

Edwards Biotope - Interdisziplinäres Projekt II - Abschlussbericht

01.08.2019

Contents

1 Abstract	3
2 Roadmap	4
3 Organisastion	7
3.1 Planungsgremium	7
3.2 Werbeplakat und technisches Plakat	7
3.3 Projektpräsentation und Demo	7
4 Software-Entwicklung	8
4.1 Implementationen	8
4.2 Renderer Controller Screen	8
4.2.1 Verhinderung der Input-Auswertung	8
4.2.2 Karteneinsicht	9
4.2.3 Spielobjekte	10
4.2.4 Spielabläufe	13
4.2.5 Tiefseekompass	17
4.2.6 Animationen	19
4.2.7 TitleScreen (ohne Bilddatei)	19
4.2.8 Imagebuttons für BattleScreen	21
4.2.9 Überarbeitung des Menüs im BattleScreen	22
4.2.10 Kartenauswahl	23
4.2.11 Karte aus der Hand spielen	23
4.2.12 TitleScreen ausbauen: zum BattleScreen navigieren	24
4.2.13 Deckeditor	24

4.2.14	Karten ziehen (Debug)	24
4.2.15	Visuelles Feedback innerhalb Eintrittsfelder	25
4.2.16	Klickanimation der Buttons	26
4.2.17	Deselektion der BattleCard	26
4.2.18	Spielfeld	27
4.2.19	Musik und Soundeffekte	32
4.2.20	XML befüllen	33
4.2.21	Effekte	33
4.2.22	Siegesbedingung	34
4.2.23	Starter Deck mit drei Quartieren	35
4.2.24	Auflösung auf des Title Screens und des BattleScreens auf HD anpassen . . .	35
4.2.25	Anti Aliasing hinzufügen	36
4.2.26	Camera Handle für Variable Bildschirmgrößen	36
4.2.27	Server Kommunikation Testing mit Router und Lokal	37
4.2.28	Zerstören von Karten	38
4.2.29	Handhabe aktive Spieler	40
4.2.30	Felder	42
4.2.31	Laden und Speichern von Decks	45
4.3	Konzeptionelle und Organisatorische Aufgaben	46
4.3.1	Pflege eines schemenhaften Klassendiagramms	46
4.3.2	Merging und Abnahme	47
4.4	Softwarequalität	49
4.4.1	Qualitätssicherung der Oberfläche	49
4.4.2	Unit-Tests	49
4.4.3	Metriken	50
4.4.4	Manuelle Tests	52
4.4.5	Pair-Programming und Debugging	52
5	Gamedesign	52
5.1	Feinschliff: Fokus auf Stellungsspiel	52
5.2	Die Spielmechanik der Strömung	53
6	Grafik- und Sounddesign	53
6.1	Video-Produktion	53
6.1.1	Einarbeitung in After Effects	54
6.1.2	Animation des Logos	54
6.1.3	Animation des Logos für denn Trailer	55
6.1.4	Kompression des Trailers	56

6.1.5	Storyboard	56
6.1.6	Produktion Video Intro	58
6.2	Illustrationen	59
6.2.1	Lichtgebung und Schattierung	59
6.2.2	Karten kolorieren	59
6.2.3	Spielkarten Outlines	59
6.2.4	Spielkartenrückseite Assets	59
6.2.5	Spielkarten Hintergründe	60
6.2.6	Karten exportieren	60
6.2.7	Hauptmenü-Bildschirm Assets	60
6.2.8	Duell-Bildschrim-Assets	61
6.3	Musik und Soundeffekte	61
7	Abschlussbetrachtung und Fazit	62

1 Abstract

Organisation: Die Gruppe "Edwards Biotope", bestehend aus 10 Leuten, hat sich vor Beginn des Wintersemesters 2018/19 zusammengefunden, um sich eine Projektidee zu überlegen. Es soll ein Strategie Videospiel entstehen. Das Konzept wurde durch interne Projektpräsentationen und Diskussionen fertig ausgearbeitet und in Form eines Projektpapier formuliert. Mit dem fertigen Konzept wurde bei der Themenvorstellung für die Interdisziplinären Projekte vor den Studenten und betreuenden Professoren unseres Semesters für die Idee geworben. Die finale Gruppeneinteilung und der betreuende Professor Schedel waren somit offiziell und endgültig festgelegt. Zu Beginn fanden viele Besprechungen statt, in denen sich geeinigt werden musste, um was es in dem Spiel gehen soll, wie der Stil des Spiels sein soll, welche Frameworks zur Ausarbeitung und Realisierung benutzt werden und wie die Arbeit mit den bevorstehenden Aufgaben organisiert werden soll. Im Team wurde beschlossen, nach der agilen Scrum Methode zu entwickeln (Scrum Master: Sebastian Beck, Pia Korndörfer / Product Owner: Felix Baumgarten, Robert Sabo). Die Infrastruktur zur Projektorganisation sowie Kommunikation wurde mit Hilfe von GitKraken realisiert (Kombination aus GitHub und einfache Projekt-Management Lösung). Damit entsprechend wurde eine zentrale Lösung gefunden, in der man zum einen die verschiedenen Software-Stände und zugehörige Branches verwalten kann, aber auch alle Aufgaben innerhalb des Projekts mittels einem sogenannten „Glo-Board“ dokumentieren und zuweisen kann. Die Marke "Edwards Biotope" wurde als originale IP etabliert und profiliert.

Game Design: Zunächst wurde eine Detailrecherche zu den Game Design-Anforderungen gemacht. Also die Spielmechanik bereits bestehender und erfolgreicher Trading Card Games wurde analysiert und bewertet. Aber auch auf visueller Basis wurden Moodboards zum Spieleuniversum

erstellt. Ein komplett neues, originales Gameplay wurde entwickelt und mit Spielabläufen und Spielregeln festgehalten. Um die Spielmechanik besser nachzuvollziehen, wurde ein analoger Prototyp des Spiels produziert. So konnte die Spielmechanik, vor allem am Anfang unseres Projektes, verstanden und gegebenenfalls verbessert und mit manuellen Tests optimiert werden. Aber auch die Story und das Spieluniversum wurde durch Ausarbeitung und internen Präsentationen geformt und gestaltet. Dabei war es wichtig Spielinterne Mechaniken mit einer Story zu verknüpfen. Im Zuge dessen wurde sich auch auf eine Namensfindung des Produkts "Edwards Biotope" geeinigt.

Ästhetik: Viele Grundlegende ästhetische Vorgaben wurden besprochen und festgelegt. Der Stil der Gesamt-Ästhetik und Tonalität, ein Corporate Design mit Logo, Farbpalette und verwendeten Schriftarten. Aber auch grundsätzliche Bedienelemente wie Icons und Buttons, aber auch Checkboxen und Eingabefelder wurde angefertigt. Um einen einheitlichen Zeichenstil der einzelnen Illustrationen für die Spielkarten beizubehalten, wurde auch ein Work Flow der Illustrationen definiert. Durch Aufteilung in einzelne Abschnitte (Skizzen, Outlines, Colorblocking, Highlights und Shading, Hintergrund) wurde eine einheitliche Illustration gewährleistet. Diese wurde zum letzten Semesterende bis zu etwa 50 Prozent fertiggestellt. Sowohl zum Duell-Bildschirm, als auch zum Kartendeckeditor, wurden mehrere digitale Prototypen als Vorlage erstellt. Kleinere Mechaniken konnten so getestet werden. Diese wurden mit dem Prototyping Tool Figma realisiert.

Software-Entwicklung: Mit ausreichender Recherchearbeit zur idealen Programmiersprache und dazugehörigem Framework, wurde sich gemeinschaftlich auf Java und LibGDX geeignet (für unser Projekt ausreichender Umfang und nicht zu komplex). Zusätzlich wurde jedes Mitglied mit entsprechender Entwicklungsumgebung ausgestattet und ein dezentrales Repository mit Versionsverwaltung eingerichtet. Das favorisierte Werkzeug zur Java-Entwicklung ist die Entwicklungsumgebung IntelliJ IDEA. Um nun bei steigender Komplexität sicherstellen zu können, dass auch mit neuen Implementationen alte Komponenten lauffähig bleiben, wurden automatisierte Softwaretests mit dem Framework JUnit verwendet. Recherchearbeit und Tests zu Multiplayer-Funktionen wurde mit Socket.io und NodeJS gehandhabt. Nach ersten Implementierungen eines Codestamms, sowie einfachen Erstellungen von Klassendiagrammen und weiteren UML-Diagrammen, wurde nach und nach die Architektur der Software spezifiziert und ausgearbeitet. Die ersten Codes wurden schon geschrieben, um eine Basis der noch komplexer werdenden Software zu legen.

Beteiligte Personen: Felix Baumgarten, Sebastian Beck, Pamela Schättin, Manuela Mosandl, Cigdem Bozyigit , Gabriel Veiz, Pia Korndörfer, Philadelphia Gauß, Robert Sabo, Daniel Scharrer (nur im ersten Projektabschnitt dabei)

2 Roadmap

Organisation	
--------------	--

Planungsgremium	Manuela Mosandl
Werbeplakat und technisches Plakat	Robert Sabo
Projektpräsentation und Demo	Robert Sabo, Sebastian Beck, Gabriel Veiz, Pia Korndörfer, Felix Baumgarten
Game Design	
Fokus auf Stellungsspiel	Felix Baumgarten, Robert Sabo
Die Spielmechanik der Strömung	Robert Sabo, Felix Baumgarten
Grafik- und Sounddesign	
Produktion: Musik- und Soundeffekte	
Produktion Video: After Effects Tutorials	Cigdem Bozyigit
Produktion Video: Logo animieren	Cigdem Bozyigit
Produktion Video: Ideenfindung	Alle
Produktion Video: Projekt Video Intro	Manuela Mosandl
Produktion Video: Storyboard	Pamela Schättin, Manuela Mosandl, Gabriel Veiz
Produktion Video: Logo Animation für den Trailer	Pia Korndörfer, Robert Sabo
Produktion Video: Kompression des Trailers	Pia Korndörfer, Robert Sabo
Produktion Video: Animation Spielfeld	Gabriel Veiz
Produktion Musik und Sound	Philadelphia Gaus
Illustration: Lichtgebung und Schattierung	Manuela Mosandl
Illustration: Karten kolorieren	Pia Korndörfer
Illustration: Karten exportieren	Pia Korndörfer
Illustration: Hauptmenü-Bildschirm Assets	Robert Sabo, Pamela Schättin
Illustration: Duell-Bildschirm-Assets	Robert Sabo
Illustration: Spielkartenrückseite Assets	Robert Sabo
Illustration: Spielkartenrahmen Assets	Robert Sabo
Illustration: Spielkarten Outlines	Robert Sabo
Illustration: Spielkarten Hintergründe	Robert Sabo
Software-Entwicklung	
Implementationen	
Verhinderung der Input-Auswertung	Felix Baumgarten
Verhinderung der Input-Auswertung	Felix Baumgarten
Karteneinsicht	Felix Baumgarten
Spielobjekte	Robert Sabo, Felix Baumgarten
Spielabläufe	Pamela Schättin, Gabriel Veiz, Philadelphia Gaus, Felix Baumgarten

Tiefseekompass	Pamela Schättin, Philadelphia Gaus, Manuela Mosandl, Felix Baumgarten
Animationen	Gabriel Veiz, Felix Baumgarten
TitleScreen (ohne Bilddatei)	Cigdem Bozyigit
Imagebutton für BattleScreen	Cigdem Bozyigit
Karte aus der Hand Spielen	Cigdem Bozyigit
TitleScreen ausbauen: Zum BattleScreen navigieren	Cigdem Bozyigit
Deckeditor	Philadelphia Gaus, Manuela Mosandl, Sebastian Beck
Karten ziehen (Debug)	Manuela Mosandl
Kartenauswahl	Manuela Mosandl
Visuelles Feedback innerhalb Eintrittsfelder	Manuela Mosandl, Cigdem Bozyigit
Klickanimation der Buttons	Manuela Mosandl
Deselektion der BattleCard	Manuela Mosandl
Spielablauf	Pamela Schättin, Gabriel Veiz, Philadelphia Gaus, Felix Baumgarten
Spielfeld	Pamela Schättin, Robert Sabo, Gabriel Veiz, Cigdem Bozyigit, Manuela Mosandl, Felix Baumgarten
Musik und Soundeffekte	Philadelphia Gaus
XML befüllen	Pia Korndörfer
Effekte	Pia Korndörfer, Sebastian Beck
Siegesbedingungen	Pia Korndörfer, Sebastian Beck
Starter Deck mit drei Quariteren	Pia Korndörfer, Felix Baumgarten
Auflösung auf HD anpassen	Pia Korndörfer
Anit Aliasing hinzufügen	Pia Korndörfer
Menü im BattleScreen überarbeiten	Pia Korndörfer, Gabriel Veiz, Sebastian Beck
Handhabe aktiver Spieler	Gabriel Veiz, Felix Baumgarten
Camera Handle für Variable Bildschirmgrößen	Gabriel Veiz, Sebastian Beck
Server Kommunikation Testing mit Router und Lokal	Gabriel Veiz
Zerstören von Karten	Gabriel Veiz
Felder	Felix Baumgarten, Gabriel Veiz
Konzeptionelle und Organisatorische Aufgaben	

Pflege eines schemenhaften Klassendiagramms	Felix Baumgarten
Merging und Abnahme	Sebastian Beck, Philadelphia Gaus
Softwarequalität	
Qualitätssicherung der Oberfläche	Pia Korndörfer

3 Organisastion

3.1 Planungsgremium

Aufgabe des Planungsgremiums war die Planung des Präsentationstages. Teil davon war die Organisation des Ablaufes sowie der Örtlichkeit der Präsentationen, als auch die Planung der daran anschließenden Abschlussfeier.

3.2 Werbeplakat und technisches Plakat

Zur Projektpräsentation sollten ein Werbeplakat und ein technisches Plakat erstellt werden. Das Werbeplakat vom letzten Semester wurde mit kleinen Veränderungen übernommen. Letztendlich musste nur ein Plakat zur Erklärung und Visualisierung aller technischer Komponenten hergestellt werden. Diese wurden mit InDesign in einem DIN A1 Querformat realisiert. Dabei war es wichtig im selben Stil wie das Werbeplakat zu gestalten. Es sollten minimalistische Formen und Icons genutzt werden. Bezüge sollte nur durch wenig Text erklärt werden. Bezüge und allgemeine technische Funktionen unseres Videospiels sollen verstanden und erkannt werden.

3.3 Projektpräsentation und Demo

Um nun unseren aktuellen Stand von "Edwards Biotope" an dem Projektpräsentationstag vorzustellen, musste eine Präsentation her. Diese wurde von Robert Sabo und Gabriel Veiz strukturiert und erstellt. Der Fokus lag dabei eher auf das Produkt und weniger auf das Projekt an sich. Es sollte klar gezeigt werden, was "Edward Biotope" ist, was es kann. Präsentiert haben Pia Korndörfer, Gabriel Veiz, Sebastian Beck und Felix Baumgarten. Neben allgemeinen Informationen und visuellen Erklärungen, wurden auch Spielregeln anhand des fertigen Spiel gezeigt. Bei der darauffolgenden offenen Demo konnte "Edwards Biotope" mit Studenten und Gästen getestet und gespielt werden.

4 Software-Entwicklung

4.1 Implementationen

In diesem Abschnitt sind einzeln konkrete Umsetzungen von Anforderungen aufgeführt und über Commit-IDs oder Branch-Namen weiter referenziert.

4.2 Renderer Controller Screen

Wie bereits im Zwischen-Bericht angeführt stellten wir nach mehreren Refactors unsere Grundlegende Architektur auf eine Art MVC-Architektur um. Hierbei dienen die Spielobjekte als die jeweiligen "Models", die Renderer als "View" und die Controller als die gleichnamigen "Controller". Eine besonderheit bilden noch die "Screens" in LibGDX die die zentrale Steuerung der drei komponenten Übernehmen, sobald sie aufgerufen werden. Ursprüngl hatten wir konzipiert, dass wir die einzelnen Screen-Controller-Renderer Konstrukte als separierte Einheiten betrachten und auch so implementieren. Während der Implementation stellte sich jedoch relativ früh heraus, dass es beispielsweise keine separaten Renderer benötigt, da diese immer nur eine Sammlung von Objekten erhielten, diese renderten und nach Veränderungen im Controller diese anzeigen. Daher lies sich dies auf einen zentralen Renderer reduzieren. Ähnlich verhielt es sich auch bei den Controllern - diese setzten wir in einer Vererbungshirarchie um. Die Screens wurden gegen einen AbstractScreen implementiert.

4.2.1 Verhinderung der Input-Auswertung

Für den Benutzer eines Videospiels soll die Bedienung möglichst intuitiv sein. Eine Konsequenz davon ist das so genannte „Polishing“, also der Feinschliff der Schnittstellen. Wenn etwa grafische Elemente den Spieler durch ihre Ausgestaltung durch das Spiel navigieren oder Soundeffekte auf die richtige Art und Weise Feedback geben, können schon im Vorfeld Irritationen und Fragen auf ein Minimum beschränkt werden.

So verhält es sich auch mit der Nutzereingabe in Form von Mausklicks. Daher muss gewährleistet werden, dass die theoretische Möglichkeit, zu jeder Zeit auf jeden Punkt des Bildschirms zu klicken, nicht in einen Wust aus parallelen Spielveränderungen und damit einhergehenden Objektanimationen ausartet.

So implementierte Felix Baumgarten einen Schutzmechanismus für Eingabearbeitungen, der wie folgt funktioniert: Da Nutzereingaben im Rahmen des Game Loops mit jedem darzustellenden Frame erneut abgefragt und ausgewertet werden, müssen sensible Aktionsauslöser im Programmcode lediglich sicherstellen, dass zum Zeitpunkt einer Nutzereingabe keine Animationen mehr abgearbeitet werden. Dies geschieht, indem die Methode `isGoodToGo()` des BattleControllers aufgerufen wird. Diese leitet die Anfrage schlicht an alle Komponenten weiter, die über

Instanzen der Actor-Klasse verfügen, welche wiederum im Spielverlauf animiert sein könnten. Mit der Methode hasActions() wird eine solche Actor-Instanz dann auf gegenwärtig abzuwickelnde Animationen geprüft.

Prüfung der BattleController-Klasse:

```
public boolean isGoodToGo()
{
    return BUTTON_VIEW.isGoodToGo()
        && BATTLEFIELD.isGoodToGo()
        && PLAYER_1.isGoodToGo()
        && PLAYER_2.isGoodToGo();
}

public boolean isGoodToGo()
{
    for (Field field : FIELDS) {
        for (BattleCard battleCard : field.giveCards()) {
            if (battleCard.hasActions()) {
                return false;
            }
        }
    }
    return true;
}
```

Gibt es keine Actions (also Animationen), gibt die Methode isGoodToGo() den Wert „true“ zurück und lässt somit die Auswertung der Nutzereingabe zu. Dieses Verfahren äußert sich im Spielgeschehen etwa darin, dass der Spieler mit dem Klick auf Karten, Felder oder Buttons stets warten muss, bis beispielsweise zuvor geklickte Karten über den Bildschirm geflogen sind. Diese Vorgehensweise birgt die Gefahr, dass erfahrene Spieler in ihrem Spielfluss ausgebremst werden, weswegen es wichtig ist, Animationen in einem zeitlichen Kompromiss aus Zügigkeit und Erkennbarkeit abzuwickeln.

4.2.2 Karteneinsicht

Das Spielkonzept sieht vor, dass Karten in bestimmten Positionen nur von bestimmten Spielteilnehmern eingesehen werden können. Zu diesem Zweck hat Felix Baumgarten der „BattleCard“-Klasse zusätzlich zu ihrer individuellen Vorderseitentextur eine statische Rückseitentextur hinzugefügt, die für jede Instanz die gleiche sein wird. Außerdem verfügt jede „BattleCard“-Instanz über eine „Sprite“-Instanz, die dazu verwendet wird, im Rahmen des Game Loops eine beliebige Textur

auf den Bildschirm zu zeichnen. Nun kann auf Kommando zwischen Vorder- und Rückseitentextur gewechselt werden. Indikator dafür, welche Textur der Sprite zeichnen soll, ist die „Field“-Klasse bzw. ihre zahlreichen Erweiterungen. Deren update-Methode weist den BattleCards nicht nur Positionen und Rotationen zu, sondern kommunizieren ihr auch, auf welche Textur ihr Sprite zugreifen soll. Die Anzeigelogik für eine Karte, die Spieler 1 kommandiert, während er am Zug ist, lautet wie folgt:

	Spieler 1	Spieler 2	Bedeutung
Field	Vorderseite	Vorderseite	Ausgangssituation für jedes Field – jeder Spieler hat Einsicht in die Karte
SupplyField von Spieler 1	Rückseite	Rückseite	Die Karte befindet sich im Kartendeck von Spieler 1 – kein Spieler hat Einsicht in die Karte
HandField von Spieler 1	Vorderseite	Rückseite	Die Karte befindet sich auf der Hand von Spieler 1 – Spieler 1 hat Einischt in die Karte, Spieler 2 jedoch nicht

Beispielsweise überschreibt die „HandField“-Klasse die Methode updateReadabilityOfBattleCard() ihrer Super-Klasse „Field“, die im Rahmen der update()-Methode aufgerufen wird, daher wie folgt:

```
@Override protected void updateReadabilityOfBattleCard (BattleCard battleCard)
{
    Player activePlayer = FIELD_USER.giveBattleController().giveActivePlayer();
    if (battleCard.giveCommander() == activePlayer) {
        battleCard.uncoverYourself();
    } else {
        battleCard.coverYourself();
    }
}
```

4.2.3 Spielobjekte

Als Schöpfer der zu Grunde liegenden Spielmechanik übernahmen Robert Sabo und Felix Baumgarten die Implementierung der wichtigsten Spielobjekte, die im weiteren Programmierverlauf bei Bedarf mit Methoden angereichert werden konnten. Die BattleController-Instanz dient dabei als Knotenpunkt für alle beteiligten Objekte. Sie hält ein Battlefield – den kreisförmigen Bereich für ausgespielte Karten, analog: Spieltisch – sowie zwei Player-Instanzen, die die beiden Spiel-

teilnehmer repräsentieren. Außerdem verwaltet Sie die linke und rechte Seitenspalte jeweils in einer DetailView-Instanz und einer ButtonView-Instanz. Diese Komponenten werden während der Laufzeit nicht mehr verändert und sind daher Konstanten. Einzig der aktive Spieler, die zuletzt angeklickte Karte und ein Token für einen gestarteten Spielzug werden als Variablen gespeichert.

BattleController

```
private BME_PROJECT: BMEProject
public DETAIL_VIEW: DetailView
public BUTTON_VIEW: ButtonView \\
public BATTLEFIELD: Battlefield \\
private Player_1: Player \\
private Player_2: Player \\
private activePlayer: Player \\
private started: boolean \\
private lastClickedBattleCard: BattleCard \\
```

Das Schlachtfeld beschreibt die kreisrunde Feldformation zwischen beiden Spielteilnehmern, auf der im Laufe des Spieles ausgespielte Karten Platz nehmen. Es wird durch die Battlefield-Klasse repräsentiert und ist in sechs Sektoren unterteilt, die in einer konstanten ArrayList gehalten werden. Diese kennt den BattleController und instanziert den Tiefseekompass, der sich in ihrer Mitte befindet.

Battlefield

```
public BATTLE_CONTROLLER: BattleController
public COMPASS: Compass
private SECTORS: ArrayList<Sector>
```

Der Tiefseekompass ist ein Spielobjekt, das sich in der Mitte des Schlachtfeldes befindet und als Compass-Objekt von diesem instanziert wird. Er regelt die Verteilung der drei Farbzonen auf die insgesamt sechs Sektoren des Battlefields, indem er einen Sector speichert, ab dem die Farbreihe mit Rot beginnt. Außerdem speichert er die gegenwärtige Strömungsrichtung in der Variable stream und bietet für deren Veränderungen Schnittstellen in Form von Getter- und Setter-Methoden an.

Compass

```
public BATTLEFIELD: Battlefield
public ZONE_VIEWER: ZoneViewer
private stream: Stream
private startSector: Sector
```

Der FieldUser ist eine kleine Klasse, die eine ArrayList mit Fields bereitstellt. Er fasst die gemeinsamen Bedürfnisse von Sector und Player zusammen und vererbt diesen Klassen daher seine Eigenschaften.

FieldUser

```
protected FIELDS: ArrayList<Field>
```

Der Sector stellt ein Sechstel des Battlefields dar, vergleichbar mit einem Pizzastück. Er erbt vom FieldUser und merkt sich bei seiner Instanziierung das Battlefield, von dem er stammt. Außerdem legt er vier Fields an, auf denen später die Spielkarten platziert werden und verfügt über diverse Methoden, um diese einzeln oder im Kollektiv und wahlweise nach Strömungsrichtung sortiert auszugeben.

Sector

```
private BATTLEFIELD: Battlefield
```

Die Player-Klasse repräsentiert einen Spielteilnehmer und wird daher zweimal vom BattleController instanziert. Sie erbt wie der Sector vom FieldUser und verfügt daher über eine ArrayList mit Fields, die sie während ihrer Konstruktion mit drei Field-Instanzen (SupplyField für das Kartendeck, HandField für die Handkarten, Field für den Ablagestapel) füllt. Außerdem wird ihr mit der Konstanten PARTY ein Eintrag aus der gleichnamigen Enumeration zugewiesen, der aussagt, welcher Partei der Spieler angehört.

Player

```
public BATTLE_CONTROLLER: BattleController  
public PARTY: Party
```

Die BattleCard ist eine der wichtigsten und daher umfangreichsten Klassen im Spiel. Sie erbt von der Actor-Klasse, die das Framework mitliefert, da sie als Dreh- und Angelpunkt aller Interaktionsschnittstellen positioniert, animiert und angeklickt werden können muss. Da sie pro Spieler ca. 40 Mal instanziert wird, die teilweise ein und dieselben Member verwenden, legen wir diese als Klassenattribute an; machen sie also statisch. Dazu zählt die Textur für die Kartenrückseite sowie den -rand, das Interpolationsverfahren und die Dauer für Bewegungsanimationen sowie die Breite und Höhe in Pixeln. Außerdem speichert jede BattleCard ihren Besitzer als PLAYER, ihre Eigenschaften im ihr zugrunde liegenden Datencontainer CARD und ihre individuellen Vorderseiten-Texturen sowie die zu deren Darstellung benötigten Sprites als Konstanten ab. Im Spielverlauf veränderbar und daher als Variablen werden ihr gegenwärtiger Kommandant und ihre aktuellen Hit Points gespeichert. Die BattleCard verfügt zudem über einen InputListener, der auf Mausklicks und Hover-Aktionen reagiert und wird mit zahlreichen Gettern und Settern ausgestattet, um eine solide Kapselung zu gewährleisten.

BattleCard

```
private BACK_TEXTURE: Texture  
private BORDER_TEXTURE: Texture  
private ANIMATION_INTERPOLATION: Interpolation
```

```

private ANIMATION_DURATION: float
public WIDTH: float
public HEIGHT: float
protected PLAYER: Player
public CARD : Card
public FRONT_TEXTURE: Texture
private FRONT_TEXTURE_SMALL: Texture
private SPRITE: Sprite
private BORDER_SPRITE: Sprite
protected commander: Player
private currentHitPoints: int

```

4.2.4 Spielabläufe

Der gesamte Spielablauf ist in Einzelabläufe unterteilt, die überwiegend per Spielereingabe ausgelöst werden. Die Implementierung dieser Einzelabläufe wurde unter Pamela Schättin, Philadelphia Gaus, Gabriel Veiz und Felix Baumgarten aufgeteilt, die den entsprechenden Code mitunter allein und via Pair Programming schrieben.

- **Duell starten**

Um ein Spielduell zu starten, muss der Spieler im Hauptmenü auf den entsprechenden Button klicken. Dabei wird ein BattleScreen-Objekt mitsamt Controller und Renderer generiert und von unserer BMEProject-Instanz aufgerufen. Im Konstruktor des BattleControllers werden alle Objekte angelgt, die im Spiel benötigt werden, darunter das Battlefield, die beiden Player-Instanzen und die Seitenspalten, die etwa spielrelevante Buttons beinhalten. Bei der Konstruktion der Player werden die jeweiligen Kartensätze geladen, als BattleCards instanziert und auf die zuvor generierten Fields verteilt. Anschließend weist der BattleController die beiden Player an, ihre Kartendecks zu mischen und dann die je fünf obersten Karten auf die Hand zu nehmen. Zudem legt er fest, dass Spieler 1 mit seinem Spielzug das Duell beginnt.

```

public BattleController( SpriteBatch spriteBatch , BMEProject bmeProject )
{
    [ . . . ]
    DETAIL_VIEW = new DetailView( stage );
    BUTTON_VIEW = new ButtonView( this );
    BATTLEFIELD = new Battlefield( this );
    PLAYER_1 = new Player( this , Party.ALLY );
    PLAYER_2 = new Player( this , Party.ENEMY );
    activePlayer = PLAYER_1;
}

```

```

    activePlayer.beginTurn();
    [ ... ]
}

```

- **Spielzug beginnen**

Zu Beginn eines Spielzugs nimmt der aktive Spieler stets die oberste Karte von seinem Kartendeck auf die Hand. Dabei muss geprüft werden, ob das Handkartenlimit bereits erreicht ist.

```

public void drawTopCard() {
    if (giveHand().getPileSize() <= 7) {
        BMEProject.cardSound.play(0.4f);
        BattleCard card = giveSupply().pullTopCard();
        giveHand().addBattleCard(card);
    }
}

```

- **Strömung und Farbzoneneinteilung ändern**

Bevor der aktive Spieler durch das Setzen einer Handkarte auf das Spielfeld oder die Aktivierung einer Farbzone seinen Spielzug eröffnet, hat er die Möglichkeit, nach Belieben die Strömungsrichtung und die Farbzoneneinteilung zu ändern. Hierfür stehen ihm zwei Buttons zur Verfügung, die am rechten Bildschirmrand dargestellt werden. Wie alle Buttons sind auch diese mit InputListenern ausgestattet, die auf Mausklicks reagieren. Wichtig hierbei: Laut Spielregeln soll die Änderung von Strömung und Farbzoneneinteilung unterbunden werden, sobald der Spieler seinen Spielzug wie oben beschrieben eröffnet hat. Daher wird der Alpha-Wert der entsprechenden Buttons per Animation reduziert, sodass sie transparent und damit „nicht mehr klickbar“ erscheinen.

- **Handkarte setzen**

Eine Handkarte auf das Schlachtfeld zu setzen ist eine der zwei Hauptaktionen, die ein Spieler in seinem Zug durchführen kann. Hierbei wird eine Handkarte per Mausklick markiert, indem sie im BattleController als „lastClickedBattleCard“ hinterlegt wird, und anschließend mit einem Klick auf eine zulässige Field-Instanz des Schlachtfelds ins Spiel gebracht. Details hierzu sowie Beispielcode halten die Kapitel „Implementierung: Auswahl einer Handkarte“ und „Implementierung: Verhinderung der Input-Auswertung“ bereit.

- **Zone aktivieren**

Eine Zone zu aktivieren ist die zweite Hauptaktion, die einem Spieler während seines Zuges zur Verfügung steht. Sie wird initiiert, indem der entsprechende Button am rechten Bildschirmrand angeklickt wird. Dieser hält einen InputListener, der auf den Mausklick reagiert

und die Methode activateZone() des Battlefields aufruft. Diese arbeitet die folgenden Aufgaben in der folgenden Reihenfolge ab:

- **Buttons deaktivieren**

Da es für den aktiven Spieler ab der Aktivierung einer Zone nicht mehr möglich sein soll, Strömung und Farbausrichtung zu ändern oder die gleiche Zone noch einmal zu aktivieren, müssen entsprechende Buttons abgeschalten und ein Token hinterlegt werden. Dies geschieht im BattleController:

```
public void setTurnStarted()  
{  
    if (!started) {  
        BUTTON_VIEW.fadeOutStartButtons();  
        started = true;  
    }  
}  
  
public void setTurnStarted(Zone zone)  
{  
    BUTTON_VIEW.fadeOutButtonOfZone(zone);  
    setTurnStarted();  
}
```

- **Kartenliste erstellen**

Anschließend soll eine ArrayList mit zu aktivierenden BattleCards erstellt werden, die sich auf den Feldern der Sektoren befinden, über die sich die Farbzone erstreckt. Ob eine Karte aktiviert wird, hängt dabei von ihrem Typ und der Farbzone ab. Parallel dazu sollen Strömungsrichtung und Ringtiefe beachtet werden, die die Reihenfolge vorgeben, in der die Liste später abgearbeitet wird.

- **Karten aktivieren**

Alle Karten, die die zuvor erstellte Liste beinhaltet, werden nun der Reihe nach „aktiviert“. Dazu wird die entsprechende Methode der BattleCard-Klasse aufgerufen. Je nach Kartentyp wird diese Methode nach dem Konzept der Polymorphie unterschiedlich implementiert:

- * **Quartier aktivieren**

Quartiere schieben Karten, die sich auf dem Eintrittsfeld des gleichen Sektors befinden, auf das in Strömungsrichtung nächstgelegene freie Feld.

```
@Override public void getActivated()
```

```
{
```

```
    EntryField correspondingEntryField = giveCorrespondingEntryField(
```

```

        if (correspondingEntryField.giveCards().size() > 0) {
            correspondingEntryField.moveContentStreamwise();
        }
    }
}

```

* Kreatur aktivieren

Kreaturen greifen die nächste gegnerische Karte in Strömungsrichtung an, die sich in der gleichen Zone befindet. Werden die Hit Points einer Karte im Zuge dessen auf 0 reduziert, gilt sie als besiegt und wird auf den Ablagestapel ihres Besitzers versetzt.

```

@Override public void getActivated()
{
    Zone currentZone = giveCurrentZone();
    ArrayList<Field> fields =
        PLAYER.BATTLE_CONTROLLER.BATTLEFIELD.giveRingwiseOrderedFieldsOfZone(currentZone);
    for (int i = (fields.indexOf(giveCurrentField()) + 1); i < fields.size(); i++) {
        ArrayList<BattleCard> fieldCards = fields.get(i).giveCards();
        if (fieldCards.size() > 0) {
            BattleCard potentialTarget = fieldCards.get(0);
            if (potentialTarget.giveCommander() != commander) {
                attack(potentialTarget);
                return;
            }
        }
    }
}

```

* Phänomen aktivieren

Phänomene wickeln ihre individuellen Effekte ab, die im Beschreibungstext der Karte dargelegt sind. Zum aktuellen Zeitpunkt sind solche Effekte noch nicht implementiert.

– Hit Points zurücksetzen

Sobald alle Kartenaktivierungen abgewickelt wurden, werden die Hit Points aller Karten, die sich in der Farbzone befinden, auf ihren Normalwert zurückgesetzt.

– Zone als aktiviert setzen

Eine Zonenaktivierung wird abgeschlossen, indem eine Variable in der entsprechenden Instanz gesetzt wird.

• Spielzug beenden

Die Beendigung eines Spielzugs resultiert stets im Zurücksetzen aller veränderten Umstände wie gesetzten Tokens und deaktivierten Buttons. Im BattleController wird dazu folgende Methode aufgerufen:

```

public void changeActivePlayer()
{
    resetLastClickedBattleCard();
    Zone.RED.deactivate();
    Zone.GREEN.deactivate();
}

```

```

Zone.BLUE.deactivate();
setTurnUnstarted();
BUTTON_VIEW.fadeInZoneButtons();
Player nextPlayer = giveOppositePlayerOf(activePlayer);
nextPlayer.beginTurn();
activePlayer = nextPlayer;
updateAllFields();
showActivePlayerMessage();
}

```

4.2.5 Tiefseekompass

Eine der wichtigsten Eingriffsmöglichkeiten in das Spielgeschehen ist der Tiefseekompass. Er speichert, die gegenwärtige Strömungsrichtung und teilt die sechs Spielfeldsektoren in drei Farbzonen ein. Manuela Mosandl implementierte die Handhabe der Strömung, indem sie die Enum-Klasse „Stream“ anlegte, die wiederum die statischen Einträge „CLOCKWISE“ und „COUNTERCLOCKWISE“ zur Verfügung stellt. Unsere „Compass“-Klasse hielt schließlich eine Feldvariable „stream“, die stets mit einem dieser beiden Enum-Einträge belegt sein würde, und wurde mit entsprechenden Gettern und Settern ausgestattet.

```

CLOCKWISE {
    @Override public int giveDirection()
    {
        return -1;
    }
    @Override public Stream giveOppositeStream()
    {
        return Stream.COUNTERCLOCKWISE;
    }
},
COUNTERCLOCKWISE {
    @Override public int giveDirection()
    {
        return 1;
    }
    @Override public Stream giveOppositeStream()
    {
        return Stream.CLOCKWISE;
    }
}

```

```
};
```

Philadelphia Gaus realisierte die Logik hinter den verschiebbaren Farbzonen: Sie hinterlegte in der „Compass“-Klasse einen unserer sechs in der „Battlefield“-Klasse gehaltenen Sektoren als „startSector“, der angibt, bei welchem Spielfeldsechstel unser Farbkreis der Zonen gegenwärtig mit der Farbe Rot beginnt. Mittels Gettern und Settern könnte dann auf diesen Sektor zugegriffen werden.

```
public void proceedStartSector()
{
    int index = BATTLEFIELD.giveIndexOfSector(startSector) + 1;
    index = BATTLEFIELD.increaseSectorIndex(index);
    startSector = BATTLEFIELD.giveSectorOfIndex(index);
    ZONE_VIEWER.update();
}
```

Pamela Schättin übernahm indes die Generierung eines Zufallssektors, der zu Spielbeginn zum Startsektor werden sollte. Sie machte sich dabei die Java-eigene „Random“-Klasse zunutze, die eine Zufallszahl zwischen 1 und 6 erstellte und den entsprechenden Sektor anhand seines Array-Indexes zuwies:

```
private void initStartSector()
{
    Random random = new Random();
    int randomStartingPoint = random.nextInt(6);
    startSector = BATTLEFIELD.giveSectorOfIndex(randomStartingPoint);
    ZONE_VIEWER.updateRotation();
}
```

Felix Baumgarten visualisierte das Ganze in der Klasse „ZoneViewer“. Hierbei handelt es sich um eine Erweiterung der „Actor“-Klasse, die mit ihren Positions- und Rotationsattributen sowie ihrer Fähigkeit, „Actions“ abzuwickeln (und somit animiert zu werden), die idealen Voraussetzungen dafür bietet. Der „ZoneViewer“ würde zwei Grafiken enthalten, die entsprechend der „stream“- und „startSector“-Variablen des „Compass“ positioniert und rotiert sein würden. Deren Veränderung würde stets in den update-Methoden des „ZoneViewers“ münden.

```
public void updateRotation()
{
    setRotation(giveZoneViewerRotation());
}
private float giveZoneViewerRotation()
{
```

```

    return COMPASS.BATTLEFIELD.giveIndexOfSector(COMPASS.giveStartSector())*60 f;
}

```

4.2.6 Animationen

Um dem Spieler visuelles Feedback über die Geschehnisse im Spiel zu geben und so Abläufe nachvollziehbar zu gestalten, haben Gabriel Veiz und Felix Baumgarten die Spielkarten mit Animationen ausgestattet, die allesamt auf den affinen geometrischen Transformationskonzepten Translation, Rotation und Skalierung basieren. Karten, die etwa durch einen Feldwechsel ihre Position auf dem Bildschirm ändern, würde mittels Translation verschoben. Welchem Spieler welche Karte gehört, erschließt sich aus ihrer Ausrichtung, die wiederum durch ihren Rotationswert bestimmt würde (0: Karte wird regulär angezeigt; 180: Karte wird auf den Kopf gestellt angezeigt). Wählt man etwa eine Handkarte aus, die auf das Spielfeld gesetzt werden kann, soll diese leicht hochskaliert dargestellt werden. Das Framework bietet dafür eine einfache Implementierungsschnittstelle: die Action-Klasse. Jede Instanz der Actor-Klasse, zu der auch unsere Spielkarten gehören, hält von Haus aus eine zunächst leere Liste mit Actions, die wiederum bei jeder Iteration des Game Loops auf ihren Bestand geprüft wird. Übergibt man einer Actor-Instanz eine Action und fügt sie damit dieser Liste hinzu, wird sie ab der darauffolgenden Game Loop-Iteration abgearbeitet.

```

public void moveTo(float x, float y)
{
    MoveToAction moveToAction = new MoveToAction();
    moveToAction.setDuration(0.5f);
    moveToAction.setInterpolation(Interpolation.sine);
    moveToAction.setPosition(x, y);
    addAction(moveToAction);
}

```

Das bedeutet: Ein aktueller Wert (Koordinaten, Rotation, Größe) würde zu einem angegebenen Zielwert (Zielkoordinaten, Zielrotation, Zielgröße) über eine Anzahl an Game Loop-Iterationen hinweg verändert, die sich aus der Verrechnung von angegebener Zeitspanne und festgelegter Bildwiederholungsrate ergibt.

4.2.7 TitleScreen (ohne Bilddatei)

Zwar wurde im Wintersemester bereits an dem Titelbildschirm gearbeitet, allerdings war dies nicht der letzte Stand. Im letzten Semester war es möglich, einen TextButton beziehungsweise ImageButton anzuklicken. Der Anwender konnte somit vom Titelbildschirm zum nächsten Screen navigiert werden. In diesem Semester war das Ziel die Funktionalität zu erweitern. Es sollte ermöglicht werden, zwischen den Screens zu wechseln.

Auf dem Titelbildschirm gibt es nicht nur Bilder sondern auch Buttons. Diese Buttons sind anklickbar und leiten den Nutzer durch das Spiel. Es gibt verschiedene Arten von Buttons. Der TextButton ist zum Beispiel ein Button, womit ein Button mit einem Text erstellt wird. Zusätzlich muss die Schriftart in einem uiskin.json festgelegt und anschließend mit der Skin Methode implementiert werden. Für eine Klick-Funktionalität eines Buttons wird der addListener implementiert. Zum Schluss wird die Stage gerendert.

Eine weitere Möglichkeit einen Button zu implementieren ist der ImageButton. Hier kann der Button selber designet werden. Das fertige Button Bild wird in den assets Ordner eingefügt. Der Texture Methode werden die Button Bilder in der show() Methode zugewiesen. Die Image Buttons werden in einen inititionsButton() implementiert. Hier wird nun ebenfalls ein addListener benötigt. Nach einem Klick auf den Button wird dem Nutzer der TestScreen angezeigt. Da unsere Designer im Team selber Buttons erstellt hatten, fiel die Entscheidung auf die letztere Variante, nämlich auf den ImageButton. Daher wurden die drei ImageButtons „Duell starten“, „Button“ und „Einstellungen“ hinzugefügt.

Screens implementieren

Zumeist wird einem Nutzer ermöglicht, zwischen den Screens zu wechseln. Daher ist es zum einen wichtig, einen neuen Screen aufzurufen und zum anderen wichtig, wieder auf den alten Screen zurückzugehen.

Folglich wurden auf der Titelseite drei Buttons hinzugefügt. Je nachdem welchen Button der Nutzer anklickt, wird der entsprechende Screen angezeigt. Dies wird in der render() Methode wie folgt implementiert.

```
@Override  
public void render(float delta) {  
    Gdx.gl.glClearColor(0x52 / 255.0f, 0x9D / 255.0f, 0xBF / 255.0f, 0xff /  
    255.0f);  
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);  
  
    if (imageButtonBattle.isPressed()) {  
        BME_PROJECT.activateBattleScreen();  
    }  
    if (imageButtonDeck.isPressed()) {  
        BME_PROJECT.activateDeckScreen();  
    }  
}
```

Da der Settings-Button keine hohe Priorität hat, ist dieser zunächst einmal auskommentiert. Wenn nun beispielsweise auf dem Titelbildschirm der „Duell starten“-ImageButton geklickt wird, wird dem Nutzer der Duell Screen angezeigt. Der Nutzer hat hier die Möglichkeit sein Deck zu

bearbeiten oder wieder zurück auf die Titelseite zu gehen. Die folgende Abbildung veranschaulicht diese Implementierung.

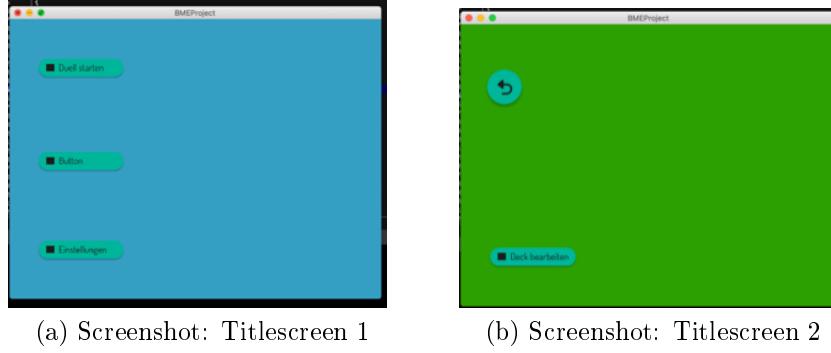


Figure 1: Screenshots Titlescreen

In der render() Methode wird festgelegt, welcher Screen nach einem Klick auf den ImageButton-Back angezeigt wird. In diesem Fall, soll das Startbildschirm erneut angezeigt werden. Nachdem mit einer If-Anweisung überprüft wird, ob der zurück Button geklickt wurde, wird das Startbildschirm angezeigt.

Schließlich wurde durch diese Implementierung ermöglicht, die Spielumgebungen durch Klicks auf die Buttons anzusteuern und zwischen den Screens zu wechseln.

4.2.8 Imagebuttons für BattleScreen

Anfangs wurden die Screens mit einer Hintergrundfarbe festgelegt. Allerdings war das nur vorübergehend. In diesem Semester erstellte Robert Sabo alles, was für den BattleScreen benötigt wurden. Alle Bilder, die hierfür auf diesem Screen benötigen werden, wurden zunächst einmal in den Assets-beziehungsweise Visuals-Ordner eingefügt. Das Spielfeld sah anfangs an den Seiten leer aus. Das liegt daran, dass alle Buttons hinzugefügt und richtig positioniert werden mussten. Das Endergebnis ist in Abbildung 2 zu sehen. Um testen zu können, ob der Klick auf den Button funktioniert, wurde zunächst einmal in der Console die Information „Zone successfully clicked“ wie folgt aus-



Figure 2: Screenshot: BattleScreen mit ImageButtons

gegeben.

```
450.0 breite : 800.0
BATTLE SCREEN SHOWN
Zone successfully clicked
```

4.2.9 Überarbeitung des Menüs im BattleScreen

Um den Spieler das Spielen etwas zu erleichtern, wurden ein paar Hilfestellungen implementiert. Zunächst sollte der Spieler wissen, wer gerade an der Reihe ist. Hierfür wurde für die Klasse "Party" dem „Spieler 1“ der Imagepath zu dem Asset „du_bist.png“ zugeordnet. Dem „Spieler 2“ wurde der Imagepath zu dem Asset „gegner_ist.png“ zugeordnet. Der Path wird der Klasse "BattleControl" übergeben. Dort wird die Methode showActivePlayer implementiert. Diese Methode zeichnet das Bild an der gewünschten Stelle. Die Methode wird in der Methode „changeActivePlayer“ ausgeführt. Des Weiteren sollte der Spieler wissen, welche Effekte die unterschiedlichen Buttons im Spiel haben. Dafür wurde die Methode „setUserMessage“ in der Klasse „DetailView“ implementiert. Dieser Funktion wird jedes mal der String „userMessage“ übergeben, wenn der Spieler über einen Button, das Grab oder seine Handkarten fährt. Folgende Objekte benötigten folgende User Message:

- Strömungsrichtung ändern Button - Die Strömung im Rücken ist vom Vorteil. Sie lässt dich zuerst Angreifen
- Handkarten - Setze deine Einheit auf die gewünschte Kampfposition
- Kompassausrichtung ändern - Drehe den Kompass um die Sektoren mit Strahlung auszusetzen. Die verschiedenen Strahlungen beeinflussen die Einheiten auf unterschiedliche Weise
- Grab - Diese Einheiten stehen dir für den Kampf nicht mehr zur Verfügung
- Rote & Blaue Strahlung Aktivieren Button - Deine Kreaturen und Phänomene werden durch die rote/blaue Strahlung zu wahren Kriegsmaschinen
- Grüne Strahlung Aktivieren Button - Die Grüne Strahlung schafft dir mehr Raum um neue Einheiten in den Kampf zu senden
- Zug beenden Button - Beende deinen Zug

Die Nachricht an den Spieler muss an der gewünschten Position, im gewünschten Absatzformat und mit automatischen Umbrüchen bei langen Texten angezeigt werden. Dies kann mit den Funktionen .setPosition, setWidth, .setHeight, .setAlignment und .setWrap aus dem Framework libGDX erfolgen. Um eine eigene Schriftart einzubinden wird der FreeTypeFontGenerator von

libGDX benötigt. Dieser wandelt eine True Type Font in eine BitMap Font in der gewünschten Größe um. Für den Font muss ein linearer Filter für die Kantenglättung aktiviert werden.

Siehe im Git-Repository in den Commits:

- 11e47f46e37f4ab279a322218272dbd34607bd0b
- 02d6bfa7794a80f599a048de649a22a8b060f047
- db2c258eb297e9354fe680d54f02bee5e6bd0800

4.2.10 Kartenauswahl

Für die Auswahl einer bestimmten BattleCard und die damit verbundene Aktivierung bzw. Aktionssauslösung mussten erst entsprechende Vorbereitungen getroffen werden. Zum einen benötigte die Klasse an sich einen InputListener des Typs “touchDown”, welcher für die Auswahl der Karte durch Anklicken benötigt wird. Das Hinzufügen des InputListeners stellte die Klasse Actor, von der die BattleCard erbt, über die Methode addListener bereit. Ihr musste nur die Klasse InputListener übergeben werden, bei der man direkt interne Anpassungen durchführen konnte. So wurde die von ihr beinhaltende Methode touchDown für unser Vorhaben genutzt, indem sie die von dem Klick betroffene Karte als “zuletzt angeklickte Karte” sicherte. Für eine flexiblere Verwendung der dadurch aktivierte Karte, wurde sie im BattleController als solche gesetzt und war damit dank Getter-Methoden auch von den restlichen Klassen abrufbar.

4.2.11 Karte aus der Hand spielen

Bei jedem Kartenspiel ist es wichtig, die Karte, welche aus der Hand gespielt wird, abzulegen. Um diese Funktionalität an der richtigen Stelle einzubauen zu können, wurde zunächst einmal der Code analysiert. Mit Hilfe der Ausgabe auf der Consolen-Ausgabe, konnte ermittelt werden, welche Klassen und Methoden hierfür benötigt wurden.

Die giveLastClickedBattleCard() Methode in der BattleController Klasse gibt die lastClicked-BattleCard, mit dem Rückgabetyp BattleCard, zurück. Mit Hilfe dessen, wurde zu aller erst ermittelt, ob die selecteCard null ist. Denn nur wenn eine Karte erst ausgewählt wird, kann diese auch wieder abgelegt werden. Nach einer weiteren if-Anweisung konnte mit Hilfe der Methoden addCard() und resetLastClicketBattleCard() die Funktionalität, die Karte aus der Hand zu legen, implementiert werden. Die folgenden Zeilen ermöglichen diesen Spielzug.

```
@Override public boolean touchDown(InputEvent event, float x, float y, int poin
{
    BattleController battleController = FIELDABLE.giveBattleController();
    BattleCard selectedCard      = battleController.giveLastClickedBattleCard();
```

```

if (selectedCard != null) {
    addCard(selectedCard);
    battleController.resetLastClickedBattleCard();
}
return true;
}

```

Damit wurde somit dem Anwender die Option gegeben, seine Karte auf dem Spielfeld abzulegen.

4.2.12 TitleScreen ausbauen: zum BattleScreen navigieren

Sobald das Projekt gestartet wurde, wurde uns der BattleScreen angezeigt. Daher musste der TitelScreen ausgebaut werden. Am Anfang des Semesters wurde diese Funktionalität zwar bereits programmiert, allerdings gab es während dem Semester Code-Änderungen. Das Resultat dieser Änderungen ergab, dass trotz eines Klicks auf den Button, der Anwender weiterhin den Titelbildschirm sah und somit nicht navigiert wurde.

Dabei wurde festgestellt, dass der Verweis auf den BattleScreen nicht sofort funktionierte, weshalb der vorhandene Code analysiert wurde. Der TitelScreen erbt von AbstractScreen. Dieser enthält die Methode createController mit dem Parameter SpriteBatch spriteBatch. Allerdings muss diese Methode in der TitelScreen Klasse erweitert werden. Daher musste der Konstruktor der TitelController Klasse mit einem zweiten Parameter, dem TitelScreen titelScreen, ergänzt werden. Des Weiteren wurde im TitelScreen der return Wert von der creatController Methode angepasst. Schließlich funktionierte wieder der Verweis auf den BattleScreen.

4.2.13 Deckeditor

Um dem Spieler die Möglichkeit zu verschaffen ein eigenes, individuelles Spielkartendeck zu erstellen, war anfangs ein Deckeditor geplant, der vom Titelscreen aus aufgerufen werden sollte. Die Klassen Deck, DeckScreen und DeckController wurden hierfür implementiert. Sie waren anfangs auch für interne Tests hilfreich, da sie alle Spielkarten hielten und dadurch der Zugriff auf Daten wie Kartename oder Typ möglich war. Da die Priorität am Ende jedoch auf einer funktionierenden, visuell ansprechenden Spielfunktion lag, wurde der Deckeditor, und damit auch die Ausarbeitung der Deck-Klassen, auf Eis gelegt.

4.2.14 Karten ziehen (Debug)

Zur Visualisierung der Kartenziehung wurde vorerst vom Kartenstapel jeweils eine Karte auf das HandField verschoben. Dafür musste die drawTopCard-Methode der Player-Klasse innerhalb des BattleControllers aufgerufen werden. Dieser Aufruf musste nur noch mittels eines Impuls ausgelöst werden, für diesen Fall bot sich die von LibGDX bereitgestellte Tastendruckabfrage an:

```

if (Gdx.input.isKeyJustPressed(Input.Keys.SPACE)) {
    activePlayer.drawTopCard();
}

```

Vorerst wurde diese Aktion mittels der Leertaste ausgeführt, wobei der Auslöser im Laufenden noch ausgearbeitet wurde.

4.2.15 Visuelles Feedback innerhalb Eintrittsfelder

Um dem Nutzer das Spielprinzip während seines Spielzuges zu erleichtern, bot sich eine visuelle Rückmeldung für die zum Ablegen von BattleCards ansteuerbaren Felder an. Um die Aufmerksamkeit des Nutzers also auf die möglichen Ablageorte zu lenken, wurde als Form der Kennzeichnung ein Asset in Form eines gelben Rahmens mit glühendem Effekt angefertigt. Dieses wird schließlich um das Feld oder die Felder pulsartig aufleuchten. Der erste Schritt in der Implementierung dieser Funktion lag im Hinzufügen des Assets in alle Ablagefelder (EntryFields), wobei seine Transparenz auf Null gesetzt wurde. Optisch änderte sich am Spielfeld also noch nichts. Bei diesem Schritt kam es jedoch zum Problem, dass sich das Verhalten des Ablagefeldes funktional veränderte, so wurde ein Mausklick auf dieses nicht mehr erkannt. Ein Kartenablegen auf dieses Feld war damit unmöglich. Der Fehler lag hierbei in der Positionierung des Assets, das fälschlicherweise über den Feld-Actor gelegt wurde, der den Mausklick erhält. Dafür musste man die Assets auf eine niedrigere Ebene als die des Actors legen. Für diese Positionierung setzte man die Z-Position der Assets neu:

```

int zCor = getZIndex() - 1;
glowBorder.setZIndex(zCor);

```

Der zweite Schritt wird durch den Mausklick auf die BattleCard angestoßen. Hier wird vorerst durch alle sechs Sektoren durchiteriert:

```

public void activateSectorOfPlayer(Party party){
    for (Sector s : BATTLEFIELD.giveSectors()) {
        if (s.giveCommander().PARTY == party) {
            s.giveEntryField().showBorder();
        } else {
            s.giveEntryField().hideBorder();
        }
    }
}

```

Liegt dabei ein Sektor im Besitz des aktiven Spielers, so startet die vorgesehene Animation auf dem im Sektor enthaltenen Ablagefeld. Für Animationen stellt LibGDX eigene Aktionen

bereit. Für den gewünschten, pulsierenden Effekt wurde die “RepeatAction” gewählt, der man eine Ausblende- und Einblendefunktion übergeben muss:

```
repeatAction.setAction(Actions.sequence(Actions.fadeOut(duration));
Actions.fadeIn(duration)));
```

Bei jedem Aufruf dieser Aktion werden diese beiden Funktionen einer Art Aktionsschleife hinzugefügt. Je länger diese Schleife wuchs, desto geringer wurde der prozentuale Anteil der Animationsdauer (Duration), daher verschnellerte sich bei jedem Klick die Rahmenanimation des Ein- und Ausblendens. Aus diesem Grund musste diese Aktionsschleife vor jedem Aufruf zuerst geleert werden:

```
glowBorder.addAction().clear();
```

Diese Methode wird auch für das Beenden der Animation verwendet. Dieser Fall tritt immer dann ein wenn der Spieler aktuell keine BattleCard aktiviert hat, also wenn er entweder eine Karte abgelegt oder sie durch Klicken auf das Spielfeld unselektiert hat. Das Beenden der Animation lässt den Rahmen jedoch nicht verschwinden, sondern fügt ihn in seiner aktuellen Transparenz, die zwischen Eins (fadeIn) und Null (fadeOut) variiert, der Stage hinzu. Für das endgültige Beenden der visuellen Rückmeldung muss die Transparenz der Assets wieder auf Null gesetzt werden:

```
glowBorder.setColor(1,1,1,0);
```

4.2.16 Klickanimation der Buttons

Für die visuelle Bestätigung der Durchführung eines Klicks wurden die Spieldfeldbuttons um ein Asset erweitert. Der Aufruf des ImageButton-Konstruktors mit zwei Parametern, imageUp und imageDown, ermöglicht es die Buttongrafiken während eines Mausklicks, als auch nach dieser Aktion, zu setzen. Das neu hinzugefügte Asset beinhaltet die Buttongrafik in einer schwächeren Belichtung:

```
final Texture BUTTON_RED_DOWN = new Texture("core/assets/visuals/buttons/3_");
final ImageButton RED_BUTTON = new ImageButton(new TextureRegionDrawable(new Te
```

4.2.17 Deselektion der BattleCard

Selektiert der Player eine Handkarte und entscheidet sich daraufhin um, weil er lieber eine oder mehrere andere Aktionen durchführen möchte, wird der Buffer der letzten geklickten Karte geleert und ihre Aktivierung damit aufgehoben. Generell geschieht dies bei jedem Mausklick außerhalb einer BattleCard. Benötigt wird hier also eine Abfrage, ob sich auf geklickter Position ein Actor befindet, der wiederum von einer BattleCard gehalten wird. Mit der hit-Methode der Stage-Klasse können wir dies leicht überprüfen. Diese benötigt lediglich die Koordinaten des Mausklicks, um

einen dort positionierten Actor auszugeben. Dieser wird anschließend auf seine Zugehörigkeit geprüft, die nicht der BattleCard angehören darf:

```

if (Gdx.input.justTouched()) {
    Stage stage = giveStage();
    Vector2 vector = new Vector2(Gdx.input.getX(), Gdx.input.getY());
    stage.screenToStageCoordinates(vector);
    Actor clickedActor = stage.hit(vector.x, vector.y, true);
    if (clickedActor != null) {
        if (!(clickedActor instanceof BattleCard)) {
            resetLastClickedBattleCard();
        }
    }
}

```

4.2.18 Spielfeld

Um das Spielgeschehen überhaupt zu gestalten, bedarf es eines Spielfeldes, auf dem sich alle Spielzüge abspielen können. Dies darf man gerne mit einer Art „Schlachtfeld“ in einem kriegerischen Strategiespiel vergleichen, auf dem es einzunehmende Basen, die sogenannten „Quartiere“ gibt. Ziel einer Partie ist es, sämtliche Basen des Gegenspielers für sich zu gewinnen, indem alle verteidigenden Kreaturen besiegt und somit vom Feld genommen werden. Zum Spielfeld von EDWARDS BIOTOPE gehören Sektoren, Zonen und die darin geschehende Feldpositionierung. Die dafür benötigten Programmelemente wurden von Robert, Pamela, Gabriel, Cigdem, Manuela und Felix implementiert.

Sektoren Das Spielfeld von EDWARDS BIOTOPE ist in sechs Sektoren aufgeteilt, die jeweils ein Quartier und drei Kreaturen und Phänomene halten können. Die im Spiel gelegten Karten werden wiederum den verschiedenen Feldern zugeordnet, die im jeweiligen Sektor angelegt sind.

```

public Sector(Battlefield battlefield, Vector2 field1Vector, Vector2 field2Vector,
               Vector2 quarterFieldVector)
{
    super();
    BATTLEFIELD = battlefield;
    FIELDS.add(new Field(this, quarterFieldVector));
    FIELDS.add(new LeadField(this, field1Vector));
    FIELDS.add(new EntryField(this, field2Vector));
    FIELDS.add(new EndField(this, field3Vector));
}

```

Es wird außerdem ausgegeben, welche Sektoren vor bzw. nach dem aktuellen Sektor stehen.

```
public Sector givePreviousSector()
{
    return BATTLEFIELD.givePreviousSectorOf(this);
}

public Sector giveNextSector()
{
    return BATTLEFIELD.giveNextSectorOf(this);
}
```

Siehe im Git-Repository in den Commits:

- daf1fc139ee620220bbe36340e67b0e3bcbeada2
- e68492971171565a17d499ab5675c5032eb09542
- a936f5cfb03662dd957ca27a2ecf8b86e4da1c9a

Zonen Es erstrecken sich drei verschiedenfarbige Zonen über das Spielfeld: Rot, Grün und Blau. Jede Zone enthält jeweils zwei Sektoren. Die Zone selbst ist als eine Aufzählung angelegt und kann je nach momentan liegender Farbe aktiviert werden.

```
public enum Zone
{
    // -----
    // ENTRIES
    // -----
    RED,
    GREEN,
    BLUE;

    // -----
    // ATTRIBUTES
    // -----
    private boolean activated;

    // -----
    // CONSTRUCTORS
}
```

```

// =====

Zone()
{
    activated = false;
}

// =====
// ATTRIBUTES
// =====

public static Zone giveZoneByColorIndex(int index)
{
    for (Zone zone : values()) {
        int ordinal = zone.ordinal() * 2;
        if (ordinal == index || (ordinal + 1) == index) {
            return zone;
        }
    }
    return null;
}

public boolean isActivated()
{
    return activated;
}

public void activate()
{
    activated = true;
}

public void deactivate()
{
    activated = false;
}
}

```

Siehe im Git-Repository in den Commits:

- cfb3bf3736c1e90fc7144eb4a110ab580337df68
- 9b915960bfaf07586067aaa2ac92173d0700350d
- 16261f7ae75d97aedf791c6af081314f70f0ebdd
- c83d620c1d9bd22af5b58b22c683b0d596c66783

Feldpositionierung Für das Spielgeschehen spielt die Feldpositionierung eine entscheidende Rolle. Hiermit wird angegeben, auf welchem Feld sich eine Karte befindet, oder ob ein Feld leer ist. So hat ein Sektor ein Feld für das Quartier und drei Felder auf denen Kreaturen abgelegt werden können, wovon das erste das Eintrittsfeld darstellt. Über dieses Feld werden die Kreaturen in das Spielgeschehen geschickt. Sollte die grüne Zone aktiviert werden, kann die Kreatur dann auf eines der anderen Felder weitergeschoben werden, damit das Eintrittsfeld wieder frei wird für neu hinzukommende Karten.

```
@Override public Field giveCurrentFieldOfBattleCard(BattleCard battleCard)
{
    for (Field field : FIELDS) {
        if (field.giveCards().contains(battleCard)) {
            return field;
        }
    }
    return null;
}
```

Mit der folgenden Funktion wird von den Sektoren abgefragt, ob das jeweilige Feld eine Karte hält oder nicht. Basierend auf diesen Flags, kann das Battlefield die Informationen dann dem Kontext entsprechend verarbeiten.

```
public boolean hasBattleCard(BattleCard battleCardToHave)
{
    for (Field field : FIELDS) {
        if (field.giveCards().contains(battleCardToHave)) {
            return true;
        }
    }
    return false;
}
```

Des Weiteren wird ausgegeben, welches Feld mit welchen Karten belegt ist. Per get() findet hier eine eindeutige Zuweisung an der jeweiligen Stelle der zugrundeliegenden ArrayList ab. So können wir einen Trading-Card Spieltisch später mit fest zugewiesenen Slots für Karten verstehen und letztendlich grafisch übersichtlich präsentieren.

```
public Field giveQuarterField()
{
    return FIELDS.get(0);
}

public ArrayList<RingField> giveRingFields()
{
    ArrayList<RingField> ringFields = new ArrayList<RingField>();
    ringFields.add(giveLeadField());
    ringFields.add(giveEntryField());
    ringFields.add(giveEndField());
    return ringFields;
}

public LeadField giveLeadField()
{
    return (LeadField)FIELDS.get(1);
}

public EntryField giveEntryField()
{
    return (EntryField)FIELDS.get(2);
}

public EndField giveEndField()
{
    return (EndField)FIELDS.get(3);
}
```

In welche Richtung die Karten auf dem Feld positioniert werden, wird an der aktuellen Strömungsrichtung ausgemacht. Sie dient durch ihre Sortierung der Array-Liste später auch dazu, Angriffsrichtungen im Stellungsspiel um die Quartierkarten zu bestimmen. Man drückt im Interface am Anfang der Runde auf einen optionalen Schalter und dreht damit den Ablauf entweder im, oder gegen den Urzeigersinn. Das bestimmt dann, wer für eine Kreaturen-Karte das nächste

mögliche Angriffsziel sein kann.

```
private ArrayList<BattleCard> reverseCardOrder( ArrayList<BattleCard> list )
{
    Collections.reverse(list);
    return list;
}

public BattleCard giveQuarter()
{
    return FIELDS.get(0).giveCards().get(0);
}

public ArrayList<RingField> giveOrderedRingFields()
{
    ArrayList<RingField> orderedRingFields = giveRingFields();
    BATTLEFIELD.COMPASS.giveStream().orderRingFields(orderedRingFields);
    return orderedRingFields;
}
```

Siehe im Git-Repository in den Commits:

- daf1fc139ee620220bbe36340e67b0e3bcbeada2
- e68492971171565a17d499ab5675c5032eb09542
- a936f5cfb03662dd957ca27a2ecf8b86e4da1c9a

4.2.19 Musik und Soundeffekte

LibGDX bietet eine Musikschnittstelle, mit der man Musik und Soundeffekte leicht in das Spiel einbinden kann. Dafür gibt es die zwei Klassen “Music” und “Sound”. Bei einer Audiodatei, deren Länge ein paar Sekunden übertrifft, handelt es sich um eine Music-Klasse. Es ist zu bevorzugen, die Datei von der Festplatte zu streamen, anstatt sie vollständig in den Arbeitsspeicher zu laden. Die Sound-Klasse bedient Soundeffekte, wie kleine Audio-Samples, die in der Regel nicht länger als einige Sekunden dauern und bei bestimmten Spielereignissen wiedergegeben werden sollen, wie zum Beispiel beim Drücken eines Buttons, beim Kartenziehen oder beim Drehen des Kompasses. Soundeffekte können in verschiedenen Formaten gespeichert werden. LibGDX unterstützt MP3-, OGG- und WAV-Dateien. Die benötigten Audio-Dateien liegen dabei im Assets-Ordner. Um die Musik in das Spiel zu laden, muss eine Instanz der Music-Klasse generiert werden. Dies passiert in der BMEProjekt-Klasse, damit man die Musik und Soundeffekte in allen Klassen benutzen kann:

```
public static Music music = Gdx.audio.newMusic(Gdx.files.internal("core/assets/
```

Die Attribute der Music-Klasse erlauben die Anpassungen einiger Einstellungen, wie beispielsweise die Veränderung der Lautstärke und die Setzung der Audio-Datei auf Wiederholung:

```
music.setVolume(1.0f);  
music.setLooping(true);
```

Das Abspielen der Musikinstanz funktioniert wie folgt:

```
music.play();
```

Um Ressourcen freizugeben wird die Musikinstanz in der BMEProjekt-Klasse entsorgt, sobald sie nicht mehr benötigt wird:

```
music.dispose();
```

Die Soundeffekte werden ähnlich implementiert und wie folgt abgespielt:

```
BMEProject.streamSound.play(0.2f);
```

Dadurch wird der Soundeffekt einmal mit der Lautstärke von 20 % wiedergegeben. Die Methode .play(); wird beim Eintreten bestimmter Ereignissen gesetzt.

4.2.20 XML befüllen

Um der Klasse Cards ihre Werte zuzuweisen, wird ein XML benötigt mit allen relevanten Informationen. Jede Karte benötigt eine ID, cardName, cardType, cardEffect, cardDescription und den cardIllustrationFilePath. Diese Informationen wurden für 44 Karten hinterlegt. Siehe hierzu den Commit:

- 1f719baad08daf434cbb0c69186e068338359ade

4.2.21 Effekte

Da jede Karte mehrere Effekte bekommen sollte, wurde eine XML mit 10 Effekten erstellt. Der bisherige XML-Reader liest bisher die CARDS XML aus und musste deshalb erweitert werden, um die Effekte XML auszulesen. Jeder Effekt hat einen Integer Wert, mit der eigener ID. Zusätzlich einen String, der den Effekt in einem Satz beschreibt. Um die Effekte den einzelnen Karten zu zuzuordnen, wurden die Karten im XML um den Tag <Effects> erweitert. In diesem Tag können die IDs der Effekte als Integer Wert eingetragen werden. Die Effekte werden mit der Karte verbunden, indem die Klassen „Card“ und „DeckCard“ um den Typ Effekt erweitert werden. Damit die Effekte angezeigt werden, wurde eine Methode implementiert, die die Karten in größerer Auflösung anzeigt. Beim Drüberfahren mit der Maus werden die Effekte, Typ und Beschreibung als Text anzeigt. Die Effekte wurden jedoch gestrichen, da die Konzeption der einzelnen Effekte nicht abgeschlossen war. Außerdem konnte die Implementierung zeitlich nicht bewältigt werden.

Der Branch wurde deshalb im Git-Repository nicht dem Master Branch hinzugefügt. Siehe im Git-Repository den Branch: 70 76 KartenDetailsAnzeigen KartenAssets

4.2.22 Siegesbedingung

Um den Spieler zu signalisieren, wer der Gewinner ist, fehlte noch eine Methode mit einer Benachrichtigung über den Sieg. Nachdem ein Quartier den Besitzer gewechselt hat, wird über die sechs Quartiere auf dem Spielbrett iteriert. In der Iteration wird geprüft, welchem Spieler die Quartiere gehören. Besitzt ein Spieler alle sechs Quartiere, hat dieser gewonnen. Nach dem Erfüllen der Bedingung wird ein Bild angezeigt mit der Nachricht „du hast gewonnen“ oder „du hast verloren“. Zusätzlich werden Buttons eingefügt, um ein neues Spiel zu starten und um zurück in das Hauptmenü zu gelangen. Siehe im Git-Repository in den Commits:

- c390e51a6058673c1509ec47513f5e58ba560900
- 649f98e2344ba5429391f988e76dc903e2d58e08
- 7bcdccc49363fc3ac3529966aa51384e178c47c
- f634203a5dd291874886739790f981ed42f5fdc8

Die entsprechende Logik wurde von Pia Korndörfer zuerst in den Klassen Battleflied und BattleController umgesetzt. Da es sich hierbei fachlich um einen eigenen Screen handelt musste der Code hierfür in einen neuen Screen umgezogen werden. Hierfür implementierte Sebastian Beck einen entsprechenden EndOfGameScreen und einen passenden Controller und übernahm die grundsätzliche logik. Die Besonderheit bei diesem Screen ist jedoch im vergleich zu den anderen, dass dieser nicht initial mit den anderen Screens im Projekt gebaut werden kann, da er vom Spielergebnis abhängig ist. Jedoch werden gleichzeitig die einzelnen Screens und Aufrufe über die zentrale Klasse "BMEProject" gesteuert. Hierzu wurde die BMEProject-Klasse um die Methode "setEndOfGameScreen()" erweitert. Diese erwartet ein vorbereitetes Objekt der "EndOfGameScreen"-Klasse, welches bereits mit einem Boolealn "isWin" beladen ist um einen Gewinn oder Verlust anzuzeigen.

BattleController:

```
if (allyCounter == 6) {  
    endOfGameScreen = new EndOfGameScreen(BME_PROJECT, this, true);  
    BME_PROJECT.setEndOfGameScreen(endOfGameScreen);  
    return true;  
}
```

BMEProject:

```

public void setEndOfGameScreen(EndOfGameScreen endOfGameScreen)
{
    setScreen(endOfGameScreen);
}

```

4.2.23 Starter Deck mit drei Quartieren

Jeder Spieler darf in dem eigenen Deck Maximal drei Quartiere haben. Bisher wird das Kartendeck zufällig mit einem Random Number Generator generiert ohne dabei eine Obergrenze für Quartiere zu berücksichtigen. Die Quartiere in Cards.XML haben die ID 1 bis 5 und die restlichen Karten-typen von 6 bis 44. Deshalb wurde der Random Number Generator aufgeteilt. Zunächst werden zufällig drei Zahlen von 1 bis 5 gezogen, anschließend werden 37 Zahlen zwischen 6 bis 44 gezogen. Die Kombination der Karten IDS wird beim Spielstart als Deck für den Nutzer gespeichert. Siehe im Git-Repository in dem Commit:

- c390e51a6058673c1509ec47513f5e58ba560900

4.2.24 Auflösung auf des Title Screens und des BattleScreens auf HD anpassen

Bisher ist das Spielfeld auf eine Auflösung 800x450 Pixel konfiguriert. Diese Einstellung wurde auf Full HD Auflösung 1920x080 angepasst. Alle Regionen für die Karten waren absolut positioniert. Deshalb mussten alle Regionen neu positioniert und skaliert werden, um die volle Auflösung zu nutzen. Die Assets waren nicht auf die HD-Auflösung angepasst und werden entweder vergrößert oder verkleinert angezeigt, dadurch entsteht ein Qualitätsverlust. Da EDWARDS BIOTYPE Full HD unterstützen soll, wurden die Assets auf die tatsächliche Größe angepasst. Um die tatsächlichen Größen auszulesen, wurde ein Screenshot des aktuellen Spiels in Adobe Photoshop vermessen und die benötigten Größen aufgeschrieben. Anschließend wurden die Assets in der tatsächlichen Größe erneut gespeichert und in das Spiel eingebunden. Die Spielkarten werden in zwei Größen benötigt. Einmal als kleine Version für das Spielbrett und eine große Version für die Detail-Ansicht. Deshalb muss die Klasse "Cards" um zwei Strings erweitert werden. Zum einen den ILLUSTRATION_FILE_PATH dieser Path setzt sich "core/assets/visuals/cards/large/" und der Variable illustrationFilePath zusammen. Zusätzlich der ILLUSTRATION_FILE_PATH_SMALL, dieser setzt sich aus "core/assets/visuals/cards/small/" und der Variable illustrationFilePath zusammen. Die Variable illustrationFilePath weist der XML Reader aus dem Card XML zu. Die Klasse "Battle Card" verwendet als Texturen für die Spielkarten den Pfad den ILLUSTRATION_FILE_PATH_SMALL und die DetailView ILLUSTRATION_FILE_PATH. Die Anzeige war nach den Anpassungen noch nicht zufriedenstellend, da LibGDX Texturen ohne Kantenglättung anzeigt. Siehe hierzu die Commits:

- 8ccd2046db2a7def1d17a7988afe598631f64f7b

- a0bab3abbb009b004659399adbbc854ee6932e8e
- 1dea80495d1dc5ba55e9909326146ac56540672d
- 90859da52af88b69d243b6d072f287102e0ca21e
- dde09c88d28081d813a9c581a4425b7425eb8a98
- 14ca0dd54308280139d0dac3d7e6e86d334a803f

4.2.25 Anti Aliasing hinzufügen

LibGDX bietet die Funktion `.setFilter` für Texturen an. Hierfür muss der Filter von „Nearest“ auf „Linear“ angepasst werden, um die Kanten der Assets zu Glätten. Deshalb wurde der Code nach allen Texturen durchsucht und für jede Textur die Funktion

`TEXTUR.setFilter(Texture.TextureFilter.Linear, Texture.TextureFilter.Linear);`
hinzugefügt. Siehe hierzu den Commit:

- 86617498da70acabe885862ce9db573342b08d6e

4.2.26 Camera Handle für Variable Bildschirmgrößen

LibGDX hat schon eine vorgefertigte Camera-Klasse, die man benutzen kann. Es gibt 2 Subklassen und zwar die OrthographicCamera und PerspectiveCamera. Bei der Orthographic Camera gibt es keine Perspektive, sondern alle Längen bleiben gleich egal von welchem Winkel man sie betrachtet. Für unser Spiel ist eine Orthographic Camera am sinnvollsten, weil wir keine Tiefen mit 3D-Meshes im Spielfeld haben. Die Implementierung der Kamera übernimmt die Renderer-Klasse

```
private final SpriteBatch SPRITE_BATCH;
private Viewport viewport;
private Camera camera;
private Stage stage;
Renderer(SpriteBatch spriteBatch)
{
    SPRITE_BATCH = spriteBatch;
    camera = new OrthographicCamera();
    viewport = new FitViewport(1920f, 1080f, camera);
    camera.position.set(1920 / 2f, 1080 / 2f, 0f);
}
```

Als Viewport wird ein FitViewPort gewählt mit Full-HD Auflösung. Es gibt in LibGDX verschiedene Viewports z.B. StretchViewport, FillViewport, ScreenViewport usw. , der FitViewport

macht keine Verzerrung, d.h. Egal in welche Größe ich das Fenster skaliere, wird das Spiel nie verzerrt. Schwarze Ränder füllen die Ränder auf, falls das Format des Fensters nicht mit dem Format des Spiels zusammenpasst. Die Position der Kamera ist die Mitte des Bildschirms, also 1920/2 für X und 1080/2 für Y. In Edwards Biotope braucht es nur eine statische Kamera die auf einer Position bleibt. Im Gegensatz zu Zelda, wo sich die Kamera mit dem Spieler bewegt. Die Render-Klasse hat ihre eigene Render-Methode. Die Methode bekommt bei jedem Render-Durchgang die stageToRender zugewiesen, das ist die Stage BattleScreen oder Titlescreen zum Beispiel, und der Viewport wird gesetzt. Das passiert 60 mal in der Sekunde.

```
void render(Stage stageToRender)
{
    stage = stageToRender;
    stage.setViewport(viewport);
    Gdx.gl.glClearColor(0f, 0f, 0f, 0f);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    stage.draw();
}
```

4.2.27 Server Kommunikation Testing mit Router und Lokal

Die Implementierung, dass 2 Spieler am gleichen PC mit 2 Fenstern spielen können wurde bereits in einem frühen Stadium vom Spiel umgesetzt und kann als über die Projektzeit weiter führende Aufgabe in dem jetzigen Stand des Spiels, bei dem beide Spieler am gleichen Computer und im gleichen Fenster spielen, integriert werden. Dazu benötigt es nur der richtigen Events, die geschrieben werden müssen, um die Server-Kommunikation zu ermöglichen. Das Testing mit einem Handelsüblichen Router ist ähnlich dem Testing an einem Computer, man muss nur einige Parameter in der Software umstellen. Sowie beim Multiplayer auf einem System verwenden wir IO-Sockets mit einem Node-JS-Server.

Die nötige Installationen sind folgend beschrieben:

- NodeJS.org öffnen
- z.B v10.14.2 LTS Downloaden
- NodeJS installieren
- Auf Mac: Terminal öffnen und „node —version“ eingeben. Wenn v10.14.2 als Antwort kommt war die Installation erfolgreich, falls nicht ist es höchstwahrscheinlich, dass der Computer den Pfad zu NodeJS nicht kennt

- Server starten: Im Terminal „node index.js“ eingeben. Hierfür muss man zum Verzeichnis wechseln wo index.js abgespeichert ist. Im Projektverzeichnis liegt dieses unter: bme-project/game/bmeProject/MultiplayerDemo/Server Note: index.js ist eine selbsterstellte Javaskript-Datei in die den Server definiert(dessen Events usw.)

Um Lauffähigkeit herzustellen muss ggf. lokal noch Socket.io installiert werden. Eine ausführliche Anleitung hierfür befindet sich im Anhang.

Da es zum Testing keinen extra Server gab, wird ein PC, der das Spiel ausführt, zum Server gemacht. Das nennt sich Hosting und wird in Online-Spielen oft verwendet. Ein PC hat in seiner connectSocket die Localhost-Adresse eingetragen von sich selbst. Der andere PC hat die IP dieses PCs im Lokalen Netzwerk in seiner connectSocket()-Methode eingetragen. Diese IP ist beispielsweise 192.168.0.7. Beide Computer werden vorher mit dem Router verbunden per LAN-Kabel für optimale Geschwindigkeit.

4.2.28 Zerstören von Karten

Die Karten, die auf dem Spielfeld vom Gegner besiegt werden, müssen auf dem Friedhof landen, wenn es sich um Kreaturen oder Phänomene handelt. Quartiere werden nicht auf dem Friedhof abgelegt, sondern eingenommen vom Gegner. Die Kreaturen-Klasse und die Phänomenen-Klasse haben die gleiche getDestroyed()-Methode implementiert.

```
@Override public void getDestroyed()
{
    BMEProject.destroySound1.play(0.5f);
    Field graveyard = PLAYER.giveGraveyard();
    graveyard.addBattleCard(this);
}
```

In dieser Methode wird dem Graveyard die BattleCard hinzugefügt mit dem Aufruf von addBattleCard(this). Außerdem wird für die Spielästhetik ein Sound abgespielt. Da jeder Spieler seinen eigenen Friedhof hat, muss dieser erst abgefragt werden mit PLAYER.giveGraveyard(). Im Spiel ist der Graveyard ein Field, das einem Spieler zugewiesen ist. Die giveGraveyard()-Methode von Player gibt das Field aus einer Liste zurück, das für den Friedhof dieses Spielers steht. Alle Fields vom Spielfeld sind in einer Liste vorhanden und jedes Feld kann addressiert werden.

```
public Field giveGraveyard()
{
    return FIELDS.get(2);
}
```

Zu der Methode addBattleCard() ist die Beschreibung etwas umfangreicher. Es hört sich einfach an was die Methode macht, dennoch muss man einige Abfragen machen. Man hat ein

currentField und einen FIELD_USER. Man muss durch die Klassenhierarchie durchgehen. Man fragt den FIELD_USER auf welchem Feld die mitgebene BattleCard sich befindet. Es kann auch sein, dass die BattleCard sich noch auf keinem Feld befindet. In den meisten Fällen ist das jedoch nicht so. Wenn das currentField also nicht null ist, wird die BattleCard von ihrem jetzigen Field entfernt mit removeCard(battleCardToAdd). Neben dieser Prüfung ist es auch wichtig nach der Existenz der Karte zu schauen. Wenn die Karte existiert wird sie der Liste CARDS hinzugefügt.

```
public void addBattleCard(BattleCard battleCardToAdd)
{
    Field currentField = FIELD_USER.giveBattleController().giveCurrentFieldOfBatt
    if (currentField != null) {
        currentField.removeCard(battleCardToAdd);
    }
    if (battleCardToAdd != null) {
        CARDS.add(battleCardToAdd);
    }
    update();
}
```

Für die getDestroyed()-Methode des Quartiers müssen andere Abfragen und Befehle ausgeführt werden, deswegen ist diese Methode anders als die Kreatur- und Phänomen-Methode. Am Rande sei erwähnt, dass der Sound auch ein anderer ist und in deutlich niedrigerer Lautstärke. Quatieri haben einen Kommandeur(commander). Der commander ist der Spieler, der das Quartier gerade kontrolliert. In der Methode wird dem commander der Gegenspieler des derzeitigen commanders vom Quartier zugewiesen. In dieser Methode wird auch ein currentField angelegt. Dies dient dazu das currentField zu updaten.

```
@Override public void getDestroyed()
{
    BMEProject.destroySound2.play(0.1f);
    commander = PLAYER.BATTLE_CONTROLLER.giveOppositePlayerOf(commander);
    Field currentField = giveCurrentField();
    if (currentField != null) {
        currentField.update();
    }
    PLAYER.BATTLE_CONTROLLER.checkForWin();
}
```

4.2.29 Handhabe aktive Spieler

Der BattleController ist verantwortlich für den Spielablauf während eines Turniers von 2 Spielern. Der BattleController muss wissen welcher Spieler am Zug ist, damit er die Methoden beim richtigen Spieler ausführt. Zum Beispiel wäre es schlecht, wenn der BattleController beim Beginn eines Zuges von einem Spieler für den Gegenspieler die Methode drawTopCard() ausruft anstatt des derzeitigen aktiven Spieler. Bei diesem Fall würde der BattleController dem gerade nicht am Zug befindlichen Spieler eine neue Karte auf die Hand geben. Dies würde einen flüssigen Spielablauf so gut wie unmöglich machen, deswegen braucht man eine Instanz für den aktiven Spieler.

```
private Player activePlayer;
```

Bei unserem Spiel gibt es genau 2 Spieler. Beide Spieler haben eine Nummer und zwar Spieler 1 und Spieler 2. Am Anfang von einem Spiel soll Spieler 1 der aktive Spieler sein.

```
PLAYER_1 = new Player(this, Party.ALLY);  
PLAYER_2 = new Player(this, Party.ENEMY);  
activePlayer = PLAYER_1;  
activePlayer.beginTurn();
```

Wer Spieler 1 und wer Spieler 2 ist, ist durch die Aufteilung des Spielfeldes festgelegt. Der Spieler der seine Hand auf der unteren Hälfte des Bildschirms hat, ist der Spieler 1. Spieler 2 ist dementsprechend der Spieler auf der oberen Hälfte.



Figure 3: Screenshot: Spielfeld mit Spielerbereichen

Jetzt funktioniert es, dass die Methoden vom activePlayer, der Spieler 1 ist, aufgerufen wird. Wenn Spieler 1 seinen Zug beendet ist er dennoch activePlayer, aber dies ist ja nicht was wir wollen. Wir brauchen eine neue Methode die den aktiven Spieler von Spieler 1 zu Spieler 2 ändert und auch in anderer Richtung.

Diese Methode muss den activePlayer kennen und diesen neu zuweisen. ActivePlayer soll also Spieler 2 zugewiesen werden.

Den activePlayer kennt die Methode, weil activePlayer eine Instanz-Variable ist. Dennoch besteht das Problem, dass man nicht einfach Spieler 2 dem activePlayer zuweisen kann. So funktioniert die Methode nur in eine Richtung und nicht rückwärts. Es wird eine weitere Methode benötigt, die den gerade nicht aktiven Spieler ausgibt, also den Gegenspieler. GiveOppositePlayerOf(Player player) wäre so eine Methode.

```
public Player giveOppositePlayerOf(Player player)
{
    if (player == PLAYER_1) {
        return PLAYER_2;
    } else {
        return PLAYER_1;
    }
}
```

Die Abfrage ist recht trivial. Man gibt der Methode den activePlayer mit. Wenn der activePlayer Player 1 ist, wird der Player 2 zurückgegeben. Beim umgedrehten Fall wird Spieler 1 zurückgegeben. Nun können wir dahin zurück den activePlayer neu zuzuweisen. Die Methode changeActivePlayer() macht genau dies in Kombination mit getOppositePlayer(Player player).

```
public void changeActivePlayer()
{
    resetLastClickedBattleCard();
    Zone.RED.deactivate();
    Zone.GREEN.deactivate();
    Zone.BLUE.deactivate();
    setTurnUnstarted();
    BUTTON_VIEW.fadeInZoneButtons();
    Player nextPlayer = giveOppositePlayerOf(activePlayer);
    nextPlayer.beginTurn();
    activePlayer = nextPlayer;
    updateAllFields();
    showActivePlayerMessage();
}
```

In der Methode wird eine neue Variable nextPlayer angelegt. Dieser wird der return-Wert von giveOppositePlayerOf(activePlayer) zugewiesen. Dem activePlayer wird dann nurnoch der nextPlayer-Wert zugewiesen und schon ist das Problem gelöst. Neben dem Opposite Player kann man natürlich auch den activePlayer, falls er gebraucht wird, ausgeben lassen.

```
public Player giveActivePlayer()
{
    return activePlayer;
}
```

Wenn ein Spieler am Zug ist, müssen auch seine eigenen Sektoren aktiviert werden und die vom Gegener deaktiviert, damit er nur auf seine eigenen Sektoren Karten legen kann. Dafür benötigt man auch den activePlayer und man geht bei ALLY von Spieler 1 aus und bei ENEMY von Spieler 2.

```
public void activateSectorOfPlayer(Party party){
    for (Sector s : BATTLEFIELD.giveSectors()) {
        if (s.giveCommander().PARTY == party) {
            s.giveEntryField().showBorder();
        } else {
            s.giveEntryField().hideBorder();
        }
    }
}
```

4.2.30 Felder

Unser Spielfeld ist in 6 Sektoren aufgeteilt. Jeder Sektor hat 4 Fields auf denen jeweils eine Karte abgelegt werden kann. Das Field muss wissen ob eine Karte auf ihr liegt oder ob sie frei ist. Das Field an sich hat X- und Y-Koordinaten sowie eine Breite und eine Höhe. Eine Karte wird in einem sogenannten Pile innerhalb des Fields abgelegt. Ein Field kann prinzipiell beliebig viele Piles haben, aber in unserem Spiel hat jedes Field ein Pile. Ein Pile hat einen eigen X- und Y-Offset innerhalb des Fields. Die Karten können einen Offset zum Ursprung des Piles haben in dem sie ablegt werden. Zur Erläuterung ein Pile ist keine eigene Klasse, sondern die Parameter PILE_X und PILE_Y legen einen Orientierungspunkt im Field für die Karten fest.

Jedes Field hat auch ein Kartenlimit, bei fast allen ist dieses 1. Nur beim Deck gibt es ein Limit von 40 Karten pro Field. Im Field ist eine Update-Methode implementiert, die die Karten neu ordnet wenn eine Karte aus einem Field genommen wird. Die Methode macht logisch folgendes, die Methode iteriert durch die Karten des Piles durch, sie gibt jeder BattleCard ihre X- und Y-Werte neu bei jedem Update. So werden die Karten immer neu geordnet, wenn eine Karte zum Field

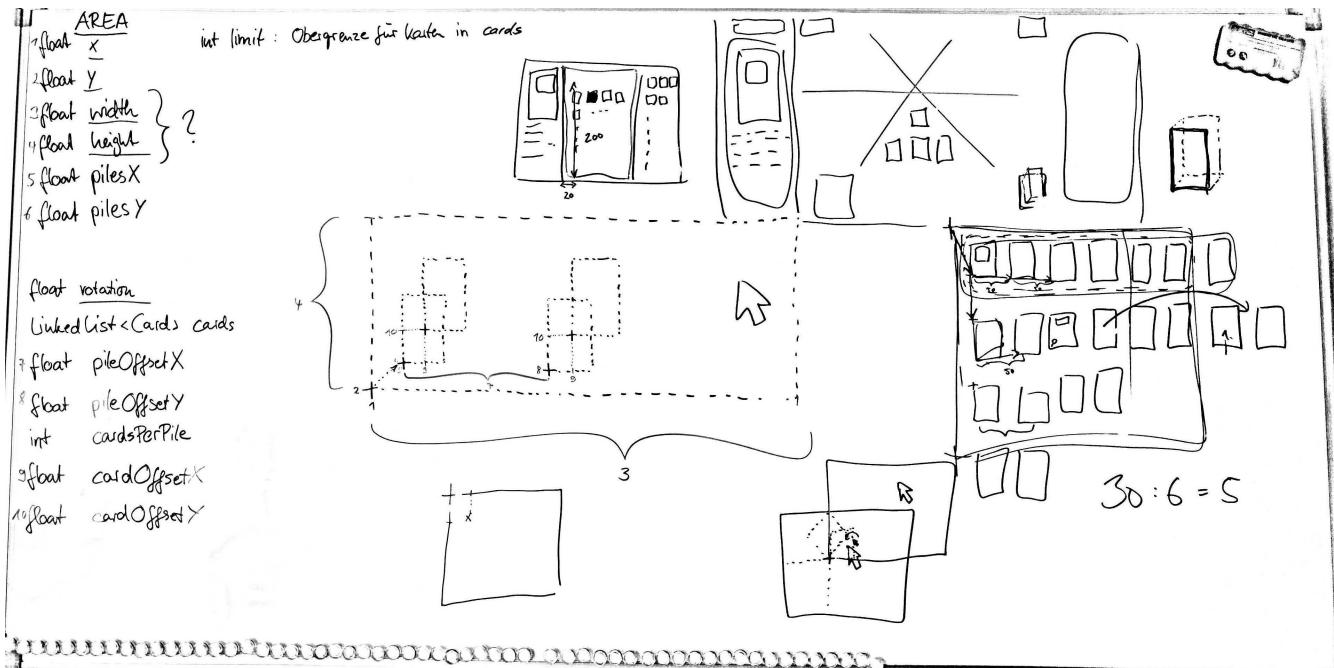


Figure 4: Zeichnung: Fields

hinzugefügt wird oder entfernt wird. Die Karten rücken also auf, damit keine Lücke entsteht. Es gibt noch einige weitere Methoden in der Field-Klasse, diese werden folgend beschrieben.

UpdateReadabilityOfBattleCard Diese Methode ruft die Methode getUncovered() von der BattleCard auf. Auf einem physikalischen Kartenspiel wäre das das Umdrehen der Karte von Rückseite auf Vorderseite. Wenn eine Karte z.B. auf dem Stapel liegt, ist diese umgedreht und nur die Rückseite der Karte ist sichtbar, welches die gleiche Textur bei allen Karten ist. Die getUncovered-Methode ändert die Textur der BattleCard von der Rückseite-Textur zur Vorderseite Textur und ist jetzt sichtbar für den Spieler.

```
protected void updateReadabilityOfBattleCard(BattleCard battleCard)
{
    battleCard.getUncovered();
}
```

AddBattleCard Wie der Name schon sagt, wird hier eine Karte dem Field hinzugefügt. Unter der Haube passiert dennoch deutlich mehr. Es wird abgefragt, ob die Karte derzeit auf einem anderen Field liegt. Falls ja wird die Karte von diesem Field erst einmal entfernt. Außerdem wird die Existenz der Karte mit einer if-not-null-Kontrolle überprüft. Die Update-Methode wird am Ende aufgerufen.

```
public void addBattleCard(BattleCard battleCardToAdd)
{
    Field currentField = FIELD_USER.giveBattleController().giveCurrentFieldOfBatt
    if (currentField != null) {
```

```

        currentField.removeCard(battleCardToAdd);
    }
    if (battleCardToAdd != null) {
        CARDs.add(battleCardToAdd);
    }
    update();
}

```

AddBattleCards Man kann nicht nur einzelne Karten hinzufügen, sondern auch ArrayListen von Karten. Hier wird auch eine Abfrage gemacht, falls eine Karte schon auf einem anderen Feld ist, diese erst von diesem Feld zu entfernen. Für jede BattleCard von der ArrayList an Karten wird diese Abfrage gemacht und dann dem Field hinzugefügt.

```

public void addBattleCards(ArrayList<BattleCard> battleCardsToAdd)
{
    for (BattleCard battleCard : battleCardsToAdd) {
        Field currentField = FIELD_USER.giveBattleController().giveCurrentFieldOf();
        if (currentField != null) {
            currentField.removeCard(battleCard);
        }
        CARDs.add(battleCard);
    }
    update();
}

```

removeCard Von der ArrayList CARDs wird die BattleCard entfernt die als Parameter übergeben wird.

```

private void removeCard(BattleCard cardToRemove)
{
    CARDs.remove(cardToRemove);
    update();
}

```

PullCard Eine Karte wird anhand von ihrem index aus der ArrayList entfernt und dann zurückgegeben. Es gibt noch die Methode pullTopCard(), bei der der Index der Karte, die entfernt wird, einfach die ArrayList-Größe -1 ist. -1 weil Indices von 0 gezählt werden.

```

public BattleCard pullCard(int index)
{
    if (CARDs.size() > 0 && CARDs.size() >= index && index >= 0) {
        BattleCard card = CARDs.get(index);

```

```

        removeCard( card );
        return card;
    } else {
        return null;
    }
}
public BattleCard pullTopCard()
{
    return pullCard( CARDS.size() - 1 );
}

```

Es gibt noch 2 weitere Methoden und zwar shuffle(), die die ArrayList CARDS durchmischt und getPileSize(), die die Größe der CARDS ArrayList zurückgibt.

```

public void shuffle()
{
    Collections.shuffle( CARDS );
}
public int getPileSize()
{
    return CARDS.size();
}

```

Am Anfang von jedem Spiel soll jedem Spieler Karten gegeben werden dies übernimmt giveCards().

```

public ArrayList<BattleCard> giveCards()
{
    return new ArrayList<BattleCard>( CARDS );
}

```

4.2.31 Laden und Speichern von Decks

Ursprünglich war angedacht, dass ein Deckeditor umgesetzt werden soll. Dieser sollte dem Spieler ermöglichen, ein Karten-Deck nach seinen Wünschen zu gestalten, zu speichern und zu laden. Als die Logik für das Feature "Effekte" jedoch vorerst nicht in das Gesamtprojekt eingefügt wurde, gab es nichts, anhand dessen sich die verschiedenen Kreaturen, Quartiere und Phenomene differenzierten. Daher machte auch darauf aufbauend eine Deck-Building mechanik erstmal keinen Sinn mehr. Trotz dessen wurde von Sebastian Beck eine einfache User-Klasse und ein SaveGameHandler implementiert. Die User-Klasse hielt die einzelnen Decks, während der SaveGameHandler als Utility-Klasse das Laden und Speichern zumindest backendseitig in ein Txt-File durchführte.

Zentral hierbei war entsprechend die Klasse "SaveGameHandler", welche sich auch im aktuellen Develop-Stand findet, jedoch nicht verwendet wird. Sie besteht aus einer File-Reader / File-Writer Logik, die in den Funktionen "saveGame" und "loadGame" umgesetzt sind.

```
public void saveGame( User user )
{
    File file = new File( "savegame.txt" );
    String cardIds = getCardIds( user );

    try {
        file.createNewFile();
        FileWriter writer = new FileWriter( file );
        writer.write( user.getName() + ";" );
        writer.write( cardIds );

        writer.flush();
        writer.close();
    }
    catch ( Exception e ) {
        System.out.println( e.getMessage() );
    }
}
```

Die Funktion erwartet ein User-Objekt um die entsprechend erstellten Decks hieraus auszulesen (über "getCardIds" werden die einzelnen Ids ermittelt) und diese samt dem User-Name in "savegame.txt" zu speichern. Die Funktion "loadGame" baut dies wieder zurück in ein Deck-Objekt. Ein letzter Schritt, dieses Deck dem User wieder zuzuführen ist noch nicht ausimplementiert, jedoch nach aktuellem Entwicklungsstand auch nicht in Nutzung.

4.3 Konzeptionelle und Organisatorische Aufgaben

In diesem Abschnitt finden sich Aufgaben konzeptioneller sowie organisatorischer Natur, die sich mit dem Software-Entwicklungsprozess beschäftigt haben.

4.3.1 Pflege eines schemenhaften Klassendiagramms

Um die Struktur unseres Programmes grob zu visualisieren und so eine bessere Orientierung im Code zu ermöglichen sowie einen einheitlichen Wissensstand unter allen Beteiligten zu etablieren, erarbeitete und pflegte Felix Baumgarten ein Klassendiagramm.

Dieses half unter anderem, die Aufgabenverteilung mittels Git-Banches und die Kapselungsorganisation zu vereinfachen. Während der Arbeit am Projekt wurde das Diagramm an einem Whiteboard abgebildet und ergänzt, da Änderungen schnell umzusetzen waren und wir uns Einarbeitungszeit in komplexere digitale UML-Diagramme sparen wollten. Eine Skizze des deutlich detaillierteren, da unter anderem mit Kardinalitäten ausgestatteten Klassendiagramms:

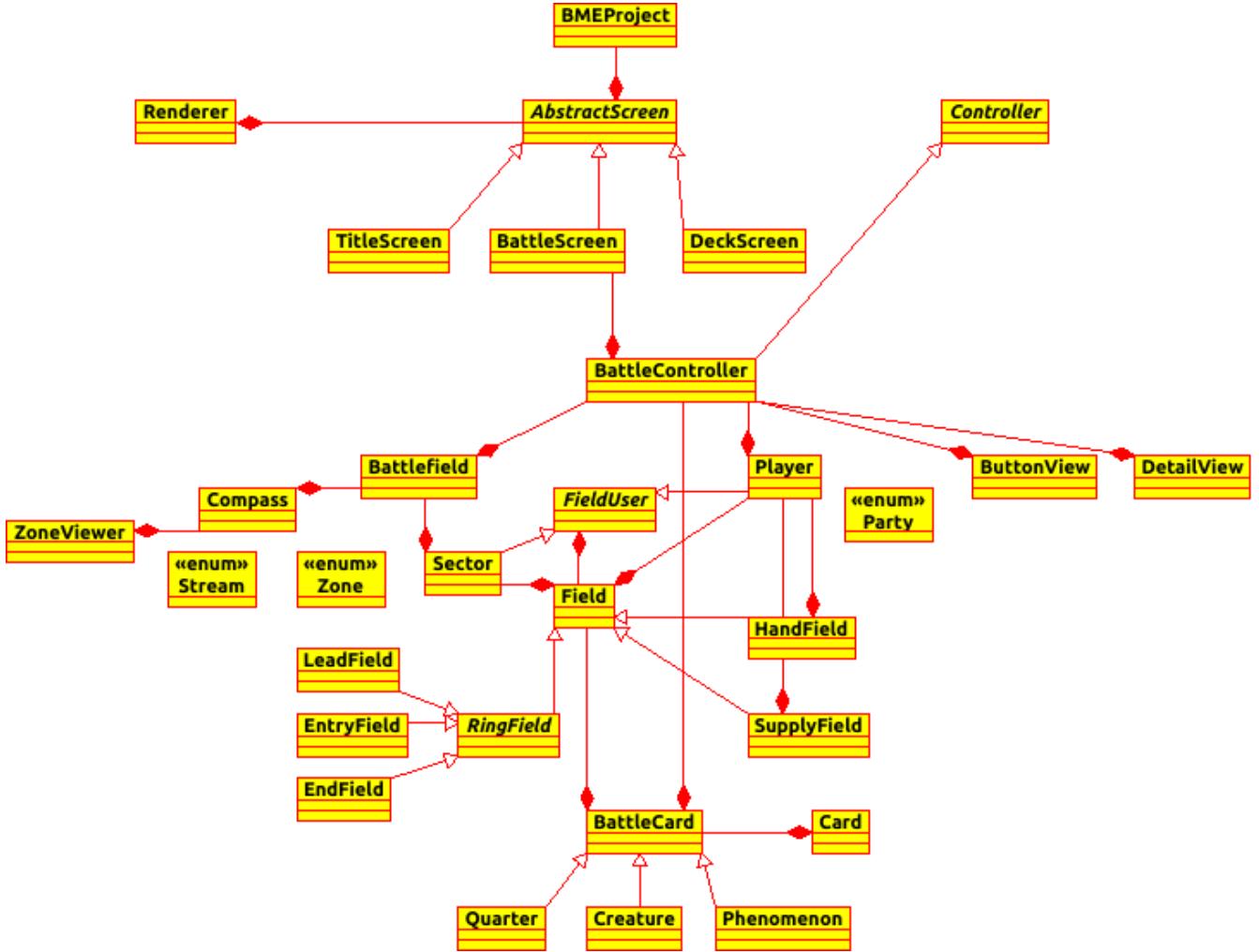


Figure 5: Klassendiagramm Edwards Biotope

4.3.2 Merging und Abnahme

Im letzten Abschnitt dieses Semesters wurde der Abnahme Prozess innerhalb des Projektteams noch einmal überarbeitet. Ziel war es dieses Semester alle Teammitglieder stärker in die konkrete Implementation von Anforderungen einzubeziehen. Unser ursprüngliches Herangehen war es, dass pro Branch bis zu drei Personen ein Review durchführen und dem jeweiligen Entwickler Feedback und ggf. Korrekturen mitteilen, die an dem Branch vorzunehmen sind. Dies erwies sich fortschreitend als ein Flaschenhals in dem Vorgehen. Der Zeitaufwand der nötig war um mehrere Personen in ein solches Review für jeden einzelnen Branch zu involvieren sorgte für fortschreitend

starke Verzögerungen was die Aktualisierung unseres Develop-Standes anging. Das wiederum sorgte entsprechend dafür, dass gewisse Anforderungen, die auf anderen Anforderungen basierten, nicht umgesetzt werden konnten, bis eben diese Aktualisierung stattfand. Wir entschieden uns entsprechend dafür, dass Sebastian Beck und Philadelphia Gauss für die Abnahme und das Merging der Branches zuständig waren und das als eine ihrer Hauptaufgaben behandeln. Damit strukturierte sich der Prozess nun neu, wie folgt:

- Erstellung eines Branches ab Develop (durch Entwickler oder einer der beiden Verantwortlichen)
- Implementation des jeweiligen Branches anhand der Anforderung die entweder im Global Board, Teammeeting oder beim Coding-Treffen definiert wurde
- Erstellen eines Pull-Requests mit Reviewer Sebastian Beck oder Philadelphia Gauss

Ab hier begann der Review-Prozess, der sich wie folgt gliederte:

- Checktout des jeweiligen Branches
- Erstes Build lokal und erster manueller Test ob Funktionalität umgestzt wurde
- einfaches Code-Review
- bei größeren Fehlern oder nötigen Anpassungen: Rückmeldung an den Entwickler, sonst kleinere Korrekturen durch Reviewer
- Prüfung ob Merge-Konflikte bestehen und diese ggf. beheben
- Erneute Prüfung der Lauffähigkeit
- Abschließender Merge nach Develop

Gerade nach der Korrektur von Merge-Konflikten fiel, vor allem bei komplexen Änderungen, immer wieder auf, dass der mitgelieferte Diff-Editor von Gitkraken sich nicht immer intuitiv verhielt. Das führte dazu, dass z. T. Methoden automatisiert oder unabsichtlich ineinander verschoben wurden oder nach dem Merge fehlten. Daraufhin führten wir vor allem bei umfangreichen Änderungen mit ein, dass der Stand nach Konfliktbehebung mittels dem Tool "Meld" erneut mit dem zu mergenden Stand abgeglichen wurde. Meld ermöglicht es relativ einfach komplett Verzeichnisse miteinander zu vergleichen und auf Unterschiede zu untersuchen. Damit konnten wir fehlenden Code oder fehlende Änderungen weitestgehend auszuschließen. Hierfür musste das Repository lediglich ein zweites mal in ein weiteres Verzeichnis ausgecheckt werden um die beiden Branches über Meld miteinander zu vergleichen. Im diesem Zuge wurde kurz auch das Tool "xxdiff" betrachtet, Meld wurde jedoch abschließend aufgrund seines einfachen UI bevorzugt.

Das beseitigen der Konflikte erfolgte in der Regel über einen Merge des aktuellen Develop-Stands in den jeweiligen Feature-Branch. Anhand dessen wurden die nötigen Korrekturen vorgenommen um die Lauffähigkeit wieder herzustellen. Der neue Stand des Features-Branches konnte nun zurück nach Develop gemerged werden.

4.4 Softwarequalität

Hier finden sich die Neuerungen zum Thema Infrastruktur

4.4.1 Qualitätssicherung der Oberfläche

Am Ende des Entwicklungsprozesses wurde die Oberfläche noch nach Problemen in der Darstellung der Assets oder nach Rechtschreibfehler geprüft. Gegebenenfalls wurden die Rechtschreibfehler korrigiert, Positionierungsfehler angepasst, Assets neu erstellt und eingebunden oder vergessene Kantenglättungen hinzugefügt.

- 934a8aed1a360ca2e1d56001e67cafcd8abcaa5
- cd8e1d9aea24ffe4f4cf9d8d9330813135ffd33
- 59e936934a9056a776462f07bc45005943dd2121
- 749b75fc52b7a87f67cf4e1256359747d0f37ed3

4.4.2 Unit-Tests

Im letzten Semester entschied sich das Projektteam dafür, Unit-Tests als einen Baustein zur Absicherung der Software-Qualität einzusetzen. Hierzu wurden relativ einfache Unit-Tests mit JUnit implementiert. Dies erfolgte leider verhältnismäßig spät, als die Code-Basis bereits komplexer war. Hierbei wurde jedoch festgestellt, dass sich einfache Model-Klassen wie "Deck" sehr einfach und isoliert testen lassen, jedoch komplexere Klassen wie der "BattleController" nur mit erheblichen Aufwand. Das mocken dieses Controllers, der die zentralen Spielmechaniken steuert, stellte sich als größeres Problem heraus. Verschiedene Versuche diesen separiert vom laufenden Projekt zu instanzieren und zu testen schlugen fehl, da die entsprechenden LibGDX-Dependencies ohne laufendes Projekt fehlten. Bei einer ersten Recherche, stießen wir auf die Option die entsprechenden Komponenten mittels dem Framework "Mockito" zu simulieren - das Team entschied sich jedoch aufgrund des hierfür nötigen Aufwands dafür, die Software-Qualität über den zentralen Abnahme-Prozess der Branches sowie die Prüfung von Metriken und manuellen Tests herzustellen. Hier exemplarisch ein Test für das Gameobject "Deck":

```
@Test  
void getSize() {
```

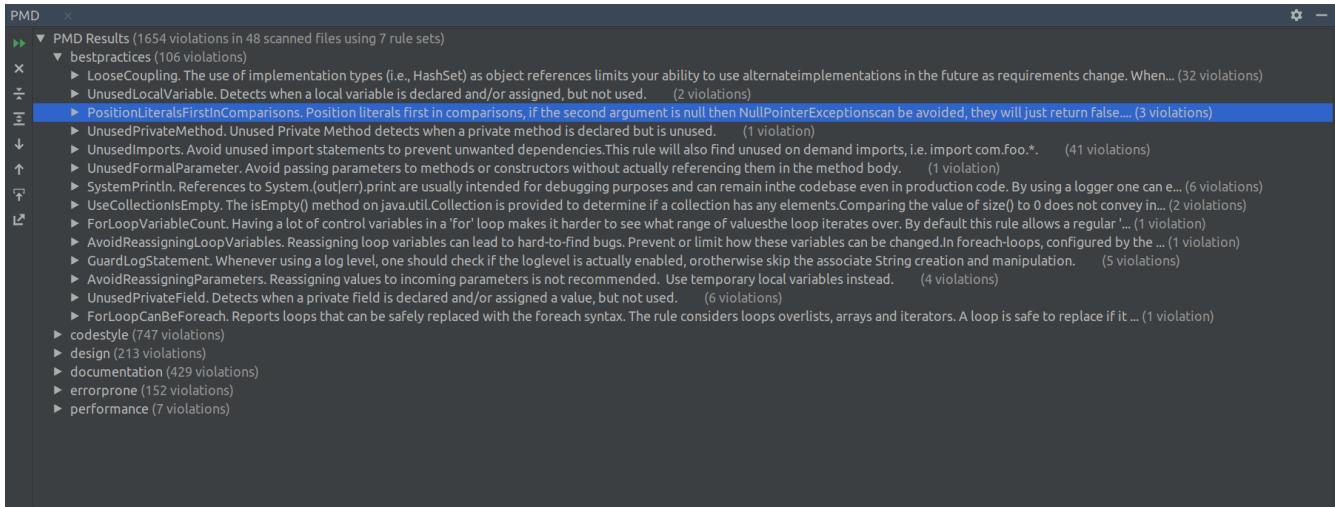


Figure 6: Screenshot: PMD Übersicht

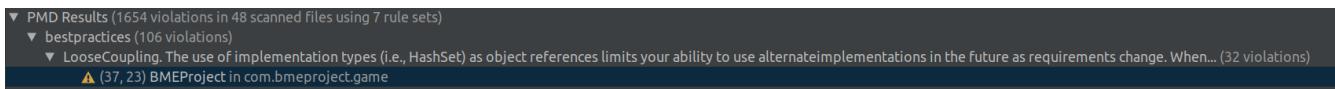


Figure 7: Screenshot: PMD Beispiel

```
    assertTrue(deck.getSize() > 0);
}
```

Vorab wird ein Deck deklariert und in diesem Test die getSize()-Methode geprüft, ob diese valide Ergebnisse zurück gibt. Abschließend lässt sich sagen, dass Unit-Tests ein elementarer Baustein unseres Software-Qualitäts-Konzepts waren und diese kaum umgesetzt wurden.

4.4.3 Metriken

Zur Analyse wurden verschiedene Tools herangezogen. Zur Auswahl standen: "PMD" und "Metrics Reloaded". Für beide liegt ein IntelliJ-Plugin vor, was die Einbindung in unser Projekt deutlich erleichterte. Beide Tools hatten entsprechend ihre Vor- und Nachteile, die hier kurz erörtert werden sollen. **PMD** PMD bietet diverse Optionen zur konkreten Code-Analyse. Dabei lässt sich nach verschiedenen "Verstößen" z. B. gegen Best Practices suchen und diese ggf. korrigieren. PMD liefert die Ergebnisse dabei in einer Art Baumstruktur und referenziert betroffene Stellen direkt im Code, sodass Änderungen einfach vorgenommen werden können. PMD liefert damit bei entsprechendem Fachwissen solide Ergebnisse über Schwachstellen unserer Software. Für einen einfachen Einsatz um einen Überblick über Problemstellen zu erhalten ist PMD jedoch potentiell zu komplex. Hier ein Auszug der umfangreichen Findings mit PMD: Beispielsweise lässt sich hier sehr einfach feststellen, dass es eine konkrete Implementation gibt die ggf. gegen Loose Coupling verstößt, was spätere Änderungen ggf. erschwert da ein höherer Grad an Abhängigkeit implementiert wurde als vielleicht notwendig. In Abbildung 7 lässt sich nun das konkrete Finding adressieren

und direkt im Quellcode öffnen. Konkret handelt es sich um folgende Zeile:

```
public static HashMap<Integer , Card> allCards ;
```

Hier hätte vorzugsweise gegen das Interface "Map" implementiert werden können:

```
public static Map<Integer , Card> allCards ;
```

Um diese Abhängigkeit aufzulösen. **Metrics-Reloaded** Metrics-Reloaded stellte sich für den Einstieg als bessere Option heraus. Es liefert zur Auswertung einen vollständigen HTML-Export, der mittels JavaScript eine konfigurierbare UI zur Verfügung stellt. Gleichzeitig liefert Metrics-Reloaded einfache Einschätzungen und Erläuterungen zu verschiedenen Metriken. Das machte es für uns leichter Problemstellen zu identifizieren und genauer zu prüfen und einen Überblick zu erhalten.

Beispielhaft ein Auszug eines Exports: In der Abbildung 8 lässt sich schnell erkennen, dass es zwei

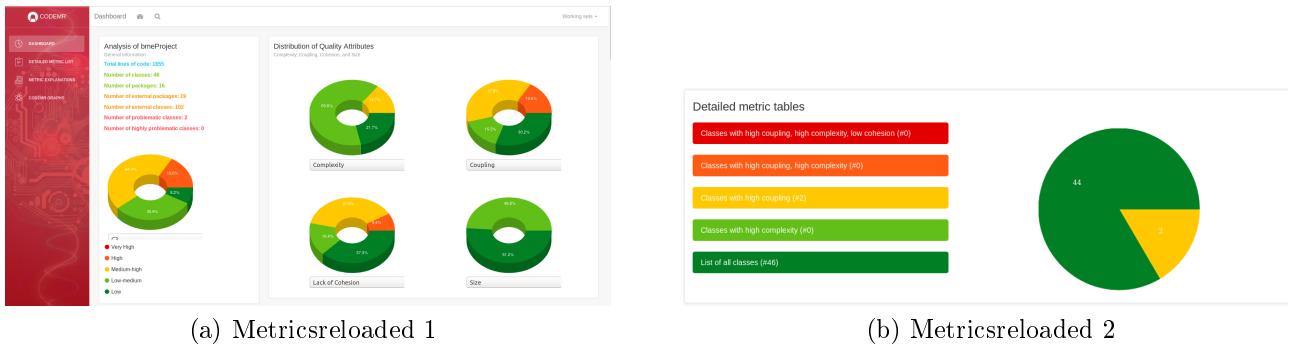


Figure 8: Screenshot: Metricsreloaded Überblick

problematische Klassen gibt, die zu analysieren sind. Wertet man dies nun weiter aus, erhält man die konkreten Klassen: Erwartungsgemäß werden die Klassen "BattleController" und "BattleCard" als potentiell problematisch erkannt. Erwartungsgemäß deshalb, da es sich um die zwei zentralen Klassen handelt, die die Spiellogik steuern und alle Objekte in diesem Zusammenhang halten und verwalten.

Classes with high coupling (#2)										
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	CBO APP	CBO LIB	RFC
1	BattleController	■	■	■	■	139	22	12	10	102
2	BattleCard	■	■	■	■	135	21	10	11	88

Figure 9: Screenshot: Metricsreloadedl High Coupling

4.4.4 Manuelle Tests

Wie bereits unter ?? angeführt, wurden unsere Feature-Banches und der nach dem Merge gebildeter neuer Stand immer manuell getestet. Hierfür besprachen wir die Anforderungen, die dieser Branch erfüllen sollte bzw. prüften diese in den jeweiligen Issues im Glo-Board und prüften daraufhin die Funktionalität manuell ab. Zusätzlich wurden hier auch ggf. Fehleingaben oder mögliche Szenarien geprüft, die als Konsequenz auftreten konnten, wie z. B. "Wird eine Null-Prüfung auf das Kartendeck durchgeführt, wenn alle Karten aufgezogen sind?". Gleichzeitig prüften wir zudem immer oberflächlich alle anderen Funktionalitäten ab, die ggf. auch unberührt von der neuen Änderung waren um Wechselwirkungen auszuschließen.

4.4.5 Pair-Programming und Debugging

Pair-Programming (und in unserem Fall oft "Pair-Debugging") ist eine praktikable Methodik um die Code-Qualität zu verbessern. Gleichzeitig ist es ein nützliches Anleitungsinstrument um neuere Entwickler in ein Projekt einzubringen. Wie im Abschnitt ?? bereits erschließen lässt, vergaben wir im Team häufig Aufgaben an mehr als eine Person auf einmal um diese gemeinsam an einer Anforderung entwickeln zu lassen. Gleichzeitig nutzten wir dies um z. B. auf unseren Team-Meetings oder Coding-Treffen gemeinsam zu programmieren und unseren Code zu debuggen. Hierbei stellten wir immer wieder fest, dass gerade komplexe Implementationen deutlich leichter umgesetzt werden können, wenn man diese einem anderen Mitglied im Prozess permanent erläutert und die logischen Operationen nochmals laut ausspricht.

5 Gamedesign

5.1 Feinschliff: Fokus auf Stellungsspiel

Bei Praxistests und in der Live Demo des letzten Semesters hat sich herausgestellt: Interessantestes Merkmal unserer Spielmechanik ist das Stellungsspiel der durch die Karten repräsentierten Spielobjekte. Wo sich vergleichbare Trading Card Games stärker auf Zahlenschieberei konzentrieren (Verrechnen von Statuswerten, Sammeln von Zählmarken, Jonglieren mit Würfelergebnissen) orientiert sich unser Produkt am Prinzip des Brettspielklassikers Schach, dessen emergentes Potenzial ausschließlich aus der Vielfalt der Anordnungsmöglichkeiten seiner Figuren erwächst. Die positive Resonanz zu diesem Umstand veranlasste Robert Sabo und Felix Baumgarten dazu, den Spielregeln einen letzten Feinschliff zu verpassen; im Zuge dessen die wenigen noch existenten Zahlen (Stärkewerte im oberen rechten Eck jeder Karte) über Bord zu werfen und das Stellungsspiel vollständig in den Mittelpunkt zu rücken. Zwar finden für konfrontative Auseinandersetzungen unter den Spielkarten noch immer Schadensberechnungen statt, doch basieren diese nicht mehr auf dem Vergleich individueller Statuswerte, sondern auf Kartentypen-abhängigen Eigenschaften:

- Jede Karte verfügt abhängig von ihrem Kartentyp über eine feste Anzahl an Trefferpunkten:
 - Quartiere: 3 Trefferpunkte
 - Kreaturen: 2 Trefferpunkte
 - Phänomene: 1 Trefferpunkt
- Ein Angriff reduziert die Trefferpunkte des Ziels stets um 1

Auf diese Weise wird das Regelwerk reduziert, die Kartengestaltung entschlackt und der Spielablauf vereinfacht – und gleichzeitig das Hauptaugenmerk unseres Gameplays unterstrichen.

5.2 Die Spielmechanik der Strömung

Im Zuge der Demoveranstaltung im vorherigen Semester, konnten Studenten, Professoren und weitere Besucher, das Spiel am analogen Prototypen testen. Viele User hatten Erfahrung mit Trading Card Games und konnten uns neben gutem Feedback auch Verbesserungsvorschläge zum Spiel geben. Der generelle Konsens fand den Fokus auf das Stellungsspiel einen interessanten Ansatz, den nicht viele Card Games berücksichtigen. Darauffolgend, haben sich Felix Baumgarten und Robert Sabo an die Spielmechanik gesetzt und ein weiteres Element des Stellungsspiels implementiert. Die Strömungsrichtung. Mann kann entweder im Uhrzeigersinn, oder gegen den Uhrzeigersinn angreifen. Ein Angriff wird jetzt immer in Strömungsrichtung ausgeführt. Aber auch Quartiere verschieben ihre Kreaturen und Phänomene in diese Richtung. Visuell wird das ganze über einen Wasserstrom angezeigt, der hinter den Spielkarten in die jeweilige Richtung rotiert. Damit ist auch diese Funktion innerhalb des Spieluniversums(Unterwasserwelt) kreativ umgesetzt und sie ist auf dem ersten Blick direkt sichtbar. Die Strömungsrichtung kann immer dann über einen Button manipuliert werden, wenn auch der Kompass verstellbar ist. Diese beiden Aktionen fallen unter dem Begriff "Spielfeld vorbereiten". Wenn diese nach Wunsch vorbereitet wurden, können Spielkarten gelegt und Farbzonen aktiviert werden. Das Element der Strömungsrichtung fördert wie der Kompass das Stellungsspiel des Kartenspiels.

6 Grafik- und Sounddesign

6.1 Video-Produktion

Ein Trailer zu unserem Produkt (Videospiel), sollte erstellt werden. Dieses ca. 1 Minute Video soll im BB-Gebäude in einer Dauerschleife mit den anderen interdisziplinären Projekten aus unserem Studiengang an großen Monitoren gezeigt werden. Der Trailer soll einen kurzen Einblick zu unserem Videospiel geben. Dabei soll die Tonalität und Spezifizierung unseres Produkts (Cardgame)

schnell erkennbar sein. Nach einer Ideensammlung einiger Teammitglieder in Form von Storyboards, wurde sich auf einen groben Aufbau geeinigt. Um einen gewissen Überblick über den dramaturgischen Ablauf zu kriegen, hat Robert Sabo ein Videoscript zum Trailer angefertigt (Siehe Datei: Video-Teaser-Trailer/Videoscript.pdf). In diesem Script steht allgemein der Zeichenstil, die Schriftarten, Länge der Abschnitte und deren Einstellungen, sowie allgemeine Schnittkonventionen. Die Handlung wurde in mehrere Einstellungen aufgeteilt, die eine gewisse Dramaturgie unterstützen. Visuelle und auditive Angaben wurden hier auch festgelegt. Die jeweils drei Abschnitte wurden unter den Beteiligten aufgeteilt (Abschnitt 1: Manuela, Abschnitt 2: Robert, Abschnitt 3: Gabriel). Sämtliche Videosegmente wurden mit Adobe After Effects realisiert. Die einzelnen Assets wurden mit Adobe Photoshop erstellt. Schlussendlich wurden die drei Abschnitte von allen Beteiligten von Robert Sabo einheitlich in After Effects zusammengeschnitten. Dabei war es wichtig die Tonalität gut herüberzubringen. Deshalb mussten Paar Abschnitte neu bzw. angepasst werden, um einheitlich zu wirken. Schlussendlich wurde die Datei im avi-Format gerendert. Da die Datei aber zu groß war, wurde sie mit Hilfe von Pia Korndörfer zu einem kleineren Format, mp4, komprimiert. Philadelphia Gauß hat noch extra für die Präsentation eine Trailer-Version mit Soundeffekten und Soundtrack erstellt, um auch einen passenden auditiven Eindruck des Videospieles für die Zuschauer zu geben.

Auf die einzelnen Abschnitte wird im Folgenden genauer eingegangen.

6.1.1 Einarbeitung in After Effects

Zunächst einmal erfolgte eine intensive Einarbeitungsphase mit dem Programm „After Effects“. Dabei waren sehr gute Youtube Tutorials eine große Unterstützung. Das Resultat von dem ersten Testvideo schaut wie folgt aus: Hierbei wurden Bilder freigestellt, ein Bild und der Titel animiert.

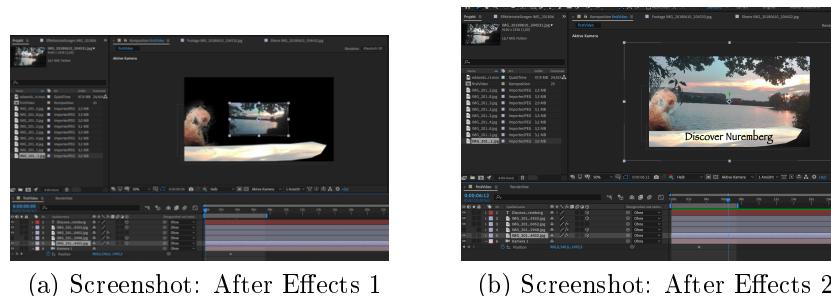


Figure 10: Screenshots After Effects

6.1.2 Animation des Logos

Das Video endet, wie in vielen bekannten Videos, mit unserem Logo. Hierfür musste das Edwards Biotope Logo animiert werden. Zunächst einmal wurde die Grafik (links) von dem Text (rechts) separiert. Die Absicht war es, zuerst das Symbol animiert einzublenden. Im Anschluss taucht der

Name „Edwards Biotope“ auf. Dieser Text wurde ebenso animiert. Zum Schluss werden beide Teile wieder zusammengefügt. Die Bildreihenfolge Abbildung 11 stellt diese Logo Animation dar.

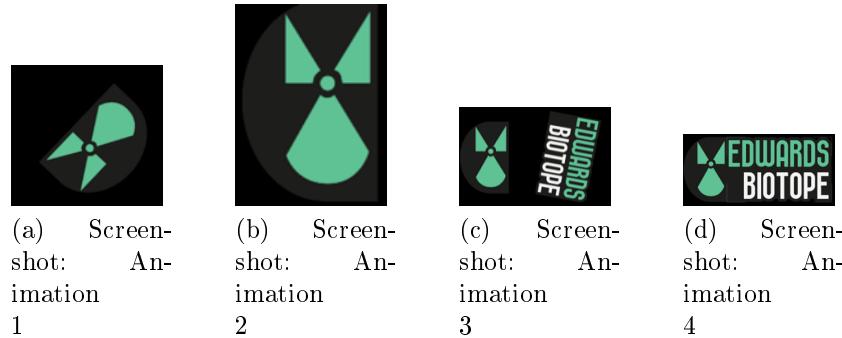


Figure 11: Screenshots Animations 1

Allerdings war das nicht sehr überzeugend. Daher wurde weiterhin experimentiert. So wurde das Logo mit dem Spielnamen nicht mehr separiert animiert, sondern von Anfang an zusammen. Der Bildablauf Abbildung 12 veranschaulicht diese Animation.

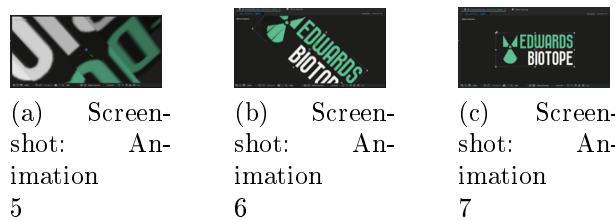


Figure 12: Screenshots Animations 2

Im Anschluss wurde ein CC Light Sweep Effekt hinzugefügt und dafür gesorgt, dass das Logo kurz glänzt. Mit der Bestimmung der Richtung verläuft dieser Effekt von links oben nach rechts unten. Dieser Effekt ist auf der Abbildung 13 zu sehen.

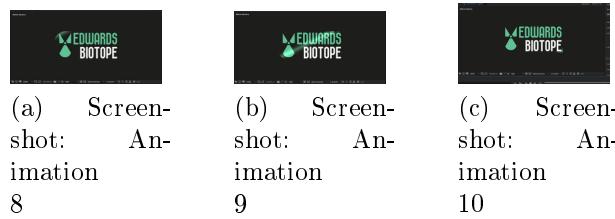


Figure 13: Screenshots Animations 3

6.1.3 Animation des Logos für denn Trailer

Für den Trailer fehlte am Ende des Trailers eine Logoanimation. Die Animation wurde mit Adobe After Effects erstellt. Als Material für die Animation wurde das Logo benötigt und ein Hintergrund,

der mit der Spielwelt von Edwards Biotope harmoniert. Deshalb wurde als Hintergrund ein Video Clip von Vimeo gesucht, das unter Creative Common Lizenz verwendet werden darf. Es wurde sich für einen Wellengang bei Sonnenuntergang im Meer entschieden. Das Logo wurde passend zu der Wellenbewegung animiert. Mit der letzten Welle wird das Logo unter das Wasser gespült und soll somit das abtauchen in die Unterwasser-Spielwelt von EDWARDS BIOTYPE symbolisieren. Die Logoanimation wurde noch einmal überarbeitet, um alle Einzelteile des Trailers aneinander anzupassen. Die Animation befindet sich entsprechend im Anhang unter Logoanimation.

6.1.4 Kompression des Trailers

Das Endprodukt des Trailers war eine 8GB unkomprimierte AVI Video-Datei. In dieser Größe konnte es nicht auf den Server der Hochschule hochgeladen werden. Deshalb musste der Trailer noch komprimiert werden. Die Komprimierung erfolgte mit dem Adobe Media Encoder. Als Dateiformat wurde MOV ausgewählt. MOV hat den Vorteil, die Datenmenge mit verlustfreier und verlustbehaftender Komprimierung auf 70 MB zu verringern ohne sichtbare Qualitätsverluste zu liefern. Die Animation befindet sich entsprechend im Anhang unter Logoanimation.

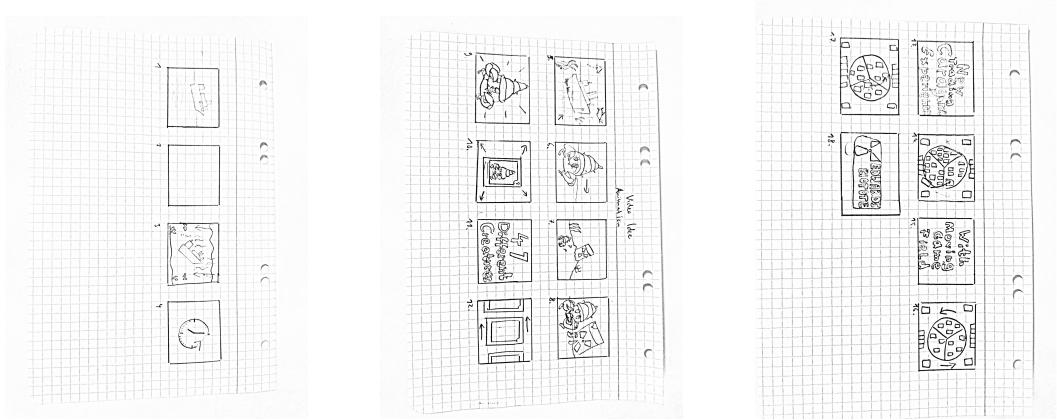
6.1.5 Storyboard

Für die Präsentation des Projekts musste ein kurzer Trailer angefertigt werden, der unser Projekt innerhalb einer Minute grob darstellt und dem Zuschauer Lust auf unser Spiel macht. Um erste Anregungen zu sammeln, wurde sich zusammengesetzt und Ideen miteinander ausgetauscht. Im Anschluss bekamen Gabriel, Pamela, Cigdem und Manuela die Aufgabe, Storyboards anzufer- tigen. Somit sollten die gesammelten Ideen veranschaulicht werden. Ein Storyboard ist eine Art visuelles Drehbuch, in dem Abläufe von Szenen konzipiert werden. Es kann skizziert, aber auch collagiert sein. Innerhalb des Storyboards können Kameraschnitte, Bewegungsanweisungen, Überblendungen und auditive Ansagen festgehalten werden. Der Trailer muss den Zuschauer in die Unterwasserwelt von EDWARDS BIOTYPE eintauchen lassen und sowohl einen Vorgeschmack für die Hintergrundgeschichte als auch eine kurze Sicht auf die Spielmechanik bieten. Dennoch sollte eine Länge von einer Minute nicht überschritten werden. Genau das galt es bei der Erstellung der Storyboards zu beachten. Das entsprechende Storyboard ist im Anhang beigefügt.

Letztendlich wurde sich nicht für eines der Storyboards entschieden, sondern es wurden die verschiedenen Umsetzungen miteinander kombiniert. Dies ergab schließlich die Vorgabe für unseren Trailer, der dann von Robert und Pia umgesetzt wurde.

In unserem Gameplay-Trailer sollte der grobe Spielablauf dargestellt werden. Als Programm wurde Adobe After Effects CC verwendet. Wie bereits Eingangs beschrieben, wurden für das Video mehrere Storyboards angefertigt. Hierzu ein Auszug in Abbildung ??

Nach einiger Überlegung wurde sich auf ein Storyboard geeinigt und den Trailer in 3 Abschnitte aufgeteilt, für die jeweils eine Person zuständig war. Abschnitt 1 ist die Vorgeschichte/Intro, Ab-



(a) Storyboard: Spielfeld 1

(b) Storyboard: Spielfeld 2

(c) Storyboard: Spielfeld 3

Figure 14: Storyboard: Spielfeld

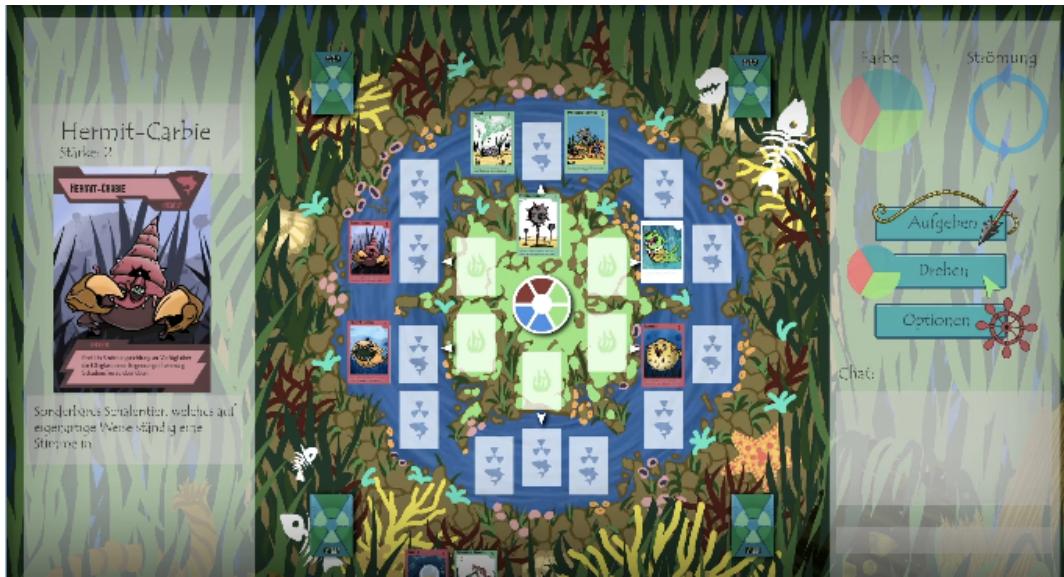


Figure 15: Darstellung des Spielfelds im Trailer

schnitt 2 die Verwandlung der Kreaturen in mutierte Monster und Abschnitt 3 Ingame-Gameplay. Hier wird die Arbeit an Abschnitt 3 beschrieben.

In Abschnitt 3 soll das Gameplay des Spiels gezeigt werden und zwar so nah am Endprodukt wie möglich. Dies war nicht sehr leicht, weil manche Ideen zum Gameplay noch nicht absolut feststanden. Die Idee war es die Grundfunktionen des Gameplays zu zeigen wie z.B. Farbrad drehen, Strömungsrichtung, Zonenaktivierung und einen ausgeführten Angriff. In dieser weise könnte ein Spielzug eines Spielers aussehen und es war wichtig diesen Ablauf einzuhalten. Zum Animieren wurden hauptsächlich die original Spiele-Assets verwendet, wobei bei den Spezialeffekten und der Seitenleiste die Original-Buttons noch nicht feststanden.

In Adobe After Effects wird anhand von Keyframes animiert. AE ist ein Compositing-Tool, das in Ebenen funktioniert, anders als Nuke(Compositing-Programm), das in Nodes aufgebaut wird.

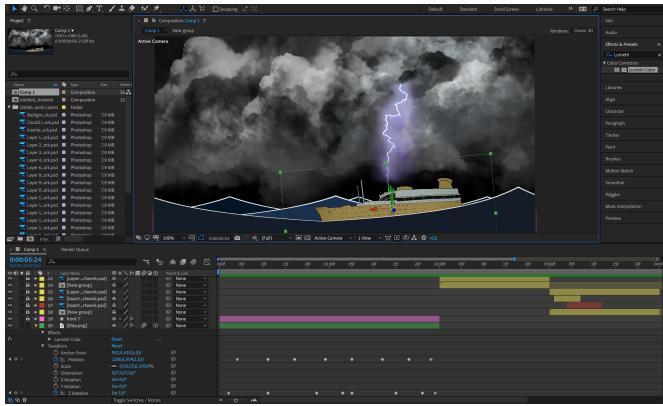


Figure 16: Screenshot: After Effects Intro

Pro Ebene ist ein Asset eingefügt. Jedes Asset lässt sich beliebig drehen, vergrößern, transparent machen und bewegen. Diese ganzen Parameter lassen sich auch über Keyframes definieren. Keyframes sind von der Zeitleiste abhängig, d.h. an dem Zeitpunkt an dem ein Keyframe ist, müssen die eingestellten Parameter angewendet sein. Von einem Keyframe zum nächsten wird diese Änderung aber interpoliert dargestellt, wodurch eine gleichmäßige Änderung im Video zu sehen ist.

Dieser Workflow ist deutlich einfacher als Gameplay aufzunehmen oder vor allem Bild für Bild selber zu bearbeiten.

6.1.6 Produktion Video Intro

Für einen Einstieg in die Geschichtsthematik des Spieles wurde der Betrachter mittels einer Kamerafahrt über das Meer an den eigentlichen Veranstaltungsort mitgenommen. Dafür wurde ein längliches Bild angefertigt, das in AfterEffects zu einer Komposition hinzugefügt wurde. Anschließend wurde eine neue “Kamera” erstellt und die 3D-Funktionen der Kompositionsebene aktiviert. Nun mussten nur noch zwei Keyframes gewählt werden, den Startkeyframe am unteren, sowie den Endkeyframe am oberen Ende des Bildes. Da Wasser, noch spezifischer gesehen Wellenbewegungen, sehr komplex zu animieren sind, wurde für die nächste Szene eine Wellenreihe in Adobe Animate angefertigt. Das Programm ermöglichte es die “Meereszacken” durch Anfertigung und Abspielen verschiedenster Frames zum Rotieren zu bringen. In After Effects importiert ergaben diese hintereinander liegenden Reihen in unterschiedlicher Ablaufgeschwindigkeit die Illusion einer sich bewegenden Meeresoberfläche. Das angefertigte Schiff musste jetzt an die Bewegungen der vordersten Welle angepasst werden. Diese Bildverschiebung kann in After Effects durch das Verändern der Position als auch Rotation sowie durch das Setzen mehrerer Keyframes (siehe Abbildung x) visualisiert werden. Auch der Hintergrund sowie der Blitz musste vorerst angefertigt und implementiert werden. Um das Aufleuchten des Blitzes naturgetreuer darzustellen, wurde seine Darstellung zwischen Erscheinen und Verschwinden noch verdoppelt.

6.2 Illustrationen

Da dieses Kartenspiel stark im Einklang mit der dahinter stehenden Geschichte und der umgebenden Atmosphäre funktioniert, hatte die Fertigstellung der restlichen Kartenillustrationen hohe Priorität. Die Kolorierung sowie die Licht- und Schattengabeung der ausstehenden Illustrationen brachten schließlich 46 individuelle Kartenmotive hervor.

6.2.1 Lichtgebung und Schattierung

Wie auch im letzten Semester gehörte das Setzen von Licht und Schatten zum letzten Schritt des Illustrationsdurchlaufes, welcher mit der Absicht der Erhaltung eines einheitlichen Illustrationsstiles erstellt wurde. Dabei unterteilte sich diese Aufgabe in die Nachbesserung der eingesetzten Farben innerhalb der geschlossenen Bereiche der Illustrationen sowie der eigentlichen Setzung von Schatten und Licht. Für diese Tätigkeiten wurde das Programm Procreate verwendet.

6.2.2 Karten kolorieren

Im 5. Semester der Projektarbeit wurden noch nicht alle Karten koloriert. Es waren noch 35 Karten, die keine Farbgebung erhalten haben. Die Kolorierung der Karten erfolgte auf einem MacBook Pro, mit der Software Adobe Photoshop CC 2019 und einem Grafiktablett von Bamboo. Um das einheitliche Gesamtbild sicherzustellen, wurden die Farben aus der Farbtabelle aus dem 5. Semester ausgewählt. Bei der Kolorierung wurde darauf geachtet, dass jede Karte einen ausgewogenen Komplementärkontrast hat. Zusätzlich benötigt jede Illustration mindestens drei Farben. Die Figuren müssen sich auch sichtbar von dem Hintergrund abheben. Entsprechende Materialien finden sich im Anhang unter "Karten_Kolorieren".

6.2.3 Spielkarten Outlines

Um die Outlines aller Illustrationen herzustellen wurde Adobe Photoshop verwendet. Die vorher angefertigten Skizzen von Pamela Schättin und Daniel Scharrer, wurden gescannt und als Vorlage zur digitalen Ausarbeitung benutzt. Mit einem Canvasformat von 850 x 560 px wurde mit einer Linienstärke von 4 px die vorher positionierte Vorlage digital nachgezeichnet. Kleinere Änderungen bzw. Verbesserungen wurden dabei vorgenommen. Die Datei konnte nun als .psd-Datei abgespeichert werden und dem nächsten Prozess, also dem Colorblocking, zur Verfügung gestellt werden.

6.2.4 Spielkartenrückseite Assets

Da im Spiel auch Spielkarten verdeckt liegen sollen, sprich im Kartendeckstapel oder die Handkarten des Gegners, müssen auch Assets für die Rückseite einer Karte vorliegen. Um die Assets herzustellen wurde Adobe Photoshop verwendet. Diese soll visuell eindeutig und schnell erkennbar

sein, da viele verschiedene bunte Farben auf der Kartenvorderseite (also Illustrationen) benutzt wurden. Um die Kartenrückseite davon abzuheben wurden wenige Farben und große Flächen benutzt. Zusätzlich wurde das Logo genutzt um eine eindeutige Zuweisung unserer IP herzustellen.

6.2.5 Spielkarten Hintergründe

Um die Hintergrüde aller Illustrationen herzustellen wurde Adobe Photoshop verwendet. Farblich wurde sich an der vorher festgelegt Farbpalette gehalten. Die vorher angefertigten Illustrationen (mit Outline, Colorblocking, Highlights und Shading) sind soweit fertig. Diese besitzen allerdings noch kein Hintergrund. Um diese Kreaturen, Quartiere und Phänomene nun noch in eine passende Umgebung zu versetzen, muss eine passende Unterwasserwelt her. Diese stellt in der Dimensionalen Wahrnehmung die hinterste Ebene dar. Das sich das Objekt (Kreatur, Quartier, Phänomen) visuell noch viel mehr vom Hintergrund absetzt, wurde zum Hintergrund ein minimalistischer Zeichenstil genutzt (also ohne Outlines). Kleine kreative Spielereien, um die Spielkarten individueller zu machen, wurden hinzugefügt. Die fertige Illustration steht.

6.2.6 Karten exportieren

In Adobe InDesign musste der Rahmen der verschiedenen Kartentypen, mit den Kartenillustrationen und dem Kartennamen aus einer CSV Datei zusammengefügt werden. Der Rahmen musste noch einmal überarbeitet werden, da keine Effekte mehr auf der Karte ausgezeichnet werden. Deshalb wurde das Textfeld für die Effekte aus dem Rahmen mit Photoshop entfernt. Nach der Zusammenführung InDesign, entsteht ein PDF, aus diesem wurde mit dem Adobe Acrobat Reader Pro einzelne PNGs exportiert. Die Benennung der einzelnen Karten erfolgte aufgrund einer ID, so lässt sich die Karten im Sourcecode durch die ID zusammenführen. Entsprechende Materialien finden sich im Anhang unter "Zusammenführung".

6.2.7 Hauptmenü-Bildschirm Assets

Um nicht direkt nachdem das Spiel gestartet wird in den Duell-Bildschirm zu kommen, wurde sich noch ein Hauptmenü-Bildschirm überlegt. Robert Sabo und Pamela Schättin kümmerten sich um die jeweiligen Assets die der Hauptmenü-Bildschirm beinhalten soll. Um die Assets herzustellen wurde Adobe Photoshop verwendet. Um einheitlich zu wirken, wurde der Illustrationsstil der Spielkarten hergenommen. Farblich wurde sich an der vorher festgelegt Farbpalette gehalten. Die Assets spiegeln das Spieluniversum, nämlich die Unterwasserwelt, visuell wieder. Es soll das Spieluniversum "Edwards Biotope" leicht "anteasern", und einen gewissen Eindruck des Spiels vermitteln. Das Background-Asset wurde von Robert Sabo entworfen und zeigt die Unterwasserwelt aus einem U-Boot heraus. links oben ist das Logo des Spiels zu sehen. Die Buttons wurden von Pamela Schättin erstellt. Die Buttons "Duell starten", "Deckeditor" und "Einstellungen" haben

jeweils immer ein passendes Icon aus der Unterwasserwelt. Ausgegraute Buttons sollen noch nicht auswählbar und "klickbar" sein.

6.2.8 Duell-Bildschrim-Assets

Robert Sabo kümmerte sich um die jeweiligen Assets die der Duell-Bildschirm beinhalten soll. Um die Assets herzustellen wurde Adobe Photoshop verwendet. Um einheitlich zu wirken, wurde der Illustrationsstil der Spielkarten hergenommen. Farblich wurde sich an der vorher festgelegt Farbpalette gehalten. Die Assets spiegeln das Spieluniversum, nämlich die Unterwasserwelt, visuell wieder. Dabei ergibt sich eine gewisse Stimmung, die erreicht werden soll. Der Aufbau des Duell-Screens ist durch Skizzen und aus dem vorher angefertigten analogen und digitalen Prototyp entstanden. Nun zum Interface Design. Grade bei so einer komplexen Spielmechanik, ist es wichtig, dass der User visuelle Eindrücke der gleichen Funktion zuordnen kann. Hierbei wurden viele grundlegende Gestaltungsgesetze für Interfaces beachtet und angewendet (Gesetz der Nähe, Gesetz der Gleichheit, Gesetz der Konstanz,...). Es soll auf dem Spielfeld klar ersichtlich sein, wo die Spielkarten abgelegt werden können, wo sich die jeweiligen Spielkartentypen wie z.B. Quartiere befinden, wo sich sowohl mein Deck, Friedhof und Handkarten, sowie die des Gegners befinden. Aber auch Interaktionsmöglichkeiten sollen auch auf dem ersten Blick ersichtlich sein. Dabei ist es wichtig das Spielfeld visuell von den Interaktionsmöglichkeiten wie Buttons und Anzeigefelder zu trennen. Das ist mit einem drei Spaltensystem gelöst worden. Die linke Spalte gibt Informationen über "Wer ist an der Reihe", einen Interaktionshinweis der Buttons und welche Karte ausgewählt ist wieder. Die rechte Spalte zeigt alle Buttons an, die mit dem Spielfeld interagieren und diese auch manipulieren können. Und die mittlere Spalte zeigt das tatsächliche Spielfeld, wo das Duell stattfindet, an. Um ein passendes Asset herzustellen müssen auch noch kommende Prozesse berücksichtigt werden. Sprich wie können diese Assets mit Hilfe der von IntelliJ IDEA vorhandenen Manipulationen, animiert werden. Rotieren, Skalieren, die Opacity ändern, Geschwindigkeit animieren etc. können später angewendet werden, um einen gewissen Interaktionsfeedback zu bekommen.

6.3 Musik und Soundeffekte

Die Musik soll für eine bestimmte Atmosphäre sorgen, im Fall von Edwards Biotope handelt es sich um eine comichafte, bunte Unterwasserwelt mit einer leicht düsteren Note. Die Aufgabe lag außerdem darin, das Gefühl zu vermitteln, sich direkt am Handlungsort zu befinden. Während die Musikproduktion im Allgemeinen schon hohe Anforderungen mit sich bringt, erhöhen sich diese speziell im Bereich der Spieleentwicklung noch um ein Vielfaches. Vor allem hat dies mit der Konzentration des Spielers zu tun: Die Musik muss eine Balance zwischen Unterhaltung und Ablenkung finden. Das Musikstück muss dabei relativ ruhig sein, damit sich der Spieler auf das Spiel konzentrieren kann, darf aber auch nicht zu monoton sein, damit es den Spieler nicht lang-

weilt oder gar nervt. Um all das zu berücksichtigen, wurden zuerst einige Ideen gesammelt. Dann wurde festgelegt, dass die Grundstimmung durch Unterwassergeräusche und Effekte repräsentiert wird. Solche Geräusche wie “Gluckern”, das Wellenbrausen, die U-Bootgeräusche, das Brüllen eines Motors, einem Sonar, das Atmen in einer Unterwassermaske, der Krach von Unterwasserexplosionen und die Kollisionen von Metallgegenständen im Wasser, wurden in einem Track mit Hilfe der Digital Audio Workstation (DAW) Ableton Live 10 gemischt. Um die Tiefe des Sounds zu erreichen, wurden verschiedene Filter wie Hi/Low-Pass, Compressor, Equalizer und zahlreiche Effekte wie Phaser, Flanger, Reverb, Delay u.a. benutzt. So entstand die atmosphärische Basis für den Track. Für die gute Laune sorgen Lieder aus den Jahren um 1920. Es wurden Lieder herausgesucht, die beim Hören im Spieler ein gutes Gefühl erzeugen und positive Stimmung verbreiten sollen und, wenn auch entfernt, im Zusammenhang mit Wasser und Meer stehen. So fiel die Wahl auf hawaiianisch anmutende Musik und auf deutsche Seemannslieder in klassischer Interpretation. Die Lieder wurden zusammen mit den Hintergrundgeräuschen in der DAW von Audacity mit Hilfe der Effekte Fade In/Out kombiniert.

7 Abschlussbetrachtung und Fazit

Am Ende der zwei Projektsemester steht ein Produkt, mit dem das Team zufrieden sein kann. Insbesondere das Spielprinzip und die audiovisuelle Präsentation wurden intern und extern bei verschiedenen Gelegenheiten als überraschend gelungen hervorgehoben, was dem im Projekt-papier festgehaltenen ursprünglichen Ansatz entspricht, technologische Innovationen oder wissenschaftliche Werte zugunsten der Annäherung an eine vertriebsreife Produktqualität zu vernachlässigen. Im Laufe der vergangenen Monate haben wir gelernt, dass persönliches Interesse für die Materie in immenser Leistungsbereitschaft münden kann, aber auch, dass Arbeitsagilität und damit letzten Endes auch -effizienz mit Teamgröße, Kommunikation und persönlichem Befinden einhergehen. Obwohl wir es schafften, regelmäßige Treffen abzuhalten und engagiert in das erste Projektsemester zu starten, stellte sich gegen Mitte eine Motivationsflaute ein, die erst dann wieder weichen sollte, als erste prestigeträchtige Spielfunktionen erfolgreich und interaktiv visualisiert waren. Schlussendlich war klar, dass das ursprünglich angestrebte Produkt mit seinen komplexen Features viel zu aufwändig sein würde, um es nach den gegebenen Rahmenbedingungen zu realisieren. Daher machten wir Abstriche im Gameplay und damit im Workload der Software-Entwicklung. So schafften wir es schließlich, die Essenz unseres Spieles – das strategische Spielduell – ansehnlich und lauffähig zu inszenieren und so unsere wichtigsten Ziele zu erfüllen.

References

- [1] After Effects Einsteigertutorial, <https://www.youtube.com/watch?v=wrKdozeXA2U1>, 01.06.2019
- [2] How to Create Cartoon Animation, <https://www.youtube.com/watch?v=QQgmXARn8aA>, 01.06.2019
- [3] After Effects Tutorial: Logo Animations, <https://www.youtube.com/watch?v=5WnaT1oFjt8>, 01.06.2019
- [4] Shine Logo Animation in After Effects, https://www.youtube.com/watch?v=uE0_nLnytp0, 01.06.2019

List of Figures

1	Screenshots Titlescreen	21
2	Screenshot: BattleScreen mit ImageButtons	21
3	Screenshot: Spielfeld mit Spielerbereichen	40
4	Zeichnung: Fields	43
5	Klassendiagramm Edwards Biotope	47
6	Screenshot: PMD Übersicht	50
7	Screenshot: PMD Beispiel	50
8	Screenshot: MetricsReloaded Überblick	51
9	Screenshot: MetricsReloaded High Coupling	51
10	Screenshots After Effects	54
11	Screenshots Animations 1	55
12	Screenshots Animations 2	55
13	Screenshots Animations 3	55
14	Storyboard: Spielfeld	57
15	Darstellung des Spielfelds im Trailer	57
16	Screenshot: After Effects Intro	58