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

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

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 Landesmuseum Birkenfeld Veranstaltungen (https://www.landesmuseum-birkenfeld.de/landesmuseum/erlebnismuseum/) 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. 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, so das verschiedene Quellen hinzugezogen werden müssen um die 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 umstellen 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 horizontal und vertikal 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. Die festgelegten Regeln für die implementierte Version werden im Kapitel 2.1 erklärt.

1.3 Zielsetzung

In dieser Arbeit wird eine Variation des Spiels Latrunculi mithilfe der Unity Game Engine (https://unity.com/) 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. Das Projekt soll 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 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 Simulation analysiert. Zum Schluss gibt es noch einen Ausblick auf mögliche Erweiterungen um den Algorithmus effizienter zu gestalten.

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 Spielregeln

Der grundsätzliche Aufbau des Spiels beinhaltet ein Spielbrett mit beispielsweise 7x6 Feldern bis hin zu überlieferten 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.

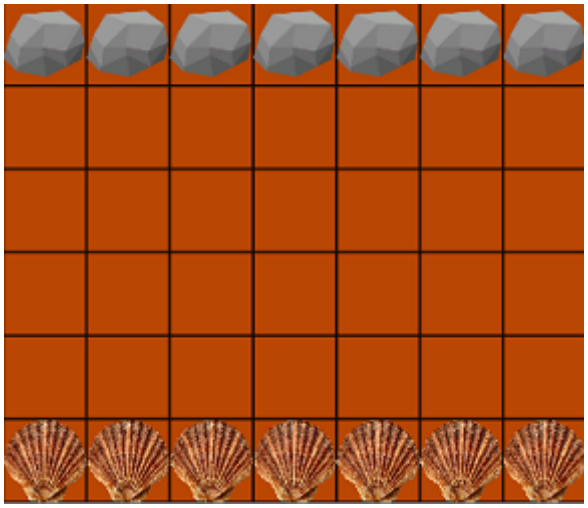

Abbildung 2.1: Startkonfiguration eines 7x6 Latrunculi Feldes

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 & 2.3).

Um gegnerische Spielsteine zu erobern, müssen diese wie in der Abbildung 2.4 von eigenen Steinen auf zwei gegenüberliegenden Feldern umstellt sein.

Wird ein gegnerischer Stein erobert, dann wird dieser aus dem Spiel entfernt und ein eigener darf nochmal bewegt werden. Hierbei gibt es auch Variationen bei denen nur

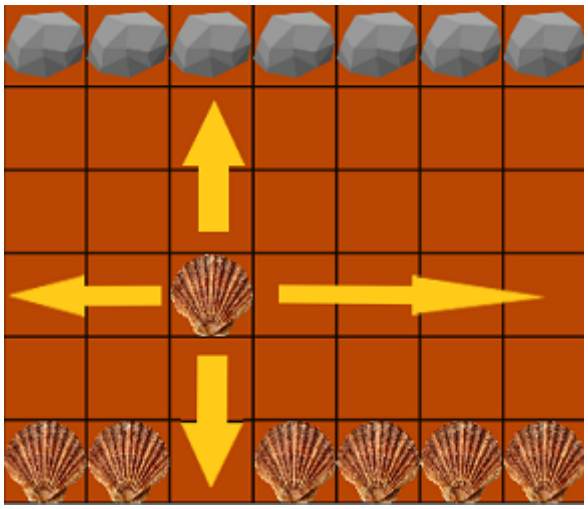

Abbildung 2.2: Legitime Bewegungen auf einem 7x6 Latrunculi-Feld

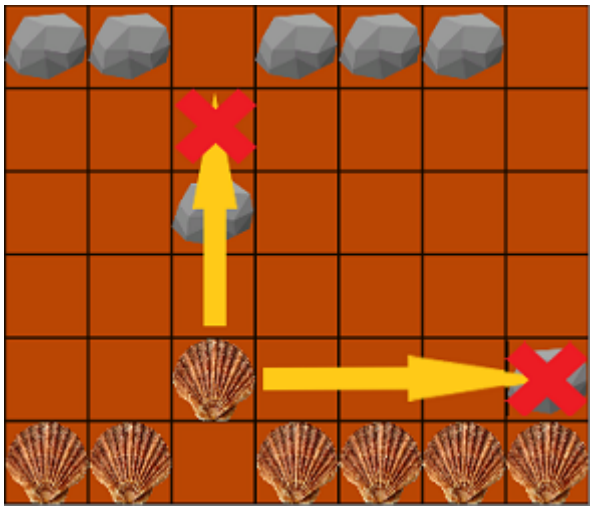

Abbildung 2.3: Nicht zulässige Bewegung auf einem 7x6 Latrunculi-Feld

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 2.5).

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

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. Um selbstgeschriebene Scripte einzubinden unterstützt Unity unter anderem C#, wie ich es in dieser Arbeit auch verwendet habe. Unity wurde hier vor allem wegen

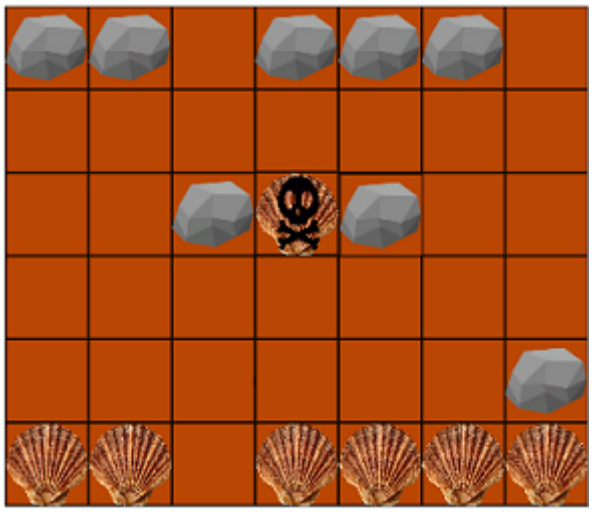

Abbildung 2.4: Eroberung eines Spielsteins

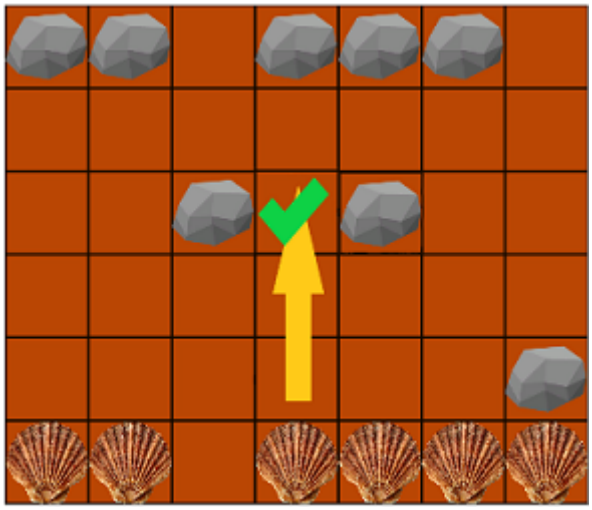

Abbildung 2.5: Keine Eroberung

der einfachen Portierung und Wiederverwendbarkeit von vorher 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, 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 als 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(vgl. [UPr]).

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. In dieser Arbeit wurde beispielsweise 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 2.6 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 Muschel 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.

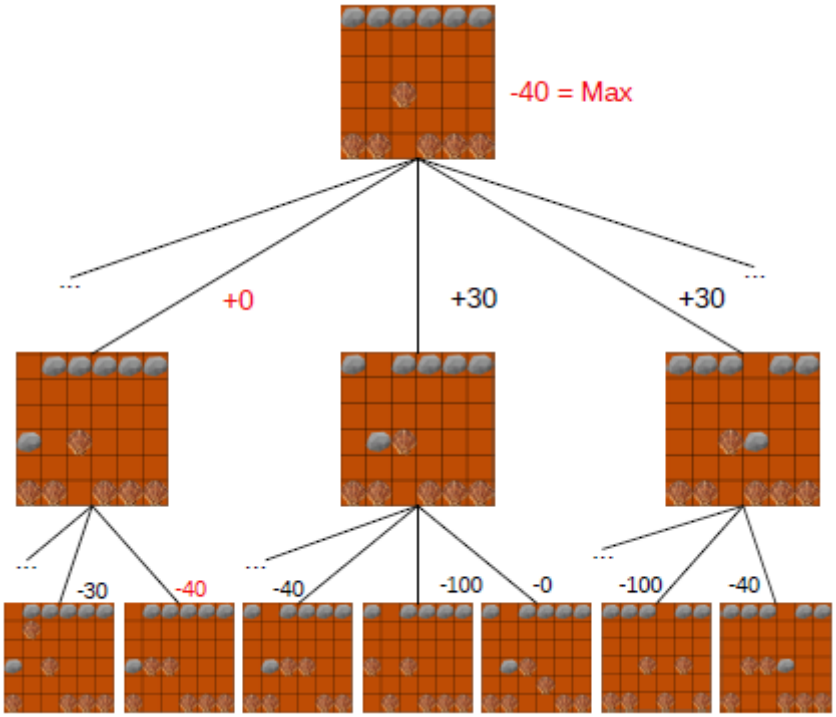

Abbildung 2.6: Ausschnitt: MinMax-Spielbaum eines 6x6 Latrunculi Feldes

2.3.3 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 Lernen 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 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 „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. Entwurf

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.

4.1 Ideen

Grundlegende Features

Diese grundsätzlichen Features entsprechen dem ursprünglichen Prototypen, der in der Abbildung 4.1 zu sehen ist, 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

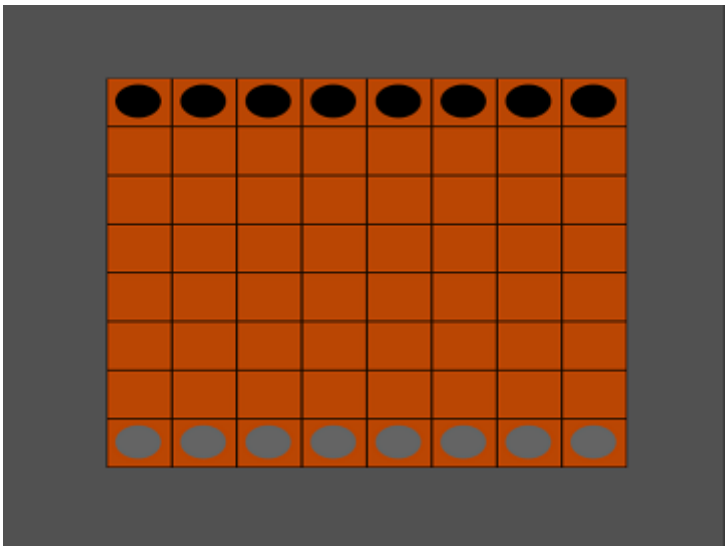

Abbildung 4.1: Prototyp

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 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 2.2. 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 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 (vgl. [Pala16]).

```
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, 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 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 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.

```
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 **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.

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

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

Spieler 1 (Steine:

- Angriffspunkte: 100
- Verteidigungspunkte: 20

- Bedrohungspunkte: 50
- Hohe Bedrohungspunkte: 80

Spieler 2 (Muscheln):

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

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 (Steine):

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

Spieler 2 (Muscheln):

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

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.

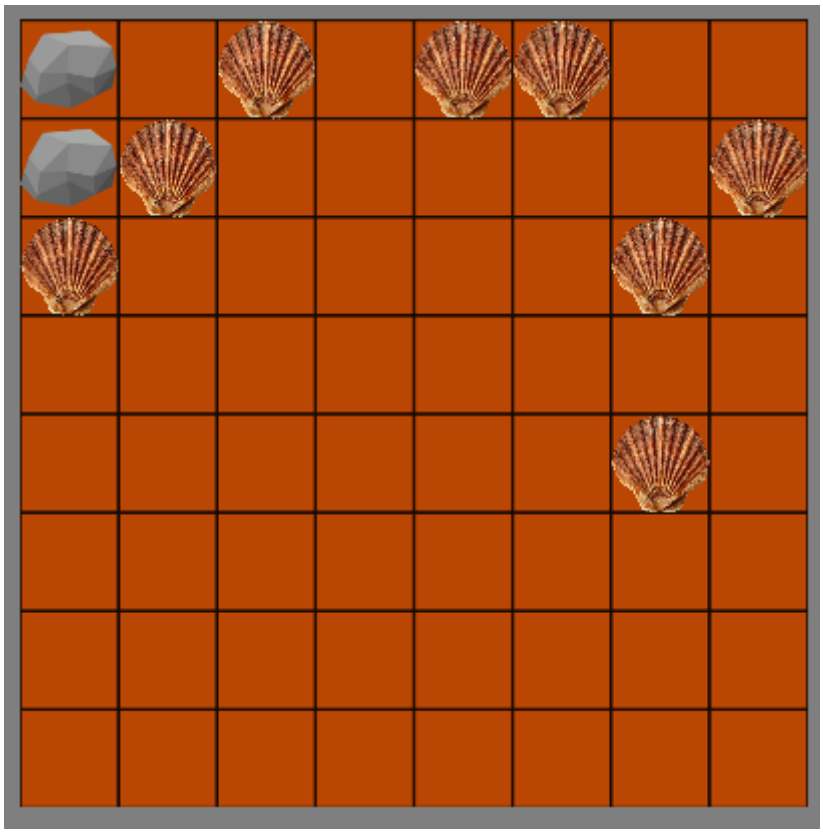

Abbildung 6.1: Spielende wird nicht erkannt

Suchtiefe 10

7. Mögliche Erweiterungen

Um den Algorithmus effizienter zu gestalten, gibt es unterschiedliche Verfahren. Der Alpha-Beta-Algorithmus ist so 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.

7.1 Alpha-Beta-Algorithmus

Der Alpha-Beta-Algorithmus ist eine Verbesserung des MiniMax-Suchverfahrens. Dabei funktionieren beide prinzipiell ähnlich, allerdings soll ersterer schneller in der Suche sein.
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.

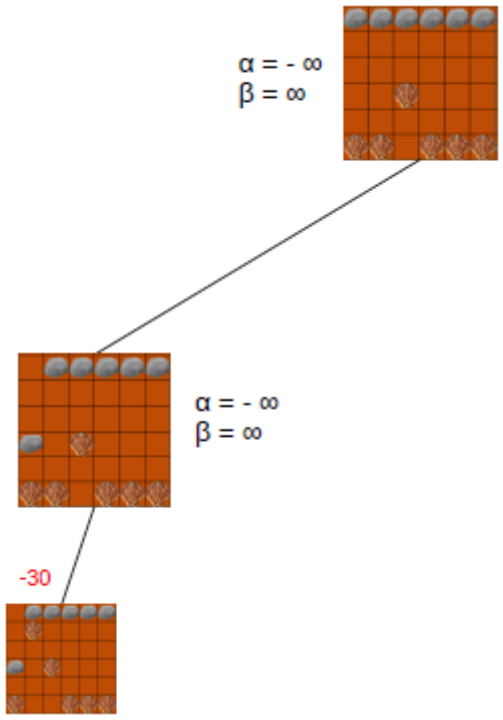

Abbildung 7.1: Schritt 1: Alpha-Beta Pruning

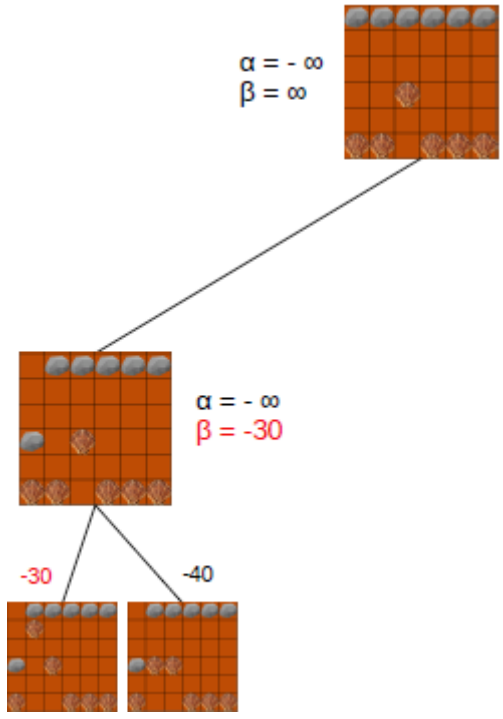

Abbildung 7.2: Schritt 2: Alpha-Beta Pruning

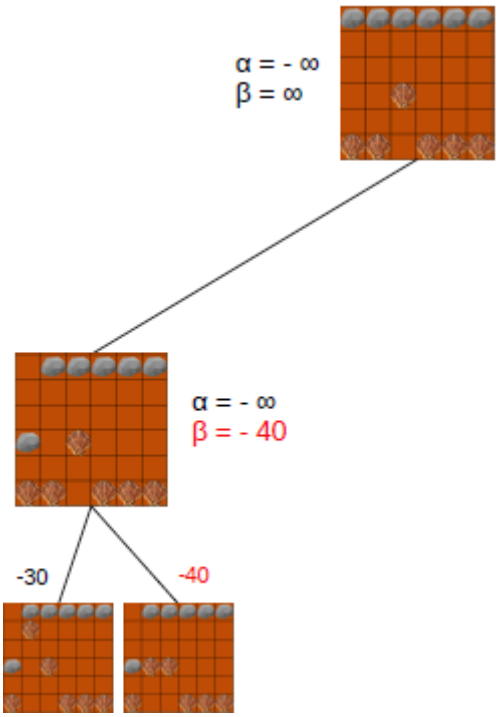

Abbildung 7.3: Schritt 3: Alpha-Beta Pruning

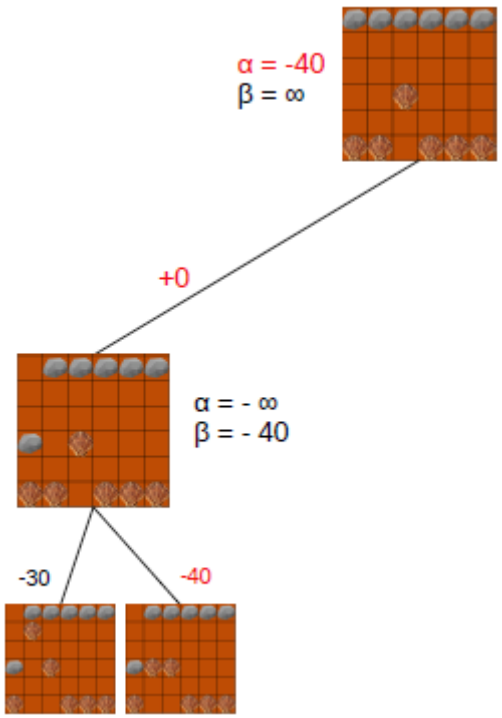

Abbildung 7.4: Schritt 4: Alpha-Beta Pruning

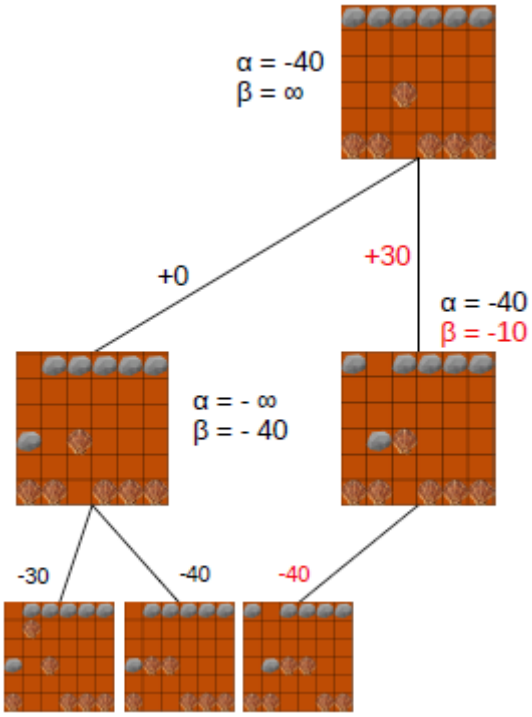

Abbildung 7.5: Schritt 5: Alpha-Beta Pruning

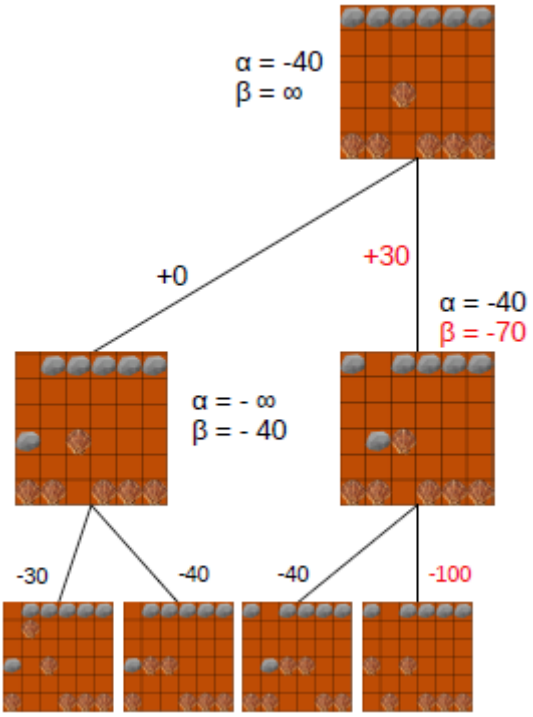

Abbildung 7.6: Schritt 6: Alpha-Beta Pruning

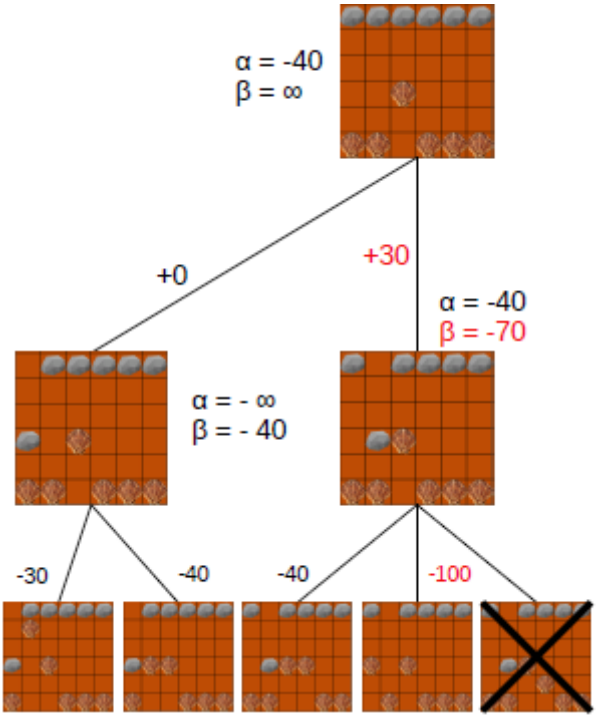

Abbildung 7.7: Schritt 7: Alpha-Beta Pruning

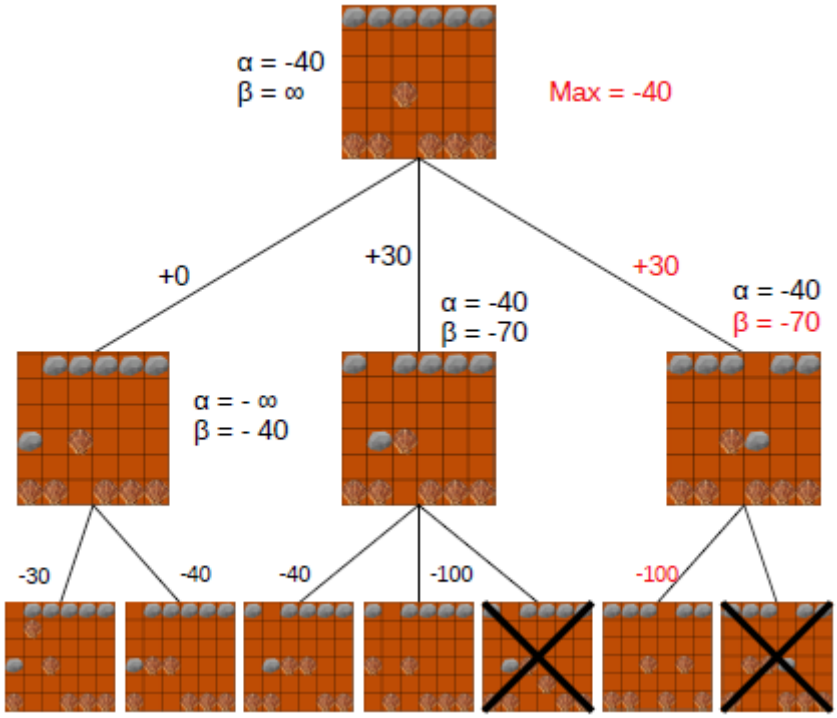

Abbildung 7.8: Schritt 8: Alpha-Beta Pruning

8. Diskussion und Ausblick

(Keine Untergliederung mehr)

Literaturverzeichnis

[EdHa61] D. J. Edwards und T. Hart. The alpha-beta heuristic. 1961.

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

[Pala16] J. Palacios. *Unity 5. x Game AI Programming Cookbook*. Packt Publishing Ltd. 2016.

[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