Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelorarbeit

**Michael Hanisch**

## Konzeption und Evaluation eines Entity-Component-Systems anhand eines rundenbasierten Videospiels

*Fakultät Technik und Informatik*
*Studiendepartment Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

Michael Hanisch

# Konzeption und Evaluation eines Entity-Component-Systems anhand eines rundenbasierten Videospiels

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Dr.-Ing. Sabine Schumann

Eingereicht am: 23. September 2016

**Michael Hanisch**

**Thema der Arbeit**
Konzeption und Evaluation eines Entity-Component-Systems anhand eines rundenbasierten Videospiels

**Stichworte**
Ashley, Freebooter's Fate, Videospiel, ECS, LibGDX, Java

**Kurzzusammenfassung**
Dieses Arbeit behandelt den Aufbau eines Videospiels mithilfe des Entity-Component-Systems (ECS) Ashley. Als Vorlage dient das Tabletop Strategiespiel Freebooter's Fate, welches auf Runden und abwechselnden Zügen basiert.
Thematisiert wird die Nutzung eines Zustandsautomaten innerhalb eines ECS. Ergebnis ist ein leicht zu erweiterndes Videospiel.

**Michael Hanisch**

**Title of the paper**
Conception and evaluation of an entity component system on the basis of a turn based video game

**Keywords**
ashley, Freebooter's Fate, video games, ECS, libGDX, java

**Abstract**
This thesis shows the implementation of a video game, using entity component system (ECS) Ashley. Original is miniature wargaming Freebooter's Fate, which is based on rounds and alternating turns.
Picked out as a central theme is the usage of a finite state machine inside an ECS. The result is an easy to expand video game.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Einleitung

Wenn das Lieblingsspiel als Videospiel umgesetzt werden soll, stellt sich neben den vielen Design-Entscheidungen zur Digitalisierung vor allem die Frage der unterliegenden Struktur. Ein objektorientierter Ansatz? Einfach drauflos coden? Oder doch lieber das Internet durchforsten nach Frameworks, die schon andere Spieleentwickler erfolgreich ans Ziel gebracht haben?
Das Entity Component System (ECS) Ashley ist eines dieser Frameworks. Es gehört zum plattformunabhängigen Java-Framework LibGDX und nutzt somit die objektorientierte Sprache Java. Veröffentlicht wurde es im Juli 2014 (Version 1.0.1) und das letzte Release (1.7.2) erfolgte im Februar diesen Jahres (2016) was die aktuelle und aktive Weiterentwicklung aufzeigt.
Umgesetzt werden soll das Tabletop-Spiel *Freebooter's Fate*, welches seit 2010 auf dem Markt ist und dessen Seite mit aktivem Forum im September 2015 einen Relaunch erfahren hat. Im Mantel und Degen Setting legen sich hier dreckige Goblins mit Piraten an oder wilde Amazonen mit der Imperialen Armada. Jeder Spieler führt seine Mannschaft an und es wird gekämpft, bis keiner mehr steht - oder acht Runden um sind.

Im Gegensatz zu den üblichen - mit ECS umgesetzten - Echtzeitspielen handelt es sich bei *Freebooter's Fate* um ein zugbasiertes Spiel. Es gilt also herauszufinden, wie im allgemeinen ein Spiel mit ECS umgesetzt werden kann und dann im Besonderen ein Spiel, in welchem einer nach dem anderen am Zug ist. Schließlich soll es in diesem Kampf gesittet zugehen.

## 1.1 Motivation

Das Tabletop Freebooter's Fate gehört zu den Lieblingsspielen des Autors. Neben der Schwierigkeit, regelmäßig Mitspieler zu finden, reizte auch eine Umsetzung als Spiel, um sich intensiv mit den Funktionalitäten und Gesetzmäßigkeiten des Spiels auseinander zu setzen. In Zukunft soll Spielen über das Netzwerk - mit räumlich weit entfernten Spielern - sowie gegen eine künstliche Intelligenz möglich sein.

Weiterhin bestand der Anreiz, eine Alternative zur Objektorientierung zu suchen. Entsprechend gilt es ein Framework zu finden, das für die spezifische Aufgabenstellung ausgelegt ist.

## 1.2 Darstellung der Problemstellung

Ashley ist ein Framework, das größtenteils für Echtzeit Spiele genutzt wird. Beispiele sind *Superjumper*[1], *Mount FuJ*, *Roto*[2], welche als Open Source auf GitHub veröffentlicht sind und häufig als Beispielprojekte herangezogen werden. Für diese Bachelorarbeit wird allerdings ein rundenbasiertes Spiel implementiert, das nur auf (zeitlich) voneinander abhängige Ereignisse reagieren muss. Trotz der abweichenden Struktur vereinfacht auch hier Ashley den Umgang mit den einzelnen Elementen und Funktionalitäten des Spiels.

Vorgestellt wird die praktische Anwendung eines Spieleframeworks, das zumeist für Echtzeit-Spiele eingesetzt wird, für ein rundenbasiertes Spiel. Vordergründig wird die Digitalisierung eines analogen Tabletopsystems durchgeführt.

## 1.3 Ziel der Arbeit

Ziel der Arbeit ist die Implementierung der Grundfunktionalitäten einer digitalen Version des Tabletops Freebooter's Fate. Hierzu gehört die Erarbeitung einer geeigneten Architektur unter Berücksichtigung des verwendeten Frameworks. Großes Augenmerk wird auf Erweiterbarkeit gesetzt, um später weitere Funktionalitäten ergänzen und das Spiel fertig stellen zu können. Weiterhin sollen äquivalent zum abgebildeten Spiel Erweiterungen hinzugefügt werden können.

### 1.3.1 Anforderungen

Ziel der Arbeit ist, eine geeignete Struktur für ein zugbasiertes Spiel zu finden. Neben traditionellen Ansätzen werden auch Entity Component Systeme in Erwägung gezogen. Hierbei sollen mindestens die Grundfunktionalitäten implementiert werden. Hierzu zählen die Bewegung der eigenen Charaktere, der Angriff gegnerischer Charaktere und korrekt funktionierende sowie alternierende Züge.

[1] Superjumper von David Saltares: github.com/saltares/ashley-superjumper
[2] Games made with Ashley - Open Source Projekte github.com/libgdx/ashley/wiki/Games-made-with-Ashley

### 1.3.2 Abgrenzung

Durch die zeitliche und thematische Begrenzung der Bachelorarbeit wird das Ergebnis kein fertiges Spiel sein. Ebenso wenig können umfangreiche Benutzer- und Plattformtests durchgeführt werden, welche für ein solches System von Nöten wären. Grafiken werden mit Genehmigung von Freebooter's Fate - vertreten durch Werner Klocke - sowohl für die Grafiken im Spiel, als auch für gesondert gekennzeichnete Abbildungen in dieser Arbeit genutzt und von einer Kommilitonin ergänzt.
Vorrangig geht es darum, die Konzepte der ausgewählten Architektur auszunutzen, um die Grundfunktionen mit einer beschränkten Auswahl an Spiel-Einheiten zu erarbeiten. Auch Sonderregeln und komplexere Spielinhalte werden vorerst weggelassen. Als Testgeräte stehen von privat ein ACER Aspire V mit Windows 10, ein ASUS Tablet mit Android 4.1 und ein LG G4 Smartphone mit Android 5.0 zur Verfügung.

## 1.4 Struktur der Arbeit

Kapitel 2 legt die Grundlagen, die für diese Arbeit nötig sind. Es werden verwandte Arbeiten vorgestellt (2.1) und auf allgemeine Ansätze in der Spieleprogrammierung (2.2) und im besonderen auf Entity Component Systeme (2.3) eingegangen. Zur Umsetzung werden konkrete Frameworks betrachtet (2.4). Zur besseren Nachvollziehbarkeit werden Tabletops im allgemeinen (2.5) erläutert und vergleichbare Videospiele aufgeführt (2.6).

Kapitel 3 stellt die von der Implementierung unabhängigen Überlegungen zu Architektur und Design des Spiels vor. Es werden unterschiedliche Ansätze verglichen, unter Berücksichtigung der Besonderheiten des Spiels (3.1). Die Elemente aus dem umzusetzenden Spiel werden in die gewählte Struktur eines ECS eingepasst (3.2) und es wird eine Auswahl getroffen, was als Komponenten und Systeme implementiert werden soll (3.3).

Kapitel 4 beschäftigt sich mit der konkreten Umsetzung. Es werden einige Überlegungen zum allgemeinen Game Design getroffen (4.1) und das verwendete Grafik-Framework vorgestellt (4.2). Es wird auf die Charaktere im Spiel eingegangen (4.3) und welche Besonderheiten das umzusetzende Spiel Freebooter's Fate mit sich bringt (4.4).

Kapitel 5 evaluiert das Ergebnis und den im Rahmen der Arbeit erstellten Prototypen. Es wird noch einmal auf die besondere Herausforderung eines zugbasierten Spiels mit einem ECS eingegangen (5.1), sowie auf den erreichten Funktionsumfang (5.2). Die Erweiterbarkeit in

Zusammenhang mit ECS und der Implementierung weiterer Funktionalitäten wird betrachtet (5.3), sowie Form und Umfang der bereits geleisteten und zukünftig benötigten Qualitätssicherung (5.4).

Kapitel 6 schließt die Arbeit ab. Es erfolgt eine Zusammenfassung der Arbeit (6.1) mit einem Rückblick auf die im Projekt gemachten Erfahrungen (6.2). Zuletzt wird auf die Zukunft des Spiels eingegangen (6.3).

# 2 Grundlagen

Das folgende Kapitel gibt einen Überblick über den aktuellen Stand der Technik. Vorgestellt werden andere Arbeiten mit dem Ashley-Framework, allgemeine Ansätze in der Spieleprogrammierung, sowie alternative Werkzeuge zur Umsetzung. Besonderes Augenmerk erhält die Vorstellung des Entity Component Systems (ECS) und verschiedene Frameworks, welche dieses umsetzen. Zuletzt werden ähnliche Spiele vorgestellt, die ein analoges Tabletop zum Vorbild haben.

## 2.1 Verwandte Arbeiten

Die vorgestellten, verwandten Arbeiten beziehen sich in der Hauptsache auf Herangehensweisen an Spielprogrammierung und ECS, da es kaum wissenschaftlich dokumentierte Umsetzungen von Videospielen gibt. Auf ähnliche Spiele wird dennoch in Abschnitt 2.6 eingegangen.

### 2.1.1 Webblog von Richard Lord

Richard Lord hat das ECS *Ash* (detaillierter vorgestellt in Abschnitt 2.4.2) entwickelt und bietet gute Grundlagen für diese Arbeit, da er in seinem Webblog grundlegende Konzepte und Gedankengänge von ECS erklärt. Vieles erläutert er anhand seiner eigenen - in *Ash* implementierten - Spiele.
Ein Beitrag beschäftigt sich damit, einen Game-Loop Schritt für Schritt in ein ECS zu portieren [11]. Ein anderer damit, welche Vorteile ein ECS für die Spieleprogrammierung bietet [12].

### 2.1.2 OOP-Pattern für die Spieleprogrammierung

Robert Nystrom stellt in seinem Buch *Design Patterns für die Spieleprogrammierung* [17] OOP Pattern für die Spieleprogrammierung vor. Besonders zu nennen sind die Pattern Component Pattern, State Pattern und Spatial Partitioning, da diese auch in ECS Anwendung finden. Das Component Pattern bildet die Grundlage für ECS. Das State Pattern wird hierbei noch detaillierter im Abschnitt 2.3.1 vorgestellt.

### 2.1.3 T-machine

Ein Webblog von Adam Martin beschäftigt sich ebenfalls detaillierter mit Entity Systemen (ES). Er stellt ES am Beispiel eines Massive Multiplayer Online Games (MMOG) vor und geht dabei auf die unterliegenden Mechanismen der Vorgehensweise ein [14]. Mit dem Spiel Bomberman beschäftigt er sich damit, welche Informationen in Komponenten umgewandelt werden sollten [16]. In einem anderen Artikel ist Thema, wie eine gute Umsetzung an Komponenten stattfindet und wie diese bei einer zu hohen Anzahl reduziert werden können [15].
Hauptsächlich thematisiert Martin die Kombination aus ECS und dem Framework Unity. Da er sich ebenfalls mit der Implementierung eines eigenen ES namens Aliqua beschäftigt, finden sich in seinem Blog auch technisch detailliertere Beschreibungen u.a. zur Speicherverwaltung.

## 2.2 Ansätze der Spieleprogrammierung

Es werden die grundlegenden Mechanismen vorgestellt, mit denen man es in der Spielprogrammierung zu tun bekommt.

### 2.2.1 Game Loop

Traditionell nimmt ein Game-Loop in vorgegebenen Intervallen Änderungen am Spielzustand vor, indem sie Ereignisse und Zustände abfragt und die Spielparameter entsprechend anpasst. Vereinfacht ist dies in Abbildung 2.1 dargestellt. Das Prinzip des Game-Loops ist sehr übersichtlich in LibGDX (4.2) umgesetzt. Hier werden von einem Launcher abwechselnd die Methoden *update* und *render* aufgerufen.

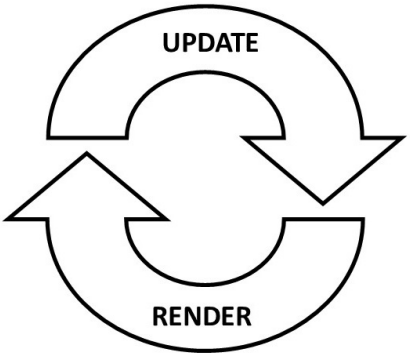


Abbildung 2.1: Game-Loop

Je komplexer das System, desto größer und unübersichtlicher wird der Codeblock und umso mehr muss der Game-Loop machen. Zudem wirkt sich die Menge der Aufrufe für Abfrage und Aktualisierung irgendwann auf die Performanz aus.
Entity Systeme entspringen dem Verlangen, den Game-Loop zu vereinfachen [11]. Ein Ansatz unter Beibehaltung des Loops ist die Verteilung von update und render auf verschiedene Threads [3, Real-time loops S. 31ff]. Weitere Optimierung bringt die Aufteilung auf verschiedene Systeme, wie bei ECS.

### 2.2.2 OOP

In der traditionellen, objektorientierten Spieleprogrammierung ist jeder Teil des Spiels ein eigenes Objekt. Diese Objekte werden über einen Game-Loop verwaltet und aktualisiert. Gerne wird hier auch die Vererbung von OO-Sprachen genutzt, um Code wiederzuverwenden [7]. Vererbung ist ein praktisches Werkzeug, um Eigenschaften und Methoden an ähnliche Objekte weiter zu geben oder zu delegieren, kommt aber an seine Grenzen, wenn z.B. ein Objekt von zwei verschiedenen Vererbungszweigen erben soll, da die meisten Programmiersprachen diese Mehrfachvererbung nicht unterstützen. Näheres hierzu wird in Abschnitt 2.3.3 erläutert.

Da der größte Teil der Logik im Game-Loop abgearbeitet wird, wird diese mit Zunahme der Komplexität eines Programms immer größer und unübersichtlicher. Außerdem werden die Abhängigkeiten einzelner Aspekte immer verworrener. Nicht zuletzt sorgt eine steigende Anzahl von miteinander asynchron interagierender Objekte für einen schwer zu erfassenden Spielzustand und Konstellationen, die kaum noch vorhergesehen werden können [5].
Ein Problem bei OOP ist auch die Erweiterbarkeit. Aufgrund der komplexen Hierarchie ist ein Objekt stark abhängig von der Struktur des Eltern-Objekts. Das Hinzufügen oder Ändern einer Funktion kann schwierig oder unmöglich sein, sodass der Code hierfür umfangreich umgebaut werden muss [8].

### 2.2.3 Entity System

Die Definition eines Entity Systems (ES) ist nicht eindeutig oder allgemeingültig. Man spricht von ES, Component Systems (CS) oder dem Component Oriented Programming (COP), wobei ES als Teilgebiet von COP angesehen werden kann. In dieser Arbeit wird die Bezeichnung Entity Component System (ECS) verwendet. ECS ist Teil des Systems, das ein bestimmtes Vorgehen nutzt, um Logik und Variablen im Quellcode umzusetzen [14].

ECS ist ein Software Architektur Pattern, das vor allem in der Spieleprogrammierung Anwendung findet. Es folgt dem „Komposition über Vererbung“-Prinzip [8, 5], um beim Definieren von Entitäten - also allem in der Spielwelt von Gegnern über Bäume und Kugeln bis zu Türen - größere Flexibilität zu ermöglichen, indem sie aus individuellen Teilen gebaut werden, die gemischt und verglichen werden können.
Die minimale Anforderung an das ECS Pattern ist zunächst eine Entität, die als ein Container dient welchem Komponenten hinzugefügt werden können. Für gewöhnlich geschieht dies hierarchisch - eine Entität kann also Kind-Entitäten haben. Weiterhin muss es Komponenten geben, die eine Art von Objekt bilden, durch die Verhalten, Aussehen und Daten zu einer Entität hinzugefügt werden können.

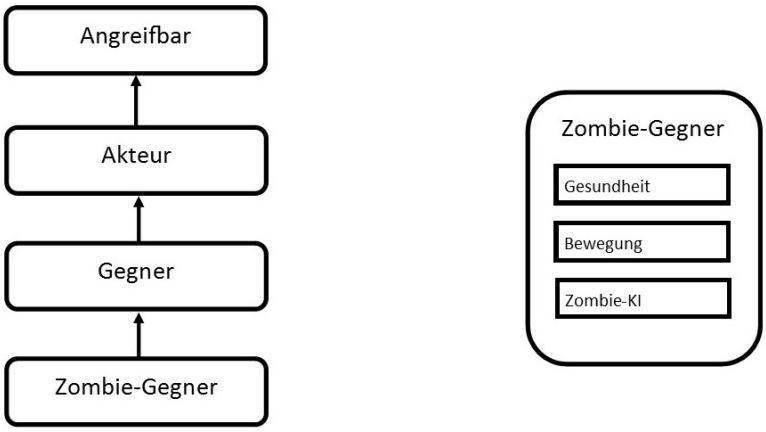


Abbildung 2.2: Zombie-Gegner; links Vererbung, rechts Entität mit Komponenten

In vererbungsbasierten Designs kann ein Gegner in eine komplexe Vererbungshierarchie integriert sein. Gezeigt ist dies in Abbildung 2.2. In der Vererbung werden die Eigenschaften durch die Eltern weitergegeben (z.B. Gesundheit durch die Elternklasse *Angreifbar*). In einem C/ES Design hat man eine Gesundheits-Komponente, eine Bewegungs-Komponente und eine Zombie-KI-Komponente, die man einer Zombie-Gegner-Entität zuweist.

Obwohl ECS viele Probleme von OOP adressieren, haben sie ihre eigenen Probleme. Isolation und Unabhängigkeit sind sowohl positive als auch negative Aspekte von ECS. Die negativen treten auf, wenn das Design das Teilen von Komponenten und somit Inter System Kommunikation bedingt [8] (siehe hierzu auch Abschnitt 2.3.3).
Im folgenden Abschnitt werden ECS genauer vorgestellt.

## 2.3 Entity Component System

Alle ECS funktionieren mit einer Engine - einer Haupteinheit - bei der Entitäten und Systeme registriert werden. Entitäten werden mit Komponenten bestückt, über welche Eigenschaften hinzugefügt werden können. Weiterhin iterieren Systeme über Entitäten mit bestimmten Komponenten.

Der traditionelle Ansatz Spieleeinheiten zu implementieren, wird mit Objektorientierung umgesetzt. Jede Einheit ist ein Objekt und Eigenschaften können per Vererbung weitergegeben werden. Je mehr verschiedene Einheiten man jedoch hat, desto schwerer wird es, diese in die Hierarchien einzubauen. Gerade wenn die neue Einheit Eigenschaften von mehreren Objekten umsetzen soll. Um dieses Problem zu umgehen haben Spieleprogrammierer angefangen, Einheiten durch Komposition anstatt Vererbung zu kreieren. Eine Einheit ist einfach eine Zusammensetzung unterschiedlicher Komponenten. Dies bietet gegenüber der Vererbung einige große Vorteile.

- Es ist einfacher neue, komplexe Einheiten hinzuzufügen. Sie entstehen durch Entitäten mit neuen bzw. einer neuen Kombination von Komponenten.
- Es ist einfacher, neue Einheiten zu definieren, da jede Einheit / Entität neben der ID über ihre Komponenten definiert wird.
- Es ist effizienter, da die Aktualisierung des Spiels durch mehrere, ggf. parallel laufende Systeme, statt über einen einzelnen Game-Loop stattfindet.

[2]

### 2.3.1 State-Pattern

Beim State-Pattern soll das Hauptobjekt sein Verhalten ändern, indem an verschiedene Objekte delegiert wird [17, Kapitel 7, State (Zustand) S.119ff]. Bei vielen Problemstellungen reicht es, sich diese mit einem Zustandsautomaten zu verdeutlichen und diesen mit Klassen und Interfaces umzusetzen.
Beispielsweise kann sich die Fortbewegung eines vom Spieler gesteuerten Charakters von *Stehen* zu *Gehen* ändern und von dort zu *Laufen*. Von *Laufen* kann aber nur über *Gehen* wieder zu *Stehen* gewechselt werden.

### 2.3.2 Strategy-Pattern

Das Strategy-Pattern oder auch Policy-Pattern gehört wie das State-Pattern zu den Verhaltensmustern. Es ermöglicht es, das Verhalten eines Algorithmus zur Laufzeit anzupassen. Der Algorithmus variiert je nachdem von welcher Klasse er genutzt wird.
Spezifizierte Werte werden jeweils in den Algorithmus eingesetzt, ohne dass Code dupliziert werden muss. So können verwandte Klassen ähnliche, aber nicht gleiche, Funktionen umsetzen.

Dieses Pattern dient als Ausgangspunkt für ECS. Die Algorithmen sind in Systemen eingebettet und die einzusetzenden Werte der Berechnungen stecken in den Komponenten der Entitäten, welche über die Zugehörigkeit zu einer Familie ausgewählt werden.

### 2.3.3 Allgemeine ECS Funktionsweise

Entitäten, Komponenten und Systeme lösen den typischen Game-Loop ab und schaffen eine leicht erweiterbare Architektur.

**Komponente:** Eine Komponente ist ein Konstrukt, welches ausschließlich als Datenspeicher fungiert. Sie enthält keinerlei Logik und hat für sich genommen nahezu keine Bedeutung. Erst durch die Kombination mehrerer Komponenten in einer Entität werden diese extrem mächtig. Auch leere Komponenten ohne Daten sind hilfreich, um Entitäten zu markieren [2, 5], was bedeutet, dass diese eine bestimmte Eigenschaft oder Fähigkeit besitzen [14].
Bespiele für Komponenten sind:

- Position(x,y)
- Geschwindigkeit (x,y)
- Gesundheit(lp)

Der beispielhafte Einsatz von Komponenten ist in Abbildung 2.3 dargestellt.

**Entität:** Eine Entität ist ein Konstrukt, das in der Spielwelt existiert. Sie ist wenig mehr als eine Ansammlung von Komponenten und sollte darüber hinaus weder Daten noch Methoden enthalten [14]. Durch diese Einfachheit ist eine Entität meistens nur eine einzigartige ID und allen Komponenten, welche die Entität ausmachen, wird die ID zugewiesen. Bei Bedarf können zur Laufzeit Komponenten zur Entität hinzugefügt oder weggenommen werden [2, 5].
Beispielsweise kann ein Frost-Zauber die Bewegungsunfähigkeit einer Entität bewirken, indem

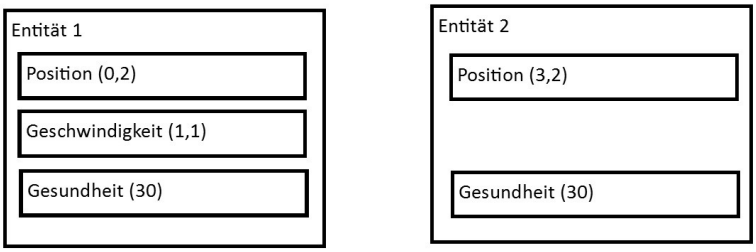


Abbildung 2.3: Komponenten in Entitäten

einfach die Geschwindigkeitskomponente entfernt wird. Wenn die Systeme nun alle Entitäten mit Bewegungs-Komponente neu positionieren, werden die „eingefrorenen“ (ohne diese Komponente) nicht berücksichtigt und bleiben stehen. Dargestellt ist dies in Abbildung 2.4.

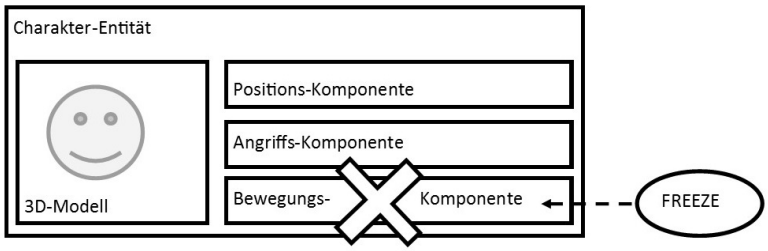


Abbildung 2.4: Entität mit Komponenten. Das Freeze-Ereignis entfernt die Bewegungs-Komponente

**System:** Die Systeme enthalten die Logik. Äquivalent zum Game-Loop wird über alle Systeme iteriert. Hierbei laufen im Idealfall alle Systeme zeitgleich, jeweils in einem eigenen Thread. Ein System arbeitet mit Gruppen von zusammenhängenden Komponenten. Ein Rendering-System beispielsweise arbeitet mit allen Entitäten, die eine Positions-Komponente und eine Texture-Komponente besitzen, um diese auf dem Bildschirm anzeigen zu können. Es stellt die jeweilige Textur an der jeweiligen Position dar. Beispielsweise das 3D-Modell einer Spielfigur auf einem Feld des Spielbretts [5, 14].

Da ein System nur dann auf eine Entität zugreift wenn alle angefragten Komponenten vorhanden sind, definieren Komponenten das Verhalten ihrer Entität. Dies kann ausgenutzt werden, indem man einer Entität „leere“ Komponenten hinzufügt (sog. Marker-Komponenten), um sie von anderen Entitäten zu unterschieden. Beispielsweise eine Gegner-Entität und eine Spieler-

Entität unterscheiden sich nur durch eine Spieler-Komponente. Beide haben außerdem eine Textur, eine Position und eine Geschwindigkeit.

### Komponenten-Pattern statt Mehrfachvererbung

Ashley setzt der Mehrfachvererbung das Komponenten-Pattern / Komposition entgegen, um den Deadly Diamond zu verhindern.

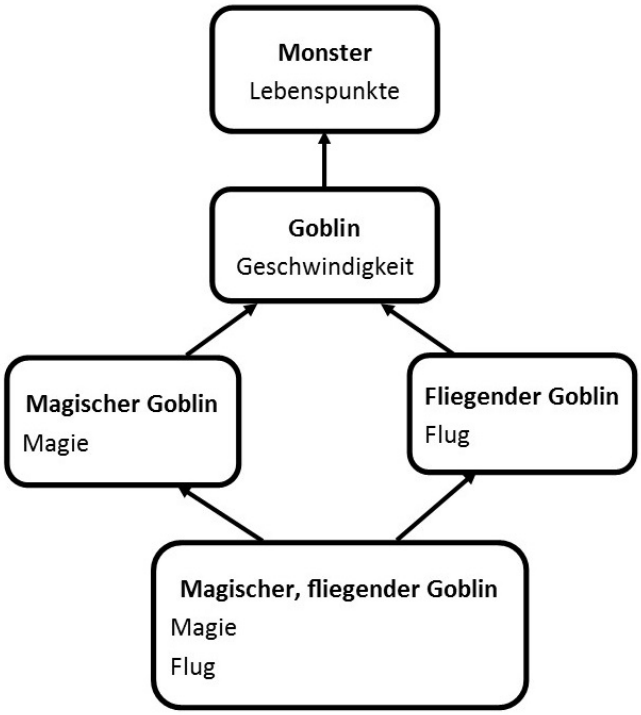


Abbildung 2.5: Deadly Diamond

Als Beispiel dafür denke man sich ein Spiel mit Monstern. Eine Art von Monster sind Goblins. Diese haben Eigenschaften wie Geschwindigkeit und Lebenspunkte. Nun wird ein fliegender Goblin entwickelt, mit einer Fluggeschwindigkeit und ein magischen Goblin mit Magiepunkten. Diese erben die Grundeigenschaften jeweils von der Goblin-Grundklasse. Soll nun ein fliegender, magischer Goblin umgesetzt werden, so ist unklar, ob vom fliegenden oder vom magischen Goblin geerbt werden soll, da Eigenschaften von beiden benötigt werden [19]. Dieses Beispiel ist in Abbildung 2.5 zu sehen.

Das Komponent-Pattern umgeht dieses Vererbungsproblem, indem es auf die Klassenhierarchie verzichtet. Stattdessen gibt es nur eine Goblin-Klasse und zwei Komponenten-Klassen: Fluggeschwindigkeit und Magie. Ein fliegender Goblin wird also zu einem Goblin mit einer Fluggeschwindigkeits-Komponente aber ohne Magie. Bei einem magischen Goblin ist es genau umgekehrt und ein fliegender, magischer Goblin erhält beide Komponenten. Kein doppelter Code, keine mehrfache Vererbung und nur drei statt vier Klassen [17, S.260f].

#### Abhängigkeiten

Da die in ECS genutzten Komponenten lediglich Sammlungen von Daten sind, haben und benötigen sie keine Abhängigkeiten. Systeme verarbeiten Entitäten, welche die entsprechende Kombination von Komponenten haben.

#### Interne System-Kommunikation

Ein Nachteil von ECS ist die interne Kommunikation zwischen verschiedenen Systemen. Dies wird umgesetzt durch Komponenten, die von mehreren Systemen abgefragt werden. Hier können Flags und weitere Informationen gesetzt werden. Diese Komponenten bewirken zusätzliche Iterationen innerhalb der Systeme und somit steigende Ineffizienz. Als mögliche Lösung für dieses Problem kann das Observer-Pattern eingesetzt werden, um die Systeme oder weitere Klassen aktiv auf Veränderungen reagieren zu lassen [20].

#### Iterationskosten

Ein weiterer Nachteil von ECS liegt in der Iteration über Entitäten. Die Systeme iterieren über eine Liste aller Entitäten und wählen die mit der richtigen Kombination von Komponenten aus. Bei vielen Entitäten sind die Kosten hierfür sehr hoch und sie steigen mit der Anzahl an Entitäten. Eine Lösungsmöglichkeit ist die Bildung von Subsets dieser Entitätenliste [20].

## 2.4 Konkrete Frameworks zur Spieleprogrammierung

Es gibt unzählige Frameworks, ausgelegt für verschiedene Plattformen, Geräte und Sprachen. Zur Auswahl stehen funktionale, objektorientierte oder gemischte Ansätze und Pattern.
Die Entscheidung fiel auf Ashley, vorrangig aufgrund von Empfehlung durch Kommilitonen und der Eigenschaft der Plattformunabhängigkeit von LibGDX. Die Entscheidungsfindung wird im folgenden Kapitel (3) thematisiert, an dieser Stelle wird es aber schon einmal vorgestellt.

### 2.4.1 Ashley

Ashley ist ein kleines, in Java geschriebenes, Entitäten Framework. Inspiriert wurde es von Frameworks wie Ash - daher der Name - und Artemis. Ashley versucht ein hoch performantes Entitäten Framework zu sein und die API transparent und einfach nutzbar zu machen (vgl. [18]).
Ashley besteht im Rahmen der *LibGDX* Familie, kann aber auch unabhängig davon eingesetzt werden. David Saltares informiert in seinem Blog salteres.com oder über Twitter @d_saltares über den neuesten Stand. Lizensiert ist Ashley unter der *Apache 2 License* was freie Verwendung, Modifizierung und Verteilung in jedem Umfeld erlaubt.

### 2.4.2 Ash

*Ash* ist ein hoch-performantes Entity System Freamework für die Spieleprogrammierung. Es ist eine kleine Library, welche die Kernklassen zur Verfügung stellt, die benötigt werden, um unter Beachtung der Architektur ein Spiel zu entwickeln. *Ash* ist speziell für Spiele geeignet und vereinfacht fokussiert die Eigenschaften, die bei der Spieleerstellung schwer zu handhaben sind.
*Ash* nutzt die Programmiersprache ActionScript [11], dient als Vorlage für Ashley und ist auch dessen Namensgeber.

### 2.4.3 Weitere

Das *Cupcake* ECS hat eine modular entworfene Architektur mit dem Ziel, besonders einfach Systeme zu entfernen und hinzuzufügen. Die Systeme müssen deswegen komplett unabhängig von einander sein. Dies soll die Erweiterbarkeit und Wiederverwendbarkeit maximieren [8].

*Artemis* ist ein ECS Framework, das für die Spieleentwicklung entworfen wurde. Das Design ähnelt dem von Cupcake. Entitäten sind einzigartige IDs, Komponenten sind Daten und Systeme führen Logik aus, unter Nutzung dieser Daten [8].

Über die genannten hinaus gibt es noch weitere ECS-Frameworks, darunter viele Ansätze, die von einzelnen Programmierern entworfen und auf ihre Bedürfnisse angepasst werden.

## 2.5 Tabletop

Als *Tabletop* oder im englischsprachigen Raum *Miniature Wargaming*, werden Strategiespielsysteme bezeichnet, bei denen Miniaturfiguren aus Metall oder Kunststoff nach vorgegebenen Regeln über eine beliebige Oberfläche bewegt werden - meist Tische oder speziell gestaltete Platten - um einen Konflikt zwischen zwei oder mehr Parteien zu simulieren. Aufgrund fehlender Spielfeldmarkierungen werden Entfernungen für Bewegungs- oder Schussreichweite meist mit Hilfe eines Maßbandes ausgemessen. Das Hobby umfasst drei große Teilaspekte: Das Sammeln und Bemalen von Figuren, das Bauen von Geländestücken und das Spielen gegen andere Spieler.

### 2.5.1 Freebooter Miniatures

*Freebooter Miniatures* [1] ist ein 2002, von dem Miniaturen-Designer Werner Klocke, gegründetes Unternehmen. Ursprünglich wurden ausschließlich Miniaturen produziert, bis 2010 ein eigenes System für diese Figuren veröffentlicht wurde.

*Freebooter's Fate* ist ein schnelles Tabletop Spiel für Miniaturen im 30 mm Maßstab, das Piraten- und Fantasyelemente miteinander verbindet [10]. Das Besondere gegenüber anderen Tabletop-Systemen ist, dass es komplett auf den Einsatz von Würfeln verzichtet. Alle Ergebnisse von Aktionen im Spiel werden über Karten abgehandelt. Es gibt zum einen Trefferkarten, die benötigt werden um zu determinieren, ob Attacken im Nah- oder Fernkampf das anvisierte Ziel treffen. Zum anderen gibt es Schicksalskarten, die Werte für zufällige Ereignisse bieten. Des Weiteren gibt es Ereigniskarten, die nur unter bestimmten Umständen erworben werden können und kleine Bonus oder Malusse im Spiel auslösen.

### 2.5.2 Weitere Tabletop Systeme und Hersteller

Es gibt Unmengen weitere Systeme und Hersteller von Miniaturen, die unterschiedlichsten Genres zugeordnet werden können. Die bekanntesten in Deutschland sollen an dieser Stelle kurz vorgestellt werden. Gemein ist diesen jedoch, dass sie als Zufallselement ausschließlich Würfel benutzen.

*Games Workshop* [2] ist wohl der größte und bekannteste Hersteller von Miniaturen am Markt. Zu den erfolgreichsten Systemen gehören das Scify System *Warhammer 40000* und die Fantasy

[1] Freebooter Miniatures; Oberhausen www.freebooterminiatures.de
[2] Games Workshop; Nottingham, Großbritannien www.games-workshop.com

Systeme *Warhammer Fantasy Battles* und *Der Herr der Ringe*. Des Weiteren haben sie einige Brettspiele veröffentlicht, deren Grenze zum Tabletop fließend sind.
Ein weiterer bekannter Hersteller ist *Privateer Press* [3] mit seinen im Steampunk/Fantasy Setting angesiedelten Spielen *Warmachine* und *Hordes*. Das Einzigartige an diesen Systemen ist, dass sie trotz unterschiedlicher Regeln miteinander kombinierbar sind.
Besondere Erwähnung soll auch noch das Spiel *X-Wing* von *Fantasy Flight Games* [4] erhalten. Da die Figuren bereits bemalt erworben und Bewegungen nicht frei sondern durch Schablonen durchgeführt werden, gehen die Meinungen auseinander, ob X-Wing ein Brettspiel oder ein Tabletop-System ist. Dank der Star Wars Lizenzen und einfacher Regeln ist dieses Spiel jedoch weit verbreitet.

## 2.6 Vergleichbare Spiele, die Tabletopspiele umsetzen

Andere Spiele, welche ein Tabletop-Spiel zum Vorbild haben, entfernen sich von den ursprünglichen Regeln und setzen zumeist ein Echzeitspiel um. Die Ähnlichkeit besteht dann meistens im Genre, dem Setting und den genutzten Charakteren. Beispiele hierfür sind *Total War: Warhammer* und die *Dawn of War* Reihe.
Weiterhin gibt es das Spiel *Bloodbowl (1 + 2)* welches die Regeln des Originalsspiels sehr genau umsetzt. Allerdings ist *Bloodbowl* mehr ein Brettspiel als ein Tabletop, was vor allem durch den in Felder unterteilten Spielplan gegeben ist.

[3] Privateer Press; Bellevue, Washington, USA www.privateerpress.com
[4] Fantasy Flight Games; Roseville, Minnesota, USA www.fantasyflightgames.com

# 3 Konzept

Im Folgenden werden abstrakte Überlegungen angestellt, die im Zuge der Projektplanung getroffen werden müssen. Konkretisiert für das ausgewählte Spiel - Freebooter's Fate - können sie die Implementierung rapide vereinfachen.
Eine Kernentscheidung muss für die grundlegende Architektur getroffen werden.

## 3.1 Auswahl der grundlegenden Architektur

Die Wahl der Architektur hat grundlegende Einflüsse auf die Designentscheidungen und bestimmt das Vorgehen bei der Entwicklung eines Programms. Die folgenden Abschnitte analysieren drei Konzepte die zur Auswahl stehen.
Zu beachten ist, dass Charaktere mit unterschiedlichen Eigenschaften modelliert werden müssen und unterschiedliche Aktionen ausgeführt werden können. Die Spieler müssen abwechselnd handeln können und das Spiel soll einfach zu erweitern sein.

### 3.1.1 OOP mit Gameloop

Dies ist der traditionelle Ansatz. Der typische Game-Loop durchläuft erst alle Aktualisierungen (Systeme) und rendert dann die grafische Darstellung anhand des Spiel-Zustandes. Zu Vergleichszwecken ist dies in Abbildung 3.1 dargestellt.

#### Konkreter Entwurf

Es gibt den Game-Loop als Schleife, um die Logik zu verarbeiten. Einzelne Charaktere sind Objekte, erben von einer Superklasse Charakter und implementieren ihre Sonderregeln. Zur konkreten Umsetzung wurden drei Ansätze erarbeitet:

- Die Superklasse enthält alle Sonderregeln und die einzelnen Charakterklassen setzen ein Flag, wenn sie die Regel besitzen.
- Die Superklasse besitzt nur die Regeln, welche für alle Charaktere gelten. Die einzelnen Charaktere implementieren dann ihre Sonderregeln selbst.

- Beispielsweise für die Imperiale Armada wird eine eigene Superklasse angelegt, in der die Zugehörigkeit und die Sonderregel *Standhaft* implementiert wird.

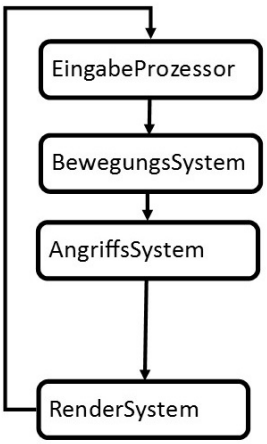


Abbildung 3.1: Geläufiger Game-Loop mit sequentieller Abhandlung der System-Funktionalitäten

### Vor- und Nachteile

Der Game-Loop ist weit verbreitet und lässt sich intuitiv nutzen. Änderungen am System müssen nur an einer Klasse vorgenommen werden.

Mit steigender Komplexität wird der Loop immer größer und undurchsichtiger. Jeder Durchlauf muss auf alle möglichen Veränderungen prüfen um darauf zu reagieren (siehe Abbildung 3.1). Die verschiedenen Ansätze bringen ebenfalls jeweils Nachteile mit sich:

- Die Superklasse wird mit zunehmender Zahl an Sonderregeln immer größer, komplexer und unübersichtlicher. Jedes Charakter-Objekt muss alle Methoden der Elternklasse enthalten, auch wenn es diese nicht nutzt. Dadurch werden diese unnötig aufgeblasen.
- Die Superklasse bleibt übersichtlich, jedoch haben einige Charaktere die gleichen Sonderregeln, sodass diese an mehreren Stellen implementiert werden müssen. Der *Single Point of Control* geht verloren und es entsteht redundanter Code, was die Fehleranfälligkeit drastisch erhöht.
- Die Superklasse ermöglicht es, Eigenschaften die nur die Imperiale Armada betreffen für alle gleich umzusetzen. Da es allerdings auch Charaktere mit der Regel *Standhaft* gibt,

die nicht zur Imperialen Armada gehören, muss dieses wieder an einem anderen Punkt beschrieben oder durch Mehrfachvererbung weitergegeben werden.

#### Fazit

Der Game-Loop ist grundsätzlich ein guter Ansatz für Spiele, jedoch wird die Unübersichtlichkeit bei steigender Komplexität eines Spiels zum Hindernis. Besonders Objekte mit ähnlichen Eigenschaften neigen dazu, in klassischen Vererbungshierarchien Eigenschaften von mehreren Elternklassen zu benötigen. Diese beiden Kriterien schließen OOP für das angestrebte Projekt aus.

### 3.1.2 ECS

Bei der Nutzung einer ECS-Architektur gilt es einige Designentscheidungen zu treffen. Entitäten, Systeme und Komponenten müssen sinnvoll gewählt und auf die Besonderheiten des umzusetzenden Spiels Rücksicht nehmen. für weitere Details siehe Abschnitt 2.3.

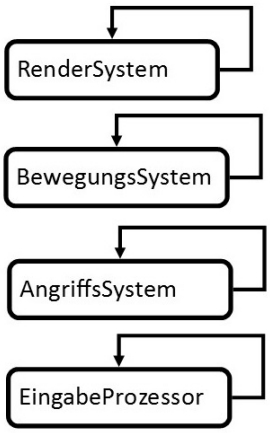


Abbildung 3.2: ECS „Game-Loop“: alle Systeme laufen parallel

#### Konkreter Entwurf

Jede Eigenschaft eines Charakters wird eine Komponente. Jeder Charakter eine Entität. Verschiedene Aktionsmöglichkeiten werden durch einzelne Systeme dargestellt. Diese laufen in eigenen Threads und prüfen in regelmäßigen Abständen, ob Handlungsbedarf besteht.

#### Vor- und Nachteile

Einzelne Charaktere können nach dem Baukastenprinzip zusammengestellt werden. Jede Regel oder Eigenschaft wird nur an einem Punkt im Code beschrieben und Änderungen betreffen alle Entitäten mit der entsprechenden Komponente.

Da es durch Sonderaktionen viele verschiedene Aktionsmöglichkeiten gibt, resultieren daraus ebenso viele Systeme. Diese sorgen für eine große Auslastung. Das Spiel soll auch auf mobilen Endgeräten laufen und da die meisten Systeme nicht gebraucht werden, sollte es vermieden werden den Arbeitsspeicher unnötig zu belasten. Für weitere Nachteile siehe auch Abschnitt 2.3.3.

#### Fazit

ECS sind gut geeignet für Spiele, die viele wiederkehrende Eigenschaften verwenden. Einzelne Systeme können dank Multithreading nebeneinander ausgeführt werden. Da Arbeitsspeicher begrenzt ist und das angestrebte Spiel ausgesprochen viele Aktionsmöglichkeiten beinhaltet - die in entsprechend vielen Systemen resultieren - ist dieser Ansatz nicht geeignet für das Projekt.

### 3.1.3 ECS mit Zustandsautomat

ECS eignen sich hervorragend zur Darstellung der Charaktere. Das Systemmanagement muss jedoch optimiert werden. Dies wird erreicht, indem nur benötigte Systeme aktiviert werden.

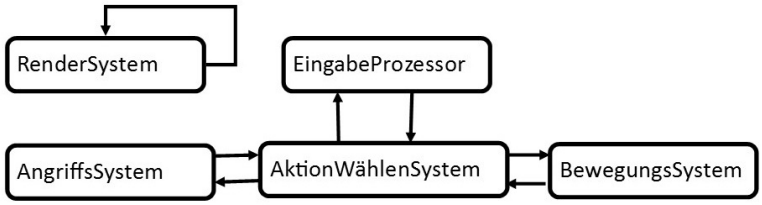


Abbildung 3.3: ECS „Game-Loop“ mit Zustandsautomat

#### Konkreter Entwurf

Wir verwenden grundsätzlich den gleichen Entwurf wie im Abschnitt 3.1.2 jedoch erweitern wir diesen um einen Zustandsautomaten (siehe auch Abbildung 3.7) und aktivieren nur die Systeme, die für die Aktionen des Spielers benötigt werden.

#### Vor- und Nachteile

Wir behalten die Vorteile des vorherigen Entwurfs und gewinnen Performanz durch fehlende aktive Systeme dazu.

Die Zustandsübergänge müssen vom Programm umgesetzt werden. Bei Erweiterungen des Spiels muss der zugrundeliegende Automat beachtet und gegebenenfalls erweitert werden. Für komplexe Spiele, die nicht durch einen Zustandsautomat dargestellt werden können, ist dieser Ansatz ungeeignet.

#### Fazit

Dieser Ansatz ist geeignet für Systeme mit einfachen Zustandsautomaten. Da nur eine begrenzte Anzahl von Aktionen zur Verfügung steht, ist dieser Ansatz gut geeignet, um das Projekt umzusetzen.

Ausgewählt wird das ECS Ashley, angereichert mit einem Zustandsautomaten.

### 3.1.4 Nachteile von ECS

**Interne System-Kommunikation** Beim Entwurf der Architektur und Aufteilung der Informationen auf Komponenten und Systeme sollte beachtet werden, welche Daten von mehreren Systemen benötigt werden, da die zusätzliche Iteration über Komponenten - um Information auszutauschen - zu Ineffizienz führen kann.

Im implementierten Spiel kommt die Nutzung von Komponenten durch mehrere Systeme häufiger vor (siehe hierzu Abbildung 3.9). Es wird die Positions-Komponente vom Rendering-System abgefragt, um das Modell an der richtigen Position anzuzeigen. Das Bewegungs-System nutzt die Position, um den Bewegungsradius zu berechnen und anzuzeigen. Außerdem fragt das Angriffs-System die Positions-Komponente ab, um Charaktere auf benachbarten Kacheln und somit potentielle Angriffsziele auszumachen (und grafisch hervorzuheben).
Diese mehrfachen Nutzungen von Komponenten dienen bisher nicht dem gezielten Austausch von Informationen zwischen Systemen, sondern werden lediglich in verschiedenen Kontexten benötigt. Bei der Erweiterung der Funktionalitäten in der Zukunft und somit voraussichtlich komplexeren Mechanismen ist die Interne System-Kommunikation jedoch weiterhin zu beachten.

**Iterationskosten** Eine hohe Anzahl an Entitäten bewirkt Ineffizienz, da die Systeme über eine Liste aller Entitäten iterieren müssen, um jene mit den richtigen Komponenten auszuwählen.
Es ist bisher nicht davon auszugehen, dass eine Senkung der Performanz aufgrund zu vieler Entitäten auftritt. Die im implementierten Spiel enthaltenen Entitäten beschränken sich auf die Spielfelder (Kacheln) und Mannschaften für zwei Spieler (mit ca. 5 Charakter-Entitäten).

### 3.1.5 Typisches Ashleyspiel

Im *LibGDX Crossplattform Development Cookbook* [13] wird Ashley am Beispiel eines Spieles vorgestellt. Ein einfaches Jump-and-Run Spiel, in dem ein Steinzeitmensch durch eine Welt mit Hindernissen bewegt wird.
Der Prototyp des Spiels zeigt die Spielfigur des Steinzeitmenschen an. Dieser reagiert auf Klicks des Spielers auf dem Bildschirm und bewegt sich in einer festen Geschwindigkeit zu dem angeklickten Punkt.
Neben dem System, das auf den Input des Spielers wartet, laufen auch Systeme zur Darstellung der Szene und Fortbewegung der Figur anhand der implementierten Geschwindigkeit. Komponenten enthalten Informationen zum Rendering, wie Position, Größe und Textur, andere zeigen an, dass die Position verändert werden soll (Bewegungs-Komponente) und dass auf die Eingabe des Spielers reagiert wird. Die Komponenten sind in Abbildung 3.4 dargestellt.

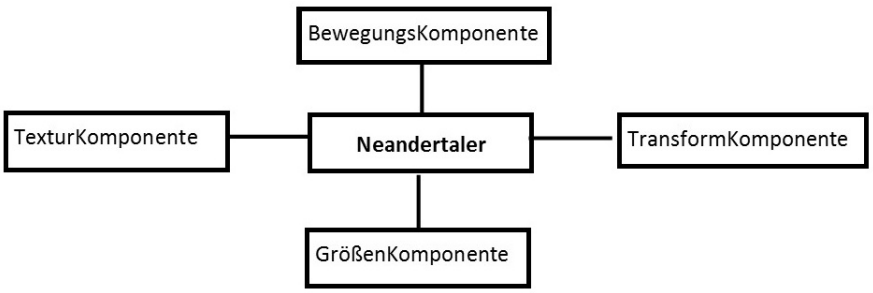


Abbildung 3.4: Komponenten und Entität im Caveman-Spiel nach [13]

### 3.1.6 States

Ein Zug hat vier verschiedene Zustände:

1. Auswählen der Einheiten
2. Auswählen der Aktion

3. Bewegung

4. Angriff

Jeder dieser Zustände wird durch ein eigenes System abgebildet. Die Übergänge zwischen den Zuständen werden entweder automatisch oder über die LibGDX Stage durchgeführt. Jeder Zustand ermöglicht es, nur erlaubte Eingaben des Nutzers zu verarbeiten.

Während der Auswahl der Einheiten können nur diejenigen angewählt werden, die zum aktiven Spieler gehören. Wurde eine Einheit ausgewählt, so wird die Kamera auf die Einheit zentriert und eine Stage eingeblendet, die in einem zirkularen Menü Informationen zum Charakter und mögliche Aktionen anzeigt. Mit der Auswahl wird auch der Zustand gewechselt und es können Aktionen angewählt werden. Gleichzeitig können aber - solange noch keine Aktion angewählt oder durchgeführt wurde - die Einheit noch gewechselt werden.
Wird eine Aktion über die Stage ausgewählt, so wird zum entsprechendem Zustand der Aktion gewechselt und die Anweisungen verarbeitet. Wenn eine einfache Aktion durchgeführt wurde und die Einheit noch Aktionspunkte übrig hat, wird zurück gewechselt zum Aktions-Auswahl-Zustand. Nun kann keine neue Einheit gewählt werden. Wurden alle Aktionspunkte verbraucht, wechselt das Zug-System automatisch den aktiven Spieler und der nächste ist am Zug.

Abbildung 3.5 zeigt Ausschnitte aus dem Spiel, zugehörig zu den verschiedenen Zuständen, die während des Zuges durchlaufen werden.

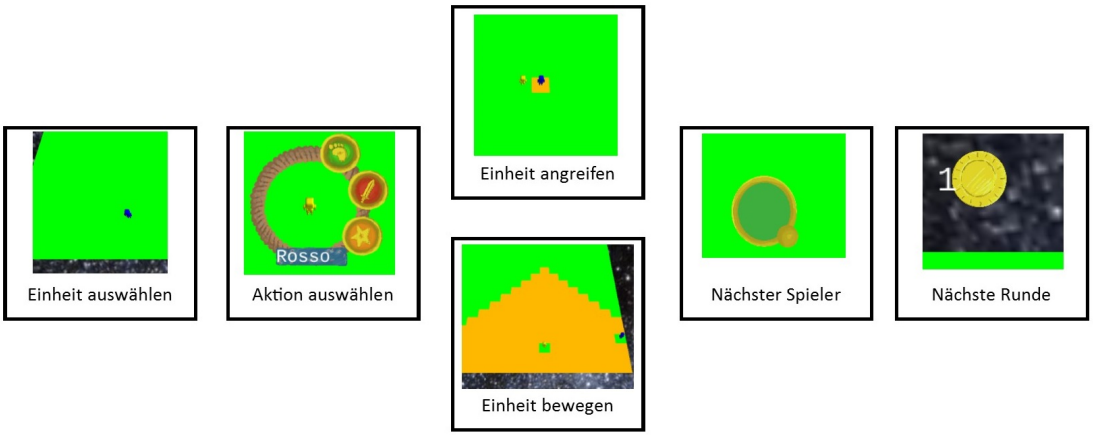


Abbildung 3.5: Relevante Ausschnitte aus dem Spiel zu den verschiedenen Zuständen

### 3.1.7 Alternierende Züge

„Der Spieler mit der ersten Handlung wählt einen beliebigen seiner Charaktere aus und kann die Aktionen der Handlung in beliebiger Reihenfolge durchführen. [...] Hat ein Charakter alle Aktionen seiner Handlung beendet und kann oder möchte er keine weiteren mehr durchführen, so wechselt das Recht der Handlung an den Gegner, und dieser wählt einen beliebigen Charakter, der noch nicht gehandelt hat, aus und führt mit diesem eine Handlung durch. [...] Hat ein Spieler keine Charaktere mehr, die handeln können, so können alle Charaktere des Gegners ihre noch nicht genutzten Handlungen ausführen“ [10, S. 20].
Dies beschreibt eine der zwei häufigsten Möglichkeiten, wie zwei Spieler ihre Mannschaften handeln lassen können. Die zweite Methode, die häufig bei Massensystemen zum Tragen kommt, ist, dass die Mannschaft des einen Spielers komplett handelt und dann die Mannschaft des anderen Spielers.

Die alternierenden Züge werden umgesetzt, indem in der *World* immer festgehalten wird, welcher Spieler der aktive ist. Wenn ein Charakter seine Aktionen durchgeführt hat, wird die Aktions-Komponente von ihm entfernt und der nächste Spieler wird zum aktiven Spieler, indem die entsprechende Methode aufgerufen wird. Nebenher läuft das Zug-System, welches darauf achtet, wie viele Charaktere noch Aktionen durchführen können. Wenn es keinen solchen Charakter mehr auf dem Spielfeld gibt, wird eine neue Runde eingeleitet, indem jedem Charakter, der eine Spieler-Komponente besitzt wieder eine Aktions-Komponente zugefügt und der Rundenzähler in der *World* um eins erhöht wird.

Der Zustandsautomat innerhalb des Spiels wechselt die Zustände aufgrund von Ereignissen, welche durch Eingaben des Spielers oder dem Anpassen von Variablen erreicht werden.

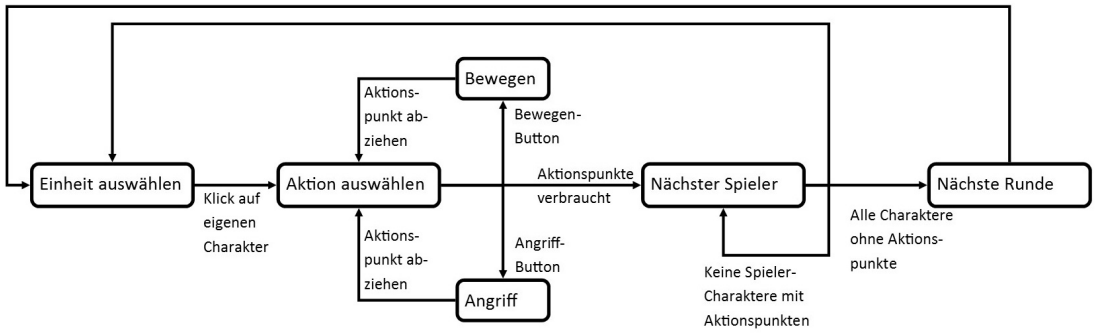


Abbildung 3.6: Zustandsänderungen ausgelöst durch Ereignisse

### 3.1.8 Zugbasiertes Spiel mit Ashley

In Gegensatz zu einem Echtzeitspiel, in dem die meisten Systeme dauerhaft aktiv sind und den Spielzustand überwachen und anpassen, gibt es in einem zugbasierten Spiel Zustände, die durch Systeme dargestellt werden. Das bedeutet, dass diese Systeme nicht gleichzeitig aktiv sein dürfen und jedes System zu einem bestimmten Zeitpunkt im Zug aktiv ist. Es darf z.B. während eines Angriffes das Bewegungssystem nicht aktiv werden, um Einheiten zu verschieben. Der Wechsel der Systeme erfolgt über das Menü und die Optionen, welche die Spieler dort wählen können. Es hat immer nur ein Spieler zur Zeit Zugriff auf das Menü seiner Mannschaft.
Gegeben durch das umgesetzte Spiel Freebooter's Fate haben die Charaktere unterschiedliche Eigenschaften. Diese müssen beim Durchführen von Aktionen berücksichtigt werden und z.B. beim Bewegen einen unterschiedlich großen Radius zulassen.

## 3.2 Freebooter's Fate

Die Umsetzung von Freebooter's Fate bedeutet einen vorgeschriebenen Pool von Aktionen, die unter bestimmten Bedingungen ausgeführt werden dürfen (abhängig von Aktionspunkten oder den Fähigkeiten des Charakters). Die Charaktere haben unterschiedliche Eigenschaften und die Berechnung von Schadenswerten findet anhand von Kartenwerten statt.

### 3.2.1 Charaktere als Entitäten

Die Charaktere der Spielwelt werden im Code als einzelne Entitäten behandelt. Jedem Charakter ist gemein, dass er eine Textur, eine Position, einen kontrollierenden Spieler und Attribute besitzt. Dies sind auch genau die Eigenschaften, die wir als Komponenten darstellen können. Hinzu kommen Schaden und Sonderregeln.

### 3.2.2 Systeme für Aktionen

Mögliche Aktionen, die ein Spieler durchführen kann, werden durch einen Zustandsautomaten festgehalten, dargestellt in Abbildung 3.7, und über ein Menü zugänglich gemacht. Jeder Zustand bekommt ein eigenes System, welches dafür Sorge trägt, dass die Aktion korrekt abgehandelt wird.

Entsprechend wird ein System für jeden Zustand benötigt, um die damit verbundene Aktion global und für die betroffenen Entitäten durchzuführen. Diese entsprechen außerdem den durch

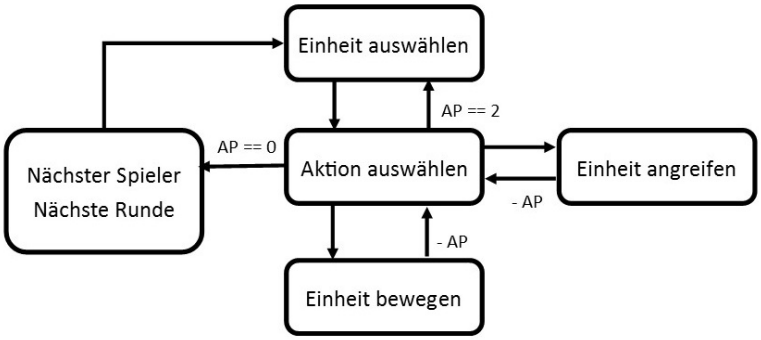


Abbildung 3.7: Automat der Spielzustände

das Spiel vorgegebenen möglichen Aktionen, welche ein Spieler seine Charaktere durchführen lassen kann.

- Bewegungs-System: Bewegt den Charakter auf dem Spielfeld
- Angriffs-System: Greift einen gegnerischen Charakter auf einem benachbarten Spielfeld an (Nahkampf)
- Weitere Aktionen sind durch das Spiel Freebooter's Fate gegeben, im Prototypen aber noch nicht umgesetzt

Darüber hinaus gibt es Systeme zur Regelung der alternierenden Züge und für die Benutzer-Interaktion. Diese werden im Kapitel 4 näher beschrieben.

### 3.2.3 Komponenten für Werte und Eigenschaften

Die Umsetzung der einzelnen Eigenschaften eines Charakters als Komponenten ist naheliegend. Hinzu kommen Komponenten für die grafische Darstellung (3D-Modell und Position auf dem Spielfeld), welche beim originalen Tabletop-Spiel nicht spezifiziert werden müssen. Die beispielhafte Umsetzung von Eigenschaften zu Komponenten ist für den Charakter *Garcia* in Abbildung 3.8 aufgezeigt.

## 3.3 Komponenten und Systeme

Wie bereits in den vorangegangenen Kapiteln erwähnt, werden Charakter-Eigenschaften als Komponenten und verschiedene Aktionen als Systeme umgesetzt. Hierbei gilt es, eine Balance zwischen Systemen und Komponenten zu finden. Für die Erstellung von Komponenten können die Daumenregeln von Martin [16] herangezogen werden:


Abbildung 3.8: Charakterkarte zu Capitan Garcia (linke Grafik mit freundlicher Genehmigung von Werner Klocke - Freebooter's Fate)

1. Komponenten sind Daten, kein Code. Sie repräsentieren den Zustand des Spiels, nicht seine Funktionalitäten.

2. Komponenten sollten so klein wie möglich gehalten werden. Die kleinstmögliche Kombination von Daten zur Repräsentation eines Aspekts.

3. Komponenten sollten sich keine Daten teilen, bzw. es sollte keine Redundanz geben. Im Zweifelsfalle wird eine neue Komponente für die gemeinsamen Daten erstellt.

### 3.3.1 Komponenten

Als Komponenten sollten Verhaltensweisen und Daten umgesetzt werden. Es gibt nun Eigenschaften, die für die Charaktere von Freebooter's Fate mitgebracht werden und solche, die für die digitale Repräsentation nötig sind.
Aus dem Spiel übernommen werden die schon in Abschnitt 3.2.3 erwähnten Eigenschaften, die von den Spielkarten abzulesen sind, sowie die folgenden Komponenten:

- Render-Komponente: Kennzeichnung, dass der Charakter im Renderingprozess dargestellt werden soll.

- logische Positions-Komponente: Enthält x und y Koordinate des logischen Spielbretts.

- tatsächliche Positions-Komponente: Enthält die pixelgenauen x, y und z Koordinaten in der 3D-Welt.

- Modell-Komponente: Enthält 3D-Modell und Textur des Charakters, zur Anzeige in der 3D-Welt.

### 3.3.2 Systeme

Die Systeme beschreiben die möglichen Aktionen eines Spielers, wie bereits in Abschnitt 3.2.2 beschrieben. Zur Durchführung einer Aktion, was eine globale Änderung an einer oder mehreren Entitäten bewirkt, fragen die Systeme entsprechende Komponenten ab.
Abbildung 3.9 zeigt als Schaubild, welche Komponenten von welchen Systemen abgefragt werden.

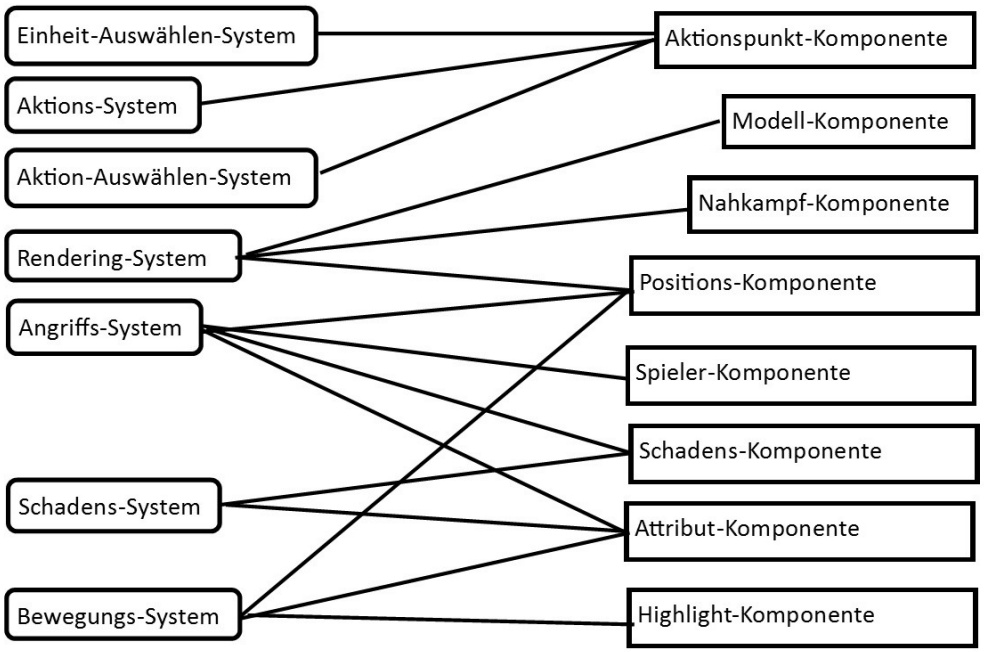


Abbildung 3.9: Abhängigkeiten von Systemen und Komponenten

# 4 Umsetzung

Bisher fand die Konzipierung dessen statt, was zur abstrakten Umsetzung des Spiels nötig ist. Die Implementierung konkretisiert die Strukturen und Automatismen.

## 4.1 Game Design

Unabhängig von der Entscheidung für eine Programmiersprache, Zielsystem und das verwendete Framework bzw. Architektur, müssen trotz der bereits vorhandenen Regeln einige Designentscheidungen getroffen werden.
Die Steuerungselemente (GUI und Kamera) müssen für den Spieler intuitiv zu bedienen sein. Sofern der Spieler mit den Regeln von Freebooter's Fate vertraut ist, soll er in der Lage sein, den Prototypen problemlos zu bedienen. Dies lässt sich allgemein unter dem Stichpunkt Usability sammeln.
Darüber hinaus sollte das Spiel eine gewisse Stimmung vermitteln, was hauptsächlich durch entsprechende Grafik und ggf. Audio-Untermalung umgesetzt werden kann. Minimales Storytelling sollte das Spiel über ein stures Bewegen von Figuren erheben [9].

Einen Rundumschlag zum Thema Spieleentwicklung gibt es zudem im Buch *Core techniques and algorithms in game programming* [3].

### 4.1.1 GUI

Die Interaktionsmöglichkeiten des Users mit dem Spiel sind über eine Stage und per Gestensteuerung möglich.
Eine Auswahl der zur Verfügung stehenden Aktionen ist über Bild-Buttons möglich, die bei Bedarf eingeblendet werden. Näheres zur Stage wird in Abschnitt 4.2.2 behandelt.

Gesten werden zur Steuerung der Kamera genutzt, mit welcher der Blick auf das Spielfeld ermöglicht wird. Mit der Pinch-Geste kann gezoomt werden. Die Auswahl von Charakteren - um den zu agierenden Charakter oder das Ziel eines Angriffes auszuwählen - wird durch

Ray-Picking erreicht. Der Touch oder Klick auf dem Bildschirm wird als Strahl weitergeführt und geprüft, welche Kachel getroffen wird. Steht auf dieser Kachel ein Charakter, so wird dieser ausgewählt.

### 4.1.2 Storytelling

Das implementierte Szenario ist eine so genannte *Fehde*. Zwei Spieler kämpfen hierbei mit ihren Mannschaften gegeneinander, bis 8 Runden gespielt sind oder eine Mannschaft komplett aufgerieben wurde.

Hintergrund der Fehde sind nach dem Regelwerk irgendwie geartete Streitigkeiten zwischen den Mannschaften. Hier ein Beispiel: „[...] ‚Außerdem haben sie Gold und Rum und wir haben demnächst die Verlobungsfeier von unserer Vanessa. ‘‚Aye! Auf sie mit Gebrüll! ‘“ [10, S.96]

### 4.1.3 Architektur

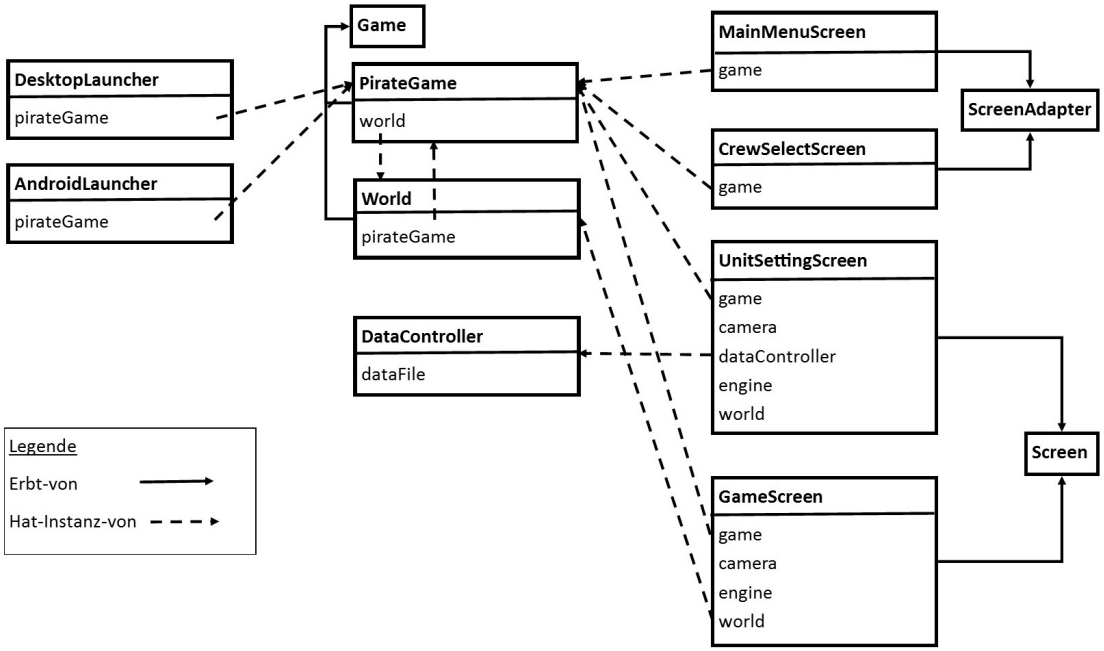


Abbildung 4.1: Von Ashley unabhängige Architektur des Projekts

Neben dem ECS ist die Architektur durch LibGDX gegeben. Hierfür existiert ein Launcher, der in der Hauptklasse den Game-Loop startet. Der Loop organisiert die unterschiedlichen Screens und führt das Spiel über die Klasse *World*. Gezeigt wird dies in der Abbildung 4.1.

Abschnitt 3.3.2 mit Abbildung 3.9 gibt bereits einen Überblick, welche Systeme welche Komponenten abfragen.

## 4.2 LibGDX

„LibGDX ist ein OpenGL Framework mit dem Ziel Programme für unterschiedliche Plattformen zu kompilieren. Unterstützt werden unter anderen auch Android Geräte und Desktop PCs. Dies bietet den Vorteil, dass man seine Programme auf dem Rechner schnell testen und debuggen kann und nur ab und zu auch auf ein echtes Android Gerät exportieren muss.“ [1]
Ashley nutzt das WebGL Framework LibGDX. Dieses wird für plattformunabhängige Implementierung und für graphische Darstellungen genutzt. Die durch das Framework vorgegebene Programmiersprache ist Java.

### 4.2.1 3D Darstellung

Das Spielfeld setzt sich aus 3D-Kacheln von LibGDX zusammen. Die Figuren sind einfache Dummy Modelle, jedoch kann jeder Figur auch ein eigenes Modell zugewiesen werden. Hierfür kann auch die Textur-Komponente angepasst werden. Eine Navigation innerhalb des dargestellten Raumes ist möglich, um Spielsituationen genauer zu betrachten.

### 4.2.2 Interaktionsmöglichkeiten

Der Spieler hat mehrere Möglichkeiten, mit dem Spiel zu interagieren. Diese sollen möglichst intuitiv sein und auch bei anderen Spielen Anwendung finden.
Die Aktionen, welche ein Charakter durchführen kann, werden durch ein rundes Menü dem Spieler zugänglich gemacht. Dieses Menü wird eingeblendet, sobald ein Charakter der aktiven Mannschaft ausgewählt wird. Da die einzelnen Aktionen immer an der gleichen Stelle stehen, kann sich ein Nutzer schnell an das Menü gewöhnen und häufige Menü-Führungen werden schnell auswendig gelernt.

Die Figuren werden durch Ray-Picking ausgewählt. Dies bedeutet, dass wenn ein Spieler auf den Bildschirm etwas auswählt, berechnet wird, was vom Nutzer aus gesehen unter der Maus oder dem Finger liegt. Diese Kachel wird intern markiert und wenn dort eine Figur steht wird diese aktiviert.

Andere Eingaben beziehen sich auf die Kamera. Die Kamera kann in x-y-Richtung verschoben werden und in z-Richtung per Zoom bewegt werden.

**Stage** Stages sind Ebenen, die sich vor das Spiel schieben können und in denen Bilder und Inputabfragen geschehen können. In dieser Arbeit werden Stages genutzt um das Menü darzustellen.
Über die Buttons der Stage nimmt der Spieler Einfluss auf den unterliegenden Automaten, indem er von der Aktion-Auswählen-Aktion aus Aktionen wie Bewegung oder Angriff auswählt. Zudem können auf der Stage Informationen an den Spieler weitergegeben werden. So wird bei der Auswahl eines Charakters dessen Name und mögliche Aktionen angezeigt. Die Stage des Protoypen wird in Abbildung 4.2 gezeigt.

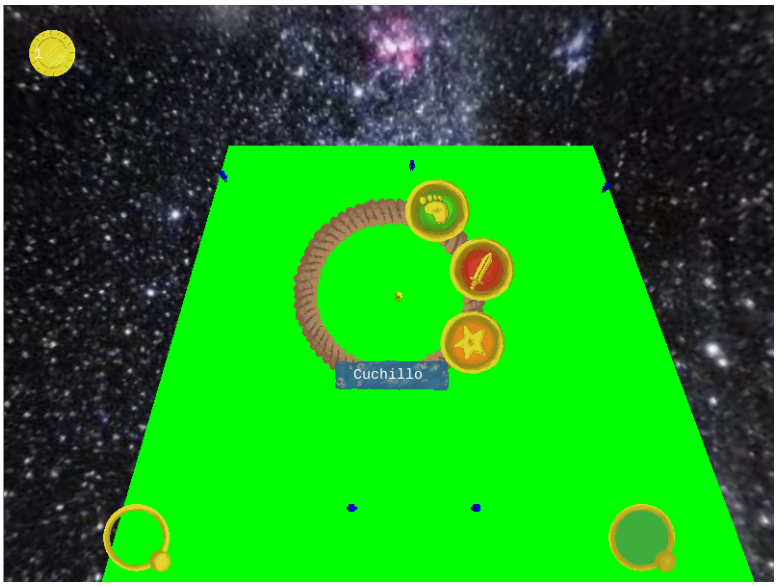


Abbildung 4.2: Screenshot des Spiels bei Anzeige der Stage

### 4.2.3 Unterstützte Plattformen

LibGDX ermöglicht es mit einmalig geschriebenen Code das Spiel sowohl auf Desktop-Rechnern als auch auf Android-Geräten auszuführen. Die Unterstützung von iOS ist möglich, wurde aber aufgrund fehlender Devices nicht thematisiert. Nativ unterstützt es auch die Ausspielung als Webapplikation. Dies wurde im Rahmen dieser Arbeit ausgelassen, da eine installierbare Anwendung angestrebt wird.

## 4.3 Charaktere

Da es mittlerweile beinahe 200 verschiedene Charaktere für Freebooter's Fate gibt, die alle unterschiedliche Regeln besitzen, wurde entschieden, für den Umfang dieser Thesis nur die vom Hersteller angebotenen Starterboxen zum Grundregelwerk umzusetzen. Diese beeinhalten insgesamt 17 verschiedene Charaktere. 4 x Bruderschaft, 4 x Imperiale Armada, 4 x Piraten und 6 x Goblin Piraten. Mit 4 bis 6 Charakteren sind sie gut ausgeglichen um faire, kurze Spiele zum Einstieg in die Welt von Freebooter's Fate austragen zu können, haben aber auch genug unterschiedliche Eigenschaften um die Vielfalt darzustellen. Dies ist genau das, was im Rahmen dieser Arbeit erreicht werden soll.

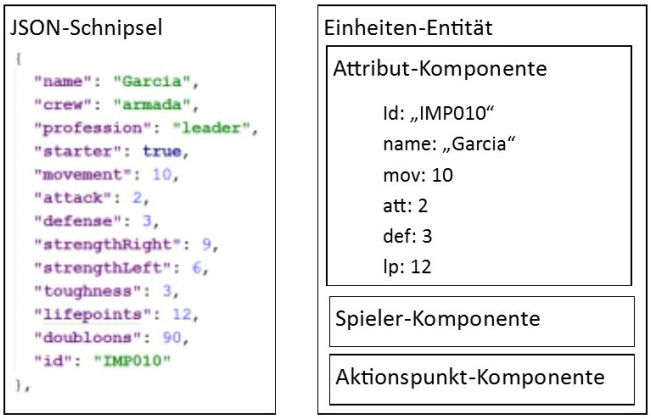


Abbildung 4.3: Ausschnitt aus der JSON-Datei und daraus erstellte Entität mit Komponenten

Die Informationen zu den Charakteren sind in einer Json-Datei festgehalten und können dort problemlos geändert und erweitert werden. Bei Spielstart werden die benötigten Einträge zu den Charakteren eingelesen und daraus neue Entitäten mit Komponenten versehen. Wie die Komponenten aus diesen Informationen gebildet werden, stellt Abbildung 4.3 dar. So kann trotz einer Änderung der Komponenten im Spielverlauf bei einer neuen Spielrunde alles wieder neu eingelesen werden.

## 4.4 Besonderheiten von Freebooter's Fate

Neben den Einzigartigkeiten der Charaktere und den spezifischen Regeln [10] ist die Abwicklung von Kämpfen die Eigenschaft, welche Freebooter's Fate neben anderen Tabletop-Systemen hervor hebt.

Die Trefferermittlung durch Trefferzonen-Karten ist nicht im Prototypen enthalten. Die Schicksalskarten zur Ermittlung des Schadens bei erfolgreichem Treffer werden im folgenden Abschnitt erläutert.

### 4.4.1 Schicksalskarten

Das besondere an Freebooter's Fate ist, dass im Gegensatz zu den meisten anderen Tabletop Spielen, komplett auf den Einsatz von Würfeln verzichtet wird. Stattdessen gibt es Trefferzonenkarten und Schicksalskarten. Da jeder Spieler selbst in der Hand hat, welche Trefferzonen seines Charakters er angreift bzw. verteidigt, sind Treffer keine reine Glückssache sondern hängen von den Pokerfähigkeiten der Spieler und somit vom Können dieser ab.

Des Weiteren gibt es das Schicksalskartendeck, mit dem alle zufälligen Ergebnisse determiniert werden. Dieses Deck besteht aus 40 Karten mit Zahlenwerten von 1 bis 10, die nach Gauß-Verteilung verteilt sind. Die Hälfte der Kartenhintergründe ist entweder schwarz oder weiß, so dass man mit Ziehen einer Karte eine 50 % Wahrscheinlichkeit abbilden kann. Zusätzlich gibt es fünf weitere Symbole, die mit unterschiedlicher Wahrscheinlichkeit auftreten und für Unterschiedliche Ereignisse genutzt werden. Da das Kartendeck für beide Spieler genutzt wird, kann man sich auf die Wahrscheinlichkeit eine bestimmte Karte zu ziehen einstellen und weiß z.B. - wenn eine 10 gezogen wurde - dass sich nur noch eine weitere 10 im Deck befindet. Somit kann man seine Aktionen anders planen als in herkömmlichen Tabletop Spielen.

Umgesetzt wird dies durch das Schicksalskarten-System. Dieses System liest die Karten aus einer Textdatei ein und legt für jede Karte eine Entität an. Diese werden dann zu einem Deck zusammengefasst und per Fischer-Yates-Algorithmus gemischt. Nach außen hin wird nur die DrawCard() Methode angeboten, welche die erste Karte des Decks zurück gibt. Kann keine Karte zurückgegeben werden, wird das Deck erneut gemischt.

### 4.4.2 Zieldevice Tablet

Hauptaugenmerk bei der Implementierung liegt auf einem (Android) Tablet. Ziel ist es, durch den Device möglichst nahe an das Spielgefühl des Tabletops heranzureichen, da dies ebenfalls flach auf dem Tisch liegt und mit der Hand (per Touch-Geste) Figuren verschoben werden können.
Die Verwendung von LibGDX ermöglicht gleichzeitig eine Auslieferung für den Desktop-PC,

sodass auch potentielle Spieler ohne Tablet angesprochen werden können, auch wenn das Spieleerlebnis am Monitor und durch Bedienung mit der Maus deutlich anders ist.

# 5 Evaluation

Das folgende Kapitel evaluiert, ob das gesetzte Ziel dieser Arbeit erreicht, und ein Spiel mit den Grundzügen der Funktionalitäten des Spiels Freebooter's Fate mit Hilfe des ECS Ashley umgesetzt wurde.

## 5.1 Zugbasiertes Spiel mit Ashley

Ziel der Arbeit ist es aufzuzeigen, wie ein Entity Component System möglichst sinnvoll für ein zugbasiertes Spiel genutzt werden kann (siehe hierzu Abschnitt 1.3.1).
Normalerweise ist Ashley für Echtzeitspiele ausgelegt bei denen viele Ereignisse zeitgleich ablaufen. Bei einem zugbasierten Spiel hat man mehr Zeit, um die Auswirkungen eines Ereignisses auszuwerten. Außerdem können nur endlich viele Aktionen gewählt werden, die nicht parallel ausgeführt werden können, was es erlaubt, jeder Aktion ein System zuzuweisen. Diese Systeme müssen nur aktiv werden, wenn sie gebraucht werden. Diese Abhängigkeiten können durch einen Zustandsautomaten visualisiert werden (siehe Abbildung 3.7).

### 5.1.1 LibGDX

Das plattformübergreifende Framework LibGDX ermöglicht es bei gleicher Codebasis, das Spiel für Windows und Android zu entwickeln. Die Entwicklung mit LibGDX und vor allem mit grafischen Komponenten war sehr ungewohnt. Nach dem Einsetzen von Dummy-Grafiken konnte sich allerdings schnell auf die Logik konzentriert werden, auch wenn noch grafische Unstimmigkeiten vorlagen.
Die Architektur von LibGDX mit Launcher und der Game-Loop ist einfach zu adaptieren. Besonderheiten stellten noch das Laden der grafischen Komponenten mit einem Assets-Manager, die Aufteilung in Screens (siehe hierzu auch Abschnitt 5.2.1) und Abhängigkeiten von Bildschirmvariablen (besonders Abmessungen) dar, auf die per *GDX* Zugriff erhalten werden kann.

### 5.1.2 ECS

Der schwierigste Part bei der Umsetzung des Spiels mit Ashley war die Aufteilung von spielrelevanten Informationen in Systeme und Komponenten. Diese Problematik wird ausführlicher in Abschnitt 3.3 beschrieben. Es wurden einige Umsetzungen ausprobiert und verworfen, bevor eine passende Struktur ersichtlich wurde. Auch aktuell könnten durch nähere Analyse noch einige Systeme und Komponenten umstrukturiert werden, wobei hierbei auch schon viel Vorarbeit für etwaige Erweiterungen geleistet wurde.

Der durch das Framework gegebene Umgang mit den ECS-Komponenten - konkret der Engine und den Component-Mappern - war intuitiv und gut dokumentiert. Das zwischenzeitliche Versionsupdate von sowohl LibGDX als auch Ashley ließ sich durch wenige Code-Änderungen durchführen.

### 5.1.3 Zustandsautomat

Die Notwendigkeit eines Zustandsautomaten wurde während der Modellierung der benötigten Systeme ersichtlich. Es durfte nicht zeitgleich auf das Auslösen zwei verschiedener Aktionen (Bewegung und Angriff) gewartet werden; vor allem nicht im Hinblick auf einen erweiterten Aktionspool (mit Fernkampf und Spezialaktionen).
Die verschiedenen, erlaubten Zustände waren hierbei durch die Regeln von Freebooter's Fate vorgegeben [10]. Nach der Zuordnung passender Komponenten musste der Wechsel der Systeme in Ashley umgesetzt werden. Da diese Funktionalität nicht ursprünglich vorgesehen war, gestaltete sich der Zustandswechsel etwas knifflig.

Die erste Idee einer Speicherung des Zustands in einer Variable wurde verworfen, da dies nicht in die ECS-Architektur passen würde. Stattdessen wechseln die Systeme selbst zu ihrem Nachfolger.
Von der Auswahl einer Aktion aus (Aktion-Auswählen-System) wird direkt zum Bewegungs- bzw. Angriffssystem gewechselt. Ein Wechsel findet statt, indem das eine Systeme aus der Engine entfernt und das andere hinzugefügt wird. Die anderen Systeme, welche im Durchlauf des Zustandsautomaten aktiviert werden, wechseln jeweils eigenständig zu ihrem Nachfolge-System.
Diese Aufteilung erscheint zunächst nachteilig, da bei einer Erweiterung um Systeme neue Wechsel eingebaut werden müssen. Allerdings finden diese Wechsel immer von und zum

Aktion-Auswählen-System statt, sodass die Änderungen ausschließlich hier vorgenommen werden müssen.

## 5.2 Funktionsumfang des Prototypen

Der Prototyp soll mindestens die Grundfunktionalitäten implementieren. Hierzu zählen die Bewegung der eigenen Charaktere, der Angriff gegnerischer Charaktere, sowie korrekt funktionierende, alternierende Züge (siehe hierzu Abschnitt 1.3.1).
Zur Umsetzung der grundlegenden Funktionen sollten die Spieler ihre Charaktere auf dem Spielfeld setzen und bewegen können. Die Aktionen sind beschränkt auf Bewegung und Nahkampf. Systeme sorgen dafür, dass die Spieler immer abwechselnd am Zug sind und nur Aktionen durchführen, die auch legal sind. Dies wurde näher im vorhergegangenen Abschnitt (5.1) betrachtet.
Die Spieler können ihre Mannschaften aus vier vorgegebenen auswählen.

### 5.2.1 Screens

Android-Anwendungen - aber auch LibGDX-Anwendungen im Besonderen - sind in Screens organisiert. Diese ermöglichen die Navigation des Benutzers durch die Anwendung und enthalten Menüs oder eigene Anwendungen.


(a) Haupt-Screen (b) Screen zu Auswahl der Fraktionen (c) Screen zur Platzierung der Charaktere

Abbildung 5.1: Screens im Spiel

Ein Ladebildschirm wird angezeigt, während der Assets-Manager die im Spiel benötigtigten Grafiken lädt. Hierzu gehören Texturen, Sprites und 3D-Modelle. Als nächstes wird der Hauptbildschirm angezeigt. Hier kann aktuell nur ein neues Spiel gestartet werden. Zu sehen ist dies in Abbildung 5.1a. Für ein neues Spiel können im folgenden Bildschirm beide Spieler eine Fraktion auswählen, mit welcher sie das Spiel bestreiten wollen. Gezeigt ist dies in Abbildung

5.1b.
Für die Spielaufstellung können die Spieler nacheinander ihre Charaktere auf dem Spielfeld platzieren. Dies geschieht in einem eigenen Screen, der bereits die Spielewelt enthält und in Abbilgund 5.1c dargestellt ist.
Als Haupt-Screen gibt es den Spiel-Screen, in welchen die Hauptanwendung - das Spiel - eingebettet ist.

### 5.2.2 Bewegung

Im Gegensatz zum Tabletop-Vorbild ist das digitale Spielfeld in Kacheln unterteilt. Eine Navigation anhand von Winkeln und Entfernungen wäre für eine zukünftige Version zwar möglich, die Organisation in Kacheln vereinfacht zunächst allerdings die Zugreichweite und Ermittlung von gegnerischen Charakteren in Reichweite (wichtig für die Angriffs-Aktion).
Die Bewegungsreichweite eines Charakters hängt von seiner Bewegungskomponente und seinem Profilwert ab. Erreichbare, nicht bereits besetzte Kacheln, werden hervorgehoben und wenn eine angeklickt wird, so wird die Figur auf dieses Kachel gesetzt.

### 5.2.3 Aktionen

Ein Nahkampf kann nur durchgeführt werden, wenn mindestens ein gegnerischer Charakter auf einer horizontal oder vertikal angrenzenden Kachel steht. Die Trefferermittlung für den Nahkampf entfällt beim Prototypen. Für die Schadensermittlung wird durch das Schicksalskarten-System für die beiden betroffenen Modelle eine Karte gezogen und die Werte der Karte auf die relevanten Werte der Charaktere hinzu addiert. Erleidet der Verteidiger erstmalig Schaden, bekommt er eine Schadenskomponente zugewiesen, in welcher der erlittene Schaden festgehalten wird. Erreicht der Schaden den Lebenspunkte-Wert, dann wird die Figur aus dem Spiel genommen.

Die Aktionen können per Menü ausgewählt werden. Hier sind bereits Aktionsflächen für zukünftige Erweiterungen vorgesehen.

### 5.2.4 Runden

Eine Runde geht so lange, bis alle Modelle beider Mannschaften ihre Handlungen durchgeführt haben. Die aktuelle Runde wird am oberen Bildschirmrand angezeigt. Welcher Spieler aktuell am Zug ist lässt sich dadurch erkennen, dass die Grafik des inaktiven Spielers ausgegraut ist.

Das Spiel endet nach acht Runden oder wenn eine Mannschaft keine Charaktere mehr zur Verfügung hat.

## 5.3 Erweiterbarkeit

Dank Ashley kann das Spiel sehr einfach erweitert werden. Für die Erweiterung um Funktionalitäten und Attribute müssen meist lediglich neue Komponenten und Systeme hinzugefügt werden.

### 5.3.1 Aktionen

Neben Bewegung und Angriff (Nahkampf) gibt es noch viele weitere Aktionen in den Regeln [10]. Diese können durch das Hinzufügen von entsprechenden Systemen und einer Auswahlmöglichkeit im Aktionsmenü ergänzt werden. Weiterhin müssen nötige Komponenten ergänzt werden.
Um z.B. Fernkampf einzubauen bekommt ein Charakter, der eine Fernkampfwaffe besitzt, eine Komponente, in der die Reichweite der Waffe, der Schaden sowie die Komplexität des Nachladens vermerkt wird. Im Menü wird dann für jeden Charakter mit einer Fernkampf-Komponente der Menüpunkt Fernkampf angezeigt. Für den eigentlichen Fernkampf wird dann ein System benötigt, das die Entfernung zum Ziel misst und mögliche Ziele markiert. Wird nun ein Ziel ausgewählt, so wird - wie im Nahkampf - die Trefferermittlung durchgeführt und der Schaden ermittelt.

### 5.3.2 Gelände

Aktuell ist lediglich einfaches Gelände umgesetzt, dass die Bewegung und den Angriff der Charaktere nicht einschränkt. Als Erweiterung gibt es schweres Gelände (z.B. Sumpf), welches die Reichweite verringert. Ebenso soll es Hindernisse - hierunter auch Gebäude - geben, die überwunden werden müssen, oder hinter denen in Deckung gegangen werden kann.

Geländeschwierigkeiten haben in erster Linie Auswirkung auf die Bewegung. Hier müsste bei der Ermittlung der erreichbaren Felder jeweils der Schwierigkeitsgrad des Feldes mit einbezogen werden.
Hindernisse und Gebäude erfordern eine Spielfelderweiterung in der dritten Dimension. Die zugehörigen Regeln machen die Berechnungen der Systeme komplexer. Die Spiel-Kacheln können um jeweilige Komponenten angereichert werden und die Systeme fragen diese ab.

### 5.3.3 Screens

Neben der Erweiterung von Spielfunktionalitäten kann auch die umgebende Anwendung noch erweitert werden. Zusätzliche Screens für Einstellungen können Animationen und Audio-Wiedergabe regeln und informative Screens Credits enthalten und auf die Freebooter's-Internetseite verlinken.
Zukünftig erhalten auch die Netzwerkverbindung zu anderen Spielern und Minispiele eigene Screens.

### 5.3.4 Storytelling

Um das Spieleerlebnis noch anzureichern, bietet sich das Integrieren von Storytelling an. Möglich sind hierfür einfache Sätze, die von den Charakteren kommend eingeblendet werden, oder die Einbettung in einen entsprechenden Kontext. Näheres hierzu beschreibt Abschnitt 6.3.2.

## 5.4 Qualitätssicherung

Die Funktionsweisen, Datenabfragen und Abhängigkeiten können wie für Softwareprojekte üblich mit JUnit überprüft werden.
Um die darüber hinausgehende Qualität des Spiels zu überprüfen, sind Usabilitytests angebracht. Diese haben im Rahmen dieser Arbeit nicht stattgefunden.

Die Usability und Design Überlegungen unterscheiden sich zu denen von andersartigen Softwareprojekten. Die Zufriedenheit des Nutzers steht hier vor Effizienz und Effektivität, da ein Spiel in erster Linie zu Unterhaltungszwecken erworben wird. Wenn es keinen Spielspaß bietet wird es sich schlecht verkaufen [6, S.5].
Zur Überprüfung des Spielspaßes können sowohl Metriken [6] als auch eine Bewertungsskala wie HEP („Heuristic Evaluation for Playability“) [4] angewendet werden.

Da Freebooter's Fate bereits ein erfolgreiches Spielprinzip vermarktet, konzentriert sich die Evaluation auf die digitale Umsetzung, Steuerung über die GUI und möglichst genaue Umsetzung der vorhandenen Regeln.

# 6 Fazit

Das Genre der Tabletop-Spiele und das umgesetzte Spiel Freebooter's Fate im Besonderen können durch eine digitale Version in ihrer Bekanntheit unterstützt werden.
Die Umsetzung mit Hilfe eines ECS bietet viele Vorteile und die Plattformunabhängigkeit des verwendeten Frameworks sorgt für eine zusätzliche Verbreitung.

Es wird das Ergebnis dieser Arbeit noch einmal zusammengefasst, über gemachte Erfahrungen berichtet und zum Schluss ein Ausblick auf weiteres Vorgehen und ein vielleicht zukünftiges, digitales Freebooter's Fate Spiel gegeben.

## 6.1 Zusammenfassung

Im Verlauf dieser Arbeit wurde ein Prototyp entwickelt, der die grundlegenden Funktionen umfasst. Es ist noch kein fertiges Spiel entstanden, jedoch wurde eine Architektur verwendet, die eine einfache Erweiterung ermöglicht und das Einfügen weiterer Elemente vereinfacht.
Zwei Spieler können in alternierenden Zügen ihre Charaktere über das Spielfeld bewegen und gegnerische Charaktere angreifen, wenn sie neben ihnen stehen. Pro Zug hat ein Charakter dabei zwei Aktionspunkte, die für Bewegung oder Angriff verwendet werden können. Ist der zugefügte Schaden gleich oder größer der Lebenspunkte, so ist ein Charakter besiegt und scheidet aus dem Spiel aus.

Die Grafiken sind an der Vorlage orientiert, um den Stil beizubehalten. Das Spiel wurde Werner Klocke vorgeführt und war Grundlage für die weiterführenden Überlegungen, welche in Abschnitt 6.3.2 vorgestellt werden.

## 6.2 Erfahrungen

Das Ashley Framework hat nach einer Eingewöhnungszeit den Prozess der Spiele-Implementierung sehr angenehm und unkompliziert werden lassen. Der Wechsel von OOP zu ECS brachte mehr Flexibilität in der Entwicklung. Die Trennung von Daten und Logik hat es sehr einfach gemacht,

neue Inhalte umzusetzen.
Dadurch, dass das vorliegende Spiel bekannt war, motivierte dies zur möglichst genauen Umsetzung, auch über die Thesis hinaus.

## 6.3 Ausblick

Da das Spiel nur über die Grundfunktionen verfügt sollten weitere Regeln umgesetzt werden. Das Spielen mit den immer gleichen Mannschaften würde Spieler auf Dauer nicht binden, sodass weitere Charaktere und ein Modul zum Zusammenstellen von eigenen Mannschaften in das Spiel integriert werden soll.
Die einfache Fehde - die gegnerische Mannschaft kampfunfähig machen - ist das simpelste Szenario. Es gibt viele weitere Szenarien, die sich oft mit dem Sammeln von Gegenständen, dem Verteidigen bzw. Angreifen von Positionen oder Ähnlichem beschäftigen. Diese Spielmodi bereichern das Spiel ungemein und sollten unbedingt aufgenommen werden.

Gewünscht wird auch ein Rollenspiel (RPG - „role play game“) oder Abenteuer-Spiel, in dem das hier entwickelte Spiel zum Austragen der Konflikte genutzt wird (siehe Abschnitt 6.3.2).

### 6.3.1 Nächste Schritte

Die dringlichsten Erweiterungen, die benötigt werden, sind Fernkampf und Treffer-Ermittlungen. Der Charme von Freebooter's Fate liegt in dem pokerartigen System der Kartenauswahl zum verhindern/erzielen der Treffer, weshalb diesem Schritt besondere Bedeutung zukommen muss.

Bisher werden größtenteils Dummy-Grafiken und -Texturen benutzt. Für ein erfolgreiches Spiel müssen diese angepasst werden.

Seit der zweiten Regelerweiterung ist die Mystik, eine Art Zaubern, ein fester Bestandteil des Spiels. Auch diese Mechanik soll übernommen werden.

Da nur die Mannschaften aus dem Grundregelwerk implementiert wurden gibt es zur Zeit noch drei weitere Mannschaften, die in das Spiel aufgenommen werden können. Dies sind die Dschungelkriegerinnen der Amazonen, die Mystiker des Kults und die Söldner der Ostleonerischen Handelsgesellschaft. Außerdem gibt es noch viele weitere Charaktere für die Grundmannschaften.

#### KI

Die künstliche Intelligenz (KI) sollte es in verschiedenen Schwierigkeitsstufen geben. Da das Schätzen von Entfernungen ein integraler Bestandteil der Regeln ist sollte eine leichte KI sich beispielsweise öfter verschätzen als eine schwere KI, die sich nie verschätzt oder andere sicherere Optionen wählt.
Die KI muss Prioritäten bei den Zielen setzen können. Wenn ein Ziel mehr Punkte bringt, wird es attraktiver, wenn ein anderes schon verwundet ist, sollte aber lieber Dieses angegriffen werden. Ist ein Szenario-Ziel in der Nähe und kann erfüllt werden, so ist der Kampf zu vermeiden.

Weiterhin gibt es Möglichkeiten mit Machine Learning eine KI gegen einen Spieler zu trainieren. Sie lernt seine Vorgehensweise kennen und wird mit jeder Partie effizienter gegen ihn.

#### Netzwerk

Das Spiel kann zur Zeit nur auf einem Gerät gespielt werden. Da die Spieler sich dafür jedoch treffen müssen, könnten sie auch ein normales Spiel mit ihren Figuren austragen. Um die Anreisezeit zu verringern oder mit Spielern aus weit entfernten Regionen spielen zu können, wird eine Netzwerkverbindung benötigt. Eine Möglichkeit wären globale Server, die über das Internet erreichbar sind, auf denen sich die Spieler einloggen können um gegeneinander zu spielen. Da dies jedoch mit weiteren Kosten verbunden ist, sollte hier der Ansatz gewählt werden, dass ein Spieler einen Server auf seinem Gerät öffnet und ein anderer Spieler sich mit diesem verbindet.

### 6.3.2 Freebooter's Fate RPG

Nach Rücksprache mit Werner Klocke - dem Gründer von Freebooter Miniatures - ergab sich, dass eine digitale Umsetzung des Spiels eine Möglichkeit wäre, neue Spieler zu gewinnen und Bestandsspielern erweiterte Inhalte zu bieten.

Das Wunsch-Spiel von Werner Klocke sieht vor, dass der Spieler in die Rolle eines jungen Piraten schlüpft und Geschichten erlebt, die vor den Ereignissen aus den Regelbüchern [10] liegen. Der Pirat muss nicht unbedingt namenlos sein, man könnte beispielsweise in die Rolle des jungen Kapitän Rosso schlüpfen, um diesen auf seinem Weg zu führen. Das eigentliche Spiel wäre ein Rollenspiel (RPG) oder auch ein Point-and-Click-Adventure und wenn es zum Konfliktfall kommt, wird dieser mit dem hier erarbeiteten Spiel abgehandelt.

# Literaturverzeichnis


[1] Benz, Armin ; Schwörer, Mike: Entwicklung eines 2D Tiled Map LibGDX Game Framework. (2015), Nr. April

[2] BorealGames: *Understanding Component-Entity-Systems*. 2013. – URL http://www.gamedev.net/page/resources/{_}/technical/game-programming/understanding-component-entity-systems-r3013. – Zugriffsdatum: 2016-08-14

[3] Dalmau, Daniel Sanchez-Crespo: *Core techniques and algorithms in game programming*. New Riders, 2004

[4] Desurvire, Heather ; Caplan, Martin ; Toth, Jozsef A.: Using heuristics to evaluate the playability of games. In: *CHI'04 extended abstracts on Human factors in computing systems* ACM (Veranst.), 2004, S. 1509–1512

[5] Ehrlich, Justin: The Component Entity System for Virtual Environments. In: *Proceedings of the International Conference on Computer Graphics and Virtual Reality (CGVR)* The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) (Veranst.), 2013, S. 32

[6] Federoff, Melissa A.: *Heuristics and usability guidelines for the creation and evaluation of fun in video games*, Citeseer, Dissertation, 2002. – 52 S

[7] Garcia, Franco E. ; de Almeida Neris, Vânia Paula: A Data-Driven Entity-Component Approach to Develop Universally Accessible Games. In: *International Conference on Universal Access in Human-Computer Interaction* Bd. 8514 Springer (Veranst.), 2014, S. 537–548

[8] Hall, Daniel ; Poly SLO, Cal ; Wood, Zoe: ECS Game Engine Design. (2014)

[9] James, Derek: *Android Game Programming*. John Wiley & Sons, 2013. – 27 – 50 S

[10] Klocke, Werner ; Sander, Franz (Hrsg.): *Freebooter's Fate: Regelbuch Korrigierte Auflage*. 2. korrigierte Auflage. Oberhausen : Freebooter Miniatures, 2011. – 112 S

[11] Lord, Richard: *What is an entity system framework for game development?* 2012. – URL http://www.richardlord.net/blog/what-is-an-entity-framework. – Zugriffsdatum: 2016-08-13

[12] Lord, Richard: *Why use an entity system framework for game development?* 2012. – URL http://www.richardlord.net/blog/why-use-an-entity-framework. – Zugriffsdatum: 2016-08-13

[13] Márquez, David S. ; Sánchez, Alberto C.: *Libgdx Cross-platform Game Development Cookbook*. Packt Publishing Ltd, 2014. – 436 – 443 S

[14] Martin, Adam: *Entity Systems are the future of MMOG development - Part2: What is an Entity System?* 2007. – URL http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/. – Zugriffsdatum: 2016-08-22

[15] Martin, Adam: *Entity Systems: what makes good Components? good Entities?* 2012. – URL http://t-machine.org/index.php/2012/03/16/entity-systems-what-makes-good-components-good-entities/. – Zugriffsdatum: 2016-08-22

[16] Martin, Adam: *Designing Bomberman with an Entity System: Which Components?* 2013. – URL http://t-machine.org/index.php/2013/05/30/designing-bomberman-with-an-entity-system-which-components/. – Zugriffsdatum: 2016-08-22

[17] Nystrom, Robert ; Lorenzen, Knut (Hrsg.): *Design Patterns für die Spieleprogrammierung*. 1. Auflage. mitp, 2015. – 400 S

[18] Saltares, David: *Ashley entity framework*. 2014. – URL http://saltares.com/blog/games/ashley-entity-framework/. – Zugriffsdatum: 2016-08-13

[19] Steinke, Lennart: *Spieleprogrammierung*. 3. Auflage. Heidelberg : BHV, 2008. – 750 S

[20] Wikipedia: *Entity component system: From Wikipedia, the free encyclopedia*. 2016. – URL https://en.wikipedia.org/wiki/Entity_component_system. – Zugriffsdatum: 2016-08-18

*Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.*

Hamburg, 23. September 2016 Michael Hanisch