen muss ich verstehen was er machtünd anhand des Gegners sehen wie man selbst vorgehen kann) - Community-Aspekt durch Turniermodus: Evaluation gegen andere Mitschüler, zusätzliche Motivation -> "besser als meine Mitschüler sein", sich unbekannten Herausforderungen stellen

3.3.1 Gameplay

- Ein zweiter Roboter + zweite Lebensanzeige muss hinzu gefügt werden - Ziel: Bessere/-überlegtere Strategien sollen sich auch im Matchergebnis niederschlagen - Verschiedene Ansätze sollen zum Erfolg führen können, z.B. effizientes Ausweichen/Positionieren, oder besonders raffiniertes Verfolgen - Stand zuvor: Sehr schnelle Bewegungen und Scanvorgang ermöglichen dies nicht und erschweren Evaluation (was passiert) - Verlangsamung der Bewegungen, Scanvorgang muss sich ausbreiten

3.3.2 Trainingsmodus

3.3.3 Turniermodus

- Wie sollten Turniere für den größten Lernerfolg aufgebaut werden?
- Wie sollten Kämpfe innerhalb des Turniers aufgebaut werden?
- Wie können Strategien entwickelt werden?
- Wie können Strategien für ein Turnier bereitgestellt werden?
- Wie wird das Verhältnis zwischen Schüler und Klasse hergestellt?
- Wie werden Turniere erstellt und daran teilgenommen?
- Welche Strategien sind im Trainingsmodus verfügbar?
- Wie können bestimmte CT-Aktivitäten unterstützt werden (z.B. Debugging)?

3.4 Vergleich zu verwandten Spielen

Nicht explizit sondern integriert

4 Implementierung

(TODO Einführung noch in Entwurfsphase) Die Architektur des CTGameStudio in der Erstversion musste angepasst werden, um die Anforderungen des neuen Spielmodus zu erfüllen. Das Hinzufügen eines zweiten Roboters erfordert, mehrere Instanzen der blockly-Komponente und des Strategie-Interpreters zu erzeugen (TODO etwas generischer formulieren). Die gleichzeitige Ausführung mehrerer Strategien muss ermöglicht werden. Die Gameloop und Roboterspielobjekte müssen so angepasst werden, dass die Roboter sinnvoll miteinander interagieren und sich so verhalten, dass sinnvolles Gameplay möglich ist. Zudem muss das Spiel Dialoge mit Formularelementen erweitert werden, und das Speichern und Laden von Daten über HTTP-Schnittstellen des Servers unterstützen. (Statemanagement, + HTML-Komponenten/Render-Strategie)

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. Neben Standard-HTML-Inputs wird Phaser [2] genutzt, um mittels WebGL Spielelemente zu rendern.

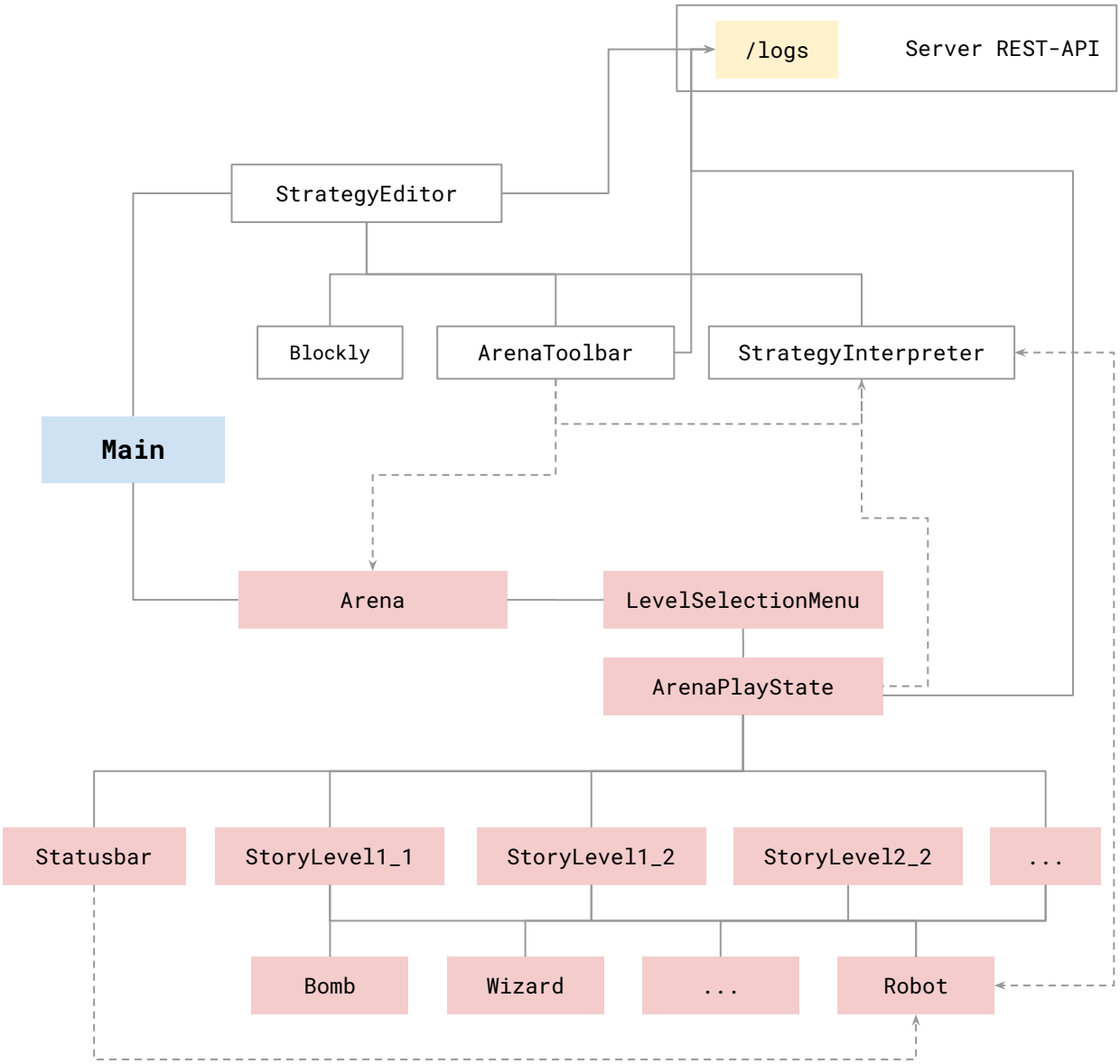
Komponenten sind als als Javascript-Funktionen, Objekte oder Klassen definiert. Während eine Strukturierung dadurch erfolgt, dass Komponenten auf mehrere Dateien aufgeteilt sind, liegen die Komponenten, Instanzen der Komponenten als auch interne Zustandsvariablen in einem globalen Namensraum. Dadurch 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. 4.1.3). 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. 4.1.3, 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. 4.1.3), 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. 4.1.4).

[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,
  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 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

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();
  }
  ...
}
```

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.

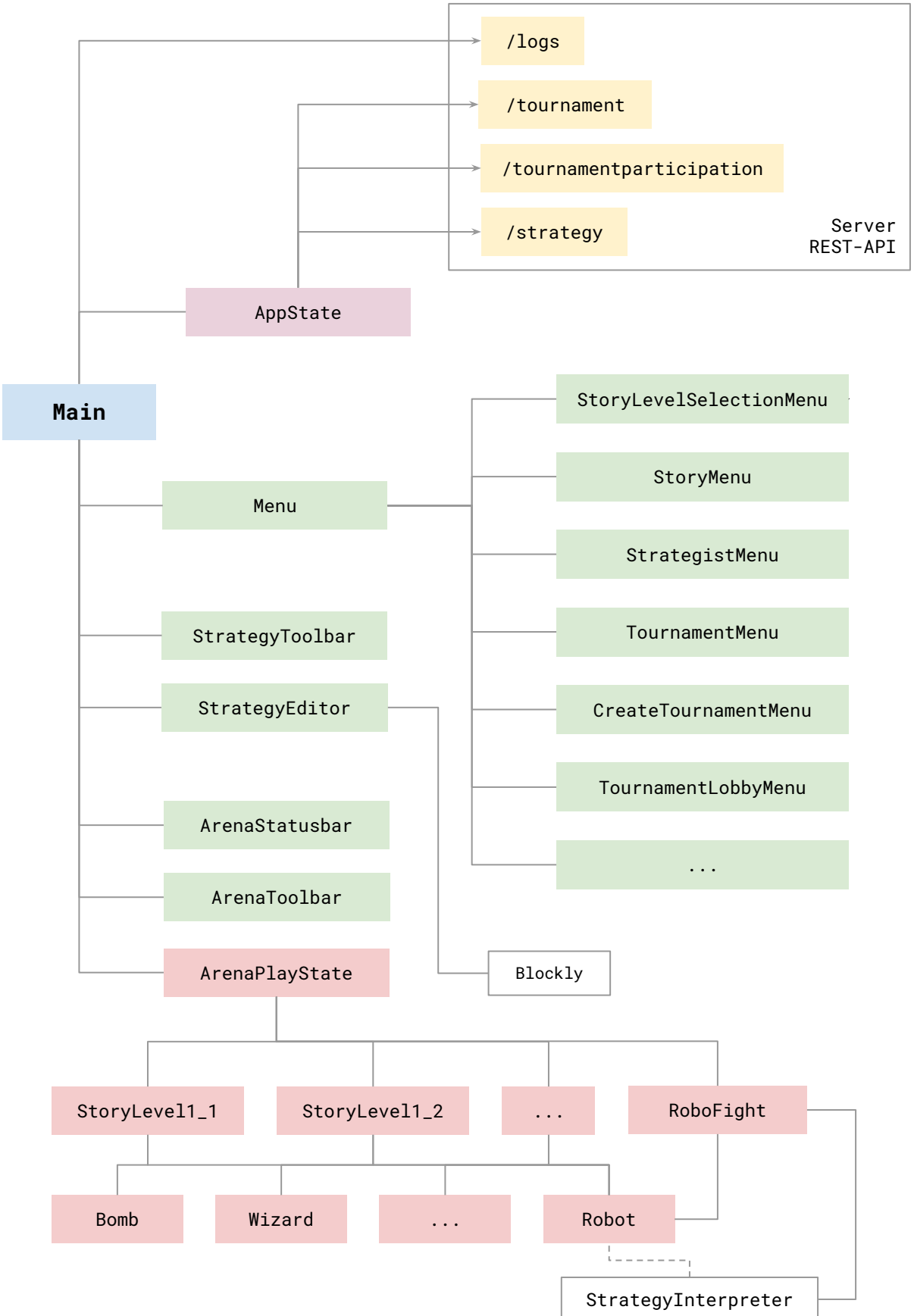
Die Statusleiste enthält die Lebensanzeige des Roboters und den Pausebutton und das Pausemenü, und ist ebenfalls als Objekt mit eigener Create- und Update-Funktion implementiert, die vom PlayState aufgerufen werden.

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

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.



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**??**-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 (**??**). Das Objekt 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. Dieses System 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. **??**). Die update-Funktion (Abb. **??**) 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 Veräderungen des Zustands nimmt die Bibliothek nur die minimalen Änderungen am DOM vor, die nötig sind, um die veränderte Struktur darzustellen. So ist es möglich, bei jeder Zustandsaktualisierung die render-Funktion auszuführen, ohne das DOM komplett neu zu konstruieren und durch den Browser darzustellen.

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. Bei Programmfehlern ist leicht nachvollziehbar, an welcher Stelle ein unkorrekter Zustand gebildet wurde. Durch die explizite Modellierung des Zustands, der Actions und der Aktualisierung des

Abbildung 4.7: Ausschnitt aus dem Zustandsmodell.

??

```
// AppState-Modul
const initialState = Immutable({
  selectedStrategyId: 'default',
  selectedEnemyStrategyId: 'default',
  strategies: [],
  view: VIEW_MENU,
  currentMenu: MENU_MAIN,
  menuHistory: [],
  roboProgrammerExtended: false,
  tournamentsList: {
    isLoading: true,
    success: false,
    failureReason: '',
    tournaments: []
  }
  ...
});
```

Zustands in Form der Update-Funktion ist übersichtlich zu sehen, welche Formen der Zustand annehmen kann. Diese Informationen werden im AppState-Modul gebündelt.

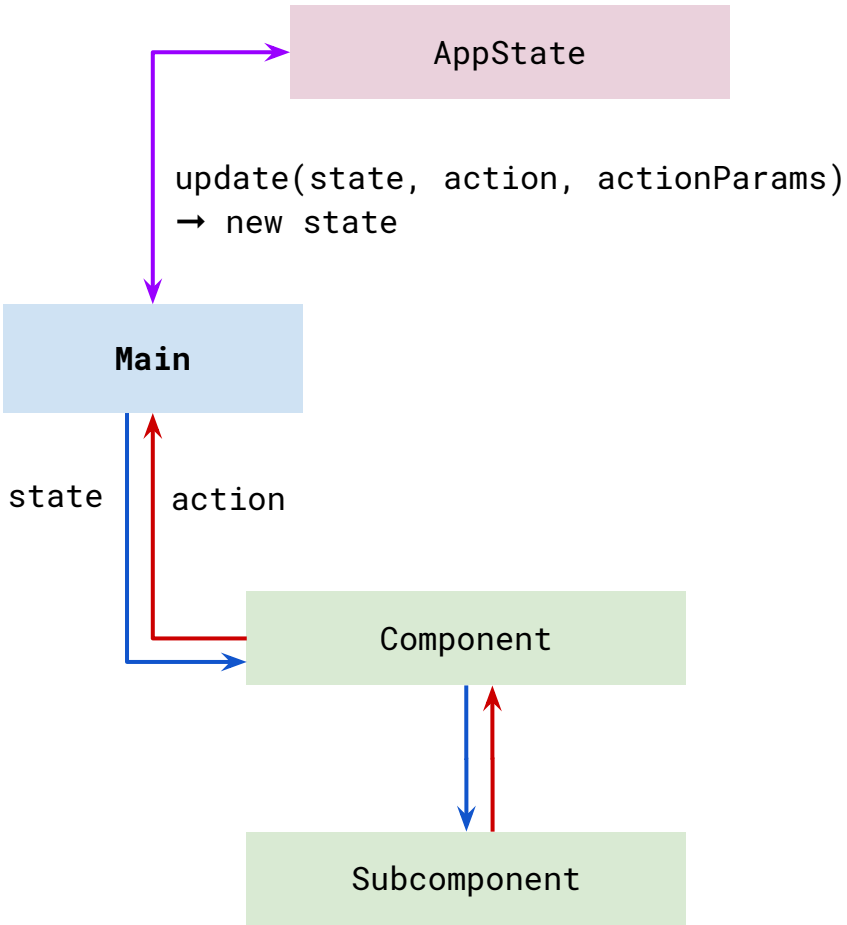
Das beschriebene Zustandsmanagement lässt sich nicht auf Phaser übertragen. Wie in 4.1.5 beschrieben, arbeitet Phaser mit regulären, veränderbaren Javascript-Objekten und einem eigenen Update/Render-Loop. Daher wird die Arena-Komponente nicht bei jeder Aktion neu dargestellt, sondern bei Start des Trainings oder Start eines Turnierkampfes einmal 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 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 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 4.8: 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
    // ...
  }
}
```

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. 4.2.2). 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 4.10: Vereinfachter Auszug aus dem PlayState und dem StrategyInterpreter, der die vereinfachte Strategieausführung verdeutlicht.

```
// PlayState
function create () {
  ...
  player1Interpreter = StrategyInterpreter(player1StrategyCode, player1Rob
  player2Interpreter = StrategyInterpreter(player2StrategyCode, player2Rob
  ...
}

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.

4.3.1 Turnierausführung

4.4 Datenmodell und Client-Server-Kommunikation

- Strategien, Turnier, Teilnahmen - RESTful API

5 Fazit

Ausblick

- ua. Auch sinnvolle Sicht für TTurnierteilnehmer

Literaturverzeichnis

[Ikeda et al., 1997] Ikeda, M., Go, S., and Mizoguchi, R. (1997). Opportunistic group formation. In *Proceedings of the Conference on Artificial Intelligence in Education (AI-ED)*, pages 167–174, Amsterdam. IOS Press.