Bane of Asphodel:
Entwicklung eines Action-Rollenspiels mit der GameEngine Unity 5

Vorgelegt von:

Konrad Leichtle
Schülen-Ennerhaus 2
6130 Willisau

Robin Schmidiger
Gutenegg
6125 Menzberg

Referent:

Herr Igo Schaller
Gartenstrasse 23
6130 Willisau

Korreferent:

Herr Erwin Hofstetter
Vonmattstrasse 20
6003 Luzern

Abstract

Im Rahmen dieser Arbeit wurde ein Computerspiel mit der GameEngine Unity 5 entwickelt. Alle 3D-Modelle wurden im Modellierprogramm Blender erstellt und der Programmcode wurde in der Programmiersprache C# geschrieben. Das Spiel ist ein Action Role-Playing-Game (RPG) namens *Bane of Asphodel* mit einer Spielzeit von etwa ein bis zwei Stunden. Im Spiel können fünf verschiedene Spielwelten erkundet und zahlreiche Gegner bekämpft werden. Die Geschichte des Spiels wurde selbst geschrieben und lehnt sich an moderne RPGs an. Neben den technischen Aspekten wurde grossen Wert auf die künstlerische Gestaltung gelegt. Mit dieser Arbeit konnte ein tieferer Einblick in den Aufbau und die Produktion von Videospielen gewonnen werden.

The goal of this project was to develop a videogame with the GameEngine Unity 5. All 3D-models were created in the 3D animation software Blender and all scripts were written in the programming language C#. The game is an action Role-Playing-Game (RPG) called *Bane of Asphodel* and takes about two hours to complete. In the game, the player can visit five different regions and combat many enemies. The story is an original creation and takes inspiration from other modern RPGs. While developing the game, we focussed not only on the technical aspect of game development, but also on the artistic aspect. With this project we managed to gain a deeper understanding of both the structure of video games and their development process.

1. Einführung

1.1. Abgrenzung des Themenbereichs und Fragestellung

Das Ziel dieser Arbeit war es, ein vollwertiges Computerspiel zu erstellen. Das Spiel sollte alle Elemente eines modernen *Role-Playing-Games (RPG)* enthalten und mit Tastatur und Gamepad spielbar sein. Für die Entwicklung des Spiels wurde die *GameEngine Unity 5* verwendet. Dabei bot sich die Gelegenheit, die Programmiersprache C# sowie das Modellieren, UV-Mapping, Riggen und Skinnen von Modellen mithilfe des 3D-Modellier-Programms Blender zu erlernen.

Die Hauptfragestellungen lauteten:

- Wie sind moderne Videospiele aufgebaut?
- Wie wird ein vollwertiges Spiel erstellt?

1.2. Motivation und Darlegung der Themenwahl

Videospiele sind besonders in der jungen Generation allgegenwärtig. Seien es virale Apps wie *Pokémon Go* oder kontroverse Shooter wie *GTA V*, Computerspiele sind immer aktuell. Seit *Tennis for Two* im Jahr 1958 hat sich das Medium stark verändert. Die Gaming-Industrie ist einer der grössten Märkte weltweit. Laut dem *2015 Global Games Market Report* von *Newzoo*, generierte die Industrie 2015 einen Umsatz von 91.5 Mrd. US Dollar [1]. Grosse Unternehmen, wie z. Bsp. *CD Project Red* oder *Rockstar Games*, beschäftigen mehrere hundert Angestellte, die über Jahre hinweg an einem einzelnen Spiel arbeiten.

Bevor wir mit der MATA begannen, hatten wir uns beide schon ein Jahr lang mit der Entwicklung von Spielen auseinandergesetzt. Neben dem Unterricht hatten wir uns, mithilfe zahlreicher Videotutorials und Lernwebsites, die Grundlagen der Entwicklungsumgebung *Unity 5* angeeignet. Wir wussten, dass die Entwicklung eines Computerspiels zeitaufwändig ist. Deshalb entschlossen wir uns, gemeinsam an einem Spiel zu arbeiten. Das ermöglichte uns, unsere unterschiedlichen Kompetenzen optimal einzubringen.

1.3. Begriffserklärungen

1.3.1. GameEngine

Eine GameEngine ist eine Entwicklungsumgebung zum Erstellen von Computerspielen. Je nach Spiel besteht die Engine aus den folgenden Bestandteilen: Einer Grafik-Engine für die grafische Darstellung auf dem Bildschirm, einem Physik-Engine für eine realistische Simulation von physikalischen Gesetzen, einem Soundsystem, einem Netzwerk-Code für Multiplayer- oder Online-Games und einem Scripting, oftmals in der Form einer vereinfachten Programmierumgebung. Häufig kreieren Entwickler eine eigene Engine, wie es zum Beispiel das Studio *Hello Games* für ihr Spiel *No Man's Sky* tat, damit diese genau auf das Spiel abgestimmt ist. Für unser Spiel wurde die Engine Unity 5 benutzt. [2] [3]

1.3.2. Role Playing Game (RPG)

Die ersten sogenannten RPGs, wie zum Beispiel *Dungeons and Dragons*, waren Brettspiele. Dabei übernahmen die Spieler die Rollen von Fantasiefiguren. Diese Art von interaktivem Fantasy-Roman wurde in den 70er-Jahren von Videospielentwicklern adaptiert und zu einer der grössten Branchen der Gaming-Industrie. In den meisten RPGs muss der Spieler Entscheidungen treffen, welche den Verlauf des Spiels beeinflussen. Man unterscheidet oftmals zwischen *Western RPG* und *Eastern RPG*. *Eastern RPGs*, auch *JRPG* genannt, sind meistens linear. Kampfsysteme sind oft Zug-basiert. Das bedeutet, dass die Spieler und Gegner in einer bestimmten Reihenfolge zum Zug kommen und nur dann angreifen können. Im Gegensatz zu *JRPGs* setzen *Western RPGs* wie *Skyrim*, *Fallout* oder *The Witcher 3* einen grösseren Schwerpunkt auf die Freiheit des Spielers. Dem Spieler werden zwar Aufgaben gegeben, doch er kann diese ignorieren und etwas anderes tun. Die Grenze zwischen *JRPG* und *Western RPG* ist in den letzten Jahren immer verschwommener geworden. Spielemacher nehmen immer häufiger die besten Elemente der beiden Genres und kreieren somit Computerspiele, welche man keiner der beiden Kategorien eindeutig zuordnen kann. [4] [5]

1.3.3. AAA-Games vs. Indie-Games

Spiele, die von grossen Studios entwickelt wurden, nennt man *AAA-Games*. Sie erhalten meistens starke finanzielle Unterstützung und investieren viel in Marketing. *Indie-Games* hingegen werden meistens von kleinen Studios entwickelt. Sie kosten weniger und haben meist weniger Spielzeit. AAA wird immer öfter mit guter Qualität

gleichgesetzt. Das ist jedoch nicht immer vorteilhaft. *Hello Games*, ein kleines Studio aus England, brachte im August 2016 ein Spiel namens *No Mans Sky* heraus. Obwohl *Hello Games* ein Indie-Studio ist, wurde das Spiel so vermarktet wie ein AAA-Game. Als das Spiel den Erwartungen der Spieler nicht gerecht wurde, löste das heftige Kritik gegenüber *Hello Games* aus. Das Studio wurde wie ein hundertköpfiges AAA-Studio behandelt, obwohl es aus nur 16 Mitarbeitern bestand. [6] [7]

1.3.4. Frames per Second (FPS)

FPS ist die Frequenz der Bilder, die angezeigt werden. Der Begriff kann bei Filmen und bei Computerspielen verwendet werden. Ein Entwickler sollte auf eine möglichst hohe Framerate zielen, denn eine zu tiefe Framerate führt zu schlechten Reaktionszeiten und lenkt vom Spielvergnügen ab. [8]

1.4. Grundlagen zum Objektorientierten Programmieren (OOP)

OOP ist ein fundamentaler Programmierstil. Beim Objektorientierten Programmieren wird ein System durch das Zusammenspiel von Objekten beschrieben. Jedem Objekt sind gewisse Eigenschaften (Attribute) und Methoden zugeordnet. Dem Objekt übergeordnet steht das Konzept der *Klasse*. In einer solchen Klasse werden Objekte zusammengefasst. Ein Objekt wird oft als eine Instanz einer Klasse definiert. Als Beispiel dafür kann man die Klasse *Einfamilienhaus* genauer anschauen. In einer Strasse können mehrere Einfamilienhäuser stehen, doch alle sind leicht verschieden. Sie sind zwar alle Instanzen der Klasse *Einfamilienhaus*, doch ihre Attribute sind unterschiedlich. Die Klasse *Einfamilienhaus* besteht aus drei Aspekten: einem Namen, den Attributen und den *Methoden*. Die Einfamilienhäuser können alle ihre Methoden ausführen. So können sie z. Bsp. Türen oder Fenster öffnen. Diese Methode kann von allen Instanzen ausgeführt werden, da sie alle von der Klasse *Einfamilienhaus* abstammen. Um eine Klasse benutzen zu können, muss man sie zuerst definieren. Im Fall der Klasse *Einfamilienhaus* sieht das ungefähr so aus:

```
public class Einfamilienhaus {}
```

Zwischen den Klammern definiert man die Attribute und Methoden. Wenn die Klasse *Einfamilienhaus* definiert ist, kann man die einzelnen Einfamilienhäuser instanziieren:

```
Einfamilienhaus MeinHaus = new Einfamilienhaus ();
```

Objektorientiertes Programmieren beruht auf vier Hauptkonzepten: Datenkapselung, Abstraktion, Vererbung und Polymorphie. [9] [10]

1.4.1. Datenkapselung

Unter Datenkapselung versteht man den kontrollierten Zugriff auf die Attribute und Methoden einer Klasse. Dadurch kann eine Klasse nur über definierte Schnittstellen auf die Attribute oder Methoden einer anderen Klasse zugreifen. Man nennt Datenkapselung auch *Information Hiding*, da die inneren Vorgänge einer Klasse versteckt und nicht von aussen einsehbar sind. Eine andere Klasse kann sozusagen das «Was» anfragen, jedoch nicht das «Wie». [9] [11]

1.4.2. Abstraktion

Abstraktion ist der Prozess des Weglassens von überflüssigen Einzelheiten, um auf etwas Allgemeineres und Einfacheres zu schliessen. Dies steht im direkten Zusammenhang mit Datenkapselung, da dadurch die Klasse vereinfacht wird und unwichtige Details verborgen werden. [9] [12]

1.4.3. Vererbung

Bei der Vererbung nimmt eine neue Klasse eine bereits existierende Klasse und baut darauf auf.

```
public class Gebäude {

}
public class Einfamilienhaus : Gebäude {

}
```

Mit der Klasse *Einfamilienhaus* kann man nun die Klasse *Gebäude* leicht ausbauen oder einschränken, ohne eine komplett neue Klasse erstellen zu müssen. Die Klasse *Zweifamilienhaus* erbt ebenfalls von der Klasse *Gebäude*, verändert sie jedoch auf eine andere Art und Weise als *Einfamilienhaus*. [9] [13]

1.4.4. Polymorphie

Polymorphie ist der griechische Begriff für Vielgestaltigkeit. Im Zusammenhang mit OOP ist damit gemeint, dass eine Methode oder ein Attribut abhängig von seiner Verwendung unterschiedliche Datentypen annehmen kann. Polymorphie tritt vor allem in drei Formen auf: Method overloading, Operator overloading und Method overriding. [9] [14]

Method overloading

Method overloading bewirkt, dass man mehrere Methoden mit demselben Namen definieren kann. Als einfaches Beispiel kann das Öffnen von Fenstern und Türen in

unserem Gebäude betrachtet werden. Die Funktion muss innerhalb der Klasse *Gebäude* definiert werde.

```
public class Gebäude {
   public void öffnen (Fenster dasFenster)
   {
   }

   public void öffnen (Türe dieTüre)
   {
   }
}
```

In diesem Beispiel ist die Funktion *öffnen ()* zweimal definiert. Man kann mit ihr beides eine Türe oder ein Fenster öffnen, obwohl dies unterschiedliche Variablen sind. [9]

Operator overloading

Operator overloading verhält sich genau wie Method overloading, jedoch sind es Operatoren wie '+', '-', oder '==' die doppelt definiert werden. Dies ist in den meisten Programmiersprachen bereits der Fall. Zwei Ganzzahlen addiert ergeben eine Ganzzahl, und zwei Kommazahlen addiert ergeben eine Kommazahl, doch beide benutzen den Operator '+'. [9]

Method overriding

Method overriding verhält sich anders als Method overloading. Method overriding erlaubt es einer Klasse, die bereits definierte Methode oder Funktion, einer ihr übergeordneten Klasse, zu überschreiben. Dafür muss sie denselben Namen haben und dieselben Parameter annehmen, wie die bereits definierte Funktion.

Dies sieht so aus:

```
public class Einfamilienhaus : Gebäude {
   public override void öffnen (Fenster fenster1, Fenster fenster2)
   {
         c = 2*a + b;
         return (c);
   }
}
```

In diesem Fall habe ich die vorher definierte Methode *öffnen ()* so überschrieben, dass mit einer Funktion zwei Fenster gleichzeitig geöffnet werden können. [9]

1.4.5. Abstract Class und Interface

Eine Abstract Class ist eine Klasse, die nicht als Objekt instanziiert werden kann. Sie kann nur als übergeordnete Klasse mittels Vererbung verwendet werden. Eine untergeordnete Klasse kann jeweils nur eine Abstract Class implementieren. Wenn eine Methode innerhalb der Abstract Class mit *abstract* gekennzeichnet ist, muss diese

von der untergeordneten Klasse überschrieben werden. Das Beispiel des Einfamilienhauses ist nicht gut für diesen Teil geeignet. Aus diesem Grund wird als neues Beispiel die abstract class *Nummer* benutzt.

```
abstract class Nummer {
   public int a = 0;
   abstract int b {get; set;}
}
```

In diesem Beispiel kann die Variable *a* von allen Klassen, die auf die Untergeordnete Klasse zugreifen können, aufgerufen werden. Die Variable *b* muss von der untergeordneten Klasse implementiert werden. [9] [12] [15]

Ein Interface definiert nur die Struktur einer Klasse. Es kann ebenfalls nicht als Objekt implementiert werden, doch im Gegensatz zur Abstract Class müssen die Methoden und Attribute in der Klasse, die vom Interface erbt, implementiert werden.

```
public interface nummer {
   int a {get; set;}
}
```

Hier kann nun das Interface *nummer* von einer Klasse implementiert, und in dieser die Variable *a* genauer definiert werden.

```
public class INummer : nummer {
   private int zahl;

   public int a {
         get {return zahl;}
         set {zahl = value;}
   }
}
```

Die Variable *zahl* kann nicht von einer anderen Klasse aufgerufen werden, die Variable *a* jedoch schon. Ruft man nun von aussen die Variable *a* mit *get* auf, gibt sie einem den Wert der Variable *zahl* aus. Benutzt man jedoch *set*, so wird der Variable *zahl* der jeweilige Wert zugewiesen. [9] [16]

1.5. 3D-Modellierung:

1.5.1. Polygone (Meshes)

Polygone, oder *Meshes*, sind die Grundlage des 3D-Modellierens. Ein Polygon besteht immer aus mindestens drei sogenannten *Vertices*, also Ecken, welche relativ zu einem Mittelpunkt im dreidimensionalen Raum stehen. Zwei Vertices bilden eine *Edge*, also eine Kante und drei oder mehr Vertices eine *Face*, eine Fläche. Jedes 3D-Modell besteht aus mindestens einem Polygon. Je mehr Vertices ein Polygon hat, desto mehr muss der Computer rechnen, um es darzustellen. Deshalb wurde in unserem Spiel ein Stil namens *Low-Poly* verwendet. Ein Low-Poly-Modell hat möglichst wenig Vertices. [17]

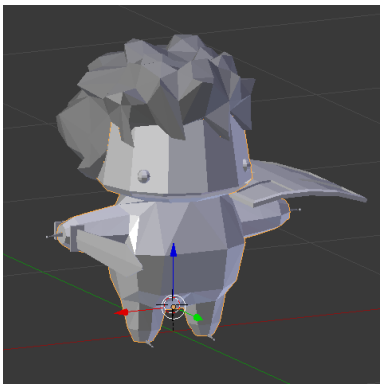

Abbildung 1: Modell der Spielfigur

1.5.2. UV-Mapping

Die Textur eines Modells wird durch ein *Material* bestimmt. Ein einfaches Material ist einfarbig und jede Face wird mit dieser Farbe bemalt. Um dem Modell eine mehrfarbige Textur zu geben, muss man eine sogenannte *UV-Map* erstellen. Die Buchstaben U und V stehen für die zwei Achsen der UV-Map, da die Buchstaben X, Y und Z bereits für die dreidimensionale Darstellung benutzt werden. Beim UV-Mapping wird das Polygon auf eine Fläche projiziert. Dazu muss man das Polygon *zerschneiden* und die Einzelteile auf der UV-Map anordnen. Diese Teile können mit Paint.net oder Photoshop eingefärbt werden. [18]

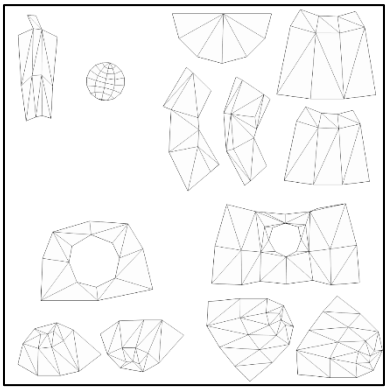

Abbildung 2: UV-Map der Spielfigur

1.5.3. Normal Maps

Da Low-Poly Meshes wenige Details vorweisen können, muss man das Modell mit *Normal Maps* bestücken. Durch eine Normal Map werden Schlagschatten auf eine flache Ebene gezeichnet. Somit wird ihnen das Aussehen einer *3D-Mesh* verliehen. Dies funktioniert natürlich nur bis zu einem gewissen Grad und ist kein Ersatz für echte 3D-Meshes. [19]

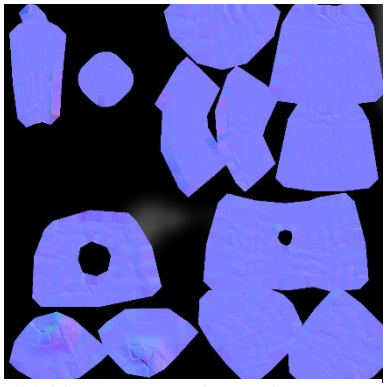

Abbildung 3: Normal Map der Spielfigur

1.5.4. Rigging

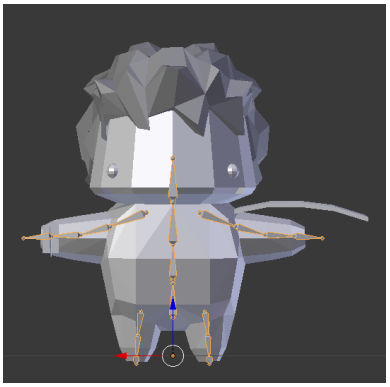

Abbildung 4: Rig der Spielfigur

Es gibt verschiedene Arten, ein Modell zu animieren. Man kann die Vertices bewegen und somit, wenn auch umständlich, eine Animation speichern. Eine viel effizientere Methode des Animierens bietet die Verwendung einer *Armature*. Diese fungiert als Skelett für das Modell. Das Skelett besteht aus *Bones*, also Knochen, welche über Gelenke miteinander verbunden sind. Das Erstellen dieses Skeletts nennt man *Rigging*. In Unity 5 kann man einen *Rig* als *Humanoid* definieren. Dazu muss das Skelett aus mind. 16 Knochen bestehen und eine menschenähnliche Form haben. Das Nützliche daran ist, dass alle Animationen, die für Humanoid-Rigs gemacht wurden, für alle Humanoid-Rigs funktionieren. Dies erleichtert die Arbeit enorm, da man nun Animationen für andere Modelle wiederverwenden kann. [20]

1.5.5. Skinning

Sobald das Skelett, auch Rig genannt, fertig ist, muss das Modell *geskinnt* werden. Beim *Skinning* wird definiert, welcher Bone welche Vertices des Polygons bewegen kann. Bei einfachen Rigs kann ein Modell automatisch vom Computer geskinnt werden, doch bei komplexeren Rigs muss dies manuell gemacht werden. Der gesamte Prozess von Modellieren, UV-Mapping, Rigging und Skinning kann in *Anhang VIII* in einem Zeitraffer von Robin Schmidiger angesehen werden. [21]

1.5.6. Animationen

Ist ein Modell geriggt und geskinnt, kann man mit den Animationen beginnen. Animationen bestehen aus sogenannten *Keyframes*. Eine solche Keyframe kann die Position, Rotation, Grösse, etc. der einzelnen Bones speichern. Die Bewegung zwischen den zwei Keyframes wird vom Computer berechnet und kann mit weiteren Keyframes justiert werden. Man muss also nicht für jedes Frame einzeln eine Keyframe erstellen, sondern kann dies z. Bsp. für jedes zehnte Frame tun. Die fertigen Animationen können in Unity 5 importiert und gegebenenfalls weiterverwendet werden. [22]

1.6. Binäre Serialisierung

Serialisierung ist ein Prozess bei welchem man Datenstrukturen in ein Format konvertiert, welches permanent abgespeichert werden kann. Den Prozess, bei dem man diese Daten wieder rekonstruiert, nennt man Deserialisierung. Es gibt verschiedene Arten, wie man Serialisierung angehen kann. Binäre Serialisierung konvertiert die Daten in ein Binäres Format. Unity 5 unterstützt mit einer Klasse namens BinaryFormatter die Binäre Serialisierung. Aus diesem Grund wurde in *Bane of Asphodel* diese Variante verwendet. [23]

1.7. Kamera und Rendering

Ein Animationsfilm besteht aus unzähligen aneinandergereihten Bilder, die von einem Computer berechnet und ausgegeben wurden. Das Berechnen und Ausgeben von Bildern nennt man *Rendering*. Bei Computerspielen müssen diese Bilder in Echtzeit gerendert und ausgegeben werden. Dieser Prozess braucht eine hohe Rechenleistung. Um eine möglichst hohe Framerate zu erzielen, muss die benötigte Rechenleistung möglichst geringgehalten werden. Damit trotzdem eine gute Bildqualität erreicht werden kann, können einige Techniken angewandt werden. Die folgenden Methoden oder Effekte werden zum einen zur Optimierung und zum anderen zur grafischen Bereicherung verwendet. Veranschaulichungen der folgenden Effekte können im Anhang VIII gefunden werden.

1.7.1. Shading und Shader

Shading oder Schattierung bezeichnet die Simulation der Oberflächen-eigenschaften von Objekten in 3D-Computergrafiken. Dafür wird ein Shader verwendet. Das ist ein Hardware- oder Software-Modul, der bestimmte Rendering-Effekte bei der 3D-Computergrafik implementiert. Software-Shader wurden ursprünglich in Assembler geschrieben, doch heute gibt es noch andere Sprachen für deren Programmierung. [24]

1.7.2. Temporal Antialiasing (TXAA)

TXAA ist eine Methode, bei der an mehreren Orten der vom Pixel gedeckten Fläche eine Stichprobe genommen wird. Dieser Prozess wird für einige Frames wiederholt. Mithilfe dieser Stichproben wird die Farbe des Pixels berechnet. Mit dieser Methode kann man eine höhere Auflösung erzielen, und die Übergänge zwischen zwei verschiedenfarbigen Flächen wirken geschmeidiger. Der Vorteil dieser Methode im

Vergleich zu anderen Antialiasing-Algorithmen ist, dass bei stark bewegten Bildern keine flackernden Pixel entstehen. Die gleiche Methode wird oft auch bei Animationsfilmen eingesetzt. [25]

1.7.3. Bloom

Bloom ist ein Effekt, der verwendet wird, um die Lichteffekte einer echten Kamera zu reproduzieren. Helle Hintergründe bewirken mit diesem Effekt Lichtschlieren an den Grenzen zu einer dunkleren Fläche. [26]

1.7.4. Screen Space Ambient Occlusion (SSAO)

SSAO ist eine Shading-Methode, mit der mit relativ kurzer Renderzeit eine realistische Beschattung von Szenen erreicht wird. Das Ergebnis ist zwar nicht physikalisch korrekt, doch die Alternative *globale Beleuchtung* ist sehr rechenintensiv. [27]

1.7.5. Chromatische Aberration (CA)

Chromatische Aberration ist, wie auch Bloom, ein Effekt mit dem die Nachahmung einer Kamera erzielt wird. In der Fotografie ist CA ein Abbildungsfehler optischer Linsen, der dadurch entsteht, dass Licht unterschiedlicher Wellenlänge oder Farbe verschieden stark gebrochen wird. Das führt zu Farbsäumen an Hell-Dunkel-Übergängen. Dieser Effekt kann auch in Computerspielen verwendet werden, wird dort jedoch künstlich mit einem Script generiert. [28]

1.7.6. Tonemapping

Tonemapping ist ein Prozess, welcher 16bit Farbwerte in 8bit Farbwerte umwandelt. Da viele Bildschirme heutzutage lediglich 8bit Farben darstellen können, müssen die 16bit Farben, welche von Unity ausgegeben werden, auf 8bit Farbwerte reduziert werden. [43]

2. Produktidee

Unser Ziel war es, ein gutes, anspruchsvolles Spiel zu machen. Das Spiel sollte so gestaltet sein, dass der Spieler durch die Story und das Gameplay dazu motiviert wird, weiter zu spielen. Es sollte sich an Open-World-RPGs orientieren und möglichst viele klassische Elemente dieser enthalten. Dazu gehören: Ein Inventar-System, eine interessante Spielwelt, ein Kampfsystem, intelligente und vielseitige Gegner und eine interessante Story die den Spieler vorantreibt. Die Welt des Spiels sollte aus mehreren Teilgebieten bestehen, welche jeweils verschiedene Gegner und Aufgaben enthalten. Zudem sollte jede Welt einen eigenen Boss-Gegner haben, einen speziellen und schwierigen Gegner. Wenn der Spieler gewisse Teile der Story abgeschlossen hat, sollte er Begleiter (Companions) mit auf den Weg bekommen. Diese Begleiter sollten ihm in Kampfsituationen helfen. Der Spieler sollte zudem zwischen den verschiedenen Figuren wechseln können. Ein ähnliches System ist in den LEGO Computerspielen zu finden. Dort gibt es meistens mehr als 100 Charaktere freizuschalten. Der Spieler sollte zudem die Wahl haben, ob er mit Gamepad oder mit Tastatur spielen möchte.

Wir wollten jedoch nicht nur den technischen Aspekt von Gamedesign beachten, sondern auch den gestalterischen. Die Ästhetik und das Design eines Spiels sind enorm wichtig für die positive Bewertung durch die Spieler und Kritiker und den kommerziellen Erfolg eines Spiels. Wir wollten für unser Spiel einen eigenen Stil verwenden und die Umgebung und Gegner gemäss diesem Stil gestalten. Das Spiel sollte interessant und ansprechend gestaltet sein. Die verschiedenen Umgebungen sollten thematisch schlüssig und trotzdem aufeinander abgestimmt wirken. Die Gegner sollten ebenfalls dem Thema ihrer Umgebung angepasst sein.

3. Vorgehen und Methode

3.1. Erlernen der Programmiersprache und Wissensbeschaffung

Die Grundlagen des OOP waren dank dem Freifach Programmieren von Igo Schaller bereits bekannt, wenn auch mit der Sprache Processing. Unity 5 unterstützt mehrere Programmiersprachen, unter anderem *C#* und *JavaScript*, doch Processing gehört nicht dazu. Als erster Schritt musste die gewählte Programmiersprache *C#* und der Umgang mit der Engine erlernt werden. Dabei halfen Video-Tutorials von *EteeskiTutorials* [41], *BurgZergArcade* [42], *HardlyBriefProgramming* [43] und anderen YouTube-Kanälen. Die Scripting API und das Scripting Manual von *unity3d.com* boten ebenfalls exzellente Hilfestellungen.

3.2. Produktion des Spiels

3.2.1. Tränke-Inventar

Die Arbeit am Inventarsystem startete bereits kurz nach der Abgabe der Projektskizze. Durch die Tutorial-Serie *Unity 5 RPG Series – Item System* von BurgZergArcade wurden die Grundlagen zu Interfaces und ScriptableObjects erlernt. Als Erstes wurde eine Klasse namens ScriptableObjectDatabase erstellt. Diese erbt von ScriptableObject, einer Klasse welche von Unity 5 bereitgestellt wird. Die ScriptableObjectDatabase speichert eine Liste von Instanzen einer Klasse T. In der Klasse ScriptableObjectDatabase wurden die verschiedenen Methoden definiert, welche die Datenbank haben muss. Dazu gehören *Add ()*, um Objekte hinzuzufügen, *Remove ()*, um sie zu entfernen, *Get ()*, um sie aufzurufen und noch mehr. Die gesamte Klasse ist im Anhang einzusehen. Sobald diese Grundlage für die Datenbank fertiggestellt war, wurde ein Interface namens IISPotion erstellt, welches definierte, welche Variablen zu einem Trank gehörten. Daraufhin wurde die Klasse ISPotion geschrieben. Diese erbte von IISPotion und implementierte die Variablen. Zudem wurden auch Konstruktoren für ISPotion definiert. Ein Konstruktor kreierte eine neue, leere Instanz und der andere kopierte eine bereits bestehende ISPotion. Zuletzt wurde eine kurze, aber wichtige Klasse definiert. Die Klasse namens ISPotionDatabase sieht so aus:

```
namespace Konrad.ItemSystem{
   [CreateAssetMenu]
   public class ISPotionDatabase : ScriptableObjectDatabase <ISPotion> {}
}
```

Als erstes wird festgesetzt, dass die Klasse Teil der Namespace Konrad.ItemSystem ist. [CreateAssetMenu] bewirkt, dass eine neue ISPotionDatabase einfach mit Rechtsklick erstellt werden kann, was schneller geht, als indirekt über einen Editor oder ein Script. Ursprünglich wurde ein selbsterstellter Editor benutzt, doch dieser führte zu Problemen beim Exportieren des Spiels und wurde daraufhin entfernt. Zuletzt wird die Klasse ISPotionDatabase definiert. Sie erbt von ScriptableObjectDatabase und speichert die Klasse ISPotion. Nun konnte die Datenbank verwendet werden.
Mit dem Speichermedium abgeschlossen, wurde nun das Inventar-GUI erstellt. Vorerst wurde es komplett mit Scripts erstellt, doch schlussendlich wurde das Canvas-Objekt von Unity 5 verwendet. Kontrolliert wurde dieses mit einem Script. Das Einzige, was mit Script angezeigt werden musste war eine Liste der Tränke, denn diese musste in jedem Frame aktualisiert werden.

3.2.2. Spielfigur und Kampfsystem

Das Modell der Spielfigur wurde in Blender erstellt. Sie wurde geriggt, geskinnt und die verschiedenen Animationen wurden erstellt. Dank dem Humanoid-Rig konnten einige Animationen kostenlos aus dem Internet heruntergeladen und verwendet werden. Dies sparte viel Zeit, vor allem auch beim Kreieren der Companions. In Unity 5 muss man, um Animationen vernetzen zu können, einen *Animator Controller* erstellen. Ein *Animator Controller* besteht aus Animations-Clips und aus den Übergängen von einem Clip zum anderen. Diese Übergänge haben bestimmte Bedingungen. Solche Bedingungen rufen eine Variable ab und überprüfen, ob sie einen bestimmten Wert angenommen hat. Ist dies der Fall, findet der Übergang statt, ansonsten nicht. Die Spielfigur wird von einem Script namens *BasicCharacterController* gesteuert. Unter anderem nimmt dieser die Inputs des Spielers an und wandelt sie in die richtige Variable des *Animator Controllers* um. Sobald man die Spielfigur kontrollieren konnte fing die Arbeit an einem Kampfsystem an. Das in *Bane Of Asphodel* verwendete System basiert auf sogenannten *Triggern*, eine von Unity 5 bereitgestellte Unterklasse von *Collidern*. Im Gegensatz zu *Collidern* sind Trigger durchlässig. Mit einfachen Funktionen kann man feststellen, ob ein bestimmter Collider in den Trigger eingetreten oder ausgetreten ist. Ein Script auf einem Gegner detektiert das Eindringen der Waffe des Spielers und zieht zufolge dem Gegner Lebenspunkte ab. Dasselbe geschieht auch umgekehrt, wenn ein Gegner einen Spieler angreift. Nun sollten jedoch nur bestimmte Animationen Schaden verursachen. Dies wurde mit einer Kurve auf den

Angriffs-Animationen gelöst. Animationen können mit Kurven versehen werden, welche einen Zahlenwert im Verlauf einer Animation darstellen. Ist der aktuelle Wert der Kurve namens *damage* grösser als 1, so fügt die Animation dem Gegenüber Schaden zu, ansonsten nicht. Um den Kampf etwas einfacher zu gestalten, wurde eine Funktion namens *FindClosestEnemy* definiert. Diese bewirkt, dass der Spieler beim Angreifen in die Richtung des ihm am nächsten stehenden Gegners schaut. Es ist dank dieser Funktion einfacher, die Gegner mit den Attacken zu treffen. Die Spielfigur hat eine bestimmte Anzahl an *Lebenspunkten*. Wird ein Spieler angegriffen, werden ihm Lebenspunkte abgezogen. Sobald die Lebenspunkte auf null sinken, «stirbt» die Spielfigur. In diesem Spiel bedeutet das, dass man auf den letzten Speicherstand zurückgesetzt wird.

3.2.3. Spielwelten

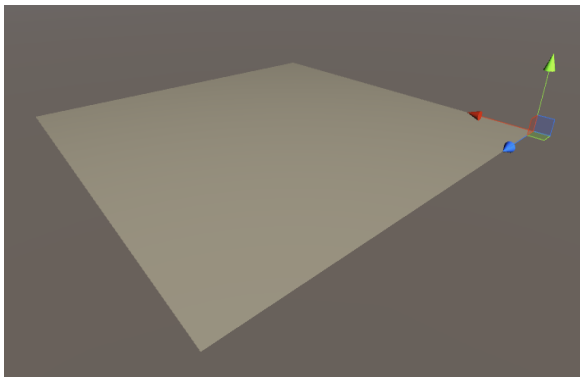

Abbildung 5: Unbearbeitetes Terrain

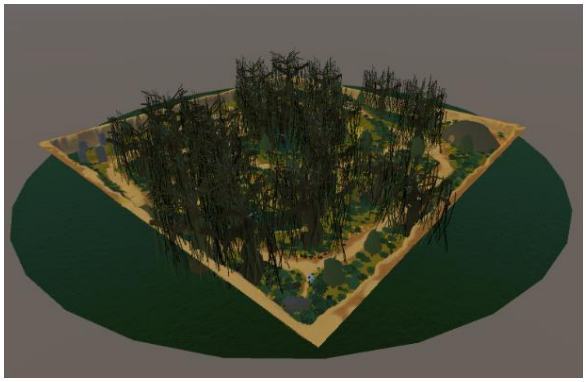

Abbildung 6: Fertig bearbeitetes Terrain

Bereits zu Beginn waren fünf verschiedene Maps geplant. Diese wurden gemeinsam konzipiert und geplant, damit sie ein möglichst breites Spektrum an Erlebnissen bieten können. Als erstes musste in jedem Fall das Terrain erstellt werden. Das Terrain ist im Grunde genommen die Geographie der Spielwelt. Es ist eine Art Plane, welche an gewissen Stellen erhöht werden kann und somit eine 3D-Landschaft darstellt. Unity 5 stellt einen Terrain-Editor zur Verfügung, mit welchem durch einfache Werkzeuge das Terrain modelliert und texturiert werden kann. Zudem erlaubt es, Gras und Bäume in grossen Mengen zu platzieren. Sobald das Terrain fertiggestellt war, musste die Welt mit Objekten wie Gebäuden oder Brücken bestückt werden. Diese wurden mithilfe von Blender erstellt und in Unity 5 importiert.

3.2.4. Gegner

Genauso wie bei der Kreation der Spielfigur musste für einen Gegner zuerst ein Modell erstellt und animiert werden. Daraufhin wurde ein Script namens *Enemy* geschrieben.

Darin wurden die Lebenspunkte des Gegners sowie *Funktionen* wie *takeDamage* definiert. Letztere kontrolliert den Schaden, der einem Gegner zugefügt werden kann. Als nächstes wurde die Künstliche Intelligenz (KI) erstellt. Dafür wurde das kostenlose Plug-In für Unity 5 namens *RAIN AI* verwendet. Die KI besteht aus einem einfachen BehaviourTree (ein Flussdiagramm aus Wenn-Dann Befehlen) und mehreren Sensoren mit welchen der Gegner den Spieler wahrnehmen können. Ein paar weitere Elemente wurden von RAIN AI bereitgestellt und nicht weiter verändert. Es wurden drei BehaviourTrees erstellt. Einer wurde für alle normalen Gegner verwendet. Die vier ersten Boss-Gegner wurden mit einem komplexeren BehaviourTree ausgestattet. Der letzte BehaviourTree wurde spezifisch für den finalen Boss-Gegner erstellt.

3.2.5. Story

Die Grundzüge der Story wurden zeitgleich mit dem Konzept der Arbeit geschrieben. Vorerst wurde nur eine grobe Skizze des Plots geschrieben an der man sich beim Erstellen der Welten und Gegnern orientieren konnte. Um möglichst viele Spieler zu erreichen, wählten wir Englisch als Sprache des Spiels. In den folgenden Monaten wurde ein Drehbuch geschrieben. Basierend auf diesem wurden in Unity 5 Standbilder aufgenommen, welche in Photoshop mit dem Dialog in Form von Textblöcken versehen wurden. Zuletzt wurde für die einzelnen Szenen eine ScriptableObjectDatabase erstellt, welche die Bilder speichert. Jede Szene wurde in einem separaten ScriptableObject gespeichert und im Quest System weiterverwendet.

3.2.6. Questsystem

In RPGs werden die Aufgaben des Spielers *Quests* genannt. Das Questsystem ist in drei Teile aufgeteilt: Kill-Quests, Location-Quests und Cutscenes. Alle drei sind jeweils in einer Klasse definiert und können mit einem Konstruktor implementiert werden.

Kill-Quests

Bei einem Kill-Quest muss der Spieler eine bestimmte Anzahl gewisser Gegner besiegen. Dies wurde mithilfe von *Events* realisiert. Wenn ein Gegner stirbt, sendet dieser ein Signal aus und überträgt eine String-Variable an alle Scripts die «zuhören». Das Script, in welchem der aktive Quest sich befindet, ist ein solcher Zuhörer. Es überprüft die String-Variable, und wenn diese mit der des zu tötenden Gegners übereinstimmt, schreitet der Quest fort. Die Aufgaben wurden in einem ScriptableObject gespeichert. Dieses hält eine Klasse mit Variablen, welche unter anderem den Typ und die Anzahl der Gegner speichern.

Location-Quests

Bei einem Location-Quest muss der Spieler an einen definierten Ort gelangen. Auch dieser Quest wurde mit Events realisiert. Zudem wurden auch Trigger verwendet. Der Trigger erkennt, wenn ein Objekt hineingerät. Ist dieses Objekt der Spieler wird, genauso wie beim Kill-Quest, ein Signal ausgesendet und von einem anderen Script überprüft.

Cutscenes

Die Cutscenes sind streng genommen keine Quests. Hat der Spieler einen gewissen Quest abgeschlossen, wird eine Serie von Standbildern angezeigt, welche die Story vermitteln. Wie vorhin bereits erwähnt, wurde dies mit ScriptableObjects gelöst. Der Spieler kann sich die Bilder beliebig lange ansehen, und auf Knopfdruck schaltet es zum nächsten Bild. Sobald die Szene vorbei ist, wird der nächste Quest eingeleitet.

Fertigstellen des Questsystems

Zuletzt mussten die drei Quest-Arten kombiniert werden. Dazu wurde eine Klasse namens *ThreeTypeQuest* erstellt. Diese kann im alle drei Arten von Quests in sich speichern. Je nachdem welche Art von Quest im Konstruktor eingegeben wird, erhält die Variable *QuestTypeID* einen anderen Wert. Nun wurde für jede Welt ein Script geschrieben, welche eine Liste von ThreeTypeQuests enthält. Wurde eine Aufgabe erfüllt, schreitet das Script weiter in der Liste fort und der nächste Quest wird aktiv. In den Maps *Fields*, *Forest*, *Swamp* und *Ruins* gibt es nur einen linearen Quest. Dieser besteht meist aus einer Kombination aus Kill- und Location-Quests und Cutscenes. Jeder dieser Quests kulminiert in einem Boss-Kampf. Da die Town-Map als eine Art Nexus fungiert, gibt es dort mehrere unzusammenhängende Quests. Je nachdem, welcher Boss gerade besiegt wurde, muss der Spieler daraufhin in der Town Map eine andere Aufgabe erfüllen. Dies wurde dadurch gelöst, dass die verschiedenen Quests jeweils in einer anderen Liste gespeichert wurden. Durch eine Variable wird erkannt, welcher Quest gerade aktiv ist. Dieser Quest wird dann vom Script verarbeitet.

3.2.7. Companions

Die Companions selbst wurden genauso erstellt wie die erste Spielfigur. Sobald das Modell und die Animationen fertiggestellt waren, wurde eine Companion-KI erstellt. Mithilfe dieser können Companions dem Spieler folgen und diverse Gegner angreifen. Von einem Script aus wird gesteuert, welcher der Charaktere momentan vom Spieler gesteuert wird. Die anderen werden von der Companion-KI gesteuert. In diesem Script

wird zuerst nach allen Objekten mit der Bezeichnung *Playable* gesucht. Diese werden dann in einem Array gespeichert. Der Spieler kann durch Knopfdruck zwischen den Charakteren wechseln. Das ist möglich, da alle fünf spielbaren Charaktere sowohl mit KI als auch mit dem BasicCharacterController ausgestattet sind. Wenn der Spieler die Figur steuert, wird die KI deaktiviert und der BasicCharacterController ist aktiv. Wechselt der Spieler Charaktere, wird die KI wieder aktiv und der BasicCharacterController wird ausgeschaltet.

3.2.8. Speichern

Das Speichern des Spielstandes basiert auf dem Prinzip der *Binären Serialisierung*. Als erstes musste festgelegt werden, welche Daten überhaupt gespeichert werden sollten. Dazu wurde eine Klasse namens *Playerstatistics* erstellt, welche alle relevanten Daten halten kann. Daraufhin musste ein Script für die Speicher- und Lade-Funktion geschrieben werden. Das Objekt, auf welchem dieses Script geladen ist, wird beim Wechseln von Scenes nicht zerstört. Dies bewirkt, dass die Funktionen immer aufgerufen werden können und die Daten beim Szenenwechsel nicht verloren gehen. Schlussendlich wurde das Script *AllPlayerVariables* erstellt. Dieses sammelt alle Daten wie die Lebenspunkte des Spielers, die ID der Scene, etc. und gibt sie weiter an die Speicherfunktion. Diese serialisiert dann die Daten und speichert sie ab.

3.3. Überarbeiten und Verbessern des Spiels

Januar 2018 wurden wir in den Nationalen Wettbewerb von Schweizer Jugend Forscht aufgenommen. Deshalb entschlossen wir uns, die Arbeit am Spiel wieder aufzunehmen und einige Kernaspekte zu verbessern. In der Zwischenzeit hatten wir beide mehr Programmier-Erfahrung gesammelt und konnten dieses neue Wissen gut anwenden.

3.3.1. Tränke-Inventar

Die Funktion des Tränke-Inventars liessen wir unverändert, doch der Code wurde komplett neu geschrieben. Das Script basiert jetzt stärker auf Funktionen als zuvor und funktioniert ähnlich wie eine *State-Machine*. Nebst dem Haupt-Inventar überarbeiteten wir die Kisten, welche in den Welten verteilt sind und in denen man neue Tränke finden kann. Das GUI wurde verbessert und die assoziierten Scripts neu geschrieben. Die meisten Spieler werden diese Änderungen nicht bemerken, doch der Programmcode ist viel besser geschrieben und weniger Fehleranfällig.

3.3.2. Spielfigur und Kampfsystem

Das Kampfsystem war der Aspekt des ursprünglichen Spiels, an welchem Testspieler am meisten auszusetzen hatten. Wir waren ebenfalls nicht sonderlich zufrieden damit und entschlossen uns, es komplett zu überarbeiten.

Während dem Überarbeitungsprozess versuchten wir, das ganze System zu vereinfachen. Wir überlegten uns, welche Features wir beibehalten, welche wir entfernen und welche wir nur leicht abändern wollten. Danach schrieben wir ein komplett neues Script und arbeiteten unsere Ideen ein. Die von uns vorgenommenen Veränderungen können im Abschnitt 4.3. nachgelesen werden.

3.3.3. Spielwelten

Bei der Verbesserung der Spielwelten versuchten wir möglichst viel Feedback von unseren Spielern einzuarbeiten.

Eines der Anliegen unserer Testspieler war eine bessere Leitung des Spielers. Viele meldeten uns zurück, dass Sie oft nicht wussten, was genau sie in einer Welt tun und wo sie hingehen sollten. Deshalb fügten wir kleine aber hilfreiche Details zu den einzelnen Welten hinzu, um den Spielern zu helfen.

In der Swamp-Map zum Beispiel muss der Spieler die Leiche eines verstorbenen Ritters finden. Um die Suche zu vereinfachen fügten wir eine Blutspur, welche zur Leiche führt, hinzu und machten eine der Laternen in ihrer Nähe ein wenig heller. In der Ruins-Map, wo der Spieler aus einem Labyrinth herausfinden muss, fügten wir kleine Partikel-Effekte hinzu, die den Ausweg einfacher zu finden machen.

Diese Änderungen sind zwar klein, doch sie tragen massiv zum Spielvergnügen bei. Ein Spiel sollte niemals frustrierend sein und vor allem nicht, wenn es passiert, weil der Spieler nicht weiss wo er hinmuss.

Ausserdem versuchten wir, den Stil innerhalb einer Welt etwas einheitlicher zu machen. Das bedeutete, einige Texturen von Models und Terrain zu überarbeiten.

Nebst den Texturen nahmen wir in manchen Spielwelten kleine Änderungen an den Models vor. So änderten wir zum Beispiel die Bäume im Swamp und verdichteten die Vegetation in der Town-Map. Die grössten Änderungen nahmen wir jedoch an der Ruins-Map vor, da diese unserer Meinung nach die schwächste Welt im Spiel war. Wir verbesserten die Models des Labyrinths und verteilten mehr Steine und Vegetation. Ausserdem gestalteten wir das Plateau interessanter, indem wir es dichter mit Ruinen besiedelten.

Die Hauptänderungen nahmen wir am *Lighting* und *PostProcessing* vor. Seitdem wir Unity zum letzten Mal verwendet hatten wurden einige neue *PostProcessing*-Tools zur Engine hinzugefügt, von denen wir nun Gebrauch machen konnten. Dank diesen vereinfachten Tools konnten wir ohne grossen Aufwand die visuelle Qualität unseres Spiels deutlich steigern.

3.3.4. Gegner

Die Verwendung von RAIN AI war uns bereits im ursprünglichen Projekt ein Dorn im Auge. Das Plug-In war zwar relativ einfach zu verwenden, doch es führte oft zu Problemen mit der Navigation von Gegnern und war insgesamt nicht besonders zuverlässig. Aus diesem Grund entschlossen wir uns, unsere eigene KI unter Verwendung von *ScriptableObjects* zu programmieren. Dazu mussten wir uns vorerst in das Thema einlesen, was dank Unitys Scripting Manual und kostenlosen Video-Tutorien relativ einfach ging. Nichtsdestotrotz war die Implementierung des Systems keineswegs einfach.

Die neue KI basiert auf *States*, welche in *ScriptableObjects* gespeichert sind. Die *States* werden von einem Script namens *BasicEnemyStateController* kontrolliert. Ein *State* besteht aus *Actions* und *Transitions*.

Actions sind ebenfalls ScriptableObjects. Sie enthalten die Funktion *Act*, welche jedes Frame vom *BasicEnemyStateController* ausgeführt wird. *Act* bestimmt, was gerade gemacht wird. Das kann bedeuten, dass der Gegner läuft, eine gewisse Animation abspielt, oder einfach nur dasteht und nichts tut.

Transitions hingegen sind Instanzen einer Klasse, welche jeweils aus dem Boolean *AlwaysTransition*, der Fliesskommazahl *Delay*, einem weiteren ScriptableObject namens *Decision* und einem *True*- und einem *False*-State bestehen. Die Transitions werden ebenfalls vom *BasicEnemyStateController* ausgeführt. Dabei geht dieser wie folgt vor. Ist *AlwaysTransition* nicht wahr, so wird die Funktion *Decide* des Decision-*ScriptableObjects* ausgeführt. Diese liefert entweder wahr oder falsch, je nach Code und Umständen. So liefert zum Beispiel die *Decision PlayerCloseDecision* wahr, wenn der Spieler sich innerhalb eines gewissen Radius um den Gegner befindet, und falsch, wenn er dies nicht tut. Ist *AlwaysTransition* wahr, so wird dieser Schritt übersprungen und die Entscheidung wird behandelt als hätte *Decision* wahr herausgegeben.

Nachdem die Entscheidung gefällt wurde, geht nach der Anzahl von *Delay* angegebenen Sekunden der *BasicEnemyStateController* in den neuen Zustand über.

Dies ist natürlich eine Übersimplifikation des neuen Systems. Wer sich mehr für die Funktion der KI interessiert kann den Programmcode im Anhang lesen.
Wir konnten mit dem neuen System drei verschiedene Verhaltensweisen erstellen. Eine für patrouillierende Gegner, eine für stationäre Gegner und schlussendlich noch eine für Bosse.
Wir basierten diese Verhaltensweisen auf den BehaviourTrees, welche wir bereits für RAIN erstellt hatten. Grundlegend verhalten sich die also Gegner gleich wie vorher, doch die Navigation funktioniert um einiges besser.
Die KI war die einzige Änderung, die wir an den Gegnern vornahmen. Die Animationen und Models beliessen wir gleich, wir entfernten keine Gegner und fügten auch keine hinzu.

3.3.5. Story

Das alte Cutscene-System war zwar ausreichend, um unsere Geschichte zu erzählen, doch es war uns nicht dynamisch genug. Wir hatten, seitdem wir die Bilder machten, viele kleine Details an den Welten geändert, was in den Standbildern stark bemerkbar gewesen wäre. Zu unserem Glück gab es in einem Update für Unity ein neues Tool namens *Timeline*. Dieses erlaubt es, ganz einfach Objekte innerhalb der Engine zu animieren und positionieren. Ausserdem kann man für Timeline gut eigene *Scripts* schreiben, welche den Prozess einfacher machen. So schrieben wir zum Beispiel ein System, welches dynamische Untertitel ermöglicht.
Wir mussten alle Cutscenes komplett neu implementieren, was enorm viel Zeitaufwand benötigte. Dabei wichen wir ein wenig vom Drehbuch ab und entfernten aus technischen Gründen die Szenen, welche jeweils nach einem Bosskampf spielten. Dies taten wir, da die Position von Objekten nach einem Kampf zu unberechenbar ist und das vermehrt zu Problemen führte.

3.3.6. Questsystem

Mit der Funktionsweise des alten Questsystems waren wir grösstenteils zufrieden, doch fanden wir den Programmcode ineffizient und versuchten das System zu vereinfachen.
Zuerst wurden die drei Quest-Arten in eine Klasse namens *StandardQuest* zusammengefasst. Diese enthält sowohl alle relevanten Informationen zu einem Quest als auch die Funktionen welche benötigt werden um den Quest zu aktivieren, updaten und deaktivieren. Ob es ein Kill-, ein Location-Quest oder eine Cutscene ist wird durch

eine Variable in der Klasse angegeben. Dies ist eine viel effizientere Lösung als jeweils verschiedene Klassen für verschiedene Arten von Quests zu benutzen.
Ausserdem wollten wir, dass man dasselbe Script in allen Welten benutzen kann. Zuvor hatten wir nämlich für jede Spielwelt ein separates Script geschrieben, was das ganze System um einiges komplizierter machte. Um das System zu vereinfachen wurden zwei Scripts geschrieben, eines für die Town-Welt und eins für die anderen Spielwelten. Dies musste getan werden, da in der Stadt mehrere Quests möglich sind und somit das System komplizierter ist als in den anderen Welten.
Die Quests sind in einer Liste gespeichert, welche im Inspektor anpassbar ist. Dies macht es einfacher, die Quests in anderen Scripts aufzurufen. Somit konnten wir weitere kleine Anpassungen anbringen um das Spiel zu verbessern. Zum Beispiel konnten die Location-Quests so anpassen, dass sie nur aktiv sind, wenn sie auch benötigt werden. Das führt zu weniger Verwirrung da Spieler nun nicht mehr ein Dialogobjekt finden können das erst später verwendet wird.
Nachdem die Scripts geschrieben wurden, mussten wir nur noch die Quests in jeder Welt neu implementieren. Dank des einfachen Systems ging dies jedoch schnell.

3.3.7. Companions

Die drastischen Änderungen am Kampfsystem betrafen auch die Companions und alles was im Abschnitt 4.3. diskutiert wird trifft auch für sie zu.
Nebst diesen Änderungen entschlossen wir uns, das System des Spielerwechsels Grundlegend zu verändern. Die Anwesenheit von mehreren Spielfiguren in den Welten führte bei manchen Spielern zu Verwirrung und trug nicht sonderlich zum Spielvergnügen bei. Deshalb entschlossen wir uns den KI-gesteuerten Companion, den man bisher in die Welten mitnehmen konnte, zu entfernen. Stattdessen sollte Neith nun «Die Form der Götter annehmen» können. Deshalb implementierten wir ein System, mit welchem man jederzeit zwischen den verschiedenen Göttern, welche man bereits freigeschalten hat, wechseln kann. Dabei wird einfach Neith mit dem erwünschten Gott ausgewechselt.
Dies erleichterte ausserdem unsere Arbeit, da wir nun nicht auch noch für die Companions eine neue KI schreiben mussten.

4. Produktbeschreibung, Reflexion und Diskussion

Nach mehreren Jahren Arbeit ist *Bane of Asphodel* nun fertiggestellt. Von den vielen Zielen, welche wir uns gesetzt hatten, haben wir die meisten erreicht. *Bane of Asphodel* ist ein RPG mit ein bis zwei Stunden Spielzeit. Der Spieler kontrolliert einen Charakter namens Neith, dessen Ziel es ist, den *Keeper of the Cycle* aufzufinden und zu besiegen. Wir werden nun die einzelnen Elemente des Spiels beschreiben und bewerten.

4.1. Story

Das Spiel startet mit Neith, der von einem fallenden Klavier getötet wird. Als er aufwacht, befindet er sich in Asphodel. Neben einem Tor steht ein Wachmann. Dieser beauftragt Neith damit, alle Monster in der Nähe zu töten. Im Verlauf der Geschichte befreit Neith vier Götter und besiegt den *Keeper of the Cycle*, einen unbekannten Bösewicht. Am Schluss muss Neith sich entscheiden, ob er dessen Platz einnehmen will oder nicht. Das Drehbuch in *Anhang I* diente als Grundlage für die Cutscenes im Spiel, doch änderten wir diese in der Überarbeitungsphase ein wenig ab.

Die Story ist nur ein kleiner Teil unseres Spiels. Sie dient dazu, den Spieler voranzutreiben und benötigt deshalb keine hohe Komplexität. Erzählt wird sie durch eine Abfolge von halb-animierten Standbildern mit Untertiteln, welche manchmal noch von Sound-Schnipseln unterstützt werden. Am Schluss des Spiels muss der Spieler einen Entscheid treffen. In den meisten modernen RPGs ist das Ende des Spiels von mehreren Entscheiden abhängig, welche man im Verlauf des Spiels getroffen hat. In unserem Spiel ist das nur einer, doch dieser hat schwerwiegende Konsequenzen. Wir haben lange überlegt, wie das Ende ausfallen sollte, und sind mit der endgültigen Version zufrieden. Die Geschichte ist abgeschlossen, lässt jedoch Raum für Interpretation.

Die Überarbeitung der Cutscenes hat sich enorm gelohnt. Das Spiel wirkt auf einen Schlag viel professioneller. Natürlich wären vollständig animierte Cutscenes um einiges besser, doch das wäre uns auf unserem derzeitigen Wissensstand schlichtweg nicht möglich.

4.2. Spielwelt

Die Spielwelt ist in fünf Gebiete aufgeteilt. Da das Spiel ein *Open-World-Game* ist, gibt es nicht lineare Level wie in *Super Mario.* Nachdem die Aufgaben in der Fields-Map erfüllt wurden, kann der Spieler jederzeit jede Spielwelt besuchen. Die Idee war, dass Asphodel eine Spiegelung der echten Welt ist, jedoch zerbrochen und irgendwie unnatürlich. Aus diesem Grund ist jede Welt komplett verschieden. Jede Welt widerspiegelt einen bestimmten Aspekt von Asphodel.

4.2.1. Fields

Das erste Gebiet ist eine idyllische Feld-Landschaft mit Gras, Bäumen und Kornfeldern. Die einzigen Gebäude sind zwei Windmühlen, ein Leuchtturm und das Tor zur Stadt. Die Fields-Map sollte den Aspekt der unberührten Schönheit darstellen. Sie zeigt, wie die Welt von Asphodel sein sollte, oder vielleicht sogar, wie Asphodel vor dem Beginn der Zyklen einst war. Der Spieler merkt jedoch schnell, dass der Schein trügt, und dass längst nicht alles in Asphodel perfekt ist.

Abbildung 7: Die Fields-Map

4.2.2. Town

Sobald der erste Boss-Gegner besiegt ist, darf der Spieler die Stadt betreten. Die Stadt hat als einzige Map kein richtiges Thema. Sie ist eine Art Nexus, an den der Spieler immer wieder zurückkehren muss. Links neben dem Tor, aus dem der Spieler tritt, ist ein kleiner Marktplatz. Auf der Erhöhung in der Mitte der Stadt steht eine Kathedrale.

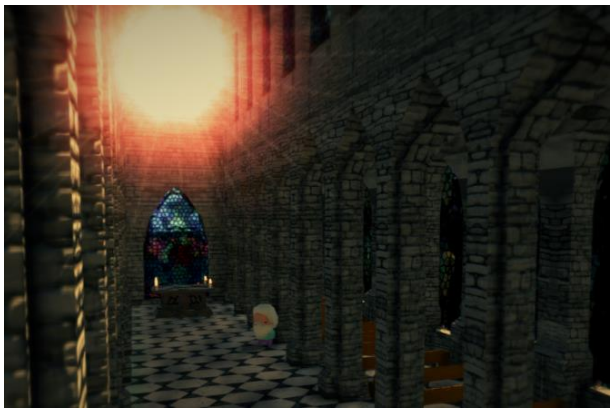

Abbildung 8: Die Kathedrale von Innen

Die Kathedrale ist das grösste und prächtigste Modell des ganzen Spiels. Sie wurde dem Baustil der gotischen Kathedralen angenähert. In ihr befinden sich mehrere Reihen Kirchenbänke und ganz hinten ein Altar mit vier Kerzen. Hier muss man die Götter von Asphodel befreien.

Die Stadt hat einen mittelalterlichen Charakter. Die zahlreichen Häuser sind alle im Fachwerk-Stil aufgebaut. Auf den Gassen laufen die Stadtbewohner ziellos umher. Für die Stadtbewohner werden beim Laden der Map eine Textur und eine Kopfbedeckung zufällig ausgewählt. So kann es vorkommen, dass man auf lauter verschiedene oder auch auf lauter genau gleiche Bewohner trifft. Hinter der Kathedrale öffnet sich der Blick auf das weite Meer. Dort gibt es auch eine Gruppe Zombies, die bei einer Boom-Box zu *Thriller* von Michael Jackson tanzen. Sollten wir das Spiel jemals veröffentlichen, müssten wir dieses Lied herausnehmen, da wir die Rechte zu *Thriller* nicht gekauft haben. Es gibt vier Ausgänge aus der Stadt, einen in jede Welt. Sobald man den ersten Gott befreit hat, kann man selbst wählen, in welche Welt man als nächstes gehen möchte.

Abbildung 9: Die Stadt

4.2.3. Forest

Eine dieser Welten ist eine dunkle Waldlandschaft. Diese Map sollte das Mysteriöse und Unbekannte an Asphodel symbolisieren. Die Landschaft ist, im Gegensatz zur Fields-Map, durch viele steile Hänge und Abgründe geprägt. In dieser Map sind Lichteffekte besonders effektiv. Es gibt zum Beispiel drei grosse Boxen in der Map neben denen jeweils zwei leuchtende Disco-kugeln stehen. Dieses Zusammenspiel der Dunkelheit und der Lichteffekte führt zu einer magischen Atmosphäre.

Abbildung 10: Die Forest-Map

Es gibt in der Forest-Map nur wenige Gebäude. Nebst ein paar wenigen Brücken gibt es einige Laternen, zwei kleine Häuschen und die bereits erwähnten *Bases*, drei grosse Boxen mit jeweils zwei Discokugeln. Es gibt einen freundlichen Charakter in dieser Map, den *Mysteriösen Unbekannten*, der dem Spieler in der Story weiterhilft.

4.2.4. Swamp

Abbildung 11: Die Swamp-Map

Eine weitere Map ist die Swamp-Map. Diese ist ein dicht bewachsener, nebliger Sumpf. Sie sollte den unheimlichen, morbiden Aspekt von Asphodel verkörpern. Sollte der Spieler vom Weg abkommen, kann er sich aufgrund des dichten Nebels leicht verirren. Der Sumpf selbst ist giftig und zieht dem Spieler kontinuierlich Lebenspunkte ab, wenn er hineingerät. Man kann jedoch den ganzen Quest dieser Map durchspielen, ohne auch nur einmal vom Weg abzuweichen. Es gibt neben mehreren Brücken auch eine Höhle und ein kleines Haus, das von einem Zaun umgeben wird. Im Sumpf sind mehrere Leichen verteilt.

Diese Map schafft es, ein Gefühl der Unsicherheit und der Furcht zu vermitteln. Man sieht die Gegner immer erst, wenn man direkt in sie hineinläuft und muss deshalb ständig auf der Hut sein. Die Tatsache, dass das Wasser giftig ist, trägt ebenfalls zu diesem Gefühl bei.

4.2.5. Ruins

Abbildung 12: Die Ruins-Map

Zuletzt gibt es noch die Ruinen. Durch sie sollte der Aspekt der Zerstörung und Verwüstung in Asphodel gezeigt werden. Ausser einem einsamen Händler lebt hier nichts mehr, und selbst die Gegner sind aus Stein. Der Spieler betritt diese Welt ebenfalls durch ein Portal. Er befindet sich nun in einem Labyrinth, aus dem er erst einmal herausfinden muss. Im Labyrinth lauern mehrere Gegner, doch es gibt auch Kisten mit wertvollen Heiltränken zu finden. Hat man den Ausweg gefunden, kann man über eine Rampe auf eine höher gelegene Ebene gelangen. Hier gibt es eine heruntergekommene Burgruine, in der sich der besagte Händler befindet. Neben der Burgruine steht ein Tempel in dem man den Boss-Gegner der Ruinen beschwören und besiegen muss.

4.2.6. Reflexion

Alle Welten sind uns gut gelungen. Wir haben viel Arbeit in das Design der einzelnen Welten gesteckt. Das hat sich auch gelohnt. Jede Welt hat eine einzigartige Atmosphäre und es macht Spass, sie zu erkunden. Trotzdem gibt es viele Verbesserungsmöglichkeiten. Im Vergleich zu anderen Spielen sind unsere Maps klein. Es wäre interessant, ein grösseres Spielfeld zur Verfügung zu haben, doch das würde einen höheren Aufwand beim Design der Welt bedeuten. Die Spielwelt wäre interessanter, wenn nicht jede Map nur eine Art von Umgebung hätte, sondern zum Beispiel eine Mischung aus der Fields-Map und der Forest-Map. Die Reduktion auf einfache Maps war ein bewusster Entscheid, den wir am Anfang des Entwicklungsprozesses gefällt haben.

Die Überarbeitung der Welten hat sich gelohnt. Spieler werden nun besser durch die einzelnen Maps geleitet, der Stil der Welten ist einheitlicher und die gesamte Qualität der Grafiken hat sich enorm gesteigert.

Man könnte natürlich noch immer vieles verbessern, doch wir sind mit dem aktuellen Stand zufrieden.

4.3. Gameplay

4.3.1. Gameplay vor dem Überarbeiten

Für die meisten Spieler ist Gameplay immer noch der wichtigste Teil eines Spiels. Ein Computerspiel kann noch so schön aussehen, aber wenn sich das Gameplay nicht gut anfühlt, wird es nicht gern gespielt. Zu Gameplay gehören die Steuerung, das Kampfsystem und andere Spielmechaniken. Die Steuerung unseres Spiels ist simpel. Man kann es mit Gamepad oder mit Maus und Tastatur spielen.

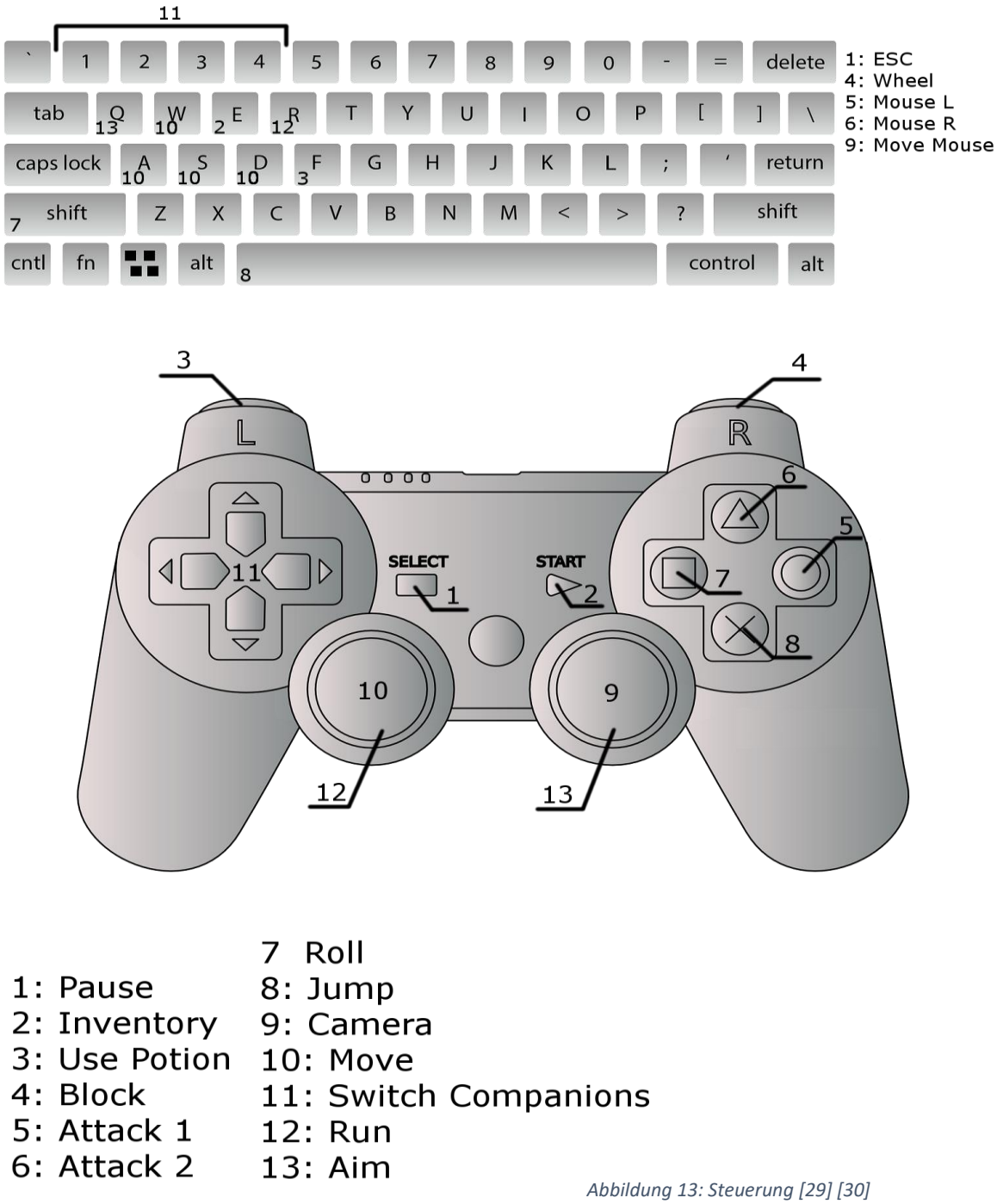

Abbildung 13: Steuerung [29] [30]

Die Steuerung der Kamera mit der Maus, respektive dem R-Stick und die Steuerung des Spielers mit WASD oder dem L-Stick ist in allen modernen Computerspielen so implementiert. Bei der Steuerung mit Maus und Tastatur achteten wir darauf, dass die Tasten für verschiedene Aktionen nahe beieinanderlagen. Somit kann der Spieler gemütlich mit einer Hand auf der Maus bleiben und muss die andere Hand nicht allzu sehr bewegen. Dies mag vielleicht banal erscheinen, doch es gibt viele moderne Spiele, die genau dieses Problem haben. Wenn die Tasten zu weit auseinanderliegen, kann das zu Unterbrechungen im Gameplay-Fluss führen und ist bei Spielern nicht beliebt. Beim Gamepad ist dies jedoch kein Problem. Wir benutzten alle Input-Möglichkeiten des Gamepads, mit Ausnahme der beiden Trigger ganz hinten. Diese werden nämlich von Unity 5 als eine Input-Achse interpretiert und nicht als Knöpfe. Dies erschwert den Umgang mit den Triggern gewaltig, weshalb wir uns schlussendlich dazu entschieden, sie nicht zu benutzen.

Eng mit der Steuerung verbunden ist das Kampfsystem. In unserem Spiel gibt es für jeden spielbaren Charakter zwei Angriffe. Mit den zwei Angriffstasten kann man jeweils einen dieser Angriffe ausführen. Um Kampfsituationen einfacher zu machen, richtet sich ein Angriff immer nach dem nächsten Gegner. Der Spieler muss also nur grob zielen können und die Feinjustierung macht das Spiel selbst. Zusätzlich kann man durch Drücken der Taste Q oder des R-Sticks die Kamera permanent auf einen Gegner richten. So muss man sich nicht gleichzeitig auf die Steuerung der Kamera und die Steuerung des Spielers konzentrieren, sondern kann seine volle Aufmerksamkeit dem Kampf widmen. Manche Charaktere haben sogenannte Combo-Angriffe. Drückt man die Angriffstaste vielmal nacheinander, führt die Spielfigur eine Art Kettenangriff aus, greift also den Gegner mehrmals hintereinander an.

Neith und die vier Companions haben unterschiedliche Attacken und spielen sich somit verschieden. Jeder der Companions hat eine spezielle Eigenschaft. Odin fügt umso mehr Schaden zu, je länger er angreift. Set fügt Gegnern von Anfang an viel Schaden zu. Hermes ist schnell, vor allem sein Sprint, doch seine Attacken fügen Gegnern wenig Schaden zu. Zuletzt ist Zao Shen, der Panda. Er ist langsam, ist jedoch resistent gegenüber Schaden. Vor allem im späteren Verlauf des Spiels ist es wichtig, seine Companions gut einzusetzen.

Eine weitere Gameplay-Mechanik, welche vor allem am Anfang des Spiels nützlich sein kann, ist Blocken. Neith hat ein Schild, mit welchem er Attacken der Gegner abwehren kann, ohne Schaden zu erleiden. Wenn man einen Angriff abwehrt, muss

man eine kurze Zeit warten bis man wieder Abwehren oder Angreifen kann. Dieses Zeitfenster gibt dem Gegner eine Angriffschance und fügt dem Blocken ein taktisches Element hinzu. Der Spieler kann ausserdem noch in die Luft hüpfen und eine Hechtrolle zum Ausweichen benutzen.

Zu guter Letzt gibt es noch Heiltränke. Man startet das Spiel mit ein paar Tränken im Inventar und kann in Kisten, welche in den Welten verteilt sind, noch mehr finden. Es gibt insgesamt zehn verschiedene Tränke im Spiel. Jeder dieser Tränke füllt die Lebenspunkte des Spielers wieder auf, jedoch unterschiedlich schnell und effektiv. Tränke sind essenziell in Kampfsituationen, da man schnell Lebenspunkte verlieren kann und sich heilen muss. Es war schwierig, die Anzahl Tränke am Anfang des Spiels richtig zu wählen. Zu viele Tränke würden das Spiel zu einfach gestalten und umgekehrt.

4.3.2. Gameplay nach dem Überarbeiten

Unsere Testspieler beschrieben die Kämpfe unseres Spiels oft als träge und beschwerten sich aufgrund von wenig taktiler Rückmeldung an den Spieler. Die Kämpfe fühlten sich oft langsam an, nicht zuletzt dank der grossen Zeitverzögerung zwischen Tastendruck und der gewünschten Aktion im Spiel.

Während dem Überarbeitungsprozess versuchten wir, das ganze System zu vereinfachen. Wir entfernten das Blocken und die Kombo-Angriffe, ohne etwas am Control-Layout zu verändern. Jede Spielfigur hat nun lediglich eine Attacke, welche kurz und präzise ist. Der Spieler kann innerhalb von einem Bruchteil einer Sekunde zwischen laufen, rollen und zuschlagen wechseln. Dadurch spielen sich die Kämpfe deutlich schneller. Spieler können jetzt präziser auf die Aktionen der Gegner reagieren.

Um das Gameplay wieder auszugleichen, verstärkten wir die Bestrafung für einen Fehler des Spielers. Den Spielfiguren ist es nicht mehr möglich die Aktionen seiner Gegner zu unterbrechen indem man auf sie einschlägt. So wird verhindert, dass der Gegner durch einen aggressiven Spieler lahmgelegt wird und nicht mehr angreifen kann. Diesen Zustand nennt man *Stun-Lock* oder *staggering*. Im Gegensatz dazu können Gegner nun die Animationen des Spielers unterbrechen. Dadurch werden die Angriffe der Gegner gefährlicher. Dies bringt den Spieler dazu, mehr über seine Handlungen nachzudenken.

Durch oben erwähnte Änderungen wird die Ausweichrolle wichtiger. Der Spieler kann auf Knopfdruck eine Ausweichrolle ausführen, welche ihn für kurze Zeit unverwundbar

macht. Die Rolle ermöglicht es dem Spieler in kurzer Zeit eine gewisse Distanz zwischen sich und dem Gegner zu bringen.
Um das Feedback klarer zu gestalten wurden einige visuelle Effekte eingebaut. Getroffene Spieler und Gegner verfärben sich rot, wenn sie von einem Angriff getroffen werden. Wird der Spieler getroffen, so wackelt die Kamera ein wenig und die Spielfigur fällt für kurze Zeit um.
Durch diese Änderungen konnten wir das Gameplay von *Bane of Asphodel* um einiges verbessern. Wir haben gelernt, dass bei Games einfache Systeme oft interessanter sind als komplizierte, da der Spieler sie einfacher lernen und meistern kann.

4.3.3. Gameplay Reflexion

Im Vergleich zu anderen Spielen ist unser Gameplay relativ oberflächlich. Die einzige Art, wie wir ein bisschen Abwechslung ins Gameplay hineinbringen, ist durch die Companions. Trotzdem sind auch sie nicht allzu tiefgründig. Wenn man zum ersten Mal mit Hermes oder Odin spielt, ist das aufregend, doch bald hat man seine Lieblingsfigur und benutzt die anderen nicht mehr.
Man könnte das Gameplay auf viele Arten verbessern. Fast alle Spiele haben heutzutage ein komplexes Inventarsystem mit verschiedenen Waffen, Rüstungen und anderen Items. Am Anfang des Entwicklungsprozesses wurde sogar an einem solchen System gearbeitet. Aufgrund von Mangel an Zeit und technischem Knowhow wurde dieser Aspekt des Spiels verworfen. Das Tranksystem, welches im Spiel vorzufinden ist, ist ebenfalls simpel. Obwohl die Tränke verschieden effektiv sind, wirken sie alle gleich. Man könnte dieses System verbessern, indem man verschiedene Tränke mit unterschiedlichen Effekten ins Spiel integriert. So könnte zum Beispiel ein Trank den Spieler schneller machen oder seinen Angriffen mehr Kraft verleihen.
Das Gameplay in Bane of Asphodel mag zwar einige Mängel haben, ist jedoch im Grossen und Ganzen ein solides Zusammenspiel mehrerer einfacher Systeme und erfüllt seinen Zweck. Für ein Spiel, das von einem so kleinen Team entwickelt wurde, ist es sogar recht gut. Es gibt viele Indie-Games auf dem Markt, welche ein schlechteres Gameplay vorweisen.

4.4. Gegner

Abbildung 14: Gegner

In *Bane of Asphodel* gibt es acht normale Gegner und fünf Boss-Gegner. In einer Welt gibt es mit Ausnahme der Town immer zwei normale und einen Boss-Gegner. Meistens muss der Spieler als Teil eines Quests eine gewisse Anzahl normaler Gegner besiegen, bis der Boss-Gegner erscheint.

Unser Ziel war es, die Gegner so unterschiedlich wie möglich zu gestalten. Wie man im obigen Bild erkennen kann, sehen alle Gegner verschieden aus. Wie schon erwähnt, hat jede Welt ein eigenes Thema. Die Gegner sind immer diesem Thema angepasst. So sind zum Beispiel die Gegner in der Fields-Map normale Tiere, denn die Welt der Fields ist unberührt und unschuldig.

Wie bereits im Teil *Vorgehen und Methode* erwähnt, basiert die KI aller normalen Gegner auf demselben BehaviourTree. Dies hat zur Folge, dass sich alle Gegner ähnlich verhalten. Sobald der Spieler in den Wahrnehmungsbereich eines Gegners eintritt, fängt dieser an, sich auf ihn zuzubewegen. Ist er in Reichweite, führt er eine von drei Attacken aus. Diese Attacken sind bei jedem Gegner verschieden. Die Hauptunterschiede bestehen in der Reichweite und der Geschwindigkeit des Angriffs.

Die Kulmination jedes Quests ist ein Kampf mit einem Bossgegner. Diese Bossgegner haben mehr Lebenspunkte, mehr Angriffe und einen komplexeren BehaviourTree als normale Gegner. Jeder Boss-Gegner wird mit einem bestimmten Musik-Stil unterstützt

und dieser Stil wird in deren Animationen widergespiegelt. So tanzt zum Beispiel der Boss der Swamp-Map den irischen River Dance und der Boss der Ruinen den Tango. Die normalen Gegner in *Bane of Asphodel* sind zwar nicht spektakulär, doch sie erfüllen ihren Zweck. Man hat schnell gelernt wie sie sich verhalten, und dann ist es nicht mehr wirklich schwierig sie zu besiegen. Die wahre Attraktion des Spiels sind die Boss-Gegner. Jeder Boss-Gegner ist wirklich einzigartig und interessant. Ausserdem sind sie schwieriger als normale Gegner und somit eine richtige Herausforderung, sogar für geübte Spieler

Es gibt viele Arten, wie wir Gegner verbessern könnten. Das Hauptproblem aller Gegner ist die KI. Selbst nachdem wir das System überarbeitet haben, bietet die von uns programmierte KI nicht genug verschiedene Verhaltensweisen und das führt dazu, dass normale Gegner eher lästig und Kampf-Situationen repetitiv wirken. Eine grössere Vielfalt an Gegnern wäre ebenfalls gut, doch auch das würde viel mehr Arbeitsstunden benötigen.

4.5. Graphical User Interface (GUI) und Grafik

Das GUI ist schlicht und unauffällig. Die Elemente, die man sieht, wenn man in der normalen Spielwelt ist, nennt man das *Heads-Up-Display* oder HUD. Das HUD in *Bane of Asphodel* sollte nicht vom Spielen ablenken. Man sieht eine Lebensleiste oben links, eine kurze Beschreibung des aktiven Quests und den ausgewählten Trank darunter. Unten links sieht man die Companions, die aktiv sind. Im Inventar kann man zwischen einem Trank- und einem Questmenü auswählen. Im Pausenmenü kann man speichern und das Spiel schliessen. Ausserdem sind dort auch die Grafik-Optionen. Wie bereits erwähnt, benutzten wir in *Bane of Asphodel* einen Low-Poly-Artstyle. Wir haben alle Texturen selbst gemacht, sei es in Paint.net, Gimp oder Photoshop. Normal-Maps nahmen wir entweder von Asset-Packs oder stellten sie selbst in Blender her. Wir versuchten zwar, unseren Stil einander anzupassen, doch vor allem am Map-Design merkt man, dass das Spiel von zwei Personen entworfen wurde.

Die Kamera wurde mit mehreren Scripts angereichert, die die Bildqualität verbessern. Dazu gehören unter anderem die Effekte, die bereits im Teil 1.8. erklärt wurden.

Die Grafiken unseres Spiels sind nichts Aussergewöhnliches, doch sie genügen unseren Ansprüchen. Mit mehr Zeit und Ressourcen hätten wir sicherlich eine höhere Qualität erreichen können, doch das war nicht das Ziel der Arbeit. Wir sind zufrieden mit dem Stil der Modelle und mit der grafischen Präsentation.

4.6. Reflexion zur Überarbeitungsphase

Wir haben gelernt, dass ein Computerspiel nie wahrhaftig „fertig“ sein kann. Man muss diese Tatsache als Entwickler akzeptieren und irgendwann mit dem Projekt aufhören. Trotzdem sind wir froh die Gelegenheit gehabt zu haben, unser Spiel zu überarbeiten. Die Auflagen von Schweizer Jugend Forscht gaben uns neue Motivation, die weniger guten Aspekte des Spiels zu überarbeiten. Wir verbesserten den Stil und überarbeiteten das Gameplay des Spiels. Genauere Angaben zu den vorgenommenen Änderungen können in *Abschnitt 3.3.* gelesen werden.

Beim Überarbeiten von *Bane of Asphodel* versuchten wir, möglichst viel Feedback von Spielern miteinzubeziehen. Als wir die Arbeit erstmals abgaben, konnten wir nur wenige Testspieler finden und mussten deswegen das Spiel vor allem selbst testen. Das führte dazu, dass wir nur sehr einseitige Rückmeldungen, nämlich unsere eigenen, erhielten. Dieses Mal vergaben wir Testversionen an mehrere Freunde und deren Feedback half uns, die Spielerfahrung interessanter zu gestalten.

Insgesamt finden wir, dass wir das Spiel drastisch verbessert haben. Die Graphiken sehen besser aus, das Gameplay macht mehr Spass und die Cutscenes wirken viel professioneller. *Bane of Asphodel* ist keineswegs ein Meisterwerk, doch wir sind sehr zufrieden mit der Qualität, die wir erreichen konnten.

4.7. Probleme

Sowohl in Indie- als auch in AAA-Spielen stossen Entwickler ständig auf Probleme. Es passiert sogar manchmal, dass ein neues Spiel noch viele Programmierfehler (Bugs) enthält, wenn es veröffentlicht wird. Um das zu verhindern engagieren Entwickler Spieltester. Diese müssen das Spiel spielen und von allen Bugs berichten, denen sie begegnen. Wir gaben ebenfalls eine Testversion an Spieler aus und konnten so viele Bugs finden.

Während des Entwicklungsprozesses gab es immer wieder kleinere Probleme. Schlussendlich konnten wir nur zwei Probleme nicht zufriedenstellend lösen.

Gegen Ende des Entwicklungsprozesses merkten wir, dass in der Town-Map vermehrt Probleme mit der Performance auftraten. Es schien, als ob die vielen Schatten der Häuser, Stadtbewohner und dem Spieler direkt mit der tiefen Framerate verbunden waren. Um die Framerate zu verbessern mussten wir die Anzahl der *Materials* in der Scene verringern, also dasselbe *Material* für mehrere Modelle verwenden. Ausserdem mussten wir die Qualität der Schatten und der Belichtung heruntersetzten.

Die Lösung zu diesem Problem kam mit einem neuen Update von Unity. Dank neuen Lighting- und PostProcessing-Tools konnten wir die Performance des Spiels drastisch steigern. Obwohl die Framerate etwas höher ist, ist die Town-Map noch immer am wenigsten gut optimiert.

Das zweite Problem betrifft den Input-Manager von Unity 5. Beim Schreiben des Companion-Scripts fiel uns auf, dass der Input mit dem D-Pad nicht immer funktionierte, je nachdem welchen Controller wir verwendeten. Durch das Forum von Unity 5 fanden wir heraus, dass die Bezeichnung für die DPad-Achsen bei einem Xbox 360 Controller eine andere ist, als die bei einem Xbox One Controller. Leider ist es mit dem Input Manager von Unity 5 unmöglich, zwischen den beiden Controllern zu unterscheiden. Das Problem liesse sich nur lösen, wenn wir einen eigenen Input Manager schreiben würden. Das wäre für uns jedoch zu zeitaufwändig gewesen. Um dieses Problem zu umgehen, mussten wir im Hauptmenü des Spiels eine Option einbauen, mit der man das Inputgerät auswählen kann. Diese Lösung ist zwar nicht benutzerfreundlich, doch sie funktioniert.

Trotz dieser Probleme haben wir es geschafft, ein zufriedenstellendes Spiel zu entwickeln. Wir konnten viele unserer Ideen umsetzten und haben positive Rückmeldungen von Testspielern erhalten. Es gab auch Ideen, die wir nur teilweise entwickelten und dann verwarfen, sei es aus Zeitgründen oder mangelnder Notwendigkeit. Ursprünglich hatten wir zusätzlich zum Trank-Inventar ein Inventar mit Rüstungen und Waffen geplant. Diese Idee wurde sogar ziemlich weit entwickelt, doch wir erkannten, dass der grosse Zeitaufwand nicht zur Bereicherung des Spiels beitragen würde.

4.8. Weiterführende Reflexion

Zu Beginn dieser Arbeit stellten wir uns zwei Fragen, die wir mit unserem Projekt zu beantworten versuchten. Als erstes fragten wir uns, wie moderne Spiele aufgebaut sind. Als zweites wollten wir wissen, wie der Entwicklungsprozess eines Spiels abläuft.

Wie sind nun moderne Spiele aufgebaut? Moderne Videospiele basieren fast immer auf einer Engine. Jede dieser Engines ist verschieden aufgebaut, doch sie alle verfügen über die Grundfunktionen, welche bereits in Teil 1.3. erläutert wurden. In Unity 5 besteht ein Spiel aus mehreren Scenes, welche wiederum aus vielen Objekten und Scripts besteht. Im Grunde genommen ist also jedes Spiel aus Polygonen und Programmcode aufgebaut, mit oder ohne einer Engine. Wir wissen nun, dass wir, wenn

auch mithilfe vieler externen Tools und einem hohen Zeitaufwand, ein Spiel erstellen können. Es ist also durchaus möglich, im Verlauf eines Jahres zu zweit ein kleines Spiel zu entwickeln. *Bane of Asphodel* kann sich natürlich nicht mit den grösseren Spielen auf dem Markt messen, doch hinter diesen stecken oft hundertköpfige Teams und Millionen von Dollars Produktionskosten.

Worauf muss man nun achten, wenn man ein Spiel entwickelt? Da jedes Genre komplett verschieden ist, behandeln wir in diesem Teil nur den Entwicklungsprozess von RPGs. Die Schwerpunkte sollten definitiv bei Gameplay und grafischer Präsentation gesetzt werden. Als erstes muss man sich für eine Perspektive entscheiden. Ein Spiel mit einer Egoperspektive führt zu mehr Immersion, aber eine *Über-Die-Schulter-Perspektive* gibt dem Spieler mehr Kontrolle. Danach muss man sich für ein Kampfsystem entscheiden. Viele RPGs verwenden Nahkampfwaffen wie Schwerter oder Äxte, doch Spiele wie *Fallout* und *Borderlands* haben gezeigt, dass Schusswaffen genauso geeignet sind. Man kann ein RPG auch mit einem komplett anderen, oder sogar ganz ohne ein Kampfsystem kreieren. Wichtig ist, dass sich das Spielen gut anfühlt.

Im Bereich der grafischen Präsentation kann man mit einem Spiel in zwei Richtungen gehen. Entweder man versucht, möglichst realistisch zu wirken, oder man verwendet einen eigenen Stil. Wenn man sich für Realismus entscheidet, sind die Erwartungen der Spieler an die Grafiken sehr hoch. Man muss also viel Arbeit in die Grafiken stecken und nach ein paar Jahren sieht das Spiel schon «alt» aus. Wenn man einen eigenen Stil verwendet, kann man das Spiel so gestalten wie man will. Viele Indie-Games verwenden ihren eigenen Stil, denn sie haben die Ressourcen nicht, die man für sogenannte *High-End-Graphics* benötigt.

Zuletzt muss man die Spielwelt aufbauen. Bevor man das erste Modell erstellt, muss man sich überlegen, was man mit der Spielwelt aussagen möchte. Die Welt sollte ein stimmiges Thema haben, das überall erkennbar ist. Zum Beispiel wollte *CDProjectRed* mit ihrer Welt in *The Witcher 3 – Wild Hunt* ein Land zeigen, das vom Krieg verwüstet worden ist. Überall sieht man zerstörte Dörfer, obdachlose Menschen und brutale Kriegsverbrechen. Wenn die einzelnen Teile der Spielwelt nicht zusammenpassen, fühlt sich die Welt unnatürlich an. Es ist auch immer gut, wenn man sich eine Hintergrundgeschichte für die Welt ausdenkt, auch wenn diese dann gar nicht im Spiel vorkommt. Dadurch hat man ein Fundament, auf der man die gesamte Spielwelt

aufbauen kann. Sollte man Gegner in der Welt vorfinden, müssen diese ebenfalls passen.

Natürlich ist der Entwicklungsprozess bei grossen Spielen um einiges komplizierter, doch die Grundzüge des Prozesses sind dieselben. Ausserdem darf man auf keinen Fall den Zeitaufwand unterschätzen. Hinter *Bane of Asphodel* stecken mehrere hundert Stunden Arbeitszeit, und hinter Spielen wie *GTA V* und *The Witcher 3 – Wild Hunt* stecken viele Jahre und hunderte Entwickler, die vollzeitlich daran gearbeitet haben. Ein Spiel zu entwickeln ist enorm zeitaufwändig und teuer. *GTA V*, eines der bis jetzt teuersten Computerspiele, kostete insgesamt 265 Millionen Dollar für Produktion und Marketing, und der Entwicklungsprozess dauerte fünf Jahre [31]. Die Ansprüche der Spieler an die Entwickler werden immer grösser, und somit steigen der Zeitaufwand und die Kosten eines neuen Spiels stetig. Trotz der hohen Kosten lohnt es sich, Spiele zu entwickeln. *GTA V* generierte am ersten Tag über 800 Millionen Dollar Umsatz [32]. Ein AAA-Game kostet in der Regel 60 Dollar in den USA oder 70 Franken in der Schweiz. Inzwischen werden die meisten Computerspiele über Online-Plattformen wie *Steam* gekauft. *Steam* verkauft nur digitale Kopien der Spiele und hat somit keine Lagerkosten. Entwickler und Verkäufer verdienen somit mehr mit ihren Spielen und können relativ schnell die Kosten der Entwicklung decken. Doch ist Geldverdienen die einzige Motivation die Spieleentwickler haben, um Spiele zu entwickeln? Neben den grossen AAA-Game-Entwicklern gibt es durchaus auch Entwickler, die das Medium als eine Kunstform benützen.

Nun stellt sich natürlich die Frage: Können Computerspiele überhaupt eine Kunstform sein? Diese Frage wird oft diskutiert und die Meinungen sind gespalten. Ein Problem beim Beantworten dieser Frage ist, dass es keine klare Definition von Kunst gibt. Viele Philosophen haben bereits versucht eine universale Definition zu finden und haben bis jetzt keine gefunden. Wir können die Frage also nicht mittels einer Definition beantworten und müssen einen anderen Weg finden. Was wir jedoch tun können, ist, das Spiel mit anerkannten Kunstformen zu vergleichen. Gemälde sind international als Kunst anerkannt, genauso wie Musik, Theater und Film. Keine dieser Kunstformen startete als solche, und oft brauchte es Zeit bis die Gesellschaft sie akzeptierte. Auf der Website des Smithsonian steht:

«Video games use images, actions, and player participation to tell stories and engage their audiences. In the same way as film, animation, and performance, they can be considered a compelling and influential form of narrative art.» [33]

Dies ist Teil der Beschreibung der Ausstellung *The Art of Video Games* die vom 16. 03. 12. bis 30. 09. 12. im *American Art Museum* in Washington DC ausgestellt wurde. Spiele sind also ein Zusammenspiel von verschiedenen Kunstformen.

Im Artikel *Video games and art: why does the media get it so wrong?* von Keith Stuart meint er, dass Kunst ein Mittel der Kommunikation darstellt. Der Künstler will mit seiner Kunst möglichst viele Menschen erreichen und verwendet deshalb das populärste Medium seiner Zeit, um sich auszudrücken. Deshalb, schreibt Stuart, schrieb Shakespeare seine Dramen für die klassenlosen, aber überfüllten Theaterhäuser Londons und aus demselben Grund publizierte Dickens seine Novellen mittels Zeitschriften. Stuart behauptet, dass Videospiele die neue Sprache sind, mit der sich Künstler ausdrücken können. [34]

Es gibt natürlich auch Menschen, die nicht dieser Meinung sind. Einer dieser Menschen war Roger Ebert (1942 – 2013), ein bekannter Kritiker. Im Jahr 2010 schrieb er einen Artikel namens *Video Games can never be art*, der heute noch von vielen zitiert wird. In diesem Artikel schreibt er unter anderem, dass Computerspiele nicht Kunst sein können, weil sie eine Form von Interaktivität beinhalten. Er behauptet, dass etwas, das man gewinnen kann und das Regeln, Aufgaben und Konsequenzen hat, nicht Kunst sein kann. Kunst sollte seiner Meinung nach etwas sein, das man «erlebt». So kann man einen Film oder ein Buch nicht gewinnen, man hat keinen Einfluss darüber und es bleibt somit die reine Vision des Künstlers. Roger Ebert gab in seinem Artikel zu, dass er in seinem Leben noch nie ein Videospiel gespielt hatte. Das erkennt man an der Art und Weise, wie er Spiele beschreibt. Er misst sie an den Vorurteilen seiner Generation. In seinen Augen ist ein Computerspiel ein einfaches Konstrukt mit Punkten, Regeln und klaren Aufgaben, doch Spiele haben sich weiterentwickelt. Viele Spiele kann man gar nicht gewinnen, sondern man muss sie, wie er so schön gesagt hat, «erleben». Ein exzellentes Beispiel für ein solches Spiel ist *Abzû*. In *Abzû* schwimmt man als ein Taucher durch einen Ozean voller Fische. Man kann nicht kämpfen, es gibt kein definiertes Ziel, das es zu erreichen gilt. Das Spiel erzählt eine Geschichte, doch sie wird dem Spieler nicht aufgedrängt. Wenn der Spieler will, kann er versuchen, sie zu finden. Er kann aber auch einfach im Ozean herumschwimmen und das ausgeklügelte Zusammenspiel von Farben und Formen geniessen. *Abzû* wurde mit dem Ziel entwickelt, ein Kunstwerk zu erschaffen, und es ist generell anerkannt, dass dieses Ziel erreicht wurde. Hätte Roger Ebert den Spielen der

Moderne mehr Aufmerksamkeit geschenkt, hätte er das künstlerische Potential von Computerspielen vielleicht erkannt. [35] [36] [37] [38] [39]

Wir sind durchaus der Meinung, dass Computerspiele eine Kunstform sein können. Wie Richard Sherry in seiner Rezension von *Abzû* sagte, sind längst nicht alle Spiele Kunstwerke, doch das müssen sie auch nicht sein [36]. Wir teilen seine Meinung. Natürlich gibt es Spiele wie *Call of Duty* oder *FIFA*, die jedes Jahr neu erscheinen und doch immer dieselben sind. Andererseits gibt es auch Spiele wie *Abzû* oder *Journey*, welche einen kompletten Gegensatz dazu bieten. Die Ausstellung im Smithsonian sollte Beweis genug dafür sein, dass Computerspiele langsam aber sicher von der Gesellschaft als Kunst anerkannt werden.

4.9. Veröffentlichung und Rezeption

Im März des Jahres 2017 veröffentlichten wir eine, nun veraltete, Version von *Bane of Asphodel* auf der Website itch.io. Diese Seite ist eine eher unbekannte Plattform auf welcher man kostenlos seine Computerspiele veröffentlichen kann. Nach etwa einem Jahr auf der Plattform wurde das Spiel 358 Mal heruntergeladen. Da wir nur wenig Werbung machten finden wir diese Zahl erstaunlich hoch. Die Rückmeldungen, die wir von Spielern erhielten, waren zunehmend positiv, und jegliche Kritik versuchten wir nun bei der Überarbeitung miteinzubeziehen.

4.10. Ausblick

Nun da wir *Bane of Asphodel* definitiv abgeschlossen haben, wollen wir es auf einer grösseren Plattform veröffentlichen. Die grösste Plattform auf PC heisst Steam. Für hundert Franken kann man dort mittels *Steam Direct* sein Spiel anmelden und veröffentlichen. Leider werden dort jeden Tag zahlreiche Spiele veröffentlicht und die meisten davon sind von niederer Qualität. Aus diesem Grund ist es für kleinere Spiele ohne Marketing schwierig, entdeckt zu werden.

AAA-Spiele haben meistens ein grosses Marketingbudget und können somit extrem viel Werbung, sei es über Plakate, Fernsehen oder Internet, machen. Indie-Spiele haben dieses Privileg nicht, deshalb machen sie vor allem über soziale Netzwerke Werbung. YouTube und Twitter sind dabei besonders beliebt. Wir haben für *Bane of Asphodel* ein wenig Werbung auf Twitter gemacht, haben dort jedoch nicht viele Leute erreicht. Ausserdem haben wir auf der Image-Sharing Site *Imgur* einen Post über unser Spiel gemacht. Mit diesem Post hatten wir dann Erfolg. Innerhalb eines Tages hatten wir über 30'000 Aufrufe und über tausend Likes. Wir erhielten viele ermunternde Kommentare und es schien, als ob ein Interesse an unserem Produkt vorhanden sei. Dies war jedoch vor über einem Jahr und trotz mehrerer Updates und neuen Posts erhielten wir nie mehr so viel Aufmerksamkeit.

Wir fänden es toll, wenn wir viele Spieler mit unserem Spiel erreichen könnten, doch wie und wann wissen wir noch nicht. Die Arbeit an *Bane of Asphodel* hat uns beiden viel Spass gemacht. Wir sind froh, dass wir im Rahmen unserer Maturaarbeit die Gelegenheit hatten, ein Spiel zu entwickeln und hoffen, dass unser Spiel vielen Spielern Spass bereiten wird.

5. Dank

Als Abschluss möchten wir allen Menschen danken, die uns bei dieser Arbeit unterstützt haben. Als erstes möchten wir der Kantonsschule Willisau für die Gelegenheit, unser Projekt zu realisieren, danken. Wir danken unseren Betreuern, Igo Schaller und Erwin Hofstetter, für die Unterstützung, die wir im Verlauf der Arbeit von ihnen erhalten haben.

Ausserdem wollen wir Schweizer Jugend Forscht dafür danken, unsere Arbeit gewürdigt zu haben. Danke auch an Ralf Mauerhofer, unserem Betreuer bei SJF.

Wir möchten auch unseren Eltern danken, die uns trotz anfänglicher Skepsis immer unterstützt haben. Ein Dank auch an BurgZergArcade und Eric Tereshinski für ihre exzellenten Videotutorien, mit denen wir die Grundlagen von Unity 5 und C# lernen konnten. Danke an alle Testspieler, die uns dabei halfen, unser Spiel besser zu machen. Vielen Dank an alle, die für uns Korrektur gelesen haben. Schlussendlich möchten wir uns gegenseitig danken. Unsere Zusammenarbeit hat gut funktioniert und wir hätten alleine niemals so ein gutes Spiel produzieren können.

6. Quellenverzeichnis

[1] https://newzoo.com/insights/articles/global-games-market-reaches-99-6-billion-2016-mobile-generating-37/, 03.10.16.

[2] http://www.gamecareerguide.com/features/529/what_is_a_game_.php, 18.08.16.

[3] https://en.wikipedia.org/wiki/Game_engine, 18.08.16.

[4] https://en.wikipedia.org/wiki/Role-playing_video_game, 18.08.16.

[5] http://www.giantbomb.com/profile/chaoskiller2000/blog/jrpg-vs-wrpg-the-difference-and-why-they-are-both-/77299/, 18.08.16.

[6] https://en.wikipedia.org/wiki/AAA_(video_game_industry), 30.09.16.

[7] http://www.hellogames.org/about-us/, 30.09.16.

[8] https://en.wikipedia.org/wiki/Frame_rate, 30.09.16.

[9] http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep#OOP, 18.08.16.

[10] https://de.wikipedia.org/wiki/Objektorientierte_Programmierung, 18.08.16.

[11] https://de.wikipedia.org/wiki/Datenkapselung_(Programmierung), 18.08.16.

[12] https://de.wikipedia.org/wiki/Abstraktion_(Informatik), 18.08.16.

[13] https://de.wikipedia.org/wiki/Vererbung_(Programmierung), 18.08.16.

[14] https://de.wikipedia.org/wiki/Polymorphie_(Programmierung), 18.08.16.

[15] https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html, 18.08.16.

[16] https://en.wikipedia.org/wiki/Interface_(computing), 18.08.16.

[17] https://en.wikipedia.org/wiki/Polygon_mesh, 19.09.16.

[18] https://en.wikipedia.org/wiki/UV_mapping, 19.09.16.

[19] https://en.wikipedia.org/wiki/Normal_mapping, 19.09.16.

[20] http://3d.about.com/od/Creating-3D-The-CG-Pipeline/a/What-Is-Rigging.htm, 19.09.16.

[21] http://blog.digitaltutors.com/weight-painting-in-blender/, 19.09.16.

[22] https://en.wikipedia.org/wiki/Skeletal_animation, 19.09.16.

[23] https://en.wikipedia.org/wiki/Serialization, 19.09.16.

[24] https://de.wikipedia.org/wiki/Shader, 01.09.16.

[25] https://www.geforce.com/hardware/technology/txaa/technology, 18.03.18.

[26] https://en.wikipedia.org/wiki/Bloom_(shader_effect), 01.09.16.

[27] https://de.wikipedia.org/wiki/Umgebungsverdeckung, 01.09 16.

[28] https://de.wikipedia.org/wiki/Chromatische_Aberration, 01.09.16.

[29] https://pixabay.com/en/keyboard-computer-laptop-technology-629234/, 01.09.16.

[30] https://pixabay.com/en/console-gaming-hand-held-controller-309614/, 01.09.16.

[31] http://www.ibtimes.com/gta-5-costs-265-million-develop-market-making-it-most-expensive-video-game-ever-produced-report, 29.09.16.

[32] http://www.ign.com/articles/2013/09/18/gta-5-makes-800-million-in-one-day, 29.09.16.

[33] http://si.edu/Exhibitions/Details/The-Art-of-Video-Games-840, 29.09.16.

[34] https://www.theguardian.com/technology/gamesblog/2014/jan/08/video-games-art-and-the-shock-of-the-new, 29.09.16.

[35] http://www.rogerebert.com/rogers-journal/video-games-can-never-be-art, 29.09.16.

[36] http://www.gameskinny.com/mdvxu/abzu-journey-games-as-art, 30.09.16.

[37] http://time.com/4038821/brian-moriarty-are-video-games-art/, 30.09.16.

[38] http://www.washingtoncitypaper.com/arts/museums-galleries/blog/13077299/still-solid-metal-gear-creator-hideo-kojima-on-the-art-of-video-games, 30.09.16.

[39] https://en.wikipedia.org/wiki/Video_games_as_an_art_form, 30.09.16.

[40] https://www.youtube.com/user/ETeeskiTutorials, 19.10.16.

[41] https://www.youtube.com/user/BurgZergArcade/, 19.10.16.

[42] https://www.youtube.com/channel/UCwWQXO0OiWY6iY03y0-QfHw/, 19.10.16.

[43] https://docs.unity3d.com/550/Documentation/Manual/script-Tonemapping.html, 18.03.18

7. Anhang

Der Anhang besteht aus dem Drehbuch und dem für diese Arbeit geschriebenen Programmcode. Da er sehr lang ist haben wir ihn nicht diesem Dokument angefügt. Stattdessen können sämtliche Dokumente unter diesem Link eingesehen werden: https://1drv.ms/f/s!Avw9ZxU1fzrqg4UlHW5S4doGTMJBfQ

8. Glossar

AAA:	Spiele oder Studios mit einem grossen Budget für Entwicklung und Vermarktung bezeichnet man als AAA, oder Triple-A.
Animator Controller:	In Unity 5 kann man Animationen mittels eines Animator Controllers vernetzen. Dieser besteht aus Animationsclips und den Übergängen dazwischen.
BehaviourTree:	Ein BehaviourTree ist das «Gehirn» der Künstlichen Intelligenz in Computerspielen und besteht aus einer Abfolge von Wenn-Dann-Befehlen.
C#:	C# ist eine objektorientierte Programmiersprache basierend auf C.
Collider:	In einer Engine ist nichts von Anfang an «Fest». Um einem Objekt Solidität zu verleihen muss man ihm einen Collider geben. Collider können bei unveränderlichen Meshes wie Gebäuden oder Bäumen die Form der Mesh annehmen. Im Falle eines animierten Objekts ist das nicht möglich und der Collider muss eine einfachere Form annehmen, etwa die einer Kugel oder eines Quaders.
Frames per Second (FPS):	Frames per Second bezeichnet die Anzahl Bilder, die innerhalb einer Sekunde vom Computer berechnet und angezeigt werden.

Funktion:	Eine Funktion ist ein vordefinierter Prozess, der durch eine Zeile Code ausgeführt werden kann. Man unterscheidet zwischen Funktionen, die einen Wert ausgeben und *Void-Funktionen*, die das nicht tun.
GameEngine:	Die GameEngine ist eine Entwicklungsumgebung, speziell für die Entwicklung von Spielen gedacht ist.
GameObject:	In Unity 5 ist ein GameObject ein Objekt in einer Scene. Das GameObject kann mit Scripts, Meshes und Materials versehen werden.
Indie:	Indie Game ist die Abkürzung für *Independent video game* und eine Bezeichnung für Spiele oder Studios mit wenigen finanziellen Mitteln.
Künstliche Intelligenz (KI):	In Games besteht die KI eines Objekts aus einem BehaviourTree und Sensoren. Der BehaviourTree bezieht Informationen von den Sensoren und diese werden verarbeitet und eine definierte Reaktion wird ausgeführt.
Klasse:	Eine Klasse ist ein Begriff in objektorientiertem Programmieren. Sie ist eine Abstraktion eines Objekts und kann als solches instanziiert werden.
Material:	Ein Material wird benutzt, um Modelle in einem Spiel oder einer Animation einzufärben. Sie sind ein fester Bestandteil von *Shading* in Computergenerierten Bildern.
Methode:	Siehe *Funktion*
Mesh/Modell:	Ein Modell ist ein virtuelles Polygon, das in einem Modellierprogramm, zum Beispiel Blender, erstellt wurde.
Objektorientiertes Programmieren (OOP):	Ein Programmierstil bei welchem Systeme mithilfe von Objekten beschrieben werden

Open-World:	In einem Open-World-Spiel kann die gesamte Welt jederzeit besucht werden. Das Gegenteil sind lineare Spiele, in denen man ein Level abschliessen muss um in das nächste zu gelangen.
Quest:	In einem RPG werden die Aufgaben, die dem Spieler gestellt werden, Quests genannt.
Rendering:	Rendering nennt man den Prozess, bei dem ein Bild vom Computer berechnet und ausgegeben wird.
Rig:	Ein Rig ist eine Art Skelett, das den Animationsprozess vereinfacht. Der Rig besteht aus Knochen (Bones), welche durch Gelenke miteinander verbunden sind.
Role-Playing-Game (RPG):	RPG ist ein Genre in der Gaming-Industrie. In einem RPG übernimmt der Spieler die Rolle einer Fantasiefigur und muss an ihrer Stelle Entscheidungen treffen.
Scene:	In Unity 5 ist eine Scene ein «Level» des Spiels. Mit gewissen Scripts kann zwischen Scenes gewechselt werden.
Script:	Ein Script ist ein Stück Programmcode. In einer GameEngine können Objekte mit Scripts versehen werden. Scripts enthalten Variablen und Funktionen, die auf andere Objekte Einfluss haben.
State Machine	State Machine ist der englische Ausdruck für einen endlichen Automaten. Das ist ein Modell zur Beschreibung von Verhalten, bestehend aus Zuständen, Übergängen und Aktionen.
Terrain:	Das Terrain ist die Grundlage für die Geografie der Spielwelt. Unity 5 hat einen Terrain-Editor, der Entwicklern ermöglicht, schnell und einfach eine Spielwelt zu erschaffen.
Trigger:	Trigger sind eine Unterart von Collidern. Sie sind durchlässig und können erkennen, wenn ein Objekt sie betretet oder verlässt.

Variable:	Genau wie in der Mathematik ist eine Variable etwas, das verschiedene Werte annehmen kann. Im Gegensatz zur Mathematik jedoch können Variablen nicht nur Zahlen sein, sondern jegliche Datentypen.

9. Redlichkeitserklärung

Ich bestätige, die vorliegende Arbeit selbständig und ohne unerlaubte Hilfe erarbeitet und verfasst zu haben. Sämtliche Textstellen, die nicht von mir stammen, sind als Zitate gekennzeichnet und mit dem genauen Hinweis auf ihre Herkunft versehen. Die verwendeten Quellen sind im Quellenverzeichnis aufgeführt.

_____

Unterschrift Ort Datum

_____

Unterschrift Ort Datum