Implementierung eines historischen Brettspiels mit einer künstlichen Intelligenz in Unity: Latrunculi

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

Universität Trier
FB IV - Informatikwissenschaften
Professur Theoretische Informatik

Gutachter:	Prof. Dr. Henning Fernau
	Petra Wolf
Betreuer:	Petra Wolf

Vorgelegt am xx.xx.xxxx von:

Alexander Pet
Hohenzollernstraße 3
54290 Trier
s4alpett@uni-trier.de
Matr.-Nr. 1205780

Zusammenfassung

In dieser Arbeit wird das Spiel Latrunculi betrachtet und eine digitale Version für das Landesmuseum Birkenfeld umgesetzt. Zum Zeitpunkt dieser Arbeit begleiten uns Kontaktbeschränkungen im Alltag, daher war die Idee dem Museum eine Alternative bieten zu können, die auch online abgerufen werden kann. Daher wurde mit Hilfe der Unity Game Engine eine unter Windows lauffähige Version, sowie eine webbasierte implementiert. Hierzu wurden überlieferte Spielregeln umgesetzt und eigene Ideen für ein Spielende festgelegt, da historisch nicht alle Informationen überliefert wurden. Zur Umsetzung habe ich mich mit dem deterministischen MiniMax-Algorithmus auseinandergesetzt, sowie die Verbesserung Alpha-Beta betrachtet. Diese implementierte künstliche Intelligenz wurde genutzt um das Spielen gegen den Computer zu ermöglichen und zur Simulation bei der zwei KIs gegeneinander antreten. Das Verhalten beider Möglichkeiten wurde im 6. Kapitel analysiert. Bei dieser Analyse hat sich herausgestellt, dass es wenig sinnvoll ist KIs mit den selben Konfigurationen gegeneinander antreten zu lassen. Diese KIs tendieren dazu sich gegenseitig auszuweichen. Weiterhin wurde das Verhalten gegen menschliche Spieler beobachtet und verbessert. Eine primitive Version wurde zu Beginn sehr leicht geschlagen, konnte aber mit einigen Anpassungen in der Analyse der Situation verbessert werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

In diesem Kapitel wird die Motivation, die Problemstellung, sowie die Zielsetzung erläutert. Zum Schluss wird die Struktur der Arbeit beschrieben.

1.1 Motivation

Da wir uns zum Zeitpunkt der Verfassung dieser Arbeit in einer durch Covid-19 verursachten Pandemie befinden und uns somit Kontaktbeschränkungen im Alltag begleiten, ist die Motivation vor allem den Besuchern des Landesmuseum Birkenfeld ein historisches Spiel online anbieten zu können. Latrunculi wurde implementiert, weil es aufgrund von nicht eindeutig überlieferten Regeln einen gewissen Spielraum in der Umsetzung bietet. Weiterhin bietet das Museum Veranstaltungen wie „So spielten die Römer"[1] für Schulklassen an und kann durch eine digitale und spielbare Umsetzung eines historischen Spiels bei den Besuchern ein größeres Interesse wecken. Außerdem habe ich ein persönliches Interesse daran, mich im Bereich der künstlichen Intelligenz und Spielprogrammierung weiterzuentwickeln.

1.2 Problemstellung

Latrunculi war historisch gesehen ein beliebtes und weit verbreitetes strategisches Brettspiel der Römer und Griechen. Die römischen Dichter Ovid und Martial haben dieses Spiel bereits erwähnt und beschrieben, dass bei sehr talentierten Spielern häufig Zuschauer anwesend waren, wie Ulrich Schaedler es bereits in „Homo Ludens"[9] erläutert hat. Die Spielsteine wurden als latrones(lat. für Soldat) bezeichnet. Da die Entstehung dieses Spiels so lange zurück liegt, wurden nicht alle Regeln überliefert, sodass verschiedene Quellen hinzugezogen werden müssen um die Spielmechanismen zu klären. Daher existieren verschiedene Variationen mit abweichenden Regeln wie zum Beispiel von Masters Traditional Games [1] und Katharina Uebel und Peter Buri[10]. Das Latrunculi-Spielfeld besteht aus einem Raster senkrechter und waagerechter Reihen. Weiterhin wurden wahrscheinlich die Spielsteine auf den Feldern und nicht auf den Linien platziert und bewegt, wie es zum Beispiel beim Go[3] der Fall ist. Dabei konnte keine fixe Größe des Spielbretts festgestellt werden, sodass verschieden große Spielfelder mit unterschiedlicher Anzahl an Zellen gefunden wurden. Bei Ausgrabungen konnten Latrunculi-Bretter mit beispielsweise 7x7, 8x8, 9x10, 7x10 und 7x6 Feldern geborgen werden. Als Spielbretter dienten hierbei unter anderem Kalksteine oder Ziegelsteine, wie sie in Mainz oder in Hadrianswall in Großbritannien gefunden wurden[9]. Betrachtet man Ovids Aussagen, kann man folgern, dass das

[1] https://www.landesmuseum-birkenfeld.de/landesmuseum/erlebnismuseum/

[3] https://www.ultraboardgames.com/go/game-rules.php

Hauptziel darin bestand mit seinen Spielsteinen einen gegnerischen von zwei gegenüberliegenden Seiten zu umstellen und somit aus dem Spiel zu nehmen. Weiterhin beschreibt er, dass es wichtig sei, seine Steine paarweise zu bewegen, dadurch wird verhindert, dass diese vom Gegner geschlagen werden können. Für diesen Angriffs- und Verteidigungsmechanismus konnten die Steine geradlinig horizontal und vertikal verschoben werden. Allerdings ist auch nicht eindeutig geklärt, wann das Spiel endet oder ob es mit besonderen Figuren funktioniert hat. Beispielsweise gibt es Variationen mit einem Feldherren mit speziellen Fähigkeiten[4], ähnlich wie die Dame beim Schach. Die festgelegten Regeln für die implementierte Version werden im Abschnitt 2.1 erklärt.

1.3 Zielsetzung

In dieser Arbeit wird eine Variation des Spiels Latrunculi mithilfe der Unity Game Engine[2] für das Landesmuseum Birkenfeld umgesetzt. Dabei wurde eine künstliche Intelligenz implementiert, sodass es die Möglichkeit gibt, gegen eine KI zu spielen, die zielführende Züge umsetzen soll. Weiterhin soll das entwickelte Spiel die Möglichkeit bieten, zwei KI's als Simulation gegeneinander antreten zu lassen. Das Projekt soll auch die Alternative bieten online abgerufen zu werden. Unity stellt relativ einfache Portierungs-Möglichkeiten zur Verfügung, sodass die Entwicklung eines Spiels für unterschiedliche Plattformen erleichtert wird.

1.4 Aufbau der Arbeit

Die folgenden Kapitel behandeln zuerst die Grundlagen, um die anschließend aufgeführten Abschnitte besser verstehen zu können. Dabei werden die umgesetzten Spielregeln erklärt, sowie die grundsätzliche Definition von künstlicher Intelligenz, als auch Umsetzungen in bekannten Spielen erwähnt. Außerdem wird die Game Engine Unity kurz erklärt und der verwendete Algorithmus für die künstliche Intelligenz. Das dritte Kapitel umfasst sowohl die technischen Anforderungen, als auch nötige Funktionalitäten des Spiels. Im Anschluss an den dritten Abschnitt werden die nötigen Features und Technologien beschrieben. Das 5. Kapitel erklärt die Implementierung des Spiels, sowie der KI. Anschließend wird erklärt, wie der Release beim Kunden im Museum unter gegebenen Umständen stattgefunden hat und die KIs als Simulation und als Gegenspieler analysiert. Zum Schluss gibt es noch einen Ausblick auf mögliche Erweiterungen um den Algorithmus effizienter zu gestalten.

[4] https://www.spielmannshof-seitenroda.de/die-spiele/latrunculi/
[2] https://unity.com/

2. Grundlagen

Dieses Kapitel beinhaltet grundlegende Informationen zu der Arbeit. Zuerst werden die Spielregeln erklärt, dann die verwendete Unity Engine und zum Schluss wird noch auf Formen der künstlichen Intelligenz eingegangen. Dabei wird unterschieden zwischen strategischen und deterministischen KIs, sowie dem Machine Learning.

2.1 Latrunculi

Ulrich Schaedler[9] hat beschrieben, wie entzückt die Römer von diesem Spiel waren. Nicht nur zum selber spielen waren sie für Latrunculi zu begeistern, sondern wohl auch beim Zuschauen von begabten Spielern. Die heute überlieferten Regeln sind nicht ganz eindeutig geklärt, daher gibt es aktuell verschiedene Variationen dieses Spiels. Die in dieser Arbeit umgesetzte Variante orientiert sich an der von Kowalski[11] beschriebenen Version. Dabei gab es keine feste Größe des Spielfelds. Schädler[9] berichtet von Ausgrabungen bei denen Größen wie 7x6 oder 10x10 gefunden wurden. Kowalski[11] erklärt die Basis-Version mit einem 8x8 Brett. Die Felder wurden in unterschiedliche Materialien geritzt. Schaedler berichtet von Kalksteinen und Marmor, aktuelle Versionen werden aber vor allem auf Holz oder Leder umgesetzt[4]. Der Ausgangszustand wird ebenfalls unterschiedlich ausgelegt. Masters Traditional Games[1] beschreibt eine Variation mit zwei Reihen an Steinen auf beiden Seiten der Spieler, wohingegen die Version von Kowalski[11] mit jeweils nur einer funktioniert. Weiterhin gab es Regeln bei denen zu Beginn entweder die Steine nacheinander platziert[9] oder aber direkt auf den ersten Reihen gesetzt wurden. Im folgenden Abschnitt wird näher auf die umgesetzten Spielregeln eingegangen.

Spielregeln

Der grundsätzliche Aufbau des Spiels beinhaltet ein Spielbrett mit minimum 6x6 Feldern bis hin zu 12x12 großen Brettern. Weiterhin gibt es zwei verschiedene Spielsteine, die entweder unterschiedliche Farben haben oder aus zum Beispiel einerseits Steinen und andererseits Muscheln, wie in dieser Ausarbeitung auch, bestehen. Dabei übernimmt ein Spieler die Steine und der Gegenspieler die Muscheln. Zu Beginn werden die eigenen Steine jeweils auf der ersten Reihe platziert, wie in Abbildung 2.1 zu sehen ist. Das Spiel beginnt mit einem der beiden Spieler. Während des Spiels ziehen die Spieler abwechselnd horizontal oder vertikal über das Feld, ohne dabei einen anderen Stein zu überspringen oder auf einer besetzten Zelle zu landen (Abbildungen 2.2). Um gegnerische Spielsteine zu erobern, müssen diese wie in der Abbildung 2.3 von eigenen Steinen auf zwei gegenüberliegenden Feldern umstellt sein. Wird ein gegnerischer Stein erobert, dann wird dieser aus dem Spiel entfernt und ein eigener

[4] https://www.spielmannshof-seitenroda.de/die-spiele/latrunculi/

darf nochmal bewegt werden. Hierbei gibt es auch Variationen bei denen nur der zuletzt bewegte Stein sich wieder bewegen darf. Bewegt sich ein eigener Stein zwischen zwei gegnerische und scheint somit umstellt zu sein, zählt dieser allerdings nicht als erobert(Abbildung 2.3). Das Ziel des Spiels ist es, dass der Gegenspieler keinen Zug mehr ausführen kann beziehungsweise keinen Spielstein mehr erobern kann.

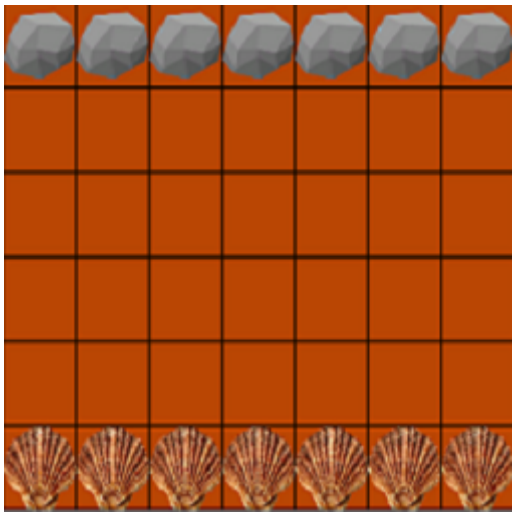

Abbildung 2.1: Startkonfiguration eines 7x6 Latrunculi Feldes

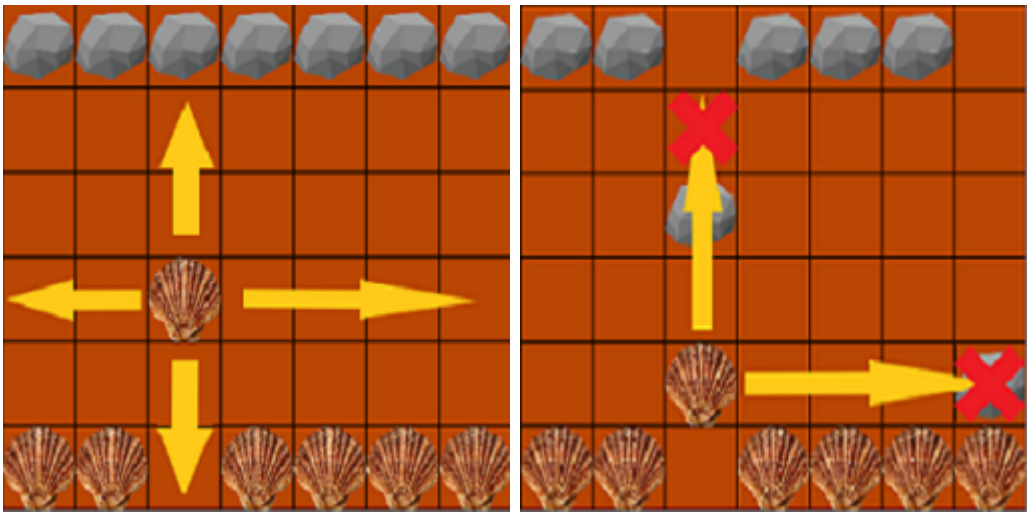

Abbildung 2.2: Legitime Bewegungen auf einem 7x6 Latrunculi-Feld(links) & nicht erlaubte Züge (rechts)

2.2 Unity Engine

Die Unity Engine ist eine von Unity Technologies entwickelte Laufzeit- und Entwicklungsumgebung für Videospiele. Die Engine bietet dabei Portierungen für PCs, Spielkonsolen, mobile Geräte wie Android und Webbrowser. Bekannte Spiele wie Pokemon Go und Hearthstone wurden mit Unity entwickelt. Weiterhin bietet Unity die Möglichkeit sowohl in 2D, als auch in 3D Spiele umzusetzen und bietet Funktionen, wie den Eventhandler für Gameobjects, um die Entwicklung zu vereinfachen.

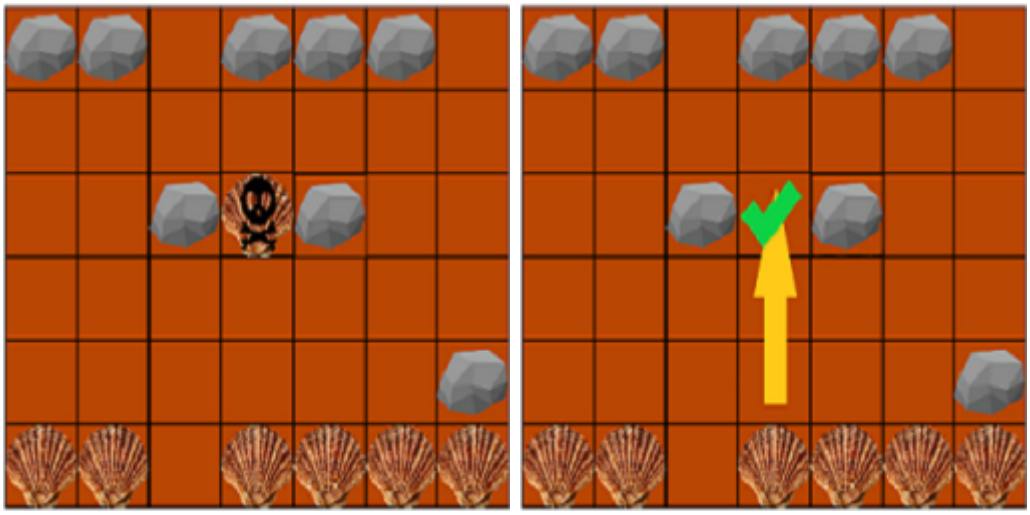

Abbildung 2.3: Eroberung eines Spielsteins(links) und keine Eroberung(rechts)

Um selbstgeschriebene Skripte einzubinden, unterstützt Unity unter anderem C#, wie ich es in dieser Arbeit auch verwendet habe. Unity wurde hier vor allem wegen der einfachen Portierung und Wiederverwendbarkeit von vorher erstellten Prefabs genutzt. Ausserdem bietet Unity eine große Anzahl an Libraries zur Unterstützung der Spielentwicklung.
Die Engine bietet die Möglichkeit einerseits per Drag & Drop Spielobjekte im Editor zu platzieren und diesen Objekten so ihren Ausgangszustand zu geben, andererseits gibt es Mittel die Szene dynamisch via Skript zu erstellen. Entwickelt man in der 2D Umgebung, hat man die Möglichkeit seinen Objekten Sprites mitzugeben. Diese Sprites sind Grafikobjekte, die zur Darstellung der Spielobjekte im Editor platziert werden. Selbst erstellte Skripte, Sprites und Szenen, sowie andere genutzte Ressourcen, wie zuvor erstellte Prefabs für eine dynamische Generierung, befinden sich bei Unity-Projekten im Assets-Verzeichniss.

Prefab System

Unitys Prefab System ermöglicht es Spielobjekte zu erstellen und als Prefabs zu speichern. Diese können wiederverwendet werden und fungieren wie eine Vorlage(Template) eines Objektes. Dadurch können wiederkehrende Spielobjekte einmalig definiert werden und durch das Prefab in der selben Form oder aber auch verändert mehrfach genutzt werden. Das heißt es können dadurch zur Laufzeit Spielobjekte instanziiert und zur Szene hinzugefügt werden, ohne jedes einzelne Objekt neu zu definieren. Weiterhin können diese so erstellten Objekte trotzdem noch separat verändert werden.

2.3 Künstliche Intelligenz

Allgemein kann man sagen, dass es drei Formen der künstlichen Intelligenz gibt. Zum einen sollte man zwischen deterministischen und strategischen KIs unterscheiden, andererseits kann man diese vom Machine Learning abgrenzen.

2.3.1 Strategische KIs

Strategische KIs werden in Videospielen zum Beispiel zur Wegfindung oder für die Atmosphäre genutzt. Am Beispiel von sogenannten Non-Player-Character (kurz:NPC) in Rollenspielen wie Gothic kann diese Form gut erklärt werden. NPCs sind Figuren innerhalb eines Spiels mit denen in der Regel interagiert werden kann oder die für die Atmosphäre eingesetzt werden und nur einen Weg ablaufen, der via Wegfinde-Algorithmus berechnet wird. Außerdem kann auch das Verhalten beeinflusst werden, indem die Spielfiguren auf Aktionen reagieren können und in einen entsprechenden Zustand versetzt werden. Betrachtet man das Spiel Gothic, erkennt man sehr leicht dieses Prinzip. Hier ist es möglich, dass der Spieler durch sein Verhalten die NPCs beeinflusst. Es existieren Figuren, die einem aggressiv gegenübertreten, falls man zum Beispiel zuvor aus seinem Haus etwas gestohlen hat. Für diese NPCs werden strategische KIs eingesetzt, um vor allem die Handlung zu unterstützen.

2.3.2 Deterministische KIs

Deterministische KIs können Anwendung in Brettspielen finden. Hier werden Algorithmen verwendet um den Spielzustand und zukünftige zu bewerten und daraus die optimale Aktion zu finden. Diese Systeme können als Baumstruktur dargestellt werden. Die Algorithmen wurden schon erfolgreich in Spielen wie Tic-Tac-Toe oder Schach[12] verwendet. Damit diese gut funktionieren ist die Voraussetzung, dass perfekte Informationen über das Spiel vorliegen. Ein Spiel mit perfekter Information ist zum Beispiel das Spiel GO oder Schach. Hier haben die Spieler immer die Informationen über den Spielzustand und den vorherigen Zügen. Als Gegenbeispiel kann man an dieser Stelle Poker nennen, wo die Spieler immer nur ihre eigenen Karten auf der Hand kennen. Da Latrunculi ein Spiel mit perfekter Information ist wurde in dieser Arbeit der MiniMax-Algorithmus implementiert.

MiniMax

Beim MiniMax-Algorithmus entspricht der Ausgangsknoten dem aktuellen Zustand und jedes Kind steht für einen zukünftigen. Diese Zustände werden nach verschiedenen Kriterien bewertet und so eine Punktzahl für jeden errechnet. An der Abbildung 2.4 wird das Prinzip deutlicher. Dabei ist im Ausgangszustand Spieler 1 an der Reihe gewesen. Die drei abgebildeten Kinder sind jeweils mögliche Züge des zweiten Spielers. Der linke Zustand hat die Wertung 0, da hier weder eine Bedrohung oder ein Abzug aus einer Bedrohung, noch ein Angriff stattfindet. Die beiden anderen Kinder haben eine Wertung von +30, da diese Zustände zu einer Bedrohung der Muscheln führen. Die nächste Reihe zeigt mögliche Züge der Steine, die aus den vorherigen Situationen resultieren. Dabei werden Züge mit 40 bewertet, falls eine Muschel sich neben die bedrohte stellt und diese dadurch vor einer Eroberung schützt. Die höchste Bewertung 100 erhalten die Züge, die den gegnerischen Stein erobern. Diese Bewertungen werden von den höchsten Punktzahlen ausgehend bis zum Ausgangsknoten aufsummiert und ergeben dabei eine Höchstpunktzahl von -40 bei einer Suchtiefe von 2.

Erfolgreiche Umsetzungen

Yannakakis und Togelius erwähnen in ihrem Buch „Artificial intelligence and games", das der MiniMax Algorithmus an Effizienz verliert, je komplexer das Spiel ist. Die

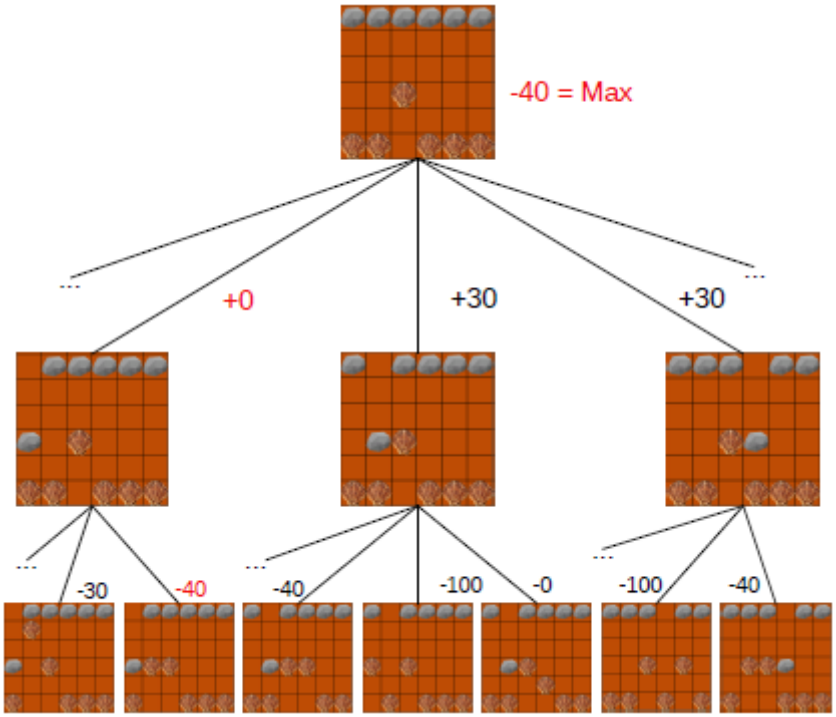

Abbildung 2.4: Ausschnitt: MinMax-Spielbaum eines 6x6 Latrunculi Feldes

Schwierigkeit liegt darin, eine gute Evaluierungsfunktion der Zustände zu finden[12]. Je mehr Züge existieren, desto schwerer wird es eine sinnvolle Punkteverteilung zu finden. Vergleicht man Schach und Go wird das Problem deutlich. Bei ersterem hat man in der Regel über 30 mögliche Züge, diese sind im Vergleich zu Go mit bis zu 300 Möglichkeiten viel geringer. Für diese weniger komplexen Spiele gibt es bereits seit Jahren erfolgreiche Umsetzungen. IBM's entwickelte KI Deep Blue (vgl. [6]) hat im Mai 1997 als erster Computer den damals amtierenden Weltmeister im Schach geschlagen[7]. Allerdings war dies mit einem hohen Rechenaufwand und speziellen Schach-Chips verbunden(vgl. [6] & [7]). Festhalten lässt sich, dass der Erfolg des Algorithmus von der Evaluierungs-Funktion und von der Rechenkraft des Computers abhängig ist.

2.3.3 Abgrenzung zum Machine Learning

Beim Machine Learning wird eine statistische Wissenbasis aufgebaut, indem das System mit Daten trainiert wird. Es ist ein Verfahren, dass Algorithmen nutzt um Daten zu analysieren und zu kategorisieren. Als Beispiel kann die Gesichtserkennung genannt werden. Hier werden die Systeme mit unterschiedlichen Gesichtern trainiert, um anschließend diese erkennen und zuordnen zu können. Generell kann man festhalten, dass diese Verfahren eher nicht in Videospielen angewendet werden. Es lässt sich zwischen dem Supervised Learning (überwachtes Lernen) und dem Unsupervised Learning (nicht überwachtes Lernen) unterscheiden. Das überwachte funktioniert so, dass ein Algorithmus Trainingsdaten erhält, sowie die Bedeutung der Daten. Dadurch soll der Algorithmus Muster erkennen und in einer Funktion speichern, um anschließend diese Muster auf neuen Daten erkennen zu können. Diese Funktion wird abhängig von der Trefferquote vom System angepasst. Beim nicht überwachten Lernen werden nicht kategorisierte Daten dem Algorithmus übergeben,

sodass dieser Cluster erkennen soll und zukünftige Daten einordnen kann.

3. Anforderungen

In diesem Kapitel werden die Anforderungen einer Lösung betrachtet um Latrunculi digital umzusetzen und auf dem gewünschten Gerät laufen zu lassen.

3.1 Technische Anforderungen

Zum Zeitpunkt der Bearbeitung dieser Arbeit befinden wir uns in einer durch Covid-19 verursachten Pandemie, daher begleiten Kontaktbeschränkungen unseren Alltag. Aus diesem Grund soll es auch eine Möglichkeit geben, dieses Spiel online via Webbrowser abrufen zu können. Außerdem soll es fähig sein auch unter Windows zu laufen und mit einem Touchscreen bedienbar sein. Bei dem vorhandenen Gerät im Landesmuseum Birkenfeld handelt es sich um einen Touch-Tisch mit installiertem Microsoft Windows 10 Enterprise LTSC (x64), 4 GB RAM als Arbeitsspeicher und einem Intel NUC10 i5 FNH. Da das Gerät auch keine dauerhafte Internetverbindung hat, soll es auch möglich sein das Spiel via USB auf dem Gerät zu starten. Unity bietet die Möglichkeit, ein entwickeltes Spiel für verschiedene Plattformen bereitzustellen, daher habe ich mich für Unity als Game Engine entschieden. Außerdem kann der Ordner mit der ausführbaren Datei prinzipiell auch auf einem USB-Stick liegen und von dort aus gestartet werden.

3.2 Weitere Anforderungen

Das Spiel soll die Möglichkeit bieten, gegen eine künstliche Intelligenz zu spielen und auch als Simulation für Vorführungszwecke zu laufen. Die KI sollte nach Möglichkeit eine sinnvolle Strategie verfolgen, sodass die Züge nicht willkürlich wirken. Das heißt es müssen einerseits die Logik & Regeln, damit das Spiel grundsätzlich spielbar ist, und andererseits der MiniMax-Algorithmus, um die erlaubten Züge vorher zu bewerten, implementiert werden. Weiterhin muss ein ansprechendes Design gewählt werden, dass auch historisch sinnvoll ist und den Spielern im Museum ein relativ realistisches Bild des Spiels bietet. Da geschichtlich überliefert wurde, dass vor allem mit Steinen gespielt wurde, wurden die Spielfiguren als Steine und Muscheln festgelegt. Die Steuerung sollte möglichst intuitiv sein, daher habe ich mich beim Prototypen für eine „OnClick()" -Steuerung entschieden. Das heißt die Steine wurden angeklickt und durch einen Klick auf eine erlaubte Zelle verschoben. Als allerdings klar wurde, dass das Spiel auf einem Touch-Gerät laufen soll, wurde Latrunculi für eine „Drag&Drop"-Bedienung angepasst. Das Menü soll leicht erreichbar sein, um das Spiel neu starten oder beenden zu können.

4. Ideen

In diesem Kapitel werden die Features erklärt, die grundlegend notwendig sind, sowie die, die sich während der Entwicklung ergeben haben und vom Kunden gewünscht wurden.

Grundlegende Features

Diese grundsätzlichen Features entsprechen dem ursprünglichen Prototypen des Spiels, der in der Abbildung 4.1 zu sehen ist, dadurch konnten die implementierten Regeln getestet und durch zusätzliche Ideen erweitert werden.

- Spielbrett, mit einer Größe von 8x8 und fix in der Szene verankert
- Schwarze und weiße/graue Kreise als Spielsteine
- Zwei Spieler können per Klick auf einen Stein und auf ein Feld Spielsteine bewegen
- Hervorheben der möglichen Positionen

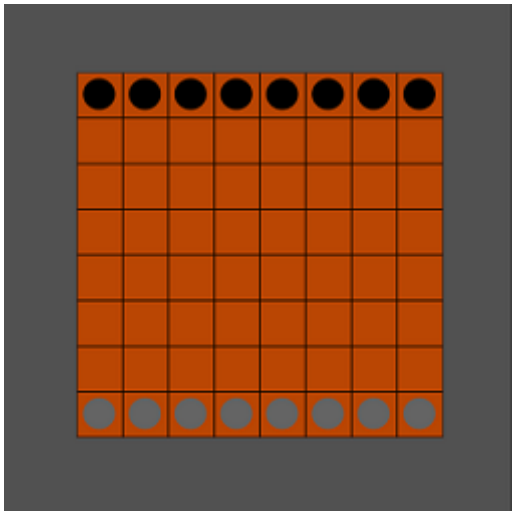

Abbildung 4.1: Prototyp

Erweiterte Features nach Testen der Logik

Die erweiterten Features wurden nach ausgiebigem Testen der grundsätzlichen Features und der implementierten Logik hinzugefügt.

- Künstliche Intelligenz als Gegenspieler
- Pausenmenü mit Neustart-, Beenden- und Start-Button
- Dynamisch generiertes Spielfeld für unterschiedliche Größen des Spielbretts
- Slider um die Punkteverteilung der künstlichen Intelligenz zu regulieren
- Touch-Unterstützung, da beim Landesmuseum ein Touchscreen verwendet wird, soll das Spiel die Bedienung ohne Maus und Tastatur unterstützen
- Simple Kreise durch Sprites von historisch verwendeten Spielsteinen ersetzen

Zusätzliche Features bei Übergabe an Kunden

Nachdem das zuvor konzeptionierte Spiel dem Landesmuseum Birkenfeld zugeschickt wurde, haben sich die folgenden noch zusätzlich benötigten Features herausgestellt:

- Unterschiedliche Schwierigkeitsgrade anstatt Slider im Spiel gegen die KI, da diese nicht so intuitiv verstanden wurden
- Zweite KI soll zuschaltbar sein, um Spiel zu simulieren
- Neustart- und Beenden-Buttons auf den Spielbildschirm verschieben

Benötigte Grafiken

Zur Umsetzung von Latrunculi wurden folgende Sprites genutzt:

- Spieler 1 als Muscheln
- Spieler 2 als Steine
- Quadratische Zellen als einzelne Felder des Spielbretts
- Quadratische Zellen mit Hervorgehobenen Rändern als Rückmeldung welche Bewegung möglich ist

Nötige Technologien

- Unity 2019.3.14f als Engine
- C# für die verwendeten Skripte
- Visual Studio 2019 als Entwicklungsumgebung
- GIMP zum erstellen und bearbeiten von Sprites

Künstliche Intelligenz

Um das Spielen auch alleine zu ermöglichen wurde auf Basis des MiniMax-Algorithmus eine künstliche Intelligenz entwickelt gegen die ein Spieler antreten kann. Hierbei können 3 verschiedene Schwierigkeitsgrade ausgewählt werden, indem die Punkteverteilung des Algorithmus und die Tiefe, die dieser berechnen soll, angepasst werden. Außerdem wurde für Vorführungen im Museum eine zweite KI mit einer eigenen Punkteverteilung hinzugefügt um das Spiel auch als Simulation laufen lassen zu können.

5. Implementierung

Aufgrund der unterstützenden Funktionen, wie das Portieren auf verschiedene Systeme, habe ich mich für die Unity Engine in der Version 2019.3.14f entschieden, um Latrunculi digital umzusetzen. Die Erklärung zu Unity befindet sich im Abschnitt 2.2. Weiterhin habe ich mich in C# eingearbeitet und damit die verwendeteten Skripte geschrieben und genutzt. Da ich vorher noch nicht mit Unity und C# gearbeitet habe, benötigte ich etwas Einarbeitungszeit um den ersten Prototypen zu erstellen.

5.1 Unity

Die ersten Versuche einer lauffähigen Umsetzung habe ich mit einem vorgefertigten Spielbrett umgesetzt. Dabei habe ich das Brett durch 8x8 einzelne Zellen beziehungsweise Quadrate dargestellt. Auf diesen Quadraten habe ich die Spielsteine als schwarze und graue Kreise platziert und diesen via Skript Funktionen gegeben. Dieser erste Entwurf (Abbildung 4.1) reagierte auf das OnClick()-Event, sodass beim Anklicken eines Kreises der mögliche Weg hervorgehoben wurde und beim Klick auf das gewünschte Feld, wurde der Kreis auf diesem neuen Feld platziert. Da dieser Entwurf sehr statisch und die Größe fest verankert war, habe ich mir Informationen gesucht um das Spiel dynamisch zu erstellen und die Größe des Spielfelds variabel zu machen. Dazu habe ich sowohl für die Zellen, als auch die Spielsteine Prefabs erstellt und somit mittels Skript in der Start-Funktion die Prefabs abgerufen und je nach angegebener Größe das Spielfeld zur Laufzeit erstellt. Außerdem wurde das OnClick()-Event durch Drag & Drop ersetzt, sodass es auf einem Touchscreen intuitiver zu bedienen ist.

5.2 GameObjects

Um die einzelnen Spielelemente darzustellen, müssen im Unity-Editor erst Spielobjekte (englisch: GameObject) erstellt werden. Die folgenden Objekte wurden dem Spiel hinzugefügt:

Main Camera

Die Kamera ist zuständig für das Sichtfeld, sodass hier festgelegt werden kann, was innerhalb der Szene sichtbar ist.

Board

Das **Board** ist das Objekt, dass das Spielbrett darstellt. Via Prefabs werden dem Board zur Laufzeit die einzelnen Zellen hinzugefügt und so das Brett aufgebaut.

PieceManager

Der **PieceManager** ist zuständig für die einzelnen Spielsteine, die ebenfalls via Skript und Prefab der Szene und dem PieceManager hinzugefügt werden.

GameManager

Der **GameManager** beinhaltet die Spiellogik und lässt das Spiel starten, sowie auch beenden. Hier wird nach jedem Zug geprüft ob ein Spieler gewonnen hat oder das Spiel noch weiter läuft. Weiterhin wird hier die Berechnung der KI angestoßen.

EscapeMenuManager

Der **EscapeMenuManager** ist zuständig für das Event-Handling des Menüs und der Buttons. Das heißt, hier wird entschieden, ob das Menü sichtbar und was in dem Menü zu sehen ist.

5.3 Skripte

Damit die einzelnen Objekte auch Funktionen und Informationen bekommen, musste jedem Objekt ein Skript hinzugefügt werden. Beispielsweise hat jeder Spielstein das Skript **SimplePiece** zugeordnet bekommen. Dieses ermöglicht dem Spielstein die Bewegungen auf dem Spielbrett durchzuführen. Es enthält Funktionen die auf das Ein- und Austreten des Drag&Drop-Events reagieren. Des Weiteren werden hier Informationen über die Ursprungszelle, sowie die aktuelle Zelle gespeichert um das Zurücksetzen des Spiels zu vereinfachen.
Das **PieceManager**-Skript ist für die Spielsteine insgesamt zuständig, das heißt hier werden die Prefabs und Skripte der Spielsteine instanziiert und der Szene hinzugefügt beziehungsweise auf dem Spielbrett platziert. Außerdem kann dieser Manager die Funktionen der Spielsteine ein und ausschalten, sodass nur mit den Steinen interagiert werden kann, die auch gerade am Zug sind, vorausgesetzt das Spiel befindet sich nicht im Simulations-Modus.
Der **EscapeMenuActivator** ist für das Ein- und Ausschalten des Menüs zuständig.
Der **GameManager** stößt jeden Spiel-relevanten Prozess an, speichert ob eine oder zwei KIs aktiviert sind und lenkt dementsprechend das Spielgeschehen. Das Spiel wird hier auch gestartet und beendet.
Das Spielbrett erhält das **Board**-Skript und initiiert den Aufbau des Spielbretts. Außerdem werden hier alle Zellen und deren Anordnung gespeichert, sodass das Board mögliche Züge validieren kann.
Für die Berechnungen der KI erbt das Skript **BoardDraught** die Funktionen und Variablen des Board-Skriptes, übernimmt aber nur die wichtigen Informationen.
Jede Zelle erhält ebenfalls ein eigenes Skript (**Cell**), dass die Position der Zelle beinhaltet und den Spielstein, der sich auf der Position befindet. Außerdem wird hier das Hervorheben der Zellen durchgeführt.
Die Klasse **Move** enthält Informationen für eine mögliche Bewegung, das heißt hier wird die aktuelle Position des gerade betrachteten Spielsteins gespeichert, sowie die Zelle auf welche sich der Stein bewegen kann. Weiterhin enthält diese verschiedene Flags, die zur Bewertung der Aktion dienen.
Im **AIManager** befindet sich der MiniMax-Algorithmus, der die Berechnungen und Bewertungen der möglichen Züge durchführt. Wie genau der Algorithmus funktioniert wird im nachfolgenden Abschnitt erklärt.

5.4 MiniMaxing

Für die Implementierung des MiniMax-Algorithmus habe ich die Klasse AIManager entworfen. Diese enthält die Funktion **Minimax()**, die sich an der beschriebenen Umsetzung von Jorge Palacios[8] orientiert. Diese übernimmt das aktuelle Spielbrett als Board, die maximale Suchtiefe, die aktuelle Tiefe und den Spieler der am Zug ist. Die Berechnungen und Evaluation der möglichen Züge werden anschließend hier rekursiv durchgeführt und es wird die maximal erreichte Punktzahl zurückgegeben, sowie die Referenzierung des zugeordneten Zuges.

Die rekursive Funktion **Minimax()** wird aufgerufen und bekommt sowohl den momentanen Spielzustand, die maximale & aktuelle Suchtiefe, als auch eine Referenzierung auf einen Zug, der nachher der beste zugeordnet wird, übergeben. Somit sind alle nötigen Informationen zur Berechnung vorhanden und es kann, wie in Abbildung 5.1 zu sehen ist, in Zeile 4 die Startbedingung geprüft werden. Falls das Spiel beendet ist oder die maximale Suchtiefe erreicht wurde, wird eine 0 zurückgegeben. Anschließend werden die möglichen Züge des gerade betrachteten Spielers innerhalb der **BoardDraught**-Klasse berechnet. Zeile 14 prüft, ob es sich um den ersten oder zweiten Spieler handelt, ruft dementsprechend die Züge des Spielers ab und speichert diese in der Variablen allMoves. In der ForEach-Schleife wird über die von diesem Zustand aus möglichen Züge iteriert. Dabei wird zuerst die Bewegung auf dem Spielbrett simuliert und auf diesem gespeichert, um nach jeder Iteration den Schritt rückgängig machen zu können. In Zeile 31 wird geprüft, ob der Spieler einen gegnerischen Stein erobern würde. Falls ein Angriff stattgefunden hat, wird der nächste Spieler auf den aktuellen Spieler gesetzt, sodass dieser in der darauffolgenden Suchtiefe wieder betrachtet wird. Zeile 38 startet die Rekursion mit dem neuen Spielzustand und speichert die Punktzahl in currentScore. Anschließend wird die aktuelle Bewegung evaluiert und die entsprechenden Punkte zurückgegeben und in newScore zwischengespeichert. Zum Schluss werden die Punkte aus newScore auf die aktuelle Punktzahl addiert, falls der betrachtete Spieler auch derjenige ist, der am Zug ist, oder subtrahiert, wenn der Gegenspieler dran wäre. Schließlich wird in Zeile 61 mit der Board-Funktion **StepBack()** der letzte Zug rückgängig gemacht und die Iteration wird fortgesetzt. Nach der Rekursion sollten alle Aktionen eine Bewertung erhalten haben. Da die Punktzahl teilweise identisch war, wurde immer der erste betrachtete Zug mit der maximal erreichten Punktzahl gewählt. Daher habe ich noch einen Zufallsgenerator implementiert, der eine zufällige Bewegung aus denen mit der höchsten und selben Punktzahl auswählt und zurückgibt.

Die **Evaluate()**-Funktionen sind für die Bewertung der Züge zuständig. Dabei ruft **Evaluate(int player)** nach der Entscheidung, welche Farbe dem Spieler zugeordnet wurde, die **Evaluate(string color)**-Funktion auf. Hier findet die eigentliche Punkteverteilung statt. Jeder Zug hat zuvor für verschiedene Spielsituationen Flags zugeordnet bekommen. Diese werden bei der Suche nach möglichen Bewegungen auf *true* gesetzt, wenn die definierte Bedingung zutrifft. Beispielsweise wird das *Attacked*-Flag gesetzt, wenn der Zug zu einer Eroberung führt und die Aktion erhält die vorher festgelegte Punktzahl für Angriffe. Das Flag *attacked2* wird gesetzt, wenn zwei Steine in diesem Zug erobert werden können. *Threaten* wird *true*, wenn die Aktion zu einer Bedrohung für den Gegenspieler führt. *HighThreat* wurde eingeführt um Bedrohungen, die im nächsten Zug zur Eroberung führen, besser bewerten zu können. Das *Hide*-Flag steht für den Abzug aus einer eigenen Bedrohung, also werden hier Punkte für ein defensives Verhalten verteilt. Zum Schluss wird geprüft

```
public static float Minimax(BoardDraught board, int player, int maxDepth,
int currentDepth, ref Move bestMove)
{
        if (board.IsGameOver() || currentDepth == maxDepth)
                return 0;
        bestMove = null;
        float bestScore = Mathf.Infinity;
        if (board.GetCurrentPlayer() == player)
        bestScore = Mathf.NegativeInfinity;
        List<Move> allMoves = new List<Move>();
        int nextPlayer = 0;
        if (player == 2){
                allMoves = board.GetMoves(player);
                nextPlayer = 1;
        }
        else if (player == 1){
                allMoves = board.GetMoves(player);
                nextPlayer = 2;
        }
        Move currentMove;
        float score = 0;
        foreach (Move m in allMoves){
                board.MakeMove(m);
                float currentScore = 0;
                currentMove = m;
                if (m.attacked) {
                        if (nextPlayer == 2)
                        nextPlayer = 1;
                        else
                        nextPlayer = 2;
                }
                currentScore = Minimax(board, nextPlayer, maxDepth,
                                currentDepth + 1, ref currentMove);
                float newScore = board.Evaluate(player);
                if (board.GetCurrentPlayer() == player) {
                        currentScore += newScore;
                        if (currentScore > bestScore)
                        {
                                bestScore = currentScore;
                                bestMove = m;
                                m.mScore = bestScore;
                        }
                }
                else{
                        currentScore -= newScore;
                        if (currentScore < bestScore){
                                bestScore = currentScore;
                                bestMove = m;
                                m.mScore = bestScore;
                        }
                }
                board.StepBack();
                if (m.attacked){
                        if (nextPlayer == 2)
                                nextPlayer = 1;
                        else
                                nextPlayer = 2;
                }
        }
        List<Move> bestMoves = new List<Move>();
        if (currentDepth == 0){
                foreach (Move m in allMoves){
                        if (m.mScore == bestScore){
                                bestMoves.Add(m);
                        }
                }
                System.Random rnd = new System.Random();
                int index = rnd.Next(bestMoves.Count);
                bestMove = bestMoves.ToArray()[index];
        }
        return score;
}
```

Abbildung 5.1: MiniMax

```
1  public float Evaluate(string color)
2  {
3          float eval = 1f;
4          int rows = sizeX;
5          int cols = sizeY;
6          if (currentMove.attacked)
7                  eval += pointAttacked;
8          if (currentMove.attacked2)
9                  eval += pointAttacked;
10         if (currentMove.threaten)
11                 eval += pointThreat;
12         if (currentMove.highThreat)
13                 eval += pointHighThreat;
14         if (currentMove.hide)
15                 eval += pointHide;
16         if (IsGameOver())
17                 eval += pointSuccess;
18         currentMove.mScore += eval;
19         return eval;
20 }
```

Abbildung 5.2: Evaluierungs-Funktion Version 1

ob das Spiel nach dem Zug beendet wäre und dementsprechend die Punktzahl dazu addiert. Die Punkteverteilung bestimmt auch den Schwierigkeitsgrad. Nach dem Testen dieses Bewertungssystem hat sich herausgestellt, dass diese Bewertung noch zu unpräzise ist. Um die KI noch intelligenter zu machen, wurden weitere wichtige Spielsituationen betrachtet und die Evaluierung um Kriterien ergänzt.

Verbesserung der Evaluierungs-Funktion

Die zuvor beschriebene Evaluierung wurde zu grob gestaltet, sodass wichtige Situationen nicht oder zu wenig betrachtet wurden. Dabei ist aufgefallen, dass die KI vor allem akute Bedrohungen der eigenen Steine nicht erkannt hat. Daher wurden die Flags „danger", „squadHide" & „isSquadHide", „prepSquadHide" & „isPrepSquad", „corner" & „isInCorner", „OOBAndFriendlyCorner" & „isOOBAndFriendlyCorner" und „highAlert" hinzugefügt wie in der Abbildung 5.4 zu sehen ist. Dabei steht „squadHide" für eine Vierer-Formation auf dem Spielfeld (vgl. Abbildung 5.3), die vom Gegner nicht erobert werden kann. Diese Formation ist aufgrund der fehlenden Möglichkeit eingenommen werden zu können, die wichtigste Verteidigungsstellung im Spiel. „PrepSquadHide" ist ein Flag, dass die Vorbereitung auf die Vierer-Stellung darstellt. Es steht also für eine L-Förmige Formation auf dem Spielfeld, wie die Steine es in 5.3 abbilden. Das Flag „danger" steht für eine akute Gefahr und wurde so implementiert, dass erkannt wird, wenn ein bedrohter Stein von einem weiteren gegnerischen erreichbar ist und beeinflusst die Puntkeverteilung insofern, dass bei aktivem Danger-Flag die Gewichtung der Verteidigungs-Flags des bedrohten Steins erhöht werden. Das Prefix „is" steht für das jeweils äquivalente Flag des Ist-Zustands und beeinflusst die Gewichtung der Punkte.

Analyse der Ausgangssituation

Das System wurde um eine Analyse der Ausgangssituation erweitert um Bedrohungen besser ausweichen zu können oder wichtige Positionen nicht zu verlassen. Hier wird geprüft ob der aktuelle Stein in Gefahr ist und dementsprechend die Gewichtung dieser Züge erhöht, um aus der Situation fliehen zu können. Des weiteren

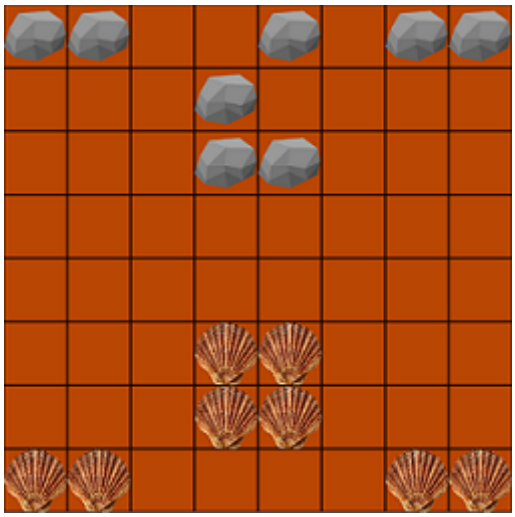

Abbildung 5.3: Vierer-Formation und L-Form

werden unschlagbare Positionen höher gewichtet. Befindet sich ein Stein in der Ecke des Spielfelds, wird die Gewichtung der Verteidigungs-Flags erhöht, sodass die Stellung nur für Eroberungen oder sichere Bedrohungen des Gegners verlassen wird. Eine sichere Bedrohung ist so definiert, dass ein Stein sich neben dem Gegenspieler positionieren kann ohne im nächsten Zug geschlagen werden zu können. Außerdem wurde das Flag „isInCorner" eingeführt, welches evaluiert, ob der Spielstein sich in einer Ecke befindet. Dann wird der Gesamtpunktzahl ein Wert x abgezogen um die Stellung nicht zu verlassen, außer es sind keine anderen Züge möglich oder es kann eine Spielfigur eingenommen werden. Außerdem wird in der Abbildung 5.4 Zeile 21 geprüft, ob der Stein sich am Rande des Spielfelds und neben einem weiteren eigenen befindet, da dies ebenfalls eine unschlagbare Position bildet.

```
public float Evaluate(string color)
{
        dangerMultiplier = 1f;
        attackWeight = 1f;
        if (danger)
        {
                dangerMultiplier = 2f;
        }
        if (isInCorner)
        {
                eval += -20;
        }
        if (isPrepSquad)
        {
                eval += -10;
        }
        if (isInCorner && attacked && !highAlert)
        {
                attackWeight = 3f;
        }
        if (isOOBAndFriendlyCorner)
        {
                eval += -50;
        }
        else if ((isOOBAndFriendlyCorner || isSquadHide) && threaten && !attacked)
        {
                attackWeight = 2f;
        }
        else if ((isOOBAndFriendlyCorner || isSquadHide) && threaten && !attacked)
        {
                attackWeight = 1.25f;
        }
        else if (isOOBAndFriendlyCorner && attacked && !highAlert)
        {
                attackWeight = 3f;
        }
        else if (isOOBAndFriendlyCorner && attacked && highAlert)
        {
                attackWeight = 3f;
        }
        else if (attacked)
        {
                attackWeight = 2f;
        }
        if (attacked)
                eval += pointAttacked * attackWeight;
        if (attacked2)
                eval += pointAttacked * attackWeight;
        if (attacked2)
                eval += pointAttacked * attackWeight;
        if (threaten)
                eval += pointThreat * attackWeight; ;
        if (hide)
                eval += pointHide * dangerMultiplier;
        if (squadHide)
                eval += pointSquadHide * dangerMultiplier;
        if (prepSquad)
                eval += pointPrepSquad * dangerMultiplier;
        if (corner)
                eval += pointCorner * dangerMultiplier;
        if (OOBAndFriendlyCorner)
                eval += pointOOBAndCorner*dangerMultiplier;
        if (highAlert)
                eval += pointHighAlert;
        if (highThreat)
                eval += pointHighThreat;
        if (IsGameOver())
                eval += pointSuccess;
        return eval;
        }
```

Abbildung 5.4: Evaluierungs-Funktion Version 2

6. Evaluation

Zur Evaluation wurde eine lauffähige Windows Version und Webversion dem Landesmuseum Birkenfeld zugesendet. Außerdem wurde das Verhalten der KI mit unterschiedlichen Konfigurationen gegen einen menschlichen Spieler, sowie als Simulation analysiert.

6.1 Launch beim Kunden

Die erste spielbare Version mit den grundlegenden und erweiterten Features wurde dem Landesmuseum mittels GitHub und E-Mail zugesendet, sodass diese Version trotz anhaltender Kontaktbeschränkungen vor Ort getestet werden konnte. Herr Decker hat uns während einem GoToMeeting-Videoanruf die Testversion auf ihrem Gerät zeigen können. Dabei wurde klar, dass die grundlegenden und erweiterten Features funktionieren. Es wurde festgestellt, dass erfahrene Spieler der implementierten KI überlegen sind und regelmäßig gewinnen. Das Spiel gegen die KI wurde zuvor von Personen, die sich nicht mit Latrunculi auseinandergesetzt haben, getestet und verloren, allerdings auch nicht in jeder Runde. Das Hervorheben der erlaubten Positionen wurde auf dem vorhandenen Gerät nicht wieder entfernt, wenn der Spieler seine Züge zu schnell erledigt hat. Außerdem sind die Spielsteine teilweise zwischen den Zellen hängen geblieben und nicht auf der korrekten Position eingerastet. Diese beiden Punkte waren auf meinen Testgeräten nicht reproduzierbar, daher liegt die Vermutung nahe, dass es ein hardwarespezifisches Problem des genutzten Touch-Tisches vor Ort im Museum ist. Aufgrund der Kontaktbeschränkungen ist es aktuell auch nicht möglich, die Version persönlich vor Ort zu testen, um Fehlerquellen einzugrenzen. Ein weiterer auffälliger Punkt war, dass das Menü auf dem Gerät vor Ort zu klein wirkte und sich nicht an die Bildschirmgröße angepasst hat. Meine Geräte haben maximal eine Größe von 22 Zoll, sodass es zuvor nicht möglich war, es auf einem vergleichbaren Gerät mit einer Größe von 55 Zoll zu testen. Ein weiterer Punkt, der sich gewünscht wurde, war, dass die KI deaktivierbar sein soll, damit das Spielen zwischen zwei Personen möglich ist.

6.2 Verhalten in der Simulation ohne Analyse der Ausgangssituation

Zur Analyse des Simulationsverhalten wurden verschiedene Konfigurationen ausprobiert und beobachtet. Die erste Einstellung hat eine Suchtiefe von 3 und folgende Punkteverteilung:

Spieler 1 (Muscheln):

- Angriffspunkte: 80
- Verteidigungspunkte: 50
- Bedrohungspunkte: 20
- Hohe Bedrohungspunkte: 0

Spieler 2 (Steine):

- Angriffspunkte: 100
- Verteidigungspunkte: 20
- Bedrohungspunkte: 50
- Hohe Bedrohungspunkte: 80

Beide KIs wirken mit diesen Einstellungen sehr aggressiv. Der erste Zug eines Spiels, ist einer auf ein Feld ohne Feindkontakt. Anschließend bedroht der zweite Spieler die bewegte Figur, indem der Spielstein sich unmittelbar daneben platziert. Im nächsten Zug wird der zuletzt bewegte Stein erobert. Insgesamt wirkt die KI auf beiden Seiten als würde sie möglichst viele gegnerische Figuren bedrohen und angreifen wollen, auch wenn es eine eigene Bedrohung zur Folge hat. Außerdem fällt auf, dass die Muscheln mehr Eroberungen erreichen. Hier lässt sich festhalten, dass in 7 von 10 Spielrunden die Muscheln gewonnen haben. Allerdings wird erkennbar, dass die Steine durch das zusätzliche Flag Bedrohungen der Muscheln einleiten, die dadurch theoretisch im nächsten Zug erobert werden könnten.

Suchtiefe: 10

Bei selber Konfiguration wie zuvor, aber erhöhter Suchtiefe, fällt auf, dass die Steine auch versuchen Muscheln zu bedrohen, die nicht in einem Zug erreichbar sind. Dieses Verhalten führt allerdings auch dazu, dass die Muscheln diesen zusätzlichen Zug teilweise ausnutzen, um die bewegte Figur zu erobern. Die Gewinnchance bei 10 gespielten Runden ist gleich verteilt, sodass beide jeweils 5 gewonnen haben. Aufgefallen ist dabei, dass es gegen Ende des Spiels zu Situationen kommen kann, in denen beide Spieler ihre verbliebenen Steine (hier waren es jeweils zwei) immer wieder hin und her bewegen, ohne dabei zu einem Angriff zu kommen. Die KIs bewerten die Bedrohungen in so einer Situation so hoch, dass sie sich dadurch zuerst scheinbar „festfahren".

Erhöhte Verteidigungspunkte und verminderte Angriffspunkte mit Suchtiefe 3

Spieler 1 (Muscheln):

- Angriffspunkte: 80
- Verteidigungspunkte: 50
- Bedrohungspunkte: 20
- Hohe Bedrohungspunkte: 70

Spieler 2 (Steine):

- Angriffspunkte: 20
- Verteidigungspunkte: 100
- Bedrohungspunkte: 50
- Hohe Bedrohungspunkte: 80

Für die folgende Beobachtung wurden die Verteidigungspunkte der Steine mit den Angriffspunkten der ersten Konfiguration vertauscht und den Muscheln wurde ebenfalls das Flag der Hohen Bedrohung mitgegeben. Die beobachteten 10 Runden endeten jedes mal mit einem Sieg der Muscheln. Dabei ist aufgefallen, dass das Verteidigungs-Flag Wirkung zeigt und die Steine sich aus Bedrohungen zurückziehen. Allerdings führt das weniger aggressive Verhalten dazu, dass die Steine im Endeffekt die Runden verlieren. Außerdem gab es die Situation aus Abbildung 6.1, in der die Muscheln nicht erkannt haben, dass die Steine eingekesselt werden können und somit das Spiel gewonnen wäre. Die Spieler haben ihre Spielsteine so lange bewegt, bis es für die Muscheln möglich war, einen zu erobern und dadurch das Spiel zu gewinnen.

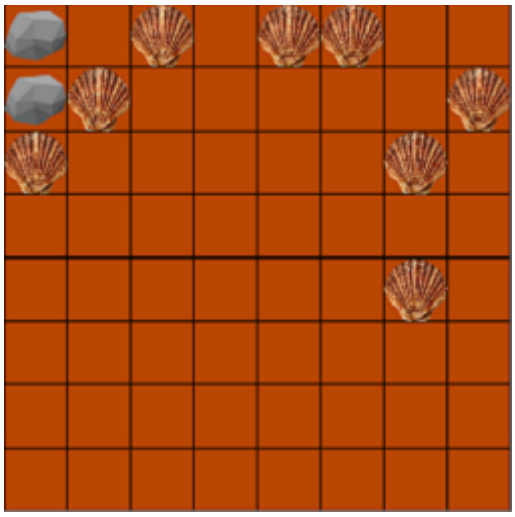

Abbildung 6.1: Spielende wird nicht erkannt

Suchtiefe 10

Bei erhöhter Suchtiefe haben die Steine 4 von 10 Runden gewonnen. Hier ist aufgefallen, dass mehr Angriffe über 2 Züge stattfinden. Dieses Verhalten kann auf die erhöhte Suchtiefe zurückgeführt werden. Dadurch können zukünftige Züge evaluiert werden und so Eroberungen die in späteren Zügen erst möglich sind durchgeführt werden. Insgesamt erweckt die KI allerdings den Eindruck, dass sie möglichst oft versucht gegnerische Spielsteine zu erobern. Aufgrund des zu aggressiven Verhaltens wurde die Analyse der Ausgangssituation implementiert, sowie weitere Flags für eine präzisere Bewertung.

6.3 Verhalten der KI ohne Analyse der Ausgangssituation gegen einen menschlichen Spieler

Spieler 2 (Steine):

- Angriffspunkte: 100
- Verteidigungspunkte: 20
- Bedrohungspunkte: 50
- Hohe Bedrohungspunkte: 80
- Suchtiefe: 3

Die KI hat mit diesen Konfigurationen eine sehr aggressive Spielweise gezeigt. Dabei ist aufgefallen, dass vor allem die Bedrohung und Angriffe im Fokus stehen. Mit dieser Konfiguration war es möglich den Computer mit simplen Spielzügen aus der Verteidigung zu locken und die Steine zu erobern. Es wurden regelmäßig Steine neben Muscheln platziert, auch wenn es im nächsten Zug eine eigene Eroberung zur Folge hat. Der Verteidigungsmechanismus kommt in dieser Konfiguration nur selten zur Geltung. Einerseits sind dazu die defensiven Punkte zu gering angesetzt, sodass diese kaum Auswirkungen auf die Spielzüge haben. Andererseits wurde hier auch eine geringe Suchtiefe gewählt, sodass die Verteidigung noch weniger ins Gewicht fällt und somit kaum Relevanz im Spiel zeigt. Um zu sehen, ob sich das Verhalten ändert, wurde die Suchtiefe auf 10 erhöht.

Spieler 2 (Steine):

- Angriffspunkte: 100
- Verteidigungspunkte: 20
- Bedrohungspunkte: 50
- Hohe Bedrohungspunkte: 80
- Suchtiefe: 10

Mit dieser Konfiguration wirkte die KI noch immer sehr aggressiv und hat kaum eine Möglichkeit ausgelassen, die Muscheln anzugreifen. Diese wurden in der Regel schnell wahrgenommen und umgesetzt, sodass allerdings im nächsten Zug wieder der angreifende Stein erobert werden konnte. Allerdings führten ein paar Bedrohungen dazu, dass die KI sich zurückgezogen und neben einem eigenen Stein eingereiht hat und nun nicht mehr so einfach einzunehmen war. Insgesamt wurde das Spielverhalten nach wenigen Runden vorhersehbar und konnte leicht überlistet werden.

Spieler 2 (Steine):

- Angriffspunkte: 80
- Verteidigungspunkte: 70
- Bedrohungspunkte: 50
- Hohe Bedrohungspunkte: 60
- Suchtiefe: 3

Für diese weiteren Tests wurden die Punktzahlen angepasst. Dabei wurde die Angriffs- und Bedrohungspunkte etwas reduziert und die Verteidigungspunkte erhöht, um dem defensiven Verhalten eine höhere Gewichtung zu geben. Diese Punkteverteilung machte sich auch im Verhalten bemerkbar. Die KI hat nicht mehr auf jede bewegte Muschel reagiert und sich so auch in der Verteidigung gehalten, allerdings wird schnell klar, dass dieses Verhalten zu passiv war und die Steine schnell eingekesselt werden konnten, sodass kein weiterer Zug mehr möglich war. Die KI hat eigene Bedrohungen noch immer nicht erkennen können, hat dafür aber nicht mehr auf jede Möglichkeit sich den Muscheln zu nähern reagiert. Anschließend wurde wieder die selbe Konfiguration mit der Suchtiefe 10 betrachtet.

Spieler 2 (Steine):

- Angriffspunkte: 80
- Verteidigungspunkte: 70
- Bedrohungspunkte: 50
- Hohe Bedrohungspunkte: 60
- Suchtiefe: 10

Mit dieser Veränderung hat die KI es geschafft, weniger durchdachte Züge zum eigenen Vorteil zu nutzen und Muscheln zu erobern. Allerdings passierte dies auf Kosten der eigenen Steine. Die KI wirkt insgesamt noch immer aggressiv und verliert in der Regel gegen geübte Spieler. Insgesamt hat sich gezeigt, dass die entwickelte KI noch zu primitiv ist und eigene Bedrohungen nicht zu erkennen scheint.
Daher wurden für weitere Analysen zusätzliche Kriterien implementiert, die unter anderem auch die aktuelle Spielsituation besser beschreiben und eigene Bedrohungen erkennen lassen. Das heißt, es wurde die Möglichkeit implementiert zu erkennen, dass ein Angriff zu eigenen Bedrohungen führen kann. Weiterhin wurden Situationen integriert, die dafür sorgen sollen, dass die Steine sich auch in Vierer-Formationen platzieren können. Diese Zählen zu den stärksten Zügen in diesem Spiel, da es schlicht nicht möglich ist Spielsteine aus dieser Formation zu erobern. Weiterhin wurden Positionen am Rand des Spielbretts und vor allem die Ecken höher priorisiert. Um das ganze System von der Situation abhängig machen zu können, wurden Multiplikatoren als Gewichtung eingesetzt, die während der Evaluierung variieren können.

6.4 Verhalten in der Simulation mit Analyse der Ausgangssituation

Für die folgenden Beobachtungen wurde die Simulation mit zusätzlichen gewichteten Analysekriterien betrachtet. Dabei gelten bei den ersten Tests die gleichen Punkteverteilungen für beide KIs und eine Suchtiefe von 10.
Dies spiegelt sich stark in ihrem Verhalten wieder. Sowohl die Steine, als auch die Muscheln finden sehr schnell die wichtigen Positionen um nicht geschlagen zu werden. Haben beide Ihre unschlagbaren Positionen gefunden, dann bewegen sie sich nur noch zwischen den Verteidigungspositionen hin und her wie in Abbildung 6.2 zu sehen ist. Dieses Verhalten hat sich endlos wiederholt.

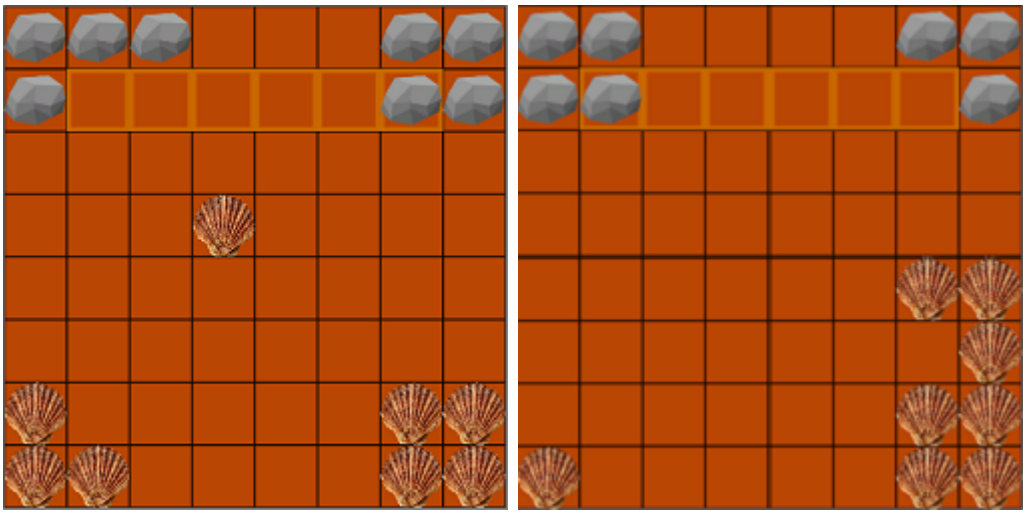

Abbildung 6.2: Steine und Muscheln mit der gleichen defensiven Konfiguration: Links mit Suchtiefe 10 und rechts mit Suchtiefe 3

Zum Vergleich wurde diese Konfiguration mit angepasster Suchtiefe auf 3 übernommen und ebenfalls als Simulation getestet. Aufgefallen ist, dass beide KIs ein paar Züge mehr benötigen, um diese Eck-Positionen, wie in Abbildung 6.2 zu sehen, zu finden und sich dort zu positionieren. Hier hat sich ebenfalls herausgestellt, dass diese Reduzierung nur dazu führt, die wichtigen Positionen später zu finden. Das Verhalten hat sich abgesehen von der längeren Suche nicht verändert. Daran wird deutlich, dass es nicht sinnvoll ist die KIs mit gleicher defensiver Konfiguration als Simulation laufen zu lassen. Hier kann man sich zwar wichtige Positionen verdeutlichen, allerdings nichts darüber hinaus, da das Spiel so auch nicht enden wird. Für den folgenden Testlauf wurden die Punkte für die Angriffs-Flags erhöht und beobachtet.

Erhöhte Angriffs- und verminderte Verteidigungspunkte mit Suchtiefe 10

Bei einer Suchtiefe von 10 fällt auf, dass die Muscheln zu Beginn sich eher in die Mitte des Felds bewegen, die Steine sich aber recht zügig in die Eckpositionen begeben. Dabei bilden erstere Reihenformationen, die eine wichtige strategische Formation im Spiel bilden, da die Steine nur noch von 2, statt von 4 Seiten angegriffen werden können. Die Simulation endet hier mit weniger aktiven Spielsteinen als im Test zuvor, allerdings fahren sich beide Spieler zum Ende hin wieder in den Eckpositionen fest.

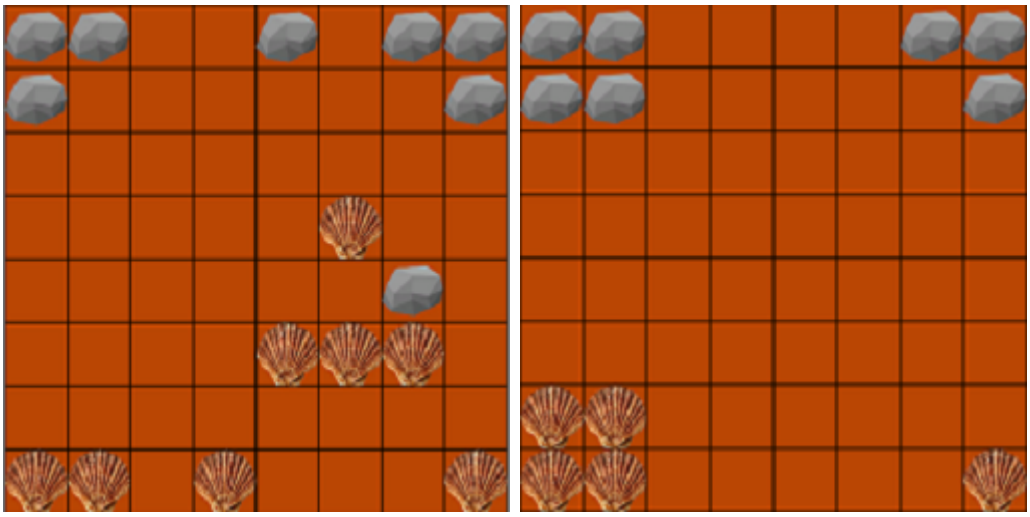

Abbildung 6.3: Steine und Muscheln mit der gleichen aggressiven Konfiguration und Suchtiefe 10: Links: Reihenformation & rechts: Endsituation.

Erhöhte Angriffs- und verminderte Verteidigungspunkte mit Suchtiefe 3

Bei diesen Einstellungen merkt man schnell, dass es länger dauert die Superpositionen zu finden. Die Spielsteine bewegen sich zuerst in die Mitte des Feldes, finden allerdings noch ihre verbündeten Steine, sodass sie sich nicht alleine offen ins Feld begeben. Allerdings tendieren sie nach wenigen Zügen dazu, sich wieder in die Ecken zu bewegen. Dabei stechen immer mal wieder Angriffe oder zumindest Angriffsversuche heraus. Zum Simulationsende hin fällt auf, dass immer wieder Versuche zur Bedrohung stattfinden, allerdings werden diese im darauffolgenden Zug wieder durch einen Rückzug hinfällig und die KIs befinden sich in einer Endlosschleife aus Bedrohung & Rückzug, wie es in der Abbildung 6.4 zu sehen ist.

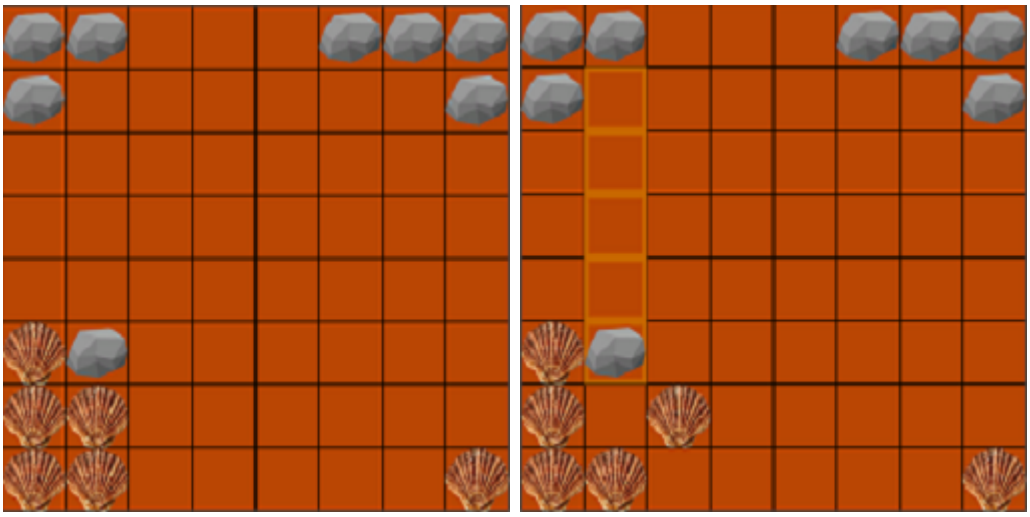

Abbildung 6.4: Steine und Muscheln mit der gleichen aggressiven Konfiguration und Suchtiefe 3: Endsituation.

Flags mit gleicher Punkteverteilung und Suchtiefe 10

Bei diesem Versuch wurden die verschiedenen Flags mit der gleichen Punktzahl belegt. Dabei ist aufgefallen, dass das Spiel insgesamt weniger intelligent verläuft

und die Spielsteine sich teilweise in bedrohliche Positionen begeben. Gute Positionen werden noch immer erkannt, allerdings scheinen die KIs direkte Bedrohungen immer identifizieren zu können. Vorteilhaft an dieser Konfiguration ist, dass es bei Vorführungszwecken unwahrscheinlicher wird, dass die Simulation sich in einer Situation festfährt und keiner der beiden gewinnt. Insgesamt erkennt man trotzdem noch, dass Positionen neben verbündeten Steinen eher aufgesucht werden als offene mitten im Spielfeld. Teilweise stellt die KI sich neben gegnerische Steine, wird aber im darauffolgenden Zug erobert.

Flags mit gleicher Punkteverteilung und Suchtiefe 3

Mit verminderter Suchtiefe wirkt das Spielgeschehen noch zufälliger. Gute Positionen wie die Eckpunkte werden erkannt, genauso wie Positionen am Rand des Spielfelds und neben verbündeten Steinen. Allerdings fällt hier wieder auf, dass akute Bedrohungen nur noch teilweise erkannt werden. Wie zuvor mit der Suchtiefe 10 begeben sich die Steine in eine Bedrohungsposition, auch wenn im nächsten gegnerischen Zug die Steine erobert werden können. Stellenweise wirkt es, als würden Spielsteine des Gegners auf der eigenen Seite nicht erkannt werden. Hier bewegen sich die Steine ein paar Runden lang hin und her, wie in Abbildung 6.5 bis die Muschel erobert wird.

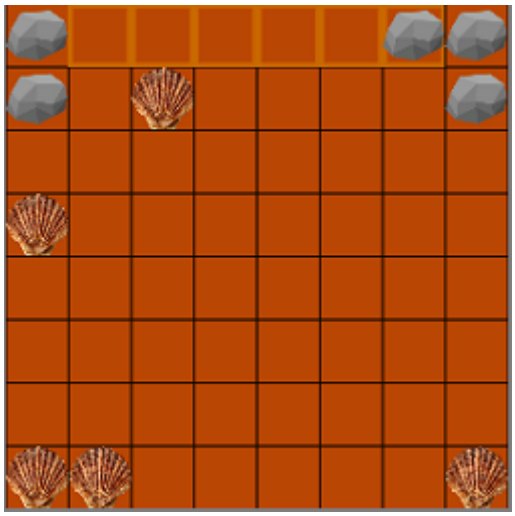

Abbildung 6.5: Steine und Muscheln mit der gleichen Konfiguration und Suchtiefe erkennen mögliche Eroberung spät.

Aggressive gegen Defensive KI

Lässt man die aggressive gegen die defensive Konfiguration spielen, werden die Spielmechaniken deutlicher. Die defensive KI wirkt insgesamt stärker und sucht sich schnell gute Spielpositionen in den eigenen Ecken. Die aggressive versucht vor allem zu Beginn noch oft Steine zu erobern, verliert bei dem Versuch aber oft eigene. Allerdings orientiert sich diese auch mit steigender Rundenzahl in die Ecken und an den eigenen Steinen. Zur Vorführung scheint diese Einstellung am sinnvollsten, da Spielmechaniken wie Eroberungen und Verteidigungen klar werden.

6.5 Verhalten der KI mit Analyse der Ausgangssituation gegen einen menschlichen Spieler

Die nachfolgenden Beobachtungen wurden bei einer Suchtiefe von 10 gemacht. Bei erhöhten Verteidigungspunkten und der Analyse des aktuellen Spielzustands hat das Spiel im Vergleich zu den Tests ohne Betrachtung der gegenwärtigen Situation bereits an Schwierigkeit gewonnen . Die KI macht einen intelligenteren Eindruck und lässt sich nicht mehr bei jeder Gelegenheit erobern. Weiterhin fällt hier auf, dass der Computer sich nach Angriffen auch wieder in eine sichere Position zurückzieht. Außerdem lassen die Steine sich auch nicht mehr aus ihrer Deckung locken, indem man die Muscheln einfach auf den mittleren horizontalen Reihen des Spielfelds zieht. Betrachtet man die Konfiguration mit verminderter Verteidigungs- und erhöhter Angriffsgewichtung, scheint das Spielen gegen die KI schwerer zu werden. Die Steine ziehen sich immer wieder in die Eckpositionen zurück und versuchen in den meisten Fällen nur anzugreifen, wenn der Angriff nach Berechnung auch sicher ist. Diese Konfiguration wirkt sinnvoller als in der Simulation mit gleichen Einstellungen. Das Verhalten lässt sich so erklären, dass bei gleicher Punkteverteilung beide KIs versuchen ihre Spielsteine zu schützen und es nur selten zu Bedrohungen oder Angriffen kommt. Die KIs ziehen sich zurück, sobald erkannt wird, dass ein Angriff auf den Stein möglich ist. Da menschliche Spieler dazu tendieren können, die gegnerischen Spielsteine zu erobern, kommt der Verteidigungsmechanismus des Computers mehr zur Geltung.

Erhöhte Angriffs- und verminderte Verteidigungspunkte mit Suchtiefe 10

Die KI hat sich mit diesen Einstellungen einkesseln lassen und so das Spiel schnell verloren. Insgesamt fällt auf, dass weniger defensiv gespielt wird als zuvor. Sie bewegt sich die meiste Zeit auf der eigenen Startseite, nutzt aber die Position clever aus, wenn sich ein Stein auf die gegnerische Spielseite bewegt hat. Die erhöhte Gewichtung der Angriffspunkte führt stellenweise zu Zügen in denen bis zu 3 Steine von mir erobert wurden.

Erhöhte Angriffs- und verminderte Verteidigungspunkte mit Suchtiefe 3

Bei geringerer Suchtiefe hat sich die KI durch ein aggressives Verhalten schnell besiegen lassen. Bedrohungen wurden zwar erkannt, allerdings hat der Computer vor allem mit vielen Spielsteinen auf dem Feld sich seltener dafür entschieden, eine Bedrohung nicht einzugehen, wenn im nachfolgenden Zug der bewegte Stein erobert werden kann. Dieses Verhalten lässt sich auf die geringe Suchtiefe zurückführen. Gegen Ende des Spiels ist die Situation aus der Abbildung 6.6 aufgefallen. Die KI hat sich nur noch in den Eckpositionen bewegt und hat sich nicht mehr in eine Situation bringen lassen, in der ich einen Stein erobern konnte.

Ausgeglichene Punkteverteilung

Diese Einstellungen werden hauptsächlich durch die Gewichtung via Multiplikatoren beeinflusst. Das fällt auch im Spiel auf. Die Züge wirken teilweise nicht zielführend, wie zum Beispiel, wenn man sich neben einen computer-gesteuerten Stein platziert, entscheidet sich die KI stellenweise dafür diesen Stein durch eine Eroberung zu

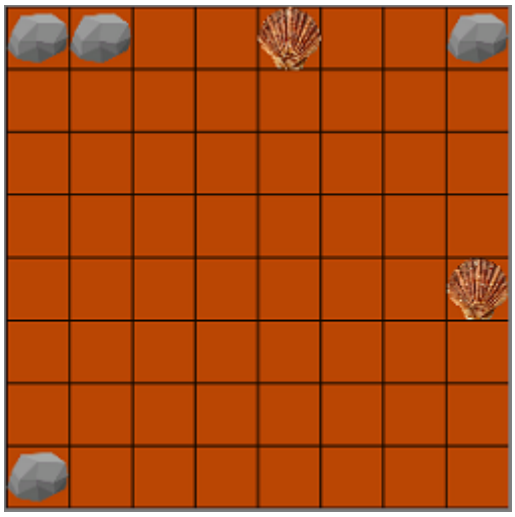

Abbildung 6.6: Hohe Angriffspunkte bei Suchtiefe 3. KI weicht in die Ecken aus.

verlieren und bringt ihn nicht in Sicherheit. In der Abbildung 6.7 erkennt man ein großes Problem bei diesen Einstellungen. Die KI hat sich zu Beginn des Spiels auf der eigenen Spielseite einkesseln lassen, sodass es nach nur einer Eroberung beendet war. Diese Situation entsteht dadurch, dass die KI sich mit ihrem ersten bewegten Stein nur eine Position nach vorne bewegt. Anschließend wird der Stein nur noch horizontal von einer Ecke zur nächsten geschoben. Diese Konfiguration ist bis hierhin die am wenigsten zielführende. Mit geringerer Suchtiefe lässt die KI sich ebenfalls wie in der Abbildung 6.7 einkesseln und verliert dadurch das Spiel. Insgesamt wirkt diese Konfiguration sehr zurückhaltend, da die KI kaum versucht gegnerische Steine zu erobern.

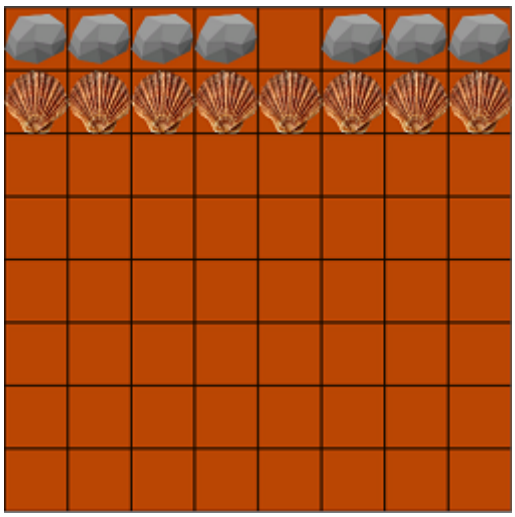

Abbildung 6.7: KI lässt sich einfach einkesseln.

7. Alpha-Beta-Algorithmus

Um den Algorithmus effizienter zu gestalten, gibt es unterschiedliche Verfahren. Der Alpha-Beta-Algorithmus ist eine Möglichkeit eine geringere Laufzeit zu erzielen, indem frühzeitig Suchen abgebrochen werden, sobald absehbar ist, dass es eine bessere Alternative gibt. Außerdem kann eine Verbesserung im Spielverhalten erzielt werden, wenn man mehr Bewertungskriterien einführt und so die Zustände präziser einordnen kann.
Der Alpha-Beta-Algorithmus ist eine Verbesserung des MiniMax-Suchverfahrens. Dabei funktionieren beide prinzipiell ähnlich, allerdings werden durch ersteren im besten Fall die betrachteten Knoten von $2N$ auf $2N^{1/2}$ reduziert und soll so schneller in der Suche sein, wie es Griffith erläutert hat[4].
Dabei entspricht Alpha dem besten bereits gefundenen Zug des maximierenden Spielers. Beta dem besten bisher gefundenen Zuges des minimierenden Spielers. Im Beispiel der Abbildungen 7.1 bis 7.8 kann das Prinzip leicht verständlich gemacht werden. Dabei entspricht der Spieler mit den Steinen dem maximierenden Spieler und der andere dem minimierenden. Alpha wird zu Beginn mit $-\infty$ und Beta mit ∞ festgelegt. Abbildung 7.1 zeigt den ersten Schritt. Dabei wird der linke Teilbaum zuerst durchsucht. Der betrachtete Spielzug des minimierenden Spielers hat eine Wertung von -30, deshalb wird in Abbildung 7.2 Beta mit -30 belegt. Nun wird das nächste Kind mit der Wertung -40 betrachtet und mit β verglichen. -40 ist kleiner als -30, deshalb erhält β in Abbildung 7.3 den kleineren Wert. Dies entspricht dem kleinsten erreichbaren Wert in dieser Situation und wird deshalb den Pfad entlang weitergereicht. In Abbildung 7.4 wird α dementsprechend mit -40 als Höchstpunktzahl des maximierenden Spielers gespeichert. Auf dem nächsten Ausschnitt 7.5 sieht man, dass α mit -40 als Vergleichswert gesetzt und β wieder mit ∞ weitergereicht wird. Der Zug des maximierenden Spielers hat eine Punktzahl von +30 erhalten und der betrachtete Folgezug des Gegenspielers eine -40, daraus ergibt sich für β -10. Wird der nächste mögliche Zug betrachtet, erreicht der minimierende Spieler eine -100 und wir erhalten ein β von -70. Da -70 kleiner als α ist, kann, wie in Abbildung 7.7 zu sehen ist, der restliche Teilbaum gestrichen werden. Der Algorithmus geht immer von den bestmöglichen Zügen aus, daher ist es nicht mehr nötig die weiteren Zustände aus diesem Teilbaum zu betrachten. Die letzte Abbildung 7.8 zeigt die Betrachtung des dritten Baums. Hier erreicht β wieder den Wert -70, sodass hier ebenfalls frühzeitig abgebrochen werden kann und wir erhalten -40 als maximale Punktzahl. Das Abbrechen der Suche führt zu der vorher erwähnten Verkürzung der Laufzeit.

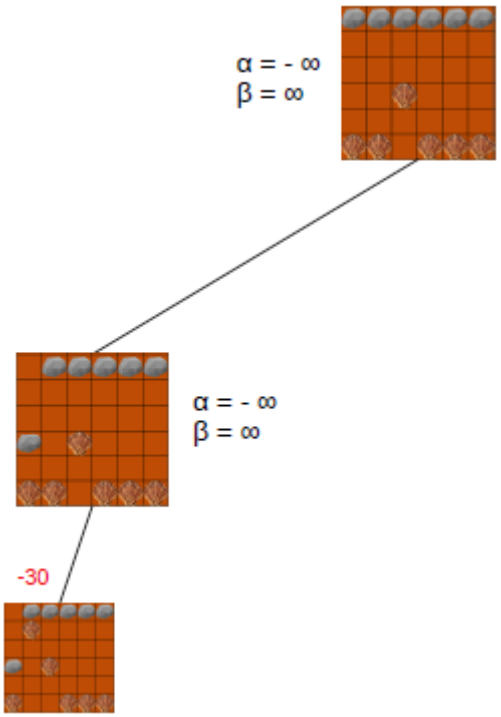

Abbildung 7.1: Alpha-Beta Pruning: Schritt 1

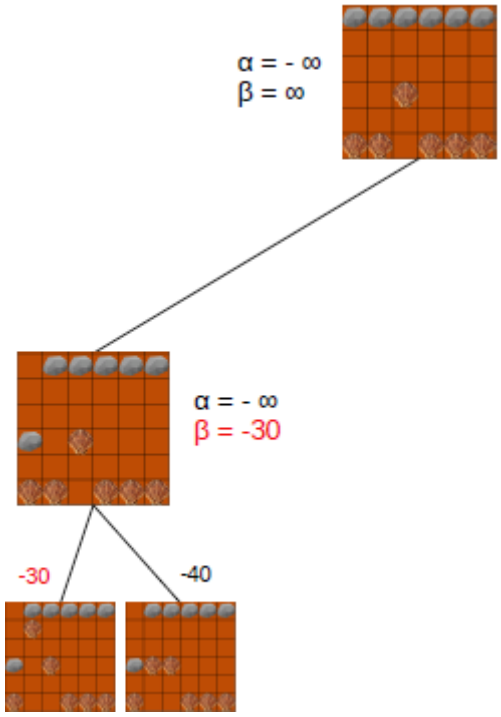

Abbildung 7.2: Alpha-Beta Pruning: Schritt 2

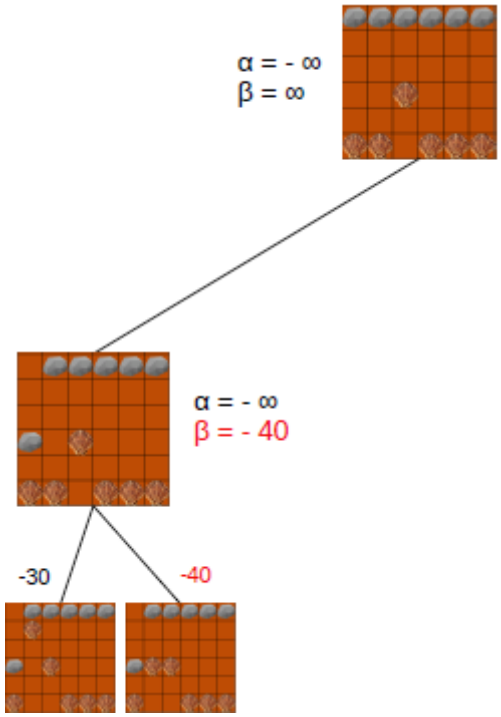

Abbildung 7.3: Alpha-Beta Pruning: Schritt 3

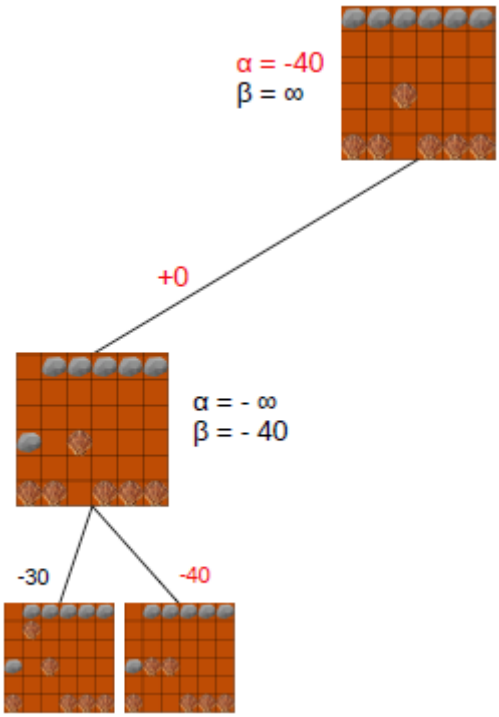

Abbildung 7.4: Alpha-Beta Pruning: Schritt 4

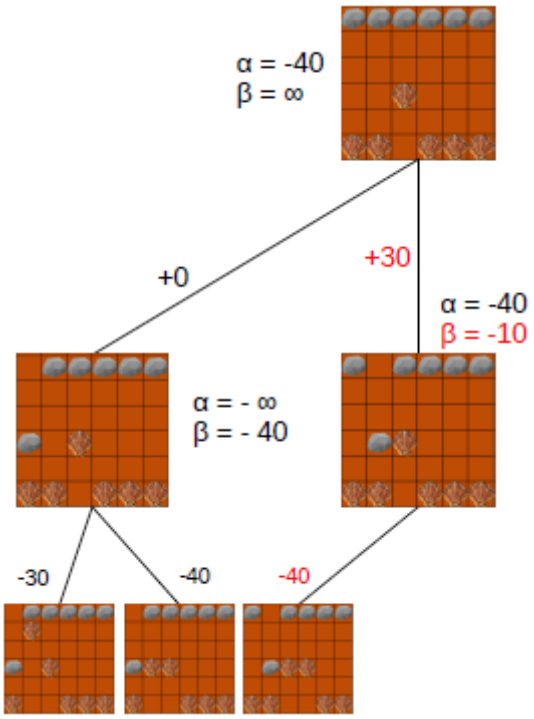

Abbildung 7.5: Alpha-Beta Pruning: Schritt 5

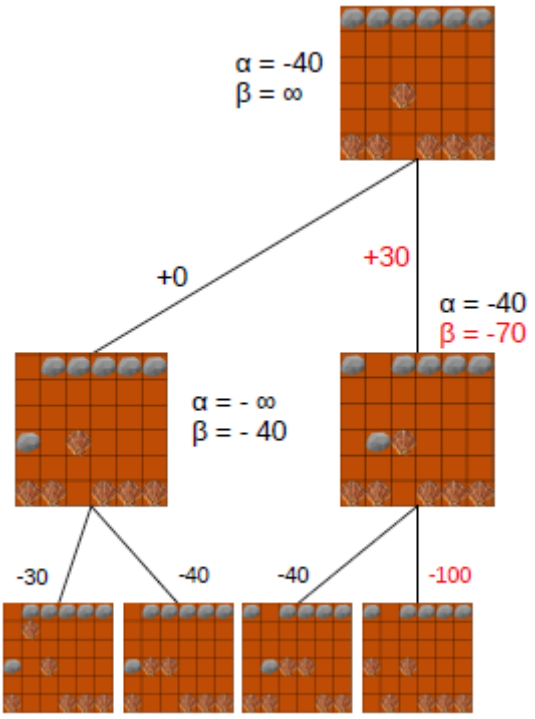

Abbildung 7.6: Alpha-Beta Pruning: Schritt 6

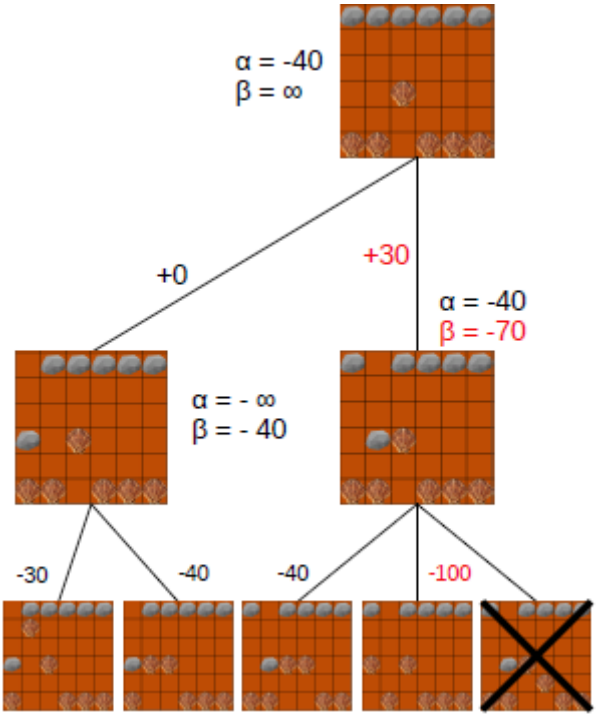

Abbildung 7.7: Alpha-Beta Pruning: Schritt 7

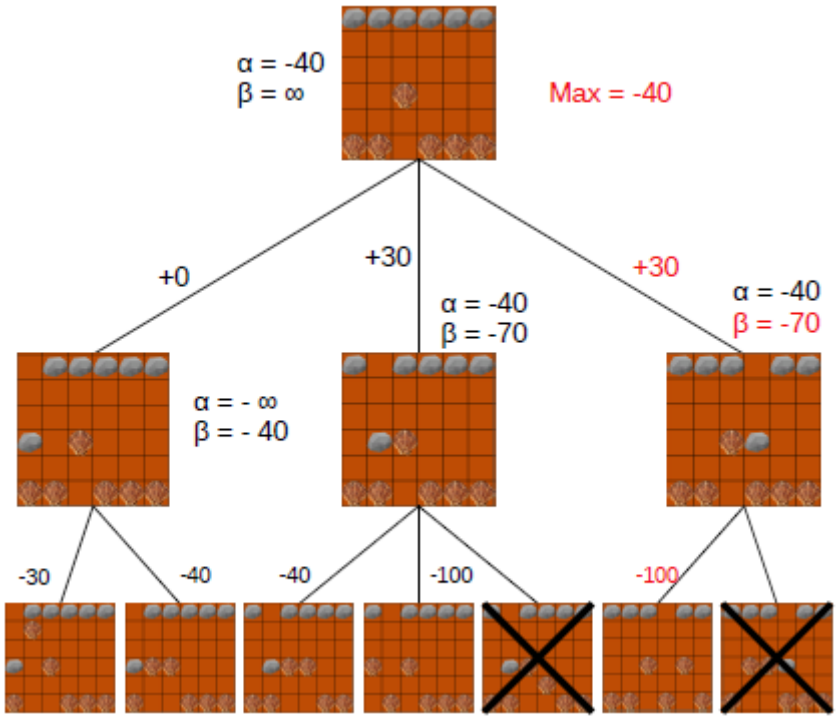

Abbildung 7.8: Alpha-Beta Pruning: Schritt 8

8. Diskussion und Ausblick

Latrunculi wurde in dieser Arbeit digital umgesetzt. Dabei wurden verschieden Einstellungen ausprobiert um zu evaluieren wie zielführend sich die implementierte KI verhält. Es wurden aggressive, defensive und ausgeglichene Konfigurationen mit verschiedenen Suchtiefen evaluiert. Bei den Versuchen ist aufgefallen, dass vor allem die ausgeglichene Variante weniger sinnvolle Spielzüge umsetzt. Am schwierigsten gegen den Computer zu gewinnen, war es mit defensiven Einstellungen. Hier hat die KI sich schnell in unschlagbare Positionen begeben und diese nur verlassen, wenn es zu einer Eroberung geführt hat. Die aggressive Konfiguration hat dafür stellenweise Spielzüge durchgeführt in denen innerhalb von drei aufeinanderfolgenden Zügen Steine erobert wurden. Dadurch, dass es möglich war die KI einzukesseln und zu gewinnen, wurde diese Einstellung als mittlerer Schwierigkeitsgrad festgelegt und die defensive Strategie als schwer. Die ausgeglichene Konfiguration wurde als leicht klassifiziert. Außerdem ist aufgefallen, dass eine höhere Suchtiefe zu insgesamt zielführenden Spielzügen führt. In der Simulation wurden die KIs zuerst mit jeweils den selben Einstellungen evaluiert. Das Problem dabei ist, dass beide die selbe Strategie verfolgen und so keine zielführenden Spielzüge Zustande kommen. Daher wurde für eine KI die defensive und für die andere die aggressive Konfiguration gewählt. Hier wurde beobachtet, dass die implementierten Spielmechaniken deutlicher werden. Die aggressive Einstellung versucht zu Beginn noch Steine anzugreifen, verliert dabei stellenweise aber eigene. Die andere KI orientiert sich schnell an den Eckpositionen und nutzt das aggressive Verhalten zum eigenen Vorteil. Da die Simulation vor allem für Vorführungszwecke genutzt werden soll, scheint diese Konfiguration am sinnvollsten zu sein.
Zusammenfassend kann man festhalten, dass die Schwierigkeit in der Implementierung einer zielführenden KI in der Evaluierungsfunktion liegt. Diese muss möglichst präzise die Situation analysieren können um daraus gute Spielzüge zu evaluieren. Das Problem bei Latrunculi ist, dass wenige Spielmechaniken und Strategien überliefert wurden, daher gibt es einen großen Spielraum an möglichen Umsetzungen. Dadurch, dass es auch zu vorzeitigen Einigungen auf den Sieger gekommen sein soll, macht es es schwieriger eine zielführende Simulation umzusetzen. Ein weiterer Punkt zur Verbesserung ist die Umsetzung des Alpha-Beta Algorithmus, der die Berechnung des MiniMax abkürzt und dadurch eine kürzere Laufzeit erreicht wird. Allerdings hat das wenig Auswirkungen auf die Effektivität der KI.

Literaturverzeichnis

[1] Ludus Latrunculorum or Latrunculi . Rules for the classic Roman game, 2011. https://www.mastersofgames.com/rules/Ludus-Latrunculorum-Rules.pdf.

[2] *Unity Documentation*, 2018. Eingesehen am 30.01.2021.

[3] Daniel James Edwards and TP Hart. The alpha-beta heuristic. Technical report, 1961.

[4] Arnold K. Griffith. Empirical exploration of the performance of the alpha beta tree-searching heuristic. *IEEE Trans. Computers*, 25(1):6–11, 1976.

[5] Maria Hartmann. Der alpha-beta-algorithmus, 2017.

[6] Feng-Hsiung Hsu. Ibm's deep blue chess grandmaster chips. *IEEE Micro*, 19(2):70–81, 1999.

[7] Feng-Hsiung Hsu. Chess hardware in deep blue. *Comput. Sci. Eng.*, 8(1):50–60, 2006.

[8] Jorge Palacios. *Unity 5. x Game AI Programming Cookbook*. Packt Publishing Ltd, 2016. Seiten: 182-184. ISBN 978-1-78355-357-0.

[9] Ulrich Schadler. Latrunculi - ein verlorenes strategisches Brettspiel der Romer. *HOMO LUDENS - Der spielende Mensch*, pages 47–67, 1994.

[10] Katharina Uebel und Peter Buri. *Römische Spiele - So spielten die alten Römer*. Regionala Verlag, second edition. Euskirchen, Germany, ISBN 978-3-939722-32-8.

[11] Kowalski Wally J. Roman board games, 2000. http://www.comscigate.com/HW/cs302/acsl/2007-2008/contest_3/Petteia%20Rules.htm.

[12] Georgios N Yannakakis and Julian Togelius. *Artificial intelligence and games*, volume 2. Springer, 2018. Seite: 45, ISBN 978-3-319-63518-7.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Bachelor-/Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vor-gelegt. Sie wurde bisher auch nicht veröffentlicht.

Trier, den xx. Monat 20xx