Implementierung eines historischen Brettspiels 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

Hier steht eine Kurzzusammenfassung (Abstract) der Arbeit. Stellen Sie kurz und präzise Ziel und Gegenstand der Arbeit, die angewendeten Methoden, sowie die Ergebnisse der Arbeit dar. Halten Sie dabei die ersten Punkten eher kurz und fokussieren Sie die Ergebnisse. Bewerten Sie auch die Ergebnissen und ordnen Sie diese in den Kontext ein.

Die Kurzzusammenfassung sollte maximal 1 Seite lang sein.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Die Einleitung besteht aus der Motivation, der Problemstellung, der Zielsetzung und einem erster Überblick über den Aufbau der Arbeit.

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, da Museumsbesuche nur eingeschränkt möglich sind. Latrunculi wurde implementiert, weil es aufgrund von nicht eindeutig überlieferten Regeln einen gewissen Spielraum in der Umsetzung. Außerdem auch historisch interessante Informationen liefert. Weiterhin bietet das Landesmuseum Birkenfeld Veranstaltungen (https://www.landesmuseum-birkenfeld.de/landesmuseum/erlebnismuseu wie „So spielten die Römer" für Schulklassen an und kann durch eine digitale und spielbare Umsetzung eines historischen Spiels bei den Besuchern ein größeres Interesse wecken.

1.2 Problemstellung

Latrunculi war historisch gesehen ein beliebtes und weit verbreitetes strategisches Brettspiel der Römer und Griechen gewesen sein. Die römischen Dichter Ovid und Martial haben dieses Spiel bereits erwähnt und beschrieben, dass bei sehr talentierten Spielern häufig Zuschauer anwesend waren. Die Spielsteine wurden als latrones(lat. für Soldat) bezeichnet. Da die Entstehung dieses Spiels so lange zurück liegt, wurden nicht alle Regeln überliefert, sodass verschieden Quellen hinzugezogen werden müssen um den Spielmechanismen zu klären. 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. 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. Betrachtet man Ovids Aussagen, kann man folgern, dass das Hauptziel darin bestand mit seinen Spielsteinen einen gegnerischen von zwei gegenüberliegenden Seiten zu einzufangen und somit aus dem Spiel zu nehmen. Weiterhin beschreibt er, dass es wichtig sei, seine Steine paarweise zu bewegen, da dadurch verhindert wird, dass diese vom Gegner geschlagen werden können. Für

diesen Angriffs- und Verteidigungsmechanismus konnten die Steine geradlinig vorwärts und rückwärts verschoben werden. Allerdings ist auch nicht eindeutig erklärt, wann das Spiel endet oder ob es mit speziellen Figuren funktioniert hat, beispielsweise gibt es Variationen mit einem König mit speziellen Fähigkeiten, ähnlich wie die Dame beim Schach, zwischen den simplen Figuren beziehungsweise Steinen.

1.3 Zielsetzung

In dieser Arbeit wird eine nachfolgend erklärte Variation des Spiels Latrunculi mithilfe der Unity Engine für das Landesmuseum Birkenfeld umgesetzt. Dabei wurde eine Künstliche Intelligenz implementiert, sodass es die Möglichkeit gibt, gegen eine KI zu spielen, die zielführende Züge umsetzt. Weiterhin soll das entwickelte Spiel die Möglichkeit bieten, zwei KI's als Simulation gegeneinander antreten zu lassen. Zum Zeitpunkt der Arbeit, befinden wir uns aufgrund von Covid-19 in einer Pandemie, sodass aktuell Kontaktbeschränkungen den Alltag bestimmen, daher soll das Projekt auch die Möglichkeit bieten online abgerufen zu werden. Unity bietet relativ einfache Portierungs-Möglichkeiten, sodass die Entwicklung eines Spiels für unterschiedliche Plattformen erleichtert wird.

1.4 Gliederung/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 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 wird das Konzept, 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.

2. Grundlagen

2.1 Spielregeln

Der grundsätzliche Aufbau des Spiels beinhaltet ein Spielbrett mit beispielsweise 7x6 Feldern bis hin zu überlieferten 12x12 großen Brettern. Weiterhin gibt es zwei verschieden Spielsteine, die entweder unterschiedliche Farben haben oder aus zum Beispiel einerseits Steinen oder 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.

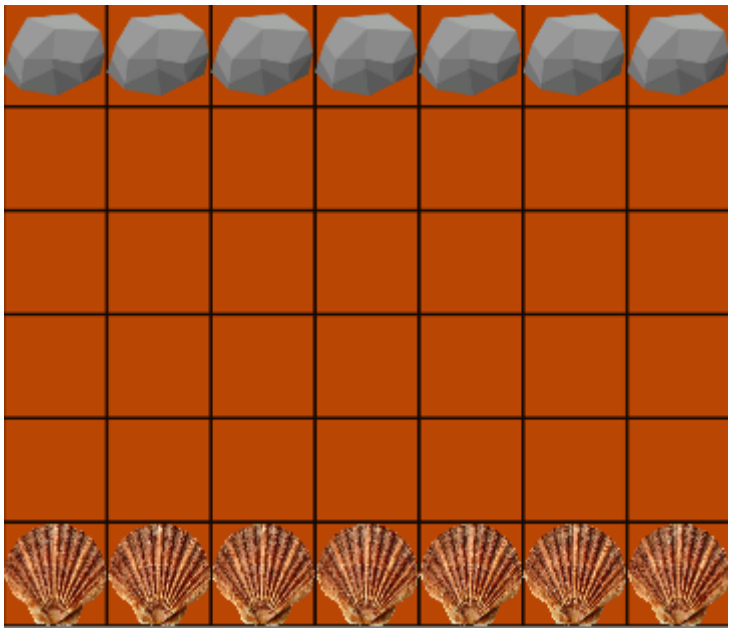

Abbildung 2.1: Startzustand

Das Spiel beginnt in dem die Spieler abwechselnd einen Stein horizontal oder vertikal über das Feld bewegen, ohne dabei einen anderen Stein zu überspringen oder auf einem besetzten Feld zu landen.

Um gegnerische Spielsteine zu erobern, müssen diese von eigenen Steinen auf zwei gegenüberliegenden Feldern umstellt sein.

Wird ein Stein erobert, dann darf ein weiterer Stein sich bewegen. Hierbei gibt es auch Variationen bei denen nur der zuletzt bewegte Stein sich wieder bewegen darf.

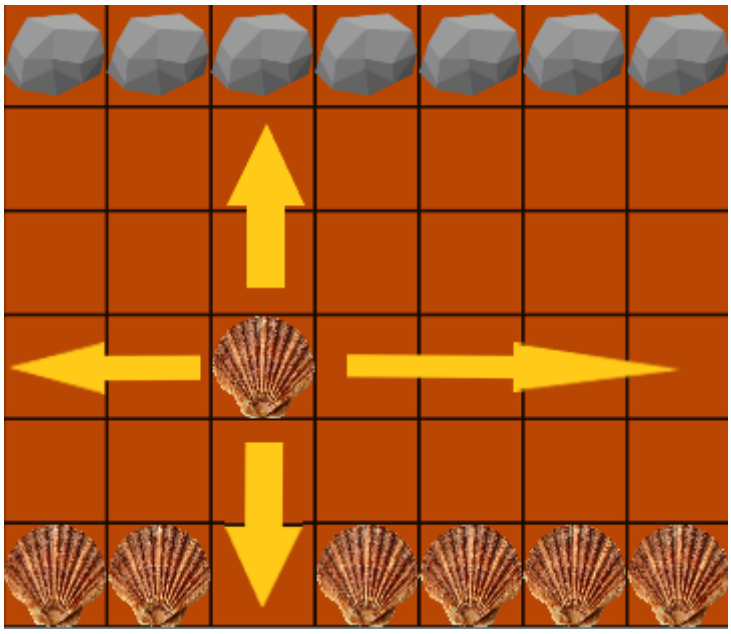

Abbildung 2.2: Legitime Bewegungen

Bewegt sich ein eigener Stein zwischen zwei gegnerische und scheint somit umstellt zu sein, zählt dieser allerdings nicht als erobert.

Das Ziel des Spiels ist es, dass der Gegenspieler keinen Zug mehr ausführen kann beziehungsweise keinen Spielstein mehr erobern kann.

2.2 Künstliche Intelligenz

Der amerikanische Informatiker John McCarthy prägte 1956 den Begriff künstliche Intelligenz (KI) auf der Dartmouth Conference. Künstliche Intelligenz findet heutzutage in vielen Bereichen Anwendung und ist eine immer wichtigere Entwicklung, um menschliche Arbeit zu vereinfachen und auch auf Computer zu übertragen. Seit der Begriff geprägt wurde, haben sich die dazugehörigen Bereiche erweitert. Heute definiert KI die Automatisierung von Prozessen bis hin zur Robotik. Hierbei geht es vor allem um den Zusammenhang zwischen Berechnung und Wahrnehmung. Aufgrund der riesigen Datenmengen (Big Data), die ein Mensch alleine nicht bearbeiten oder analysieren kann, wird dieser Bereich immer wichtiger. Maschinen können viel effizienter Daten analysieren und Muster erkennen, als ein Mensch es könnte.
Um die unterschiedlichen Typen zu kategorisieren, kann man sich die vier verschiedenen Typen, die von Arend Hintze, Assistenzprofessor für Integrative Biologie und Informatik der Michigan State University klassifiziert wurden, betrachten:

Typ 1: Reaktive Maschinen

Zu den Reaktiven Maschinen zählen solche, die den aktuellen Zustand kennen und zukünftige Entscheidungen analysieren und bewerten können. Als Beispiel kann man sich hier Deep Blue anschauen. Deep Blue ist ein von IBM entwickeltes Schachprogramm und ist in der Lage die Figuren auf dem Spielbrett zu erkennen und mögliche Züge beider Spieler analysieren. Da Deep Blue keine lernende Komponente enthält, werden vergangenen Spielsituationen nicht betrachtet und haben keinen Einfluss auf zukünftige Züge.

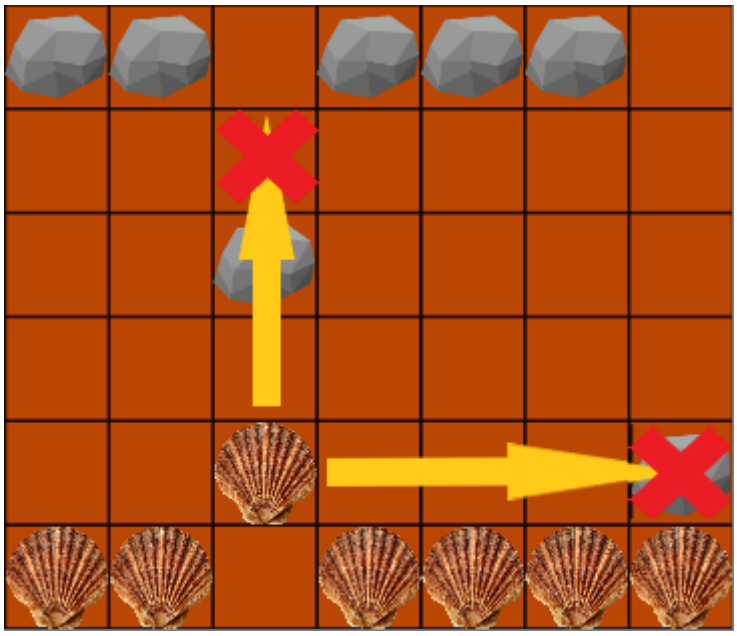

Abbildung 2.3: nicht zulässige Bewegung

Typ 2: Begrenzter Speicher

Zum zweiten Typen zählt man Systeme, die vergangene Erfahrungen nutzen können, allerdings werden hier Beobachtungen aus naher Zukunft nicht gespeichert. Als Beispiel können hier autonome Fahrzeuge betrachtet werden, die den Wechsel der Fahrspur eines Autos nicht dauerhaft speichern.

Die nachfolgenden beiden Typen existieren in der beschriebenen Form noch nicht:

Typ 3: Native Theorie

Die Native Theorie ist ein psychologischer Begriff und bezeichnet Systeme, die verstehen, dass andere eigene Überzeugungen, Wünsche und Absichten haben, die ihre Entscheidungen beeinflussen.

Typ 4: Selbsterkenntnis

Diese Kategorie umfasst Maschinen die ein Bewusstsein haben.Die sollen in der Lage sein ihren aktuellen Zustand zu verstehen und daraus Erkenntnisse über das Gefühl anderer zu ziehen.

In dieser Ausarbeitung wurde im Zusammenhang mit dem Spiel Latrunculi eine künstliche Intelligenz des Typ 1 entwickelt, die ihren aktuellen Zustand kennt und daraus Berechnungen für zukünftige Züge treffen kann.

2.2.1 Künstliche Intelligenz in Videospielen

In Videospielen werden in der Regel Reaktive Maschinen verwendet. Hierbei werden mögliche Bewegungen entweder vorher definiert und abgearbeitet oder aber die Aktionen werden mit einem Punktesystem bewertet und abhängig von der Punktzahl

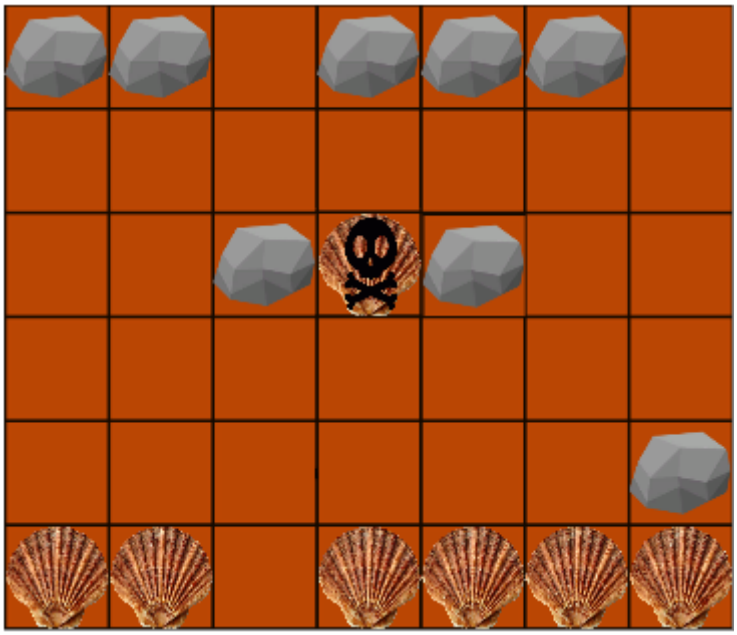

Abbildung 2.4: Erobern

ausgeführt. Dabei lässt sich festhalten, dass je mehr Bewertungskriterien die Aktionen bekommen, desto intelligenter wirkt die Künstliche Intelligenz. Betrachtet man die Entwicklung der Videospiele der letzten Jahre im Allgemeinen, fällt auf, dass vor Allem die Grafik weite Sprünge gemacht hat, im Vergleich dazu aber die eingesetzte KI sich nicht wesentlich verändert hat.
Als Beispiel kann man sich das Strategiespiel Command & Conquer aus dem Jahre 1996 anschauen, sowie den Nachfolger Command & Conquer (2007) erkennt man das Problem das sich hier abzeichnet. Während sich die Grafik innerhalb der 11 Jahre wesentlich verbessert hat, verhält sich die KI noch immer ähnlich und es passieren die selben Fehler wie schon beim ersten Teil. Spieler müssen in diesem Spiel ein Hauptquartier aufbauen und von diesem aus die gegnerischen Spieler ausschalten, dabei müssen Rohstoffe gesammelt werden um die Basis weiter auszubauen. Zur Rohstoffgewinnung verfügt man über so genannte Ernter, welche automatisch den Rohstoff Tiberium suchen. Bei der Suche fahren diese teilweise ins gegnerische Hauptquartier und werden dort zerstört. Im Nachfolger, elf Jahre später, verhalten sich die Ernter noch immer ähnlich und die Fehler passieren hier ebenfalls, allerdings mit dem Unterschied, dass das Spiel, der Ernter und die Explosions-Animation grafisch realistischer aussehen. Wenn man hier eine lernende Komponente implementieren würde, dann könnten solche Probleme der KI zwar noch passieren, aber auf Dauer wäre Sie in der Lage, das als Fehler zu erkennen.

MiniMax-Algorithmus

Der Minimax-Algorithmus dient der Entscheidungsfindung eines optimalen Spielzuges vor allem in Zwei-Spieler Spielen wie Schach oder Backgammon bei denen sich die Spieler mit ihren Zügen abwechseln. Effizient angewendet werden kann der Algorithmus vor Allem in **Nullsummenspielen**.
Ein Spiel ist ein **Nullsummenspiel**, wenn die Summe aus Gewinnn und Verlust aller Spieler nach Spielende Null ergibt, das heißt der Gewinn eines Spielers ist der

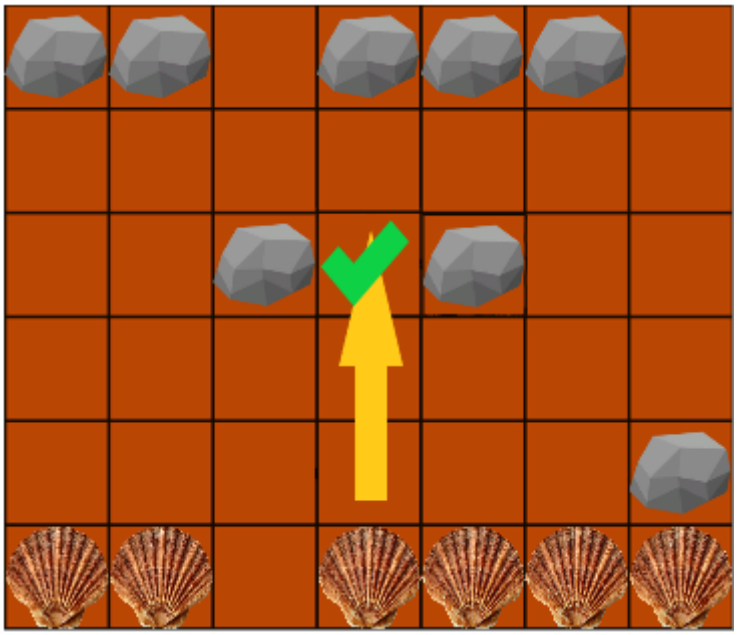

Abbildung 2.5: Keine Eroberung

Verlust beziehungsweise die Niederlage eines anderen.[Hart17].
Diesen Algorithmus kann man sich als Spielbaum vorstellen, der mit jedem Pfad einen denkbaren Spielverlauf darstellt. Dabei repräsentiert jeder Knoten des Baumes einen Spielzustand und enthält Informationen über die Verteilung der Figuren auf dem Spielbrett, welcher Spieler am Zug ist und über mögliche weitere Spielzüge. Jeder dieser Züge ist ebenfalls wieder ein Knoten und Kind der vorherigen Aktion.

Beispiel 2.1: Am einfachsten kann man sich die Funktionsweise am Beispiel von TicTacToe klar machen:
Dabei repräsentiert wieder jeder Knoten, der hier als Abbildung des Spielbretts dargestellt wird, einen Zustand des Spiels. Die Bewertung der Zustände ist in diesem Spiel vergleichsweise simpel:

- Spieler 1 gewinnt: +1 Punkt,
- Unentschieden: 0 Punkte,
- Spieler 2 gewinnt: -1 Punkt

Der Algorithmus basiert auf der Idee, anhand dieses Baumes den Verlauf mit dem höchsten Wert zu verfolgen und so das Spiel zu gewinnnen. Dabei wird wie in Abbildung 2.6 zu sehen, jedem Knoten ein Wert zugeordnet und versucht immer den höchstmöglichen Wert zu erzielen, wenn der Spieler gewinnen soll.

Analyse & Variationen

Aufgrund von tendenziell hoher Laufzeit, vor Allem bei komplexeren Spielen, gibt es verschiedene Variationen dieses Algorithmus. In der Basis-Version des MiniMax

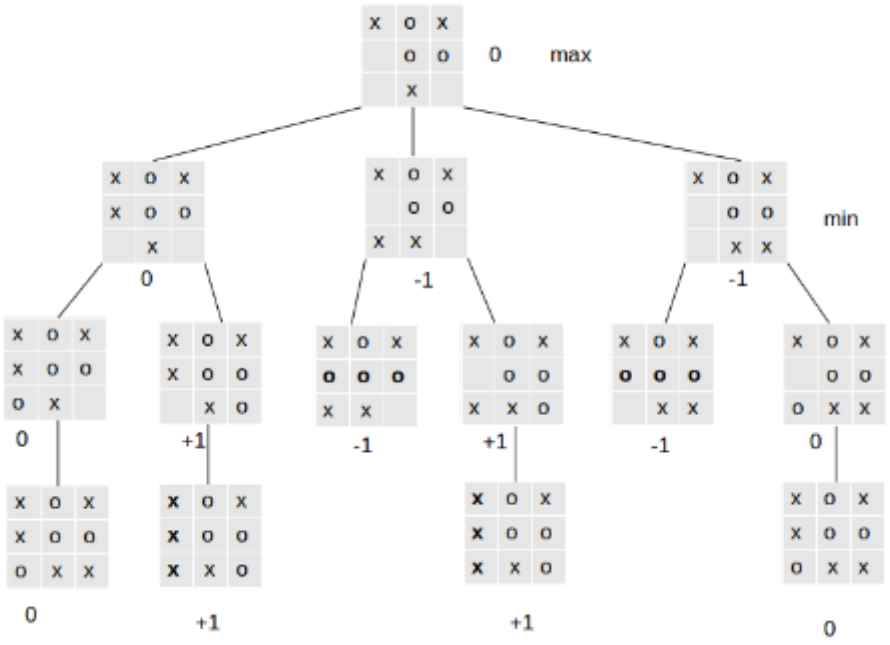

Abbildung 2.6: TicTacToe Spielbaum
[Hart17]

enden die möglichen Verläufe im ungünstigsten Fall nach d Zügen, wobei d für die Suchtiefe im Baum steht. Im extremsten Fall können wir annehmen, dass aus jedem Zustand weitere w Züge möglich sind. Daraus resultieren auf der untersten Ebene w^d Knoten und es gilt

$$\sum_{i=1}^{d} w^d \in O(w^d).$$

Zur Optimierung existieren unterschiedliche Variationen von denen eine hier näher beschrieben wird.

Alpha-Beta-Algorithmus

Aufgrund der hohen Komplexität wird auch der **Alpha-Beta-Algorithmus** verwendet. Dieser basiert auf dem MiniMax-Algorithmus, allerdings wird hier der Wert der möglichen Endzustände geschätzt und die Berechnnung eines Pfads abgebrochen, sobald klar ist, dass dieser Weg weniger zielführend ist. Dabei wird angenommen, dass Zweige ab einem Punkt in der Berechnung vernachlässigt werden können. Am nachfolgenden Beispiel wird die Idee deutlicher:

Beispiel 2.2: Nach der Durchsuchung des linken Teilbaums, sieht man, dass der der Ausgangsknoten mindestens den Wert 3 hat. Deshalb wird die Evaluation des rechten Teilbaums abgebrochen, sobald klar wird, dass der min-Knoten maximal einen Wert von 2 hat. Die Suche kann abgebrochen werde, da der rechte Teilbaum keine Auswirkungen auf den Startknoten hat.

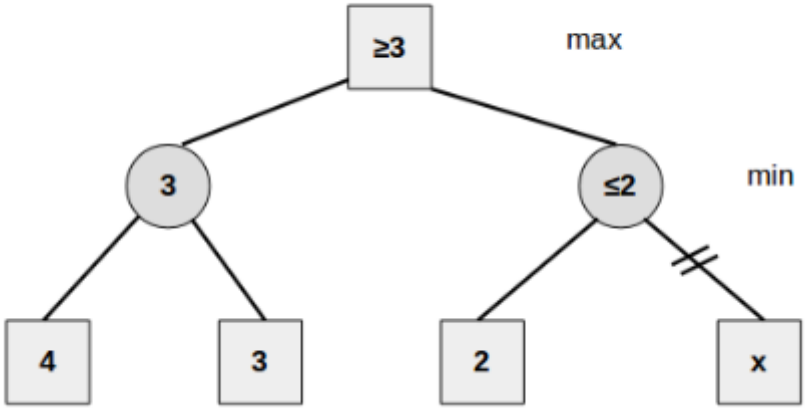

Abbildung 2.7: Alpha-Beta Spielbaum
[Hart17]

2.3 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. Um selbstgeschriebene Scripte 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 vorgefertigt erstellten Prefabs genutzt. Ausserdem bietet Unity eine große Anzahl an Libraries zur Unterstützung der Spielentwicklung.
Unity bietet die Möglichkeit einerseits per Drag & Drop Spielobjekte im Editor zu platzieren und diesen Objekten so ihren Ausgangszustand zu geben. Außerdem besteht die Möglichkeit 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 als Darstellung der Spielobjekte im Editor platziert werden. In einem Unity Projekt befindet sich ein Ordner „Assets" in dem alle selbst erstellten Skripte, Sprites und Szenen enthalten sind, sowie andere genutzte Ressourcen wie zuvor erstellte Prefabs für eine dynamische Generierung.

Prefab System

Unitys Prefab System ermöglicht es Spielobjekte zu erstellen und als Prefab zu speichern. Diese Prefabs können wiederverwendet werden und fungieren wie ein 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.

3. Analyse

Im vorherigen Kapitel sollte klar geworden sein, dass die Ansätze die bereits genutzt wurden, heute noch immer ähnlich funktionieren und ausreichend entwickelt sind diese auch in der Lage erfahrere Spieler oder Profis zu schlagen, wie IBMs Deep Blue beim Schach gezeigt hat.
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 für 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 Internettverbindung hat, soll es auch möglich sein das Spiel via USB auf dem Gerät zu starten. Da Unity die Möglichkeit bietet ein Entwickeltes Spiel für verschiedene Plattformen bereitzustellen, habe ich mich für Unity als 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, beziehungsweise auch als Simulation für Vorführungszwecke laufen zu lassen. Die KI sollte nach Möglichkeit ein sinnvolle Strategie verfolgen, sodass die Züge nicht willkürlich wirken. Das heißt es müssen einerseits die Logik & Regeln implementiert werden, damit das Spiel grundsätzlich spielbar ist und andererseits der MiniMaxing-Algorithmus um die erlaubten Züge vorher zu bewerten. Weiterhin muss ein ansprechendes Design gewählt werden, dass auch historisch sinnvoll ist und den Spielern im Museum ein relativ realistisches Bild des Spiels bieten. Da historisch 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 ÖnClickSteuerung 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&DropBedienung angepasst. Das Menü soll leicht erreichbar sein, um das Spiel Neustarten oder Beenden zu können.

4. Entwurf / Konzeption

4.1 Konzept und Ideen

Grundlegende Features

Diese grundsätzlichen Features entsprechen dem ursprünglichen Prototypen des Spiels, dadurch konnten die implementierten Regeln getestet und durch weitere 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

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:

- Unterschiedlliche Schwierigkeitsgrade anstatt Slider, da diese nicht so intuitiv verstanden wurden, im Spiel gegen die KI
- 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 unterschiedliche 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 2.3. 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.

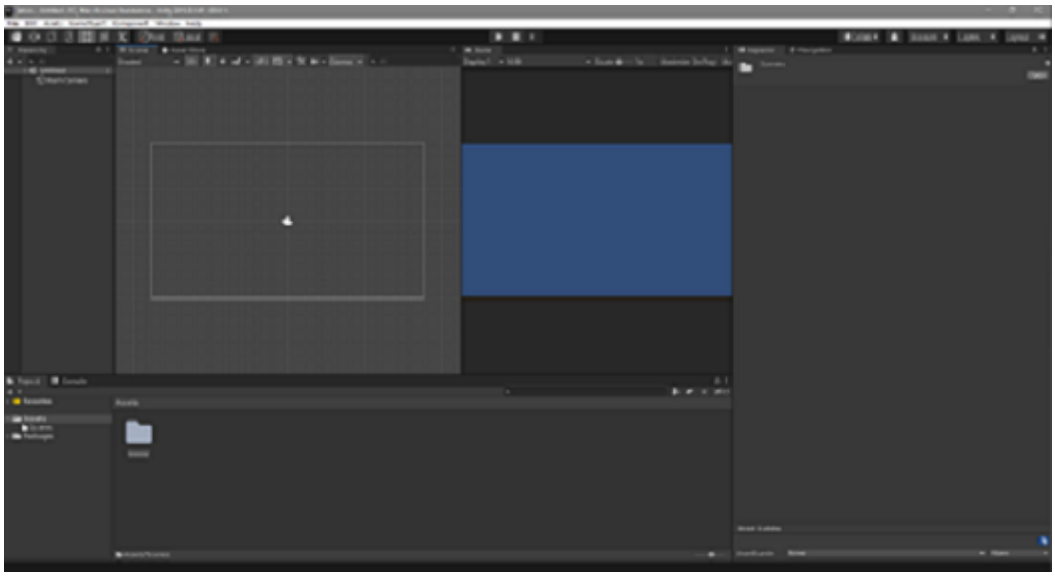

Abbildung 5.1: Unity-Editor

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 5.2) 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

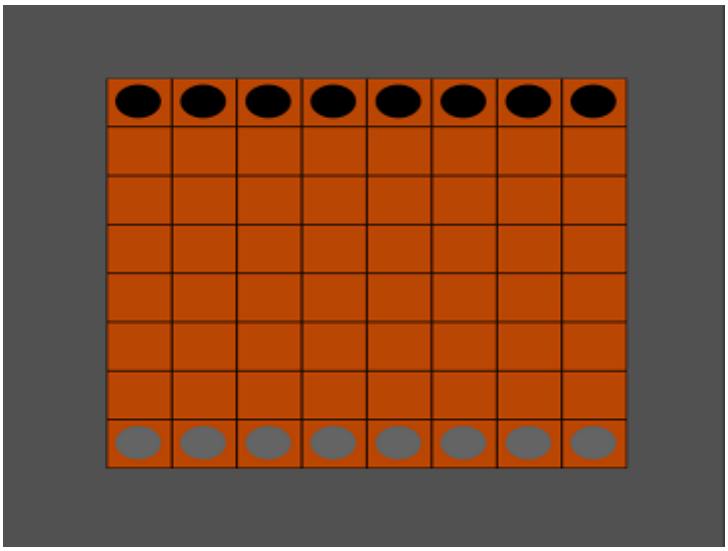

Abbildung 5.2: Prototyp

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 MiniMaxing-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 MiniMaxing habe ich die Klasse AIManager entworfen. Diese enthält die Funktion **Minimax()**, die das aktuelle Spielbrett als Board, die maximale Suchtiefe, die aktuelle Tiefe und den Spieler der am Zug ist übernimmt. 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.

```
1  public static float Minimax(BoardDraught board, int player, int maxDepth,
2  int currentDepth, ref Move bestMove)
3  {
4          if (board.IsGameOver() || currentDepth == maxDepth)
5          {
6                  return 0;
7          }
8          bestMove = null;
9          float bestScore = Mathf.Infinity;
10         if (board.GetCurrentPlayer() == player)
11         bestScore = Mathf.NegativeInfinity;
12         List<Move> allMoves = new List<Move>();
13         int nextPlayer = 0;
14         if (player == 2)
15         {
16                 allMoves = board.GetMoves(player);
17                 nextPlayer = 1;
18         }
19         else if (player == 1)
20         {
21                 allMoves = board.GetMoves(player);
22                 nextPlayer = 2;
23         }
24         Move currentMove;
25         foreach (Move m in allMoves)
26         {
27                 board.MakeMove(m);
28                 float currentScore;
29                 currentMove = m;
30                 if (m.attacked)
31                 {
32                         if (nextPlayer == 2)
33                         nextPlayer = 1;
34                         else
35                         nextPlayer = 2;
36                 }
37                 currentScore = Minimax(board, nextPlayer, maxDepth,
38                                 currentDepth + 1, ref currentMove);
39                 float newScore = board.Evaluate(player);
40                 if (board.GetCurrentPlayer() == player)
41                 {
42                         currentScore += newScore;
43                         if (currentScore > bestScore)
44                         {
45                                 bestScore = currentScore;
46                                 bestMove = m;
47                                 m.mScore = bestScore;
48                         }
49                 }
50                 else
51                 {
52                         currentScore -= newScore;
53                         if (currentScore < bestScore)
54                         {
55                                 bestScore = currentScore;
56                                 bestMove = m;
57                                 m.mScore = bestScore;
58                         }
59                 }
60                 board.StepBack();
61         }
62         List<Move> bestMoves = new List<Move>();
63         if (currentDepth == 0)
64         {
65                 foreach (Move m in allMoves)
66                 {
67                         if (m.mScore == bestScore)
68                         {
69                                 bestMoves.Add(m);
70                         }
71                 }
```

```
72                  System.Random rnd = new System.Random();
73                  int index = rnd.Next(bestMoves.Count);
74                  bestMove = bestMoves.ToArray()[index];
75          }
76          return bestScore;
77  }
```

Die rekursive Funktion **Minimax()** wird aufgerufen und bekommt den momentanen Spielzustand, sowie die maximale und 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 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 und 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 dem Spielbrett gespeichert, um nach jeder Iteration den Schritt rückgängig machen zu können. In Zeile 30 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 37 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 60 mit der Board-Funktion **StepBack()** der letzte Zug rückgängig gemacht und die Iteration wird fortgesetzt. Nach der Rekursion sollte 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 Punktzahl auswählt und zurückgibt.

```
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  }
```

Die Funktion **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 ob das Spiel nach dem Zug beendet wäre und dementsprechend die Punktzahl dazu addiert. Die Punkteverteilung bestimmt auch den Schwierigkeitsgrad. Um die KI noch intelligenter zu machen, könnte die Evaluierung noch um eine beliebige Anzahl an Flags erweitert werden, sodass die Spielsituation immer präziser bewertet wird.

6. Evaluation

Zur Evaluation wurde eine lauffähige Windows Version und Webversion dem Landesmuseum Birkenfeld zugesendet und Laufzeitmessungen des MiniMax-Algorithmus durchgeführt.

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 eines Zoom-Meetings 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 Punkt, der aufgefallen war, ist, 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 Leistungsmessung?

...

6.3 Zusammenfassung

Am Ende sollten ggf. die wichtigsten Ergebnisse nochmal in *einem* kurzen Absatz zusammengefasst werden.

7. Diskussion und Ausblick

(Keine Untergliederung mehr)

Literaturverzeichnis

[BaFe14] A. Barr und E. A. Feigenbaum. *The Handbook of Artificial Intelligence: Volume 2*, Band 2. Butterworth-Heinemann. 2014.

[Hart17] M. Hartmann. Der Alpha-Beta-Algorithmus, 2017.

[Klin08] H. Klinge. Warum Künstliche Intelligenz (KI) in Spielen stagniert, 1 2008.

[Pett] J. Petty. What is Unity 3D & What is it Used For?

[Rous18] M. Rouse. Künstliche Intelligenz (KI), 2018.

[UPr] *Unity Documentation, Prefabs.*

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