OTTO VON GUERICKE UNIVERSITÄT

Fakultät für Informatik

3D Game Project 2015

Softwareprojekt

Underlord

– Dokumentation –

Patrick Schön, Dominik Schön und Max Frick

Inhaltsverzeichnis

1. Einleitung

Im Folgenden wird die Durchführung und das Resultat des Softwareprojekts zu dem Spiel *Underlord* dargestellt, sowie ein Fazit des Projektes gezogen.
Das Projekt wurde von Dominik und Patrick Schön sowie Max Frick durchgeführt. Außerdem war zu Beginn noch ein viertes Gruppenmitglied dabei, welches jedoch aufgrund von Zeitmangel ausschied ohne einen großen, arbeitstechnischen Beitrag zum Projekt geleistet zu haben. Sie wird daher im Dokument als externe Hilfe vermerkt.

Bei *Underlord* handelt es sich um ein hexagonbasiertes 3D-Strategiespiel in dem der Spieler einen Dungeon bauen und angreifende Wellen von Helden abwehren muss, um so durch das Töten der Helden eine möglichst hohe Punktzahl zu erreichen. Dabei stehen dem Spiel zwei unterschiedliche Arten von Kreaturen, in Form von Ameisen und Skeletten, sowie zwei Zauber zur Verfügung. Die Kreaturen werden dabei in Nestern/Gebäuden produziert, welche durch Gold gekauft und durch Futter unterhalten werden müssen. Das für den Bau notwendige Gold kann hierbei durch das Abbauen von Gold- und Diamantadern, die überall im Erdreich verteilt sind, gesammelt werden. Für den eigentlichen Abbau gibt der Spieler bei seinen Helferlein, im weiteren *Imps* genannt, Jobs in Auftrag die automatisch abgearbeitet werden. Die *Imps* kümmern sich des weiteren auch um die Erweiterung des Dungeons, sowie das Ernten von Feldern zur Nahrungsgewinnung und das Füttern der Nester. Damit der Spieler Zauber wirken kann, benötigt er außerdem Mana, welches in Tempeln gewonnen wird.

Die gegnerischen Helden wiederum werden dem Spieler in zeitlichen Abständen über einen Höhleneingang auf den Hals gehetzt. Der Spieler kann selbst zusätzliche Eingänge bauen um die Anzahl an Helden zu erhöhen, den Schwierigkeitsgrad dadurch anzupassen und außerdem seine Punkteausbeute pro getötetem Helden zu steigern.

Die ursprüngliche Idee für das Spiel kam von Mareen Johannes, die das Projekt gerne an das Spiel "*Dungeon Keeper*" anlehnen wollte. Daraus entstand dann die grobe Spielidee, welche anfangs noch eine wesentlich größere Anzahl an Kreaturen und Helden, sowie ein umfangreiches Sounddesign und System zum Bauen und Aufstellen von Fallen vorsah. Dass das Projekt damit sehr umfangreich war, wurde uns sehr schnell klar, weshalb wir schon früh begannen einige der geplanten Elemente zu streichen. Vor allem durch das endgültige Ausscheiden von Mareen mussten zum Teil drastische Kürzungen am Umfang vorgenommen werden, auf die in den jeweiligen Abschnitten kurz eingegangen wird. Allerdings sind auch während des Fortschreitens des Projekts einige, neue Features hinzugekommen, die so nicht geplant waren. So zum Beispiel die torusförmige Spielwelt, welche an sich selbst anschließend ist und bereits in dem Spiel "*Die Völker*" vorkommt.

2. Projektmanagement

2.1 Zielsetzung

Die Zielsetzung des Projekts wurde bereits mit Erstellung des GDDs festgelegt. Es sollte ein Spiel entsteht, das dem Spieler Spaß macht. Außerdem sollte das Ergebnis später als Referenz verwendet werden können. Zuletzt war auch eine gute Note für das Softwareprojekt wichtig.

2.2 Aufgabenverteilung und Organisation

Die Organisation des Projekts haben wir mithilfe von Skype, Email und regelmäßigen Treffen realisiert, sowie unter Zuhilfenahme eines Redmine-Servers.

Während des Projekts selbst war es Max Aufgabe sich um die Erstellung der sowohl animierten als auch statischen Modelle zu kümmern, sowie deren späterer Anzeige im Spiel. Des Weiten war er für die GUI-Texturen verantwortlich. Hierbei hatte er die Realisierung eines GUI-Systems von Mareen übernommen und wurde dabei durch Dominik tatkräftig unterstützt. Darüber hinaus kümmerte sich Dominik um die Verwaltungsstrukturen der Spielelemente, wozu unter anderem die *Hexagone* und die *Map* gehörten. Auch die Interaktionsmöglichkeiten des Spielers bereitzustellen fiel in sein Ressort. Patrick hingen entwickelte hauptsächlich die Hintergrundlogik, wozu unter anderem die *AI,* das *Jobsystem* der *Imps* und der *Mapgenerator* gehörten.

2.3 Meilensteine

2.3.1 Meilenstein I:

Die Abgab des ersten Meilensteins war am 22.05 fällig und war eigentlich kein Meilenstein im herkömmlichen Sinne sondern stellte vor allem die Vorstellung des TDDs dar. Außerdem lag zu diesem Zeitpunkt der Fokus des Projekt vor allem bei der organisatorischen Aufbauarbeit, die ein solches Softwareprojekt zu Beginn häufig mit sich bringt. Auf Grund dessen hatten wir uns gerade auf die Kommunikationsarten Email, Skype und wöchentliche Treffen geeinigt. Außerdem wurde vereinbart, dass Max bis zum nächsten Meilenstein einen Redmine-Server zur Ticketverwaltung einrichtet. Des Weiteren hatten wir gemeinsam im GDD die grundlegenden design- und spieltechnischen Konzepte beschrieben, welche wir zukünftig im Spiel umsetzen wollten. Auch meldete sich Mareen freiwillig dafür künftige Konzeptänderungen und technische Diagramme im GDD und TDD festzuhalten und die Dokumente aktuell zu halten. Im Gegenzug dafür hatten wir ihr versprochen, dass sie sich nicht um tiefgreifende Implementationsaufgaben kümmert braucht, da sie fürchtete sich damit zu übernehmen.
Auf der technischen Seite hatten Patrick und Dominik damit begonnen, testweise ein Hexgaonfeld anzulegen und auf diesem zu navigieren. Zusätzlich hatten sie auch eine erste Maus-Kollision

geschrieben, d.h. ein durch die Maus selektiertes Hexagon wird farblich hervorgehoben. Max hatte aus seinem anfänglichen Prototyp-Spiel die Kamera- und die Partikelsystemlogik übernommen und letzter hinsichtlich der Performance etwas überarbeitet. Zudem hat er damit begonnen sich in die Animations-Thematik von XNA einzuarbeiten und versuchsweise probiert, mit einer eigenen kleinen Importer-Klasse, relevante Daten zu extrahieren. Mareen kümmerte sich derweil um das Erstellen der ersten Kreaturen und Nester, welche dann später ebenfalls durch Max texturiert und animiert werden sollten.

2.3.2 Meilenstein II:

Während es bei dem ersten Meilenstein primär darum ging vorprojektliche Aufbauarbeit zu leisten hatten wir bis zum erreichen des zweiten Meilensteins am 12.06 bereits einen Großteil der grundlegenden Datenstruktur implementiert und die Spieleprototypen ein gutes Stück voran gebracht.
So hatten Patrick und Dominik mittlerweile einen, zum damaligen Zeitpunkt, vollständige *Hexgaon*-, *Room*- und *Map*-Klasse geschrieben. Zusammen mit dem, von Patrick erstellten, *Mapgenerator* konnte zudem bereits ein Spielfeld generiert werden, welches sich aus Boden- und Wand-Modellen zusammensetzt die Max in Maya modelliert hatte. Auch hatte Dominik die Kamera mit einer Tastatursteuerung ausgestattet. Auf diese Weise war es zu diesem Zeitpunkt bereits möglich sich über die torusförmige *Map* zu bewegen. Außerdem war Dominik gerade damit beschäftige die *Interaction*-Klasse zu schreiben. Diese sollte später die einzelnen Baubefehle des Spiels registrieren und entsprechende Möglichkeiten zur Umsetzung bereitstellen. Damit hatte Dominik bereits die Möglichkeit geschaffen einen Raum aufzuziehen und ein Nest zu platzieren. Dank der *Nest*- und *Creature*-Klasse, welche sowohl Patrick als auch Dominik erstellt haben, wurde nach dem Platzieren eine erste Kreatur gespawnt. Das Nest hatte Mareen in Blender modelliert und Max anschließend in Maya textuiert. Bei der spawnenden Kreatur wiederum handelt es sich um ein einfaches Maden-Model, das im weiteren Projektverlauf nicht mehr zum Einsatz kam und auch nur speziell für die Vorstellung des Prototyp erstellt wurde. Der Grund hierfür war, dass die Animation zwar soweit schon betriebsfähig war, sprich die Klasse *Clip*, *ClipPlayer*, *CharacterModel* und *Animation-Import* nahezu vollständig implementiert waren, es jedoch noch einige Problem bei der nachträglichen Skalierung und Rotation der Model in XNA gab. Deswegen war die Made bereits passend zu dem Spielfeld in Maya rotiert und skaliert worden und diente zum bloßen Anschauungszweck. Ebenfalls fertig war das von Mareen entworfene und durch Max nachbearbeitete Echsen-Model, sowie der gegnerische Held. Patrick arbeitete des weiteren immer noch an der Maus-Kollision, da diese noch immer nicht ganz präzise arbeitete.

Trotz der genannten Erfolge schafften wir es zu diesem Meilenstein nicht, unser gewünschtes Kontingent zu erreichen. So war geplant, bis zu diesem Datum, einen lauffähigen Prototypen vorzuweisen. Das bedeutet, dass bereits Imps die Map bevölkern sollten, der Spieler mehrere

Gebäude auswählen und bauen können sollte und diese auch sogleich mit der Einheitenproduktion loslegen sollten. Darüber hinaus waren mehr Gebäude und Kreaturen, eine erste Gegner-AI sowie eine funktionierende Animation vorgesehen. Auch beim Projektmanagement gab es einige Komplikationen. Während die wöchentlichen Treffen und die Skype-Meetings reibungslos funktionierten machte das Aufsetzen des Redmine-Servers Probleme. Wie sich später herausstellen sollte, lag dies an einem Hardwaredefekt, welcher aber nicht sofort offensichtlich war.
Allgemein waren die zuvor beschrieben Schwierigkeiten in einem Mangel an Zeit begründet, welche durch einen straffen Stundenplan mit Übungen, anderen Projekten, einem Hiwi-Job und anderen Veranstaltungen verursacht wurden.

2.3.3 Meilenstein III:

Bis zum nächsten Meilenstein, der für den 10.07 angesetzt war, gelang es uns alle der zuvor offen geblieben Aufgaben zu erledigen.
Dominik hatte zwischenzeitlich die *Interaction*-Klasse um die Möglichkeiten des Vereinens und Löschens von Räumen erweitert und diese mit einer Hotkey-Steuerung für ein schnelleres Bauen ausgestattet. Außerdem war es möglich auf rudimentäre Art und Weise Wände zu Löschen, welches leider noch ohne Imps ablief. Patrick hatte zwar angefangen die *Imp-AI* zu schreiben, dieser fehlt aber das *Jobsystem*. Dagegen war es Patrick gelungen die *Creature-AI* fertigzustellen. Infolgedessen spawnten erstmals auch gegnerische Helden, welche sich anschließend umgehend in Richtung HQ auf den Weg machten, um sich dort mit den Kreaturen des Spielers einen Kampf zu liefern.
Dies funktionierte zu dem Zeitpunkt aber ausschließlich auf der Datenebene und wurde nicht visuell vermittelt. Das bedeutet, dass sich zwar zwei befeindete Kreaturen auf benachbarten Hexagonen gegenüberstanden aber keine Kampf-Animation zu sehen war. Zwar war es Max gelungen die Animationslogik noch einmal zu überarbeiten so das nun auch eine nachträgliche Rotation und Skalierung der Modelle in XNA möglich war, jedoch war er noch nicht dazu gekommen die Animation mit der *AI* zu kombinieren. Grund hierfür war, dass sich das Aufsetzten des Redmine-Servers noch einmal in die Länge gezogen hatte. Wie zuvor bereits erwähnt war der eigentliche Server kaputt, weshalb Redmine kurzerhand auf Maxs Arbeitscomputer installiert wurde. Dadurch konnte endlich das Projekt sauber bis zum Ende durchgeplant werden. Einen weitereren Zeitfresser für Max stellte das Modellieren und Animieren des HQ, des Eingangs, der Helden, der Imps sowie der Ameisen dar. Da es mittlerweile um Mareen, trotz mehrfacher nachfragen, sehr still geworden war musste Max kurzfristig die Erstellung der Modelle übernehmen. Mareen war es zwar noch gelungen die *Minimap* zu erstellen, jedoch das dringend benötigte Echsen-Nest, in welchem die *Echsen*-Kreaturen spawnen sollten, lies nach wie vor auf sich warten.
So war es nicht verwunderlich, dass auch die für diesen Termin abgemachte GUI fehlte, handelte es sich dabei doch um Mareens Aufgabenbereich. Auch ein erstes Balancing des Spiels durch Patrick und Dominik wurde nicht erreicht. Der Grund hierfür waren einige technische Schwierigkeiten wie

eine unsaubere Speicherverwaltung der Objekte was zu Performance-Einbrüchen führte und AI-Logik-Fehler, welche nur durch tiefgreifenden Änderungen an den *Hexagon*-, *Room*-, *Map*-, *Nest*- und *Creature*-Klassen behoben werden konnte. Das Projektmanagement klappt nach wie vor sehr gut. Da es auf das Ende des Semesters zu ging und die meisten abzugebende Arbeiten bereits erledigt waren konnten wir die Anzahl unserer wöchentliche Treffen auf zwei bis drei steigern.

Da es sich bei dem nächsten offiziellen Meilenstein auch bereits schon um den finalen Abgabetermin handelte, erachteten wir es für sinnvoll uns selber interne Meilensteine zu legen. Diese sollten einen kleineren Umfang als die Vorangegangenen haben, dafür aber auch enger getaktet sein.

2.3.4 Meilenstein IV (intern):

Der für den 25.07 angesetzte Meilenstein war nicht nur der erste interne Meilenstein sondern darüber hinaus auch der Zeitpunkt wo endgültig klar war, dass Mareen sich nicht mehr weiter an diesem Projekt beteiligen wird. Im Zuge dessen hatten wir angefangen den ursprünglichen Spieleinhalt radikal zu kürzen. Das geplante Fallen- und Hindernissystem wurde restlos gestrichen und die Anzahl der zu bauenden Einheiten auf zwei reduziert. Max erklärte sich bereit den kompletten Modellierungspart von Mareen zu übernehmen, während Patrick und Dominik im Gegenzug dafür das Speichersystem realisieren wollten. Hierbei sollte aber nicht mehr die gesamte Map und alle weiteren Komponenten des Spiels gespeichert werden, so dass der Spieler jederzeit wieder in das Spiel einsteigen kann, sondern lediglich der erreichte Highscore. Darüber hinaus sollten Max und Dominik das eigentliche *GUI*-System umsetzten, während Patrick noch einmal die *Minimap* überarbeiten sollte.
Da er dies wohl kommen sah, hatte Dominik bereits in weiser Voraussicht damit begonnen die *GUI-Element*-Klasse zu schreiben. Patrick hatte mittlerweile auch die *Imp-AI* komplettiert und war gerade dabei das Jobsystem mit sämtlichen *Mine*-Jobs auszustatten. Außerdem hatten Patrick und Dominik mit einem einfachen Balancing begonnen. Dies betraf vor allem die Stärke und die Spawnrate der Kreaturen des Spielers und der gegnerischen Helden. Max schaffte es bis zu diesem Datum sämtliche Modelle von Gebäuden und Kreaturen fertigzustellen und zu animieren. Es blieb sogar soviel Zeit über, dass der gegnerische Held noch einmal komplett überholt wurde, da dieser in seiner ersten Form doch recht plump geraten war.
Ansonsten waren wir tatsächlich einmal im Soll, von Mareens Aufgaben abgesehen. Zwar hatte uns ihr endgültiges Ausscheiden kurzfristig etwas aus der Bahn geworfen, uns aber auch gleichzeitig die Augen für das Wesentliche im Spiel eröffnet.
Das Management funktionierte nach wie vor reibungslos.

2.3.5 Meilenstein V (intern):

Der zweite, interne Meilenstein war für den 15.08 angesetzt. Zwischenzeitlich hatten wir uns von Mareens Ausfall erholt und dank Redmine die Aufgaben sauber an alle weiteren Beteiligten verteilt. Patrick hatte in der Zwischenzeit das *Jobsystem* fertiggestellt. Zusammen mit Dominiks, ebenfalls voll funktionsfähigen, *Interaction*-Klasse waren damit alle zentralen Spielmechaniken implementiert. Mit wenig Aufwand hatte Max zudem die Animationslogik und die Darstellung der unterschiedlichen Aktionen mit der im Hintergrund arbeitenden Mechanik verknüpft, sodass der Spieler jetzt endlich ein umfangreiches visuelles Feedback bekam. Außerdem hatte Max ein Beleuchtungssystem geschrieben, welches auf der durch XNA bereitgestellten *BasicEffect*-Klasse basiert und das Partikelsystem nutzt. Dabei war Max eigentliche Aufgabe, mit einfachen Schatten-Modellen, welche unter die entsprechenden Kreaturen gezeichnet werden sollten, den Eindruck von räumlicher Tiefe zu verstärken. Mehr oder weniger durch Zufall zeigt sich, dass die Beleuchtungsmöglichkeiten von XNA für unsere Zwecke absolute ausreichend und zudem sehr einfach zu implementieren waren, weshalb sie kurzerhand in das Spiel integriert wurden. Derweil hatte Dominik das *GUI*-System weiter ausgebaut und mit einfachen Dummy-Texturen ausgestattet, welche später von Max noch durch die passenden grafischen Elemente ersetzt werden sollten. Auf diese Weise war es möglich, sich den Gold-, Nahrungs- und Manavorrat des Spielers, die Kosten der Gebäude sowie das Leben der eigenen wie auch der gegnerischen Kreaturen anzuzeigen. Zudem hatte Dominik die *Upgrade*-Klasse sowie das *Farm*- und *Tempel*-Gebäude eingebaut. Patrick hatte noch einmal Mareens *Minimap* überarbeitet, sodass nun auch die unterschiedlichen Gebäude und Kreaturen auf dieser farblich hervorgehoben wurden. Außerdem hatte er mit der Implementierung der *Spell*-Klasse begonnen und sich zusammen mit Dominik um das Balancing der Gebäudekosten sowie der Nahrungs- und Manaproduktion gekümmert.
Als einige technische Schwierigkeit hatten wir kurzfristig mit einem flackernden Spiel-Screen zu kämpfen, diesen Lösung im Vertauschen der Updatereihenfolge von *Camera* und *Map* lag. Wie bei dem vorherigen Meilenstein konnten auch hier alle gesetzt Zwischenziele erreicht werden. Weil Patrick und Dominik vor dem 15.08 auf einem kleineren, zweiwöchigen Heimaturlaub waren, pausierten wir in dieser Zeit unsere wöchentlichen Treffen. Blieben aber nach wie vor via Email und Skype in Kontakt.

2.3.5 Meilenstein VI:

Zwar war die Abgabe des finalen Spiels für den 06.09 vorgesehen. Jedoch erschien uns dieser Termin im Falle von möglichen Komplikationen zu knapp, weswegen wir den 02.09 zur Fertigstellung unseres Spiels ansteuerten. Dies Entscheidung rettet uns, ihm Nachhinein betrachtet, ganz offensichtlich das Projekt, da wir ohne die damit zusätzliche Zeit in arge Bedrängnis gekommen wären. Der Grund hierfür war, dass sich Max kurz vor der finalen Fertigstellung eine schwere Lebensmittelvergiftung eingefangen hatte und er somit, zumindest teilweise, außer Gefecht gesetzt war.

Der Stand der Prototyps zum 02.06 war dabei wie folgt:
Vor seiner Erkrankung hatte Max glücklicherweise bereits alle benötigten *GUI*-Texturen erstellen, passende Erweiterungen an der *GUI-Element*-Klasse vorgenommen und die *BasicGUI*-Klasse geschrieben. Was an dieser Stelle noch fehlte, war die Integration in den Spielverlaufs und die Verknüpfungen zur *Interaction*-Klasse. Auch die Einbettung der von Patrick und Dominik verfassten Einsteiger-Hilfstexte war noch nicht getan. Patrick hatte derweil die *Spell*-Klasse komplettiert, einen *WaveConroller* zur Steuerung der Gegnererzeugung implementiert und war gerade dabei die *Player*-Klasse zu erweitern. In dieser sollten die aktuellen Vorräte des Spiels hinterlegt und der momentane Highscore in einer externen Datei gespeichert werden. Diese sollte zu einem späteren Zeitpunkt wieder durch die *Highscore-GUI* eingeladen werden und dem Spieler eine Möglichkeit bieten sich mit anderen zu vergleichen. Außerdem fehlte es noch an einem *Feuerball*-Model für den entsprechende Zauber.

Unter großer gemeinschaftlicher Anstrengung, bei der die ursprüngliche Aufgabenverteilung galant ignoriert wurde, konnten bis zum 06.09 tatsächlich alle der oben genannten Punkte erledigt werde. Während Max und Dominik gemeinschaftlich an der vollständigen und sauberen Integration der GUI arbeiteten, war Patrick damit beschäftige die *Player*-Klasse fertig zu schreiben. Kaum was das getan, machte sich Max daran den Feuerball zu modellieren und diesen in die *Spell*-Logik einzubinden, während Patrick und Dominik eine letzte Balancing-Phase durchführten.

Die einzigen Punkte die dabei, im Nachhinein betrachtet, auf der Stecke blieben waren, sind ein nicht immer stimmiges Verhalten der Feuerball-Partikel sowie die Tatsache, dass der Spieler zwar seinen Score, nicht aber seinen Name abspeichern konnte. Kleinere Unsauberheiten hatten damit zu tun, dass die farbliche Hervorhebung der Räume in ungünstigsten Fällen mit den Farben der Mine-, Room- und Merge-Markierung übereinstimmte und der Spieler somit nicht mehr das gerade angewählte Hexgon sehen konnte.

Nichts desto trotz waren wir über das fertige Spiel, gerade weil wir ziemlich dafür kämpfen mussten, sehr zufrieden.

3. Technische Umsetzung

3.1 Übersicht

Einleitend soll durch Abbildung 1 eine kurze, systematische Übersicht über die wichtigsten Komponenten des Spiels gegeben werden. Dabei wird aufgezeigt, wie ausgehend von der *Game1*-Klasse, die Kern-Funktion *Update* (Rot) und *Draw* (Grün) ihre Informationen auf die tieferen Logik-Ebenen propagieren und weiterleiten. Die Darstellung ist nicht vollständig sondern soll dem Leser lediglich eine Orientierungshilfe biete um die später im Detail vorgestellten Aspekte in den richtigen Zusammenhang setzten zu können.

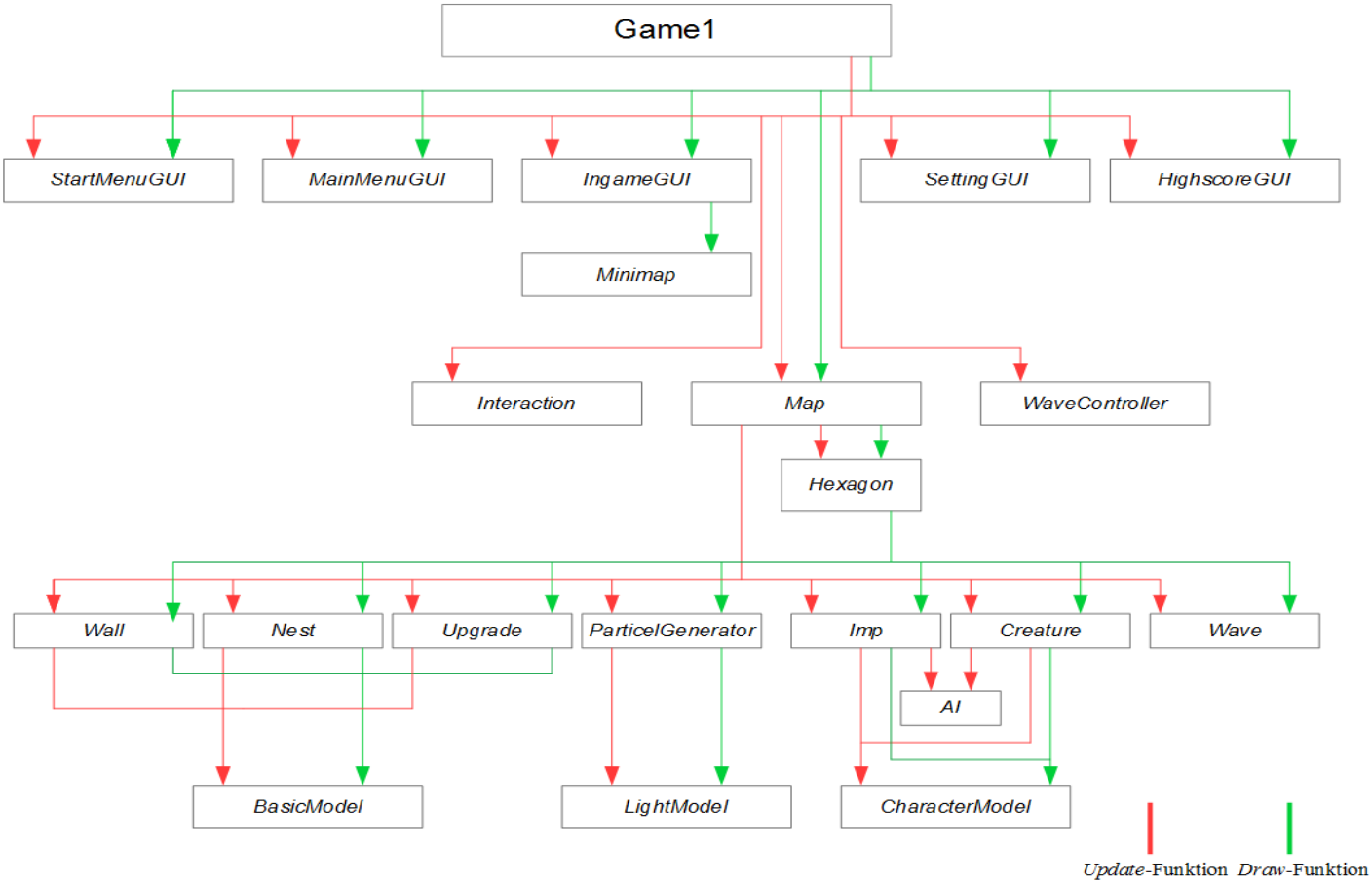


Abb. 1. Allgemeiner Systemaufbau

3.2 Grundlegende Datenstrukturen

3.2.1 Hexagon:

Das *Hexagon*-Objekt ist das zentrale Datenelement, aus dem die Spielwelt zusammengebaut ist und speichert daher einen Großteil der für das Spiel benötigten Informationen.

Diese Informationen sind:

- eine zweidimensionale Indexnummer, über die jedes Hexagon eindeutig identifizierbar ist
- die Position des Hexagons im 3D-Raum
- die Indexnummern seiner sechs Nachbar-Hexagone
- das Thing-Objekt, welches sich auf dem Hexagon befindet
- eine Liste mit den Imps, welche sich auf dem Hexagon befinden
- eine Raumnummer, durch die jedes Hexagon weiß, welchem Raum es angehört
- eine Farbe, die das Einfärben des Hexagons und des auf ihm befindlichen Thing-Objekts ermöglicht
- einen Boolean "visited" und eine Indexnummer "parent", die für Breitensuchen benötigt werden
- einen Boolean, der angibt, ob sich auf das Hexagon schon ein Nest ausgebreitet hat
- einen Boolean, der angibt, ob das Hexagon frei für den Bau von Gebäuden ist
- einen HexTyp, der bestimmt, welche Textur für das Hexagon verwendet wird
- ein GrowObject, das bestimmt, welches Objekt auf dem Hexagon für die Ausbreitung von einem Nest verwendet wird (z.B. die Pilze bei einer Farm)
- einen Boolean, der angibt, ob auf dem Hexagon eine Zielpunktflagge gezeichnet werden soll
- einige Variablen, die für das Zeichnen verwendet werden und später näher erläutert werden

3.2.2 Room:

Ein *Room*-Objekt ist eine zusammenhängende Fläche aus Hexagonen, die der Spieler anlegen kann, um dort Nester, Farmen oder Tempel zu bauen. Näheres hierzu im entsprechenden Abschnitt. Die Instanz speichert dazu eine Liste mit den zweidimensionalen Indexnummern der Hexagone, die zu diesem Raum gehören. Zusätzlich werden noch der *Nest*-Typ und die Position des *Nest*-Objekts im Raum gespeichert.

Der *Room*-Konstruktor bekommt bei der Erzeugung die Indexnummer des mittleren Raum-Hexagons sowie einen Radius übergeben. Über den Radius werden dann alle Indexnummern der Hexagone, die sich in diesem Radius befinden, aufgesammelt und zu dem Raum hinzugefügt. Welche Hexagone sich in dem Radius befinden wird mit einer Breitensuche über der *Map,* ausgehend von dem mittleren Hexagon des Raumes, festgestellt. Dabei wird sichergestellt, das die

Breitensuche nur für den Raum gültige Hexagone hinzufügt.
Über Funktionen *deleteRoom* und *mergeRoom* ist es des Weiteren möglich einen Raum zu löschen bzw. zwei Räume, welche aneinandergrenzen, zu einem Raum zu verschmelzen.

3.2.3 Thing:

Thing ist eine abstrakte Klasse, von der *Wall*, *Nest, Creature, Upgrade* und *Imp* erben. Dies sind die Objekte, die sich auf den Hexagonen befinden und von denen immer nur genau eins auf einer *Hexagon*-Instanz sein kann. Die Ausnahme davon bildet der *Imp*, welcher bis zu 6 mal zusätzlich zu anderen Objekten auf einem Hexagon vertreten sein kann.

a) Wall:

Das *Wall*-Objekt stellt das auf der *Map* am häufigsten vertretene Spielelement dar. Es handelt sich hierbei um die Steinwände die zu Beginn des Spiels fast die gesamte *Map* belegen und die der Spieler erst von seinen *Imps* abbauen lassen muss, um Platz für weitere Gebäude zu schaffen. Dabei gibt es drei verschiedene Arten von Wänden. Die Erste ist der normale Steinwall, der nichts weiter bewirkt, als das er das Spielfeld blockiert. Zweitens gibt es einen mit Goldadern durchzogenen *Wall*, durch deren Abbau der Spieler Gold erhält. Die dritte Art ist der mit Diamantadern durchzogene *Wall.* Durch diesen erhält der Spieler zwar ebenfalls Gold, wenn seine *Imps* ihn abbauen, aber im Gegensatz zu den beiden zuvor genannten *Wall*-Typen, wird dieser nicht vom Spielfeld entfernt und stellen somit eine endlose Goldquelle dar. Zu beachten ist dabei allerdings, dass der Abbau von Diamant langsamer ist als der von Gold und der Spieler somit in der selben Zeit beim Abbau von Goldadern mehr Gold erhält als es beim Abbau von Diamantadern.

b) Nest:

Bei dem *Nest*-Objekt handelt es sich um vom Spieler platzierbare Gebäude. Das Objekt speichert dabei folgende Informationen:

- den Nest-Typ bzw. welches Creature-Objekt durch das Nest instanziiert wird (Ameise, Skelett, Farm, Tempel, Eingang)
- eine Liste, die die Hexagon-Indexnummern der Hexagone enthält, auf die sich das Nest schon ausgebreitet hat
- eine Liste, die die Hexagon-Indexnummern der Hexagone enthält, auf die sich das Nest als nächsten ausbreiten kann
- die Position, auf der sich das Hauptgebäude des Nests befindet
- eine Liste, die die in dem Nest gebauten Upgrades enthält
- ein Array, in dem die gebaute Anzahl der unterschiedlichen Upgrades steht

- die Kosten für das nächste Upgrade
- den aktuellen Ernährungswert des Nests
- den maximalen Ernährungswert des Nests
- die Zielposition, zu der sich die Kreaturen dieses Nests bewegen sollen
- die Menge der zum Abbau verfügbaren Nahrung (für Farmen)

Die *Nest*-Typen, welche sich in *Ameise*, *Skelett*, *Farm* und *Tempel* unterteilen, können vom Spieler in einen zuvor angelegten Raum gebaut werden, sofern sich in diesem Raum noch kein Nest befindet. Außerdem müssen alle Nachbarhexagone von dem Hexagon, auf das das Nest gebaut werden soll ebenfalls zu diesem Raum gehören. D.h. ein Raum in welchem ein *Nest*-Objekt platziert werden soll muss mindestens aus sieben zusammenhängenden Hexagonen bestehen: das mittlere Hexagon auf welchem das eigentliche Nest gesetzt wird und all seine sechs unbebauten Nachbarn, damit sich das Nest, nach dem es gebaut worden ist, automatisch auf den Nachbar-Hexagonen ausbreiten kann. Zusätzlich werden diese sieben Hexagone für den Bau von weiteren Gebäuden gesperrt. Wenn der Raum, in dem das Nest platziert wurde, größer als die sieben benötigten Hexagone ist, werden außerdem weitere Hexagone des Raums, in die Liste der Hexagone aufgenommen, auf die sich das Nest als nächstes ausbreiten kann.
Der *Nest*-Typ *Eingang* kann von Spieler auf Hexagonen gebaut werden, die nicht in einem Raum enthalten sind, sofern sich auf diesem kein Objekt befindet. Außerdem darf sich auf dem oberen Nachbarhexagon kein weitere *Eingang* befinden und auf dem unteren Nachbarhexagon weder ein *Wall*, noch ein *Nest* oder ein anders Gebäude des Spielers.

Auf der grafischen Ebene handelt es sich bei einem *Nest* um ein einfaches, grafische Model, welches je nach Typs ein anders Aussehen aufweist. So wird dieses beispielsweise bei einem *Nest*-Typ *Skelette* durch ein *Gruf*-Model repräsentiert. Auch die Hexagone, auf denen sich das Nest bereits ausgebreitet hat, verfügen über ein typabhängiges Model. So soll, um bei dem zuvor genannten Beispiel zu bleiben, bei einem vollständig besetzten Raum vom Typ *Skelette* der Eindruck eines kleinen Friedhofs entstehen. Bei weiterem Interesse können sämtliche Model im GDD nachgeschlagen werden. Die Modele selbst sind dabei alle als *BasicModel*-Objekte instanziiert, welche eine Art Wrapper-Klasse für das klassische XNA-Model darstellt und dieses mit weiteren Funktionalitäten ausstattet. Eine detaillierte Beschreibung finde sich hierzu unter dem entsprechenden Abschnitt.

Als grundlegende Funktion implementiert die Klasse das Spawnen von *Creature*-Objekten an den entsprechenden Nestern. Hierfür ist außerdem für jedes Nest eine Zielposition festgelegt, zu der sich das jeweilige Objekt nach der Erzeugung bewegen soll. Diese Zielposition der Gebäude lässt sich vom Spieler verändern.
Auch verfügt die *Nest*-Klasse über eine Methode, mit welcher die Nester mit Upgrades ausgerüstet und verbessert werden können. Auf diese Weise werden bestimmte Statuswerte der *Creature*-Objekte verändert und erhöht.

In der *Update*-Funktion werden, mit verschiedenen Zählern, unterschiedliche Funktionalitäten der Nester realisiert. Dazu zählt:

- das Ausbreiten der *Nest*-Typen auf eines der nächsten, dafür verfügbaren, Hexagone
- das Generieren von Manapunkten für den Spieler durch den *Tempel*, wobei der Zeitintervall bis zur Erzeugung des nächsten Punkts kleiner wird, wenn der Tempel sich auf mehr Felder ausgebreitet hat
- das Erhöhen der Menge der zum Abbau verfügbaren Nahrung bei der *Farm* analog zur Managenerierung des *Tempels*. Sollte dieses Menge einen bestimmten Schwellwert überschreiten, wird ein neuer *Job* generiert, der den Abbau der Nahrung durch einen *Imp* einleitet
- das Verringern des Ernährungswertes für die Nester *Ameise* und *Skelett* . Unterschreitet dieser Wert einen Schwellwert wird ein neuer *Job* erzeugt, mit dem Ziel der Nestfütterung
- das Aufrufen der oben beschriebenen *Spawn*-Funktion in einem bestimmten Intervall. Diese Zeitspanne ist, wie bei den Gebäuden *Tempel* und *Farm*, von der Anzahl der Hexagone, auf die sich das Nest ausgebreitet hat, abhängig

c) Creature:

Neben der bereits erwähnten *Imp*-Klasse, welche im nachfolgenden Abschnitt vorgestellt wird, stellt die *Creature*-Klasse den spielerischen Kern von *Underlord* dar. Diese repräsentierten nicht nur die tatsächlich Einheiten und Gegner des Spieler sondern bringen darüber hinaus sämtliche Funktionalität unter einen Hut, angefangen bei der AI für die Steuerung der Entitäten, bis hin zur Instanziierung der *CreatureModel*-Klasse zum Zeichen und Animieren der einzelnen Objekte.

Während die *Nest*-Model durch die *BasicModel*-Klasse realisiert wird, wird die grafische Darstellung der *Creature*-Objekte mit Hilfe der *CharacterModel*-Klasse umgesetzt. Dabei handelt es sich im Prinzip, ebenso wie bei dem *BasicModel*, um einen Wrapper der vor allem die Animation des Models ermöglicht.

Die *Creature*-Objekte selbst teilen sich dabei in die Kreaturen und das Hauptgebäude des Spielers, sowie die gegnerischen Helden ein. Die Kreaturen werden von den Nestern des Spielers erschaffen und treten in Form von Ameisen- und Skelett-Modellen in Erscheinung. Gegnerische Helden, als Ritter dargestellt, werden wiederum von den Eingängen instanziiert. Bei dem Hauptgebäude, im weiteren nur als *HQ* bezeichnet, handelt es sich um einen Spezialfall, da es sich bei diesem eigentlichen auch um eine Spieler-Kreatur handelt. Diese kann zwar durch die Gegner angegriffen werden sich aber nicht selbst bewegen. Jede Instanz speichert dabei jeweils seine eigenen Information. Zu diesen gehören:

- der Typ der Kreatur (Ameise, Skelett, Ritter oder Hauptgebäude)
- der Schaden, den die Kreatur bei einem Angriff verursacht

- die Sichtweite, in der die Kreatur auf feindliche Kreaturen reagiert
- die Lebenspunkte, die die Kreatur maximal hat
- der Schaden, den die Kreatur schon erlitten hat
- das maximale Alter der Kreatur
- das aktuelle Alter der Kreatur
- den aktuellen Alters-Modifikator
- die Geschwindigkeit, mit der sich die Kreatur fortbewegt
- die Geschwindigkeit, mit der die Kreatur angreift
- die Position, auf der sich die Kreatur aktuell befindet
- einen Weg, der angibt, über welche Felder die Kreatur sich als nächstes bewegt
- das Nest, das diese Kreatur erzeugt hat

Im Konstruktor eines *Creature*-Objektes werden die Statuswerte der Entität, je nach Typus, auf unterschiedliche Werte gesetzt. Bei den beiden Kreaturtypen, die von Nestern des Spielers erschaffen werden (Ameisen und Skelette), wird bei der Instanziierung zunächst kontrolliert mit wie vielen Upgrades das Nest ausgestattet ist, bevor die Kreatur erschaffen wird und ihre Statuswerte entsprechende der Upgrade-Anzahl angepasst werden.
Als weitere Funktion implementiert die Klasse einen Alters-Modifikator. Dieser passt die Kreatur entsprechend ihres aktuellen Alters und in Abhängigkeit ihres Typs an. Dadurch werden die Kreaturen des Spielers ab einem gewissen Alter erst stärker und anschließend, wenn sie zu alt werden, wider etwas schwächer, wohingegen die Helden in einem bestimmten Intervall ständig stärker werden.
In der *Update*-Funktion der *Creature*-Klasse wird in einem ersten Schritt die *AI*-Logik der Kreatur aufgerufen. Handelt es sich bei der *Creature*-Instanz um eine Kreatur des Spiels wird zunächst geprüft, ob die Kreatur ihr maximales Alter erreicht hat und eventuell der Liste der aus dem Spiel zu entfernenden Kreaturen hinzugefügt. Anschließend wird das Alter der Kreatur erhöht und die Methode aufgerufen, welchen den Altersmodifikator bei bestimmten Schwellen verändert. Außerdem implementiert die *Update*-Methode ein *State-Machine*, welche entsprechend des aktuellen Zustands der *Creature* den passenden *Clip* in die *ClipPlayer*-Klasse des *CharacterModels* läd und diese abspielt. Die eigentlichen Zustände werden dabei, je nach Job oder gewünschtem Verhalten, durch die *AI*-Logik festgelegt. Während ein *Imp* zu jedem Job über eine dafür passende Animation verfügt, sind bei den restlichen Modeln die beiden Animationsarten *Walk* und *Fight* vollkommen ausreichend.

d) Upgrades:

Das *Upgrade*-Objekt ist, wie das *Nest*-Objekt, ein vom Spieler platzierbares Gebäude, speichert aber im Gegensatz zu diesem nur seinen *Upgrade*-Typ sowie seine Position.

Es gibt drei verschiedene Typen von Upgrades, die bei den neuen Kreaturen von ihrem zugehörigen Nest folgende Effekte haben:

- das Schadensupgrade, das den zugefügten Schaden erhöht
- das Lebenspunkteupgrade, das die maximalen Lebenspunkte erhöht
- das Geschwindigkeitsupgrade, das die Bewegungs- und Angriffsgeschwindigkeit erhöht

Jedes Upgrade wird dabei durch ein unterschiedliches Flagen-Model dargestellt, welche alle vom Typ *BasicModel* sind. Diese können nur dann in einem Raum platziert werden, wenn sich in diesem schon eine Nest des Typs *Ameise* oder *Skelett* befindet. Zusätzlich muss sich das Nest dieses Raums, analog wie bei dem Setzten eines Nestes, bereits auf mindestens sieben zusammenhängenden Hexagone ausgebreitet habe. Diese Felder dürfen außerdem nicht, auf Grund von bereits gebauten Gebäude, gesperrt sein. Mit dem Bau des Upgrades werden dann diese Hexagone für eine weitere Bebauung gesperrt und die Kosten für das nächste Upgrade in dem zugehörigen Nest werden erhöht.

3.2.4 Imp:

Der Aufbau der *Imp*-Klasse ist analog zu der zuvor beschrieben *Creature*-Klasse, von einigen Erweiterungen abgesehen. So besitzen *Imps* einen eignen *Thing*-Type *Imp* und besitzen als grafische Repräsentation kein *Character*- sondern ein *ImpModel*. Diese ist dem *CharacerModel* sehr ähnlich mit dem Unterschied, dass es eine größere Anzahl an Animation verwalten muss. Darüber hinaus ist ein *Imp* nicht angreifbar und kann nur durch einen Zauber gespawnt werden. Ein *Imp*-Objekt speichert darüber hinaus einen ab zulaufenden Pfad, seine Position in der Spielwelt, sowie einen Verweis auf ihren derzeit ausgeführten *Job*. Wie bei der *Creature*-Klasse wird die AI des *Imps* über die *Update*-Funktion ständig aufgerufen.

3.2.5 Map:

Die *Map*-Klasse repräsentiert das Spielfeld von *Underlord* und stellt die Klasse dar, welche fast alle Informationen des Spiels enthält. Sie besteht aus einem Spielfeld mit 50x50 Hexagon-Instanzen, dessen rechter mit dem linken Rand und dessen oberer mit den unteren Rand verbunden ist. Durch diese torusartige Struktur entsteht der Eindruck, dass das Spielfeld keine Begrenzung aufweist. Damit die Zugriffsgeschwindigkeit möglichst kurz ist, wird die *Map* in einem eindimensionalen *Hexagon*-Array gespeichert. Um dabei den Zugriff auf ein *Hexagon* möglichst komfortabel zu gestalten kann über eine Funktion, per Angabe der zweidimensionalen Indexnummer des Feldes, auf das entsprechende *Hexagon*-Element zugegriffen werden.
Zusätzlich werden in der *Map* mehrere Listen gespeichert, die die unterschiedlichen Entitäten des Spiels, sowie Räume und Informationen, die für die nachfolgend vorgestellte *Job*- und *Wave*-Verwaltung benötigt werden, enthalten. Diese sind:

- eine Liste mit den auf der Map existierenden Räumen
- eine Liste, die die Nester des Spielers enthält
- eine Liste, die die Farmen des Spielers enthält
- eine Liste, die die Tempel des Spielers enthält
- eine Liste, die die Eingänge von der Oberfläche enthält
- eine Liste, die die Kreaturen des Spielers enthält
- eine Liste, die die Helden-Kreaturen des Gegners enthält
- eine Liste, die die Kreaturen enthält, die aus dem Spiel entfernt werden sollen
- eine Liste, die die Imps des Spielers enthält
- eine Queue, die die Jobs enthält, die von den Imps ausgeführt werden sollen
- eine Liste, die die Jobs enthält, die gerade von den Imps ausgeführt werden
- eine Liste, die die Jobs enthält, die fertig ausgefürt sind und entfernt werden können
- eine Liste, die die Hexagon-Indexnummern der Hexagone enthält, die über MineJobs von den Imps abgebaut werden sollen
- eine Liste, die die Wellen enthält, die gerade Helden-Kreaturen erschaffen
- eine Liste, die die Wellen enthält, die alle ihre Helden-Kreaturen erschaffen haben

Die *Map* enthält verschiedene *Move*-Funktionen die *Creatures* oder *Imps* auf das nächste Feld ihres Weges bewegen, sofern dieses Feld frei ist. Außerdem verfügt sie über eine *Remove*-Funktion, die eine *Creature* aus der Spielwelt entfernt, wenn deren Lebenspunkte auf 0 gefallen sind bzw. sie ein kritisches Alter erreicht hat.
In der *Update*-Funktion werden zunächst alle, momentan registrierten, *Jobs* aktualisiert und gegebenenfalls auch beendet, bevor anschließend die jeweiligen *Update*-Funktionen von allen Nestern, Farmen, Tempeln, Eingängen, Kreaturen, Helden, Imps, Wellen, sowie des Licht- und Partikelsystems aufgerufen werden. Anschließend wird geprüft, ob Kreaturen gestorben sind, d.h ob sich Kreaturen in der Liste der zu entfernenden Objekt befinden. Diese Kreaturen werden daraufhin aus dem Spiel entfernt und für jeden gestorbenen Helden erhält der Spieler, die Anzahl der aktuell existierenden Eingänge, als Punkte gutgeschrieben. Sollte das *HQ* des Spielers zerstört worden sein, so gilt das Spiel als verloren und die Spielrunde wird beendet. Zuletzt werden die gewonnen Punkte in ein externes Dokument geschrieben, mittels derer die *Highscore-GUI* entsprechend der eingetragenen Werte aktualisiert wird.

3.2.6 BasicModel:

Wie bereits an einigen Stellen erwähnt, handelt es sich bei der *BasicModel*-Klasse um einen Wrapper der die durch XNA bereitgestellte *Model*-Klasse um einige Funktionalitäten erweitert. Zu diesen zählen unter anderem die Möglichkeit eine Farb- oder Textur-Änderung an dem Model

vorzunehmen. Als wichtigste Methode stellt das *BasicModel* eine *Draw*-Funktion bereit, welche das eigentliche Zeichnen des Models in der Spielwelt realisiert. Als Parameter werden der Funktion hierfür die Spielkamera sowie die Position, Rotation und Skalierung des Objekts als Matrix übergeben. Zusätzlich lässt sich bestimmen, ob das Model mit einem specularen Lichtanteil versehen wird. Dies ist grundlegend für die Arbeitsweise des später vorgestellten Beleuchtungssystems. Der Zeichenvorgang selbst ist hingegen relativ einfach. Hierfür wird lediglich über sämtliche im Model enthaltenen SubMeshs iteriert und durch die standardisierte *BasicEffect*-Klasse, unter Verwendung der View- und Projektionsmatrix aus der Kamera und den Orientierungswerten aus der Lokalisationsmatrix, in den dreidimensionalen Raum gezeichnet. Darüber hinaus stellt das *BasicModel* zwar einen Schnittstelle für eine *Update*-Methode bereit, implementiert diese aber nicht selbst. Einzig für die korrekte Arbeitsweise der beiden Ableitungen *CharacterModel* und *LightModel* ist diese Methode unerlässlich.

3.2.7 Vars_Func:

Die *Vars_Func*-Klasse ist eine Sammlung von Variablen und Funktionen, die sich nicht eindeutig einer anderen Klasse oder Struktur haben zuordnen lassen, da sie wiederholt von unterschiedlichen Klassen referenziert werden. Unter anderem befinden sich in ihr sämtliche *Typ*-Enums. Diese Reihe von Enums dient der Unterscheidung der verschiedenen Spielelemente, so zum Beispiel das *Spelltyp*-Enum, welche eine Aufteilung in einen *Fireball*- und *SummonImp*-Spell ermöglicht. Auch das *Gamestate*-Enum ist hier enthalten, welches für die GUI-Steuerung unerlässlich ist und beispielsweise dazu dient zu entscheiden, ob sich der Spieler gerade im *MainMenu* oder in der *Highscore*-Anzeige befindet. Des weiteren werden an dieser Stelle sämtliche im Spiel verwendeten Texturen und Model über eine *LoadContent*-Methode in das Spiel geladen. Auf diese Weise brauchen andere Klasse nur auf das jeweilige Objekt verweisen, um diese zu zeichnen, ohne diese noch einmal zu instanziieren. Aufgrund dessen lässt sich die *Vars_Func* als eine Spiel-Objekte-Bibliothek beschreiben. Außerdem enthält sie Funktionen die vor allem für die Vereinfachung der Mausinteraktion genutzt werden.

3.3 Spielelogik

Die Logik dient dazu das Spiel zu "beleben". Hierbei wird ein Großteil der Berechnungen durchgeführt von denen der Spieler im Spiel nichts direkt mitbekommt, sondern deren Auswirkungen auf die Spielwelt er nur indirekt über die Aktionen der einzelnen Entitäten wahrnehmen kann.

3.3.1 AI:

In der AI wird die Aktion der *Creature*- und *Imp*-Objekte geregelt, wobei beide jeweils eine eigene Funktion verwenden.

a) Imp-AI:

Die *Imps* verwenden das *Jobsystem* um sich ihre Aufgaben zu suchen und diese auszuführen. Zu beginn jeder Update-Base, welche für jeden *Imp* über die *Update*-Funktion der *Map* aufgerufen wird, wird eine Zeit-Zähl-Variable um den seit der letzten Aktualisierung verstrichen Zeitraum inkrementiert. Wenn diese Zeit-Zähl-Variable den Wert von einer halben Sekunde und weniger aufweist, passiert nichts weiteres. Dies dient der Entlastung des Systems und sorgt dafür, dass der *Imp* nur jede halbe Sekunde eine neue Aktion ausführen kann. Sollte die halbe Sekunde erreicht sein, überprüft die *Imp*-AI ob dieser bereits einen Job in Bearbeitung hat. Wenn dies nicht der Fall ist, wird ein neuer *Job* aus der globalen, zu erledigenden Job-Queue der *Map* genommen, dem arbeitslosen *Imp* zugewiesen und der *Job* in die Liste der sich in Arbeit befindenden *Jobs* der *Map* übertragen.
Anschließend wird der Weg vom *Imp* zum Arbeitsplatz mit Hilfe einer Breitensuche über die Hexagone berechnet. Sollte dabei herauskommen, dass der Arbeitsort nicht zu erreichen ist, wird der *Job* wieder an die Queue angehängt und der zuvor beschriebene Vorgang wird so lange wiederholt, bis der *Imp* einen erreichbaren *Job* gefunden hat. Jedoch nicht mehr als 20 mal. Hat der *Imp* einen passenden *Job* zugewiesen bekommen, beginnt er den berechneten Weg schrittweise abzulaufen, wobei er jede halbe Sekunden nur einen Schritt macht. Wurde keine erreichbare Aufgabe gefunden, wird dem *Imp* ein sogenannter *Idle*-Job zugewiesen, welcher dafür sorgt das ein *RandomWalk*-Methode ausgeführt wird. *RandomWalk* verschiebt dabei den zugehörigen *Imp* auf ein zufälliges freies Nachbarhexagon. Nachdem der *Imp* seinen Arbeitsplatz erreicht hat arbeitetet er maximal fünf Sekunden an dem Job, bevor er automatisch mit der Suche nach einem neuen *Job* beginnt.
Was genau der *Imp* bei jedem *Job* tut und wann dieser abgeschlossen ist, wird im Abschnitt *Jobsystem* näher erläutert.

b) Creature-AI:

Die *Creature-AI* verwendet anders als die *Imps* kein externes System, da das Verhalten der *Creatures* weit weniger komplex ist.
Ähnlich wie bei den *Imps* wird in der *Update*-Funktion zunächst eine Zeit-Zähl-Variable erhöht. Während der Aktionsintervall der *Imps* auf genau eine halbe Sekunde festgelegt ist, agiert eine *Creature* abhängig von seiner *Speed*-Variable, wobei diese die Anzahl an Aktionen pro Sekunde angibt.
Die *Creature-AI* überprüft nun mit Hilfe einer Breitensuche über die Hexagone, ob sich innerhalb

der eigenen Sichtweite eine angreifbare, andere *Creature* aufhält. Wenn dies der Fall ist, wird zunächst überprüft ob das Ziel sich auf einem angrenzend Hexagon, also in einer tatsächlichen Angriffsreichweite befindet und gegebenenfalls angegriffen. Fällt bei dem Angriffsziel dadurch die Lebensanzeige auf 0 wird die entsprechende *Creature* der List der zu entfernenden Objekte der *Map* übergeben und später durch deren *Remove*-Funktion aus dem Spiel entfernt. Sollte sich das Ziel hingegen außerhalb der Angriffsreichweite befinden, wird wieder mittels Breitensuche ein neuer Pfad zu dem Ziel berechnet. Wurde kein Weg zum Ziel gefunden so führt die *Creature* ebenfalls einen *RandomWalk* aus, welcher analog wie bei den *Imps* funktioniert. Sollte keine angreifbare *Creature* gefunden werden, wird, sofern *Creature* nicht schon über einen Pfad verfügt oder sich innerhalb eines fünf Hexagon Radius um den Zielpunkt seines Nests befindet, mithilfe einer Breitensuche über die Hexagone ein Pfad zu diesem ermittelt und ein Schritt auf diesem ausgeführt.

3.3.2 Jobsystem:

Die *Job*-Klasse kapselt die Arbeitsaufträge für die *Imps* und besitzt einen *Job*-Typ, welche die Art der Aufgabe festlegt, eine Zielposition, welche den Arbeitsplatz definiert, sowie die eigentliche *WorkTime*, welche von der *Imp*-AI dafür verwendet wird um die Zeit bis zum nächsten Jobwechsel zu bestimmen. Der *Job*-Typ gliedert sich in sechs Kategorien, wobei es sich bei einem um den *Idle*-Job handelt, welcher nur in der AI, nicht aber direkt im Jobsystem selbst, benutzt wird.

Bei den fünf verbleibenden Typen und ihren Funktionalitäten handelt es sich um die folgenden:

- *Harvest:*

  Das Ernten der Nahrung wird automatisch von Farmen in Auftrag gegeben und führt dazu, dass der *Imp* zum gewünschten Ziel läuft und dort die Nahrung von der Farm in den allgemeinen Vorrat überträgt. Der Job wird beendet, wenn die Farm abgeerntet ist.

- *Feed*:

  Das Füttern der Nestern wird automatisch von einem der Nester in Auftrag gegeben und führt dazu, dass der *Imp* Nahrung aus dem allgemeinen Vorrat in das Nest überträgt. Dabei ist der Auftrag zu Ende, wenn die Nahrungsvorräte des Nest wieder voll aufgefüllt sind.

- *Mine:*

  Das Abbauen von normalen Wänden wird vom Spieler durch das Markieren der entsprechenden Wände in Auftrag gegeben und führt dazu, dass der *Imp* zur markierten Wand läuft und diesen abbaut. Der Job gilt als beendet, wenn die Lebenspunkt der Wand auf 0 gesunken sind.

- *MineGold*:

  Funktioniert analog wie das zuvor beschriebene Abbauen der normalen Wände. Der einzige Unterschied liegt darin, dass während des Abbaus beständig Gold von der Wand in den allgemeinen Vorrat übertragen wird.

- *MineDiamond*:

  Funktioniert analog wie ein *MineGold*-Job, mit der Ausnahme, dass das Gold sehr viel langsamer generiert und das Objekt nie zerstört wird.

Gilt eine *Job* als beendet, so wird dieser von der Liste der sich in Arbeit befindenden *Jobs* in die List der erledigten Jobs der *Map* geschrieben und aus dem *Imp* gelöscht. Für die erledigten Jobs in der List wird dann abschließend die *EndJob*-Funktion aufgerufen. Diese setzt die Abhängigkeiten der *Job*-Variablen zurück, was im Fall eines *Mine*- oder *MineGold*-Jobs bedeutet, dass farbliche Markierung der Wände entfernt werden.

3.3.3 Mapgenerator:

Die *Mapgenerator*-Klasse wird beim Starten einer jeden Spielrunde aufgerufen und generiert eine neue *Map.* Hierfür baut er sich eine Liste mit Spezialobjekten welche er platzieren soll zusammen. Zu diesen zählen unter anderem der erste Oberflächeneingang und das *HQ* des Spielers. Außerdem eine bestimmte Anzahl an Gold- und Diamantvorkommen, welche beim Funktionsaufruf des *Mapgenerators* übergeben wird, genauso wie die eigentliche Größe der *Map*.
Ausgehend davon wird in einem ersten Schritt die gesamte *Map* mit normalen Wänden gefüllt, ausgenommen an den Stellen wo das Modulo 5 von X und Y Koordinate 0 ergibt. An diesen wird stattdessen ein zufälliges Objekt aus der Spezialobjekt-Liste eingefügt.
Anschließend wird zwischen, dem auf diese Weise gesetzten, Oberflächeneingang und dem HQ ein Pfad ermittelt. Darauf Aufbauend werden alle dazwischenliegenden Objekte sowie die Nachbarhexagone um den Eingang und das *HQ* gelöscht damit die gegnerische Held immer einen Weg zum *HQ* des Spielers finden. Abschließend werden alle Felder, welche um die bereits platzierten Gold- und Diamantvorkommen liegen, ebenfalls in Gold-Wände umgewandelt um so größere Goldadern zu erzeugen.

3.3.4 Wave/Wavecontroller:

Die *Wave*- und *Wavecontroller*-Klasse gehören zu dem Wellenverwaltungssystem, das festlegt wann, wie viele und wie starke Gegner aus den Oberflächeneingängen spawnen. Dabei werden die aktuellen Wellen in der *Map* gespeichert und entsprechend geupdatet. Die Welle selbst speichert dabei das Anfangsalter, welches die primäre Stärke der erzeugten Gegner repräsentiert, sowie die noch zu spawnende Anzahl an Gegnern. Der *Wavekontroller* wiederum erstellt nach dem Verstreichen einer bestimmten Zeit eine neue Gegner-Welle und speichert diese in der *Map*. Die Stärke und die Anzahl der Gegner hängt dabei von der Menge der schon zuvor erstellten Wellen ab, welche im *Wavecontroller* hinterlegt sind.

3.3.5 Spells:

Die *Spells*-Klasse speichert die Kosten der Zauber sowie deren Wirkung. Im Spiel selbst existieren aktuell nur folgende Zauber:

- die Impbeschwörung, welche einen neuen *Imp* spawnen lässt und mit jeder Anwendung teurer wird
- der Feuerball, welcher in einem kleinen Gebiet sowohl gegnerischen als auch eigenen Kreaturen Schaden zufügt

Beide Zauber werden durch den *castSpell*-Befehl ausgelöst und werden dabei nur durch einen Übergabeparameter unterschieden.

3.3.6 Player:

In der *Player*-Klasse werden die allgemeinen Vorräte des Spielers wie Gold, Nahrung und Mana gespeichert. Diese Vorräte können dann entweder für das Füttern von Nestern, dem Bauen von Gebäuden oder das Wirken von Zauber verwendet werden. Außerdem wird hier die derzeitige Punktzahl - der Score - des Spielers gespeichert. Auch verfügt die *Player*-Klasse über Funktionen um die Vorräte zu überprüfen, die Highscore-Liste einzusehen und zu speichern, das Spiel in den Vollbildmodus zu schalten sowie die Spiel-Hilfe ein- oder auszuschalten.

3.3.7 Interaction:

In der *Interaction*-Klasse wird ein Großteil der vom Spieler durchgeführten Interaktion behandelt. Die Interaktion findet seitens des Spieler dabei über Mausklicks und Tastatureingaben statt. Auf Softwareseite wird durch mehrere *Gamestates* unterschieden, welche Aktion im Spiel gerade ausgeführt wird. Entsprechend dieses *Gamestates* werden außerdem die *GUI-Button* abgefragt und deren Bildschirmkoordinaten mit der Mausposition im Falle eines Mausklicks verglichen, um bei Übereinstimmung die entsprechende Aktion durchzuführen. Einige Aktionen sind zusätzlich als Hotkeys über die Tastatur durchführbar. Die genau Tastenbelegung lässt sich im *Settings-Menu* nachschlagen.

3.4 Rendering

Aufgrund des Ausscheidens eines Teammitgliedes musste besonders der Umfang im Bereich Rendering deutlich gekürzt werden. Während ursprünglich noch geplant war, für das nachfolgend vorgestellte Beleuchtungssystem einen eigenen Shader zu schreiben mussten diese Pläne leider kurzfristig verworfen werden. Stattdessen fokussierten wir uns darauf, die bereits vorhandenen Möglichkeiten der Beleuchtung in XNA zu nutzen.

Auch war die anfängliche Idee mit Normalamaps zu arbeiten welche gerade bei stark strukturierten Materialien, wie beispielsweise der Stein- oder Boden-Texture, deutliche grafische Verbesserungen mit sich bringen.

3.4.1 Camera:

Die *Camera*-Klasse kapselt alle notwendigen Information der eigentlichen Spiel-Kamera und basiert auf der standardisierten *CreatePerspectiveFieldOfView*-Methode zum Aufspannen einer Projektionsmatrix. Die hierbei verwendeten Daten sind:

- die Position der Kamera (*cameraPosition*)
- die Position auf die die Kamera ausgerichtet ist (*cameraTarget*)
- ein Vector, der angibt, in welche Richtung die obere Seite der Kamera zeigt (*upVector*)
- die Projektionsmatrix, die sich zusammensetzt aus:
  - dem Öffnungswinkel der Kamera (*fieldOfView*)
  - dem Seitenverhältnis der Bildschirmgröße (*aspectRatio*)
  - der Entfernung zur Kamera ab der etwas gezeichnet werden soll (*nearClip*)
  - der Entfernung zur Kamera ab der nichts mehr gezeichnet werden soll (*farClip*)
- sowie die Seitenlänge des Spielfeldes

Alle genannten Parameter werden dabei einmalig bei der Instanziierung der Kamera dem Konstruktor übergeben. Abbildung 2 veranschaulicht noch einmal die Bedeutung der jeweiligen Werte:

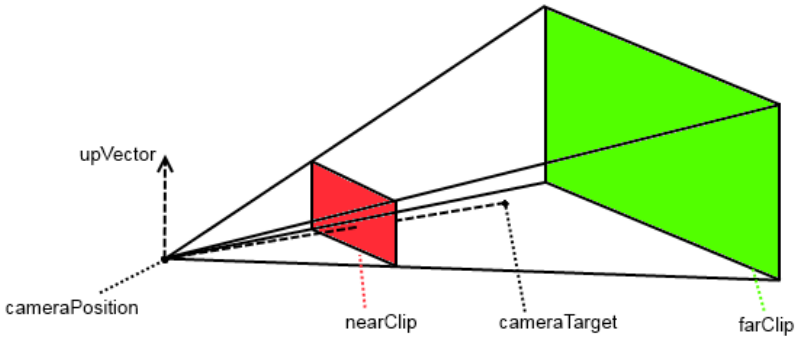


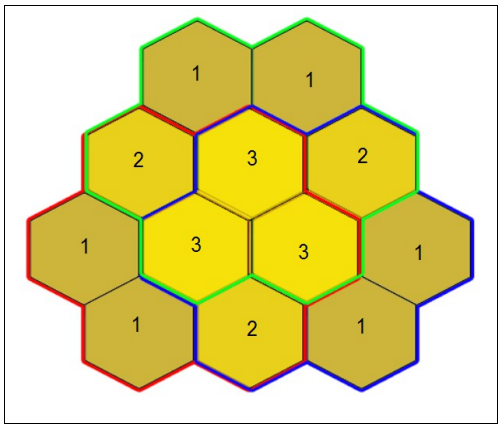
Abb. 2. Kameraaufbau

Zusätzlich verfügt die *Camera* über eine *Update*-Funktion, welche die Tastatureingaben kontrolliert und die Kamera dementsprechende über die Spielwelt bewegt. So kann der Spieler die Kamera mit den Tasten W, A, S und D entlang der X- und Y-Achse steuern und mit den Tasten Q und E um die Z-Achse rotieren. Außerdem wird in der *Update*-Funktion sichergestellt, das die Kamera, wenn sie an den "Rand" des Spielfeldes stößt, wieder auf die andere Seite des Spielfeldes gesetzt wird. Auf diese Weise entsteht der Torus-Effekt.

3.4.2 Beleuchtungssystem:

Das Beleuchtungssystems baut in weiten Teilen auf der, durch XNA bereitgestellten, *BasicEffect*-Klasse auf. Dabei handelt es sich um eine Art Standard-Shader, welcher schon für die Darstellung der Model zum Einsatz kam. Darüber hinaus verfügt die Klasse über weitere, festgelegte Schnittstellen. Diese bringen ein einfaches Ambient-, Diffuse- und Specular-Lighting mit sich und lassen sich schnell und effektiv konfigurieren. Der Nachteil dieses, eher rudimentären, Rendersystems ist, dass die Beleuchtung für jedes Model separat gesetzt wird und es so keine Möglichkeit gibt einen Modelübergreifenden Lichteffekt zu erzielen.

Für die Beleuchtung selbst wurde eine simple *LightModel*-Klasse angelegt. Analog wie zu den vorhanden Model-Klassen fungiert diese ebenfalls als Wrapper und stattet das eigentliche Model mit einigen, erweiternden Funktion aus. Bei dem Model selbst handelt es sich, wie in Abbildung 3 (b) zu sehen, um ein einfaches Säulen-Paar mit Fackeln. Um das Brennen der Fackeln bzw. das Vorhandensein von Feuer zu simulieren, implementiert die *LightModel*-Klasse außerdem ein Partikelsystem, dessen Aufbau und Funktionsweise aber erst im nachfolgenden Abschnitt erläutert wird.
Die Kern-Funktionalität der *Light*-Klasse besteht darin, ausgehenden von dem Hexagon auf welchem die Klasse instanziiert wurde, iterativ die Hexagon bis zur vierten Nachbarschaft abzuarbeiten und jedes Hexagon mit einem festen Beleuchtungswert zu versehen. Die Arbeitsweise des Algorithmus ist dabei denkbar einfach und orientiert sich vom Prinzip her an der klassischen, additiven Farbmischung. Wie in Abbildung 3 (a) vereinfacht dargestellt, iteriert jedes Hexagon über all seine direkten Nachbar und addiert diesen einen festgelegten Beleuchtungswert auf. Der addierte Wert nimmt dabei von innen nach außen kontinuierlich ab, so dass der Eindruck eines schwächer werdenden Lichts entsteht. Der auf diese Weise berechnete und im Hexagon gespeicherte Beleuchtungswert wird an die jeweiligen Model weiter gereicht und beim Rendern an die *BasicEffect*-Klasse übergeben.



(a) Grobkonzept des additiven Beleuchtungsmodel

(b) Tatsächlich Beleuchtung im Spiel (mit Säulen-Model) nach dem Anlegen eines Raumes

Abb. 3. Beleuchtungsmodel

Die Instanziierung der *LightModel*-Klasse erfolgt identisch zu den anderen statischen Model-Objekten, mit der Ausnahme, dass dieses nur an ausgewählten Stellen im Spiel instanziiert werden. Diese sind: beim Aufziehen und Anlegen eines Raumes, entlang des Start-Weges vom Oberflächeneingang zum HQ, sowie Oberflächeneingang und HQ selbst.

3.4.3 Partikelsystem:

Das Partikelsystem setzt sich aus der *Particel*- und der *ParticelGernator*-Klasse zusammen und wurden als Umsetzung des Observer-Patterns implementiert. Näheres hierzu ist dem TDD zu entnehmen.

Grundsätzlich wird das Partikelsystem, wie im zuvor beschrieben Beleuchtungssystem, immer durch eine bereits bestehende Model-Klasse implementiert. Der Grund hierfür ist, dass der *ParticelGenerator* über keinen eigene Position im dreidimensionale Raum verfügt, sondern immer relativ zu anderen Modelen gezeichnet werden muss. Das bedeutet im Klartext, dass das Objekt, welches das Partikelsystem nutzt will, in seiner eigenen *Update*-Funktion die *Update*-Funktion des Generators aufrufen muss, ebenso wie in der *Draw*-Funktion der Zeichen-Befehl ebenfalls an diesen weiter propagiert werden muss.
Für die Erzeugung des *ParticelGernators* muss dem Konstruktor das Model der Partikel, die gewünschte Anzahl der zu erzeugenden Element sowie deren durchschnittliche Lebensdauer übergeben werden. Bei den Partikel-Modellen handelt es sich grundsätzlich um einfache, zweidimensionale, teilweise transparente Texturen, welche auf eine flache Plane aufgebracht sind.

Zum bessern Verständnis zeigt Abbildung 4 das Feuer-Partikel-Model, welches im bereits zuvor besprochen *LightModel* zum Einsatz kommt:

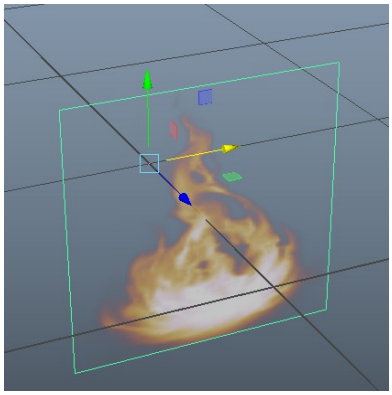


Abb. 4. Feuer-Partikel-Model

Der *ParticelGenerator* selbst übernimmt reine verwaltungstechnische Aufgaben und ist für das Erzeugen, Updaten und Zeichnen der *Particel*-Objekte verantwortlich. Die tatsächliche Logik wird durch die eigentliche *Particel*-Klasse von jeder der erzeugten Instanzen realisiert. Deren primäre

Aufgabe ist es, dass Partikel gezielt oder zufällig zu rotieren, zu skalieren und zu transponieren sowie die Lebenszeit und den damit verbundenen Alpha-Wert des zu zeichneten Partikel-Models, über eine gewisse Zeitspanne, zu verringern. Ist das Partikel vollständig verblasst, d.h. die Lebenszeit hat den Wert 0 erreicht, so löscht es sich selbständig aus dem Verwaltungssystem des Generators, was diesen dazu veranlasst ein neues Partikel zu erzeugen und der gesamte Kreislauf beginnt von neuem.

3.5 Animation

Damit das Einbinden und anschließende Abspielen von verschiedenen Animations-Clips, auch aus unterschiedlichen Quellen, reibungslos funktioniert mussten dafür zunächst einmal, in einem ersten Schritt, die dafür notwendigen Infrastrukturen geschaffen werden. Dazu musst zunächst eine Datenstruktur gefunden werden, die alle benötigten Animationsdaten in einer möglichst vorteilhaften Art speichert und diese anschließend wieder möglichst leicht für die Durchführung der Animation nutzbar macht.

3.5.1 Clip:

Auf der niedrigsten Ebene wurde hierfür eine einfache *Clip*-Klasse angelegt. Dessen Aufgabe ist es, den eigentlichen Inhalt einer einzigen Animation zu speichert und zu verwalten, so dass die Daten später leicht für die Durchführung der eigentlichen Animation genutzt werden können. Ein Clip setzt sich dabei aus einem Name, einer festen Dauer und einer Reihe von *Bone*-Objekten zusammen. Bei diesen handelt es sich um die eigentliche Struktur auf der die Animation ausgeführt wird. Die *Bone*-Klasse selbst besteht dabei nur aus einem Name, als Identifikation und Link zum animierenden Submesh des Models und einer Liste von *Keyframes*. Ein *Keyframe*, ebenfalls als Klasse implementiert, definiert dabei analog wie zum Beispiel ein Bild in einem klassischen Daumenkino nur einen kurzen Moment in dem eigentlichen Animationsverlauf. Dabei ist ein *Keyframe* nichts weiteres als Rotations- und Translationsinformation für den *Bone* zu einem bestimmten Zeitpunkt in der Animation.

3.5.2 ClipPlayer:

Aufbauend auf der Clip-Logik arbeitetet die *ClipPlayer*-Klasse. Diese ist für die Organisation und den eigentlichen Ablauf der Animation verantwortlich. Instanziiert wird der Player durch die *CharacterModel*-Klasse. Hierbei wird dem Konstruktor der abzuspielende Clip als Parameter sowie eine Referenz auf das zugrundeligende Model übergeben. Neben diesem primären Attribut verwaltet die Klasse eine einfache Positionsvariable, welche zur Bestimmung der momentanen Position im Clip dient. Darüber hinaus ist es mit der *Looping*-Property möglich den Clip in einer Endlosschleife spielen zu lassen.

Den Kern der Player-Klasse bildet, wie schon so häufig, die *Update*-Funktion. In dieser wird zunächst der Positionswert im aktuellen Clip sekundenweise erhöht. Unter Verwendung dieses Werts wird anschließend die momentane Stelle zwischen den beiden, gerade relevanten, *Keyframes* bestimmt. Anhand der so berechneten Position wird zwischen den Rotations- sowie den Tranlationswerten der beiden Frames linear interpoliert und dadurch die tatsächliche Orientierung ermittelt. Abschließend wird über die beiden Werte eine Matrix gebildet, welche auf die im *CharacterModel* enthaltenen *Bones* angewandt wird. Spezifiziert werden diese über die jeweiligen Identifikationsnamen in der *Clip*-Klasse.
Liegt der Positionwert hinter dem Zeitwert des letzten *Keyframe* im Clip und die Looping-Variable ist gesetzt, so wird die Position automatisch wieder auf 0 gesetzt und die Animation beginnt von neuem, anderenfalls stoppt diese.

3.5.3 CharacterModel:

Basierend auf der *Clips*- und der *ClipPlayer*-Klasse setzt sich die *CharacterModel*-Klasse zusammen, welche außerdem eine Ableitung der *BasicModel*-Klasse darstellt. Des weiteren bringt das Model auch erstmals die *Clip*- und die *ClipPlayer*-Klasse zusammen.
Analog wie bei dem *BasicModel* wird für die Instanziierung dem Konstruktor ein Model übergeben. Entscheidend an dieser Stelle ist, dass das Model zuvor nicht über die Standard-Pipeline-Methode eingeladen worden ist sondern über die eigens geschriebene *Animation-Import-Pipeline*. Um diese zu erreichen musste im *Content*-Ordner des Projekts der *Content Process* des Models von „*Model – XNA Framework*“ auf „*Animation Import*“ umgestellt werden. Auf diese Weise werden die benötigten Animationsinformationen aus den zu importierenden Daten gelesen und in das Model geschrieben, was zur eigentlichen Benutzung in der *CharacterModel*-Klasse gewrappt ist.
Als Attribute enthält die Klasse dementsprechend das Model selbst, eine Liste von Model-*Bones,* sowie den eigentlichen *ClipPlayer* und passend dazu eine Liste von *Clips*. Diese können über die Funktion *AddClip* hinzugefügt werden.
Während die *Update*-Funktion ausschließlich dazu dient den Player am laufen zu halten, fällt die *Draw*-Funktion etwas komplexer aus als noch die Implementierung im *BasicModel*. Um das Model und seine Animation korrekt darzustellen, müssen zuerst die durch den Player transformierten, *Bones* an das Skelett weiter propagiert werden. Das heißt die Rotations- und Translationswerte, welche beim Abspielen der Animation auf die *Bone*-Struktur angewendet worden sind, müssen ebenfalls auf das Skelett angewendet werden, an welchem wiederum die tatsächlich sichtbaren Körperteile (Arme, Beine, etc.) hängen. Anschließend kann wieder über die Submeshs des Models iteriert und das eigentliche Zeichnen vorgenommen werden. Zusätzlich zu dem bereits bekannte *BasicEffect* kommt für Visualisierung der Animation der sogenannte *SkinnedEffect* zum Einsatz. Dabei handelt es sich um eine durch XNA bereitgestellte Darstellungsmöglichkeit von *Skin*, also der Polygonteil des Models welches durch das Skelett animierbar gemacht ist. Wie bei dem *BasicEffect* wird diesem die View- und Projektionsmatrix der Kamera sowie eine

Orientierungsmatrix übergeben. Damit die Animation auch wirklich gezeichnet wird, wird dem *SkinnedEffect* darüber hinaus außerdem noch das zuvor transformierte Skelett übergeben.

3.5.4 Animation-Import:

Nachdem die grundlegende Datenstruktur implementiert war, musste in einem nächsten Schritt die eigentliche Datenbeschaffung vorgenommen werde. Um sämtliche Skelett- und Animationsdaten aus dem importierten Model zu extrahieren wurde einen neue *Animation-Import*-Klasse als Erweiterung der bestehenden, von XNA bereitgestellten, *ModelProcessing*-Pipeline geschrieben. Glücklicherweise stellt XNA hierfür einige Hilfestellung bereit, um das Verständnis für diesen Prozess zu maximieren und die eigentliche Umsetzung zu beschleunigen.
Der Aufbau der zu importierenden Daten, sei es nun ein einziges Model oder eine ganze Szene, besteht dabei immer aus einer Hierarchie von *NodeContent*-Objekten, welche einen Elternteil, eine Transformation und eine Menge von Kindern enthalten. Die Nodes selbst können unterschiedliche Typen haben und zum Beispiel vom Typ *BoneContent* oder *MeshContent* sein, welche allesamt Ableitungen von *NodeContent* darstellen. Ein *MeshContent*-Objekt setzt sich dann beispielsweise wiederum aus einer Sammlung von *GeometryContent*-Objekten zusammen usw. Mit diesem Wissen über den zugrundeliegenden Aufbau muss für den eigentlichen Import-Schritt nun noch die *Process*-Methode des *ModelProcess* überschrieben werden.

Die Abbildung 5 zeigt hierbei die Abfolge der einzelnen Schritte, welche nachfolgend genauer erläutert werden, sowie den Zusammenhang mit den zuvor beschrieben Klassen.

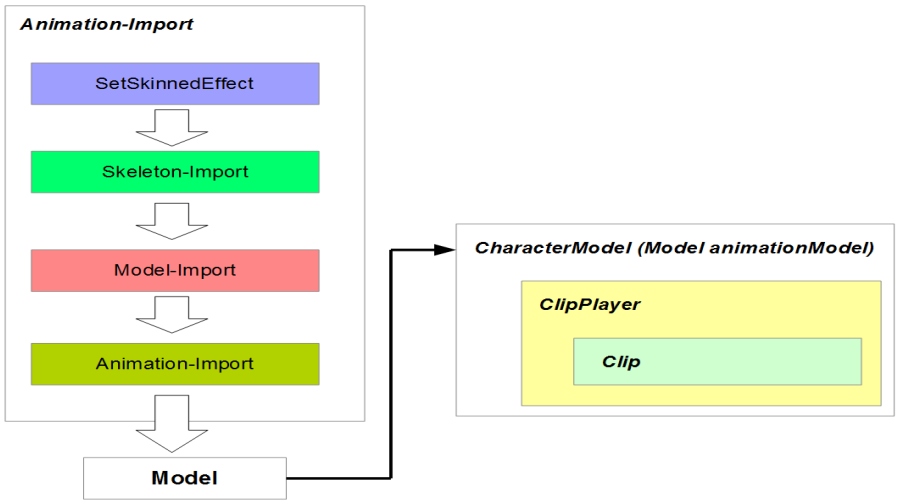


Abb. 5. Animation-Pipeline

a) SetSkinnedEffect:

Wie bereits erwähnt, müssen die animierten Submeshs des Models mit der *SkinnedEffect*-Klasse dargestellt werden. Damit diese aber überhaupt erst funktioniert, muss der standardmäßig verwendete *BasicEffect* bereits beim Import an den richtigen Stellen durch den *SkinnedEffect* ersetzt werden. Dazu werden sämtliche *GeometryContent*-Nodes durchsucht und darauf geprüft, ob sie einen *VertexChannel* mit einem *BoneWeight* besitzen. Um es kurz zu machen: Ein *BoneWeight* gibt prinzipielle an wie stark ein Mesh durch eine Animation beeinflusst wird. Hat folglich ein Mesh ein *BoneWeight* ist er für die Animation relevant und muss durch den *SkinnedEffect* gezeichnet werden, d.h. der *BasicEffect* wird an dieser Stelle ersetzt.

b) Skeleton-Import:

Da es sich bei der eigentliche Animationsart um Skelett-Animation handelt, werden im zweiten Schritt der Prozessierung die benötigten Skelettdaten aus der Import-Datei gelesen. Mittels der, intern durch XNA bereitgestellten, *FindSkeleton*-Funktion wird automatisch das *Root*-Element aus Skelett-Hierachie als *BoneContent*-Objekt zurückgegeben. Damit das Skelett später leichter wiederverwendet werden kann, wird außerdem das Skelett in eine Folge von eindeutigen Indizes zu den eigentlichen *Node*- bzw. *BoneContent*-Objekten verwandelt und in dem Model abgespeichert.

c) Model-Import:

Sind die oberen beiden Schritt getan, kann das tatsächliche Model aus den Daten geladen werden. Hierfür wird einfach die die *Process*-Funktion der Elternklasse aufgerufen.

d) Animation-Import:

In einem letzten Schritt werden nun die eigentlichen Animationsdaten extrahiert. Vorteilhafterweise bietet die *NodeContent*-Klasse bereits Möglichkeiten an wie man einfach und schnell an die notwendigen Informationen kommt. So sind im *NodeContent* Animation als *KeyValuePairs* hinterlegt. Hierbei stellt der *Key* den Clip-Namen dar und der *Value* einen sogenannten *AnimationContent,* aus welchem sich einfach und bequem die *AnimationKeyframes* auslesen lassen. Darauf aufbauend wird zunächst ein neuer *Clip* angelegt, welcher mit dem *Key*-Name und den Model-*Bones* versehen wird. Anschließend wird für jeden *AnimationKeyframe* ein normaler *Keyframe* erzeugt und die Dauer sowie die Orientierungsdaten von einem in den anderen kopiert. Zum Schluss wird der besagte Frame dann dem entsprechenden *Bone* im *Clip* zugeordnet. Sind alle *Keyframes* abgearbeitet wird der Clip, wie alle anderen Zusatzinformationen, im Model hinterlegt.

Ursprünglich war die Idee, die Pipeline dahingegen zu erweitern auch mehrere Clips aus einer Animation zu extrahieren. Leider ist dieses Vorhaben ebenfalls der notwendigen Kürzung zum Opfer gefallen, weswegen manche Model mehrfach vorhanden sind, da diese jeweils einen Animations-Clips mit sich bringen und somit den Content des Spiels nur unnötig aufblähen.

3.6 GUI

Das *GUI*-Sytem ist die grafische Abbildung der Gamestates auf eine objektorientierte Klassenstruktur. Die Klassen selbst implementieren dabei das eigentliche GUI-Layout und dessen Funktionalitäten im jeweiligen Gamestate. So wird beispielsweise im *Main-Menu*-State die besagte *Main-Menu-GUI*-Klasse aufgerufen. Diese kümmert sich anschließend sowohl um die grafische Darstellung, als auch um die Abarbeitung der Nutzereingaben, wie das Betätigen der *Settings*-Taste.

Sämtliche *GUI*-Klassen sind dabei von der abstrakten *BasicGUI*-Klasse abgeleitet, welche grundlegende Komponenten und Schnittstellen mit sich bringt. Als Komponenten werden dabei zwischen einfachen Textfeldern, grafischen Elementen und Schaltflächen unterschieden. Die Komponenten selbst sind dabei als Instanzen der sogenannten *GUI-Element*-Klasse implementiert, welche eigene Funktionen mit sich bringen. Die Nutzung dieser Klassenstruktur hat den Vorteil, dass jede Komponente selbst unabhängig von anderen Elementen ihre eigentliche Logik implementiert und verwaltetet. Durch die Wahl der entsprechenden Konstruktor-Parameter werden die jeweiligen *GUI-Element*, d.h Textfeld, Grafik oder Schaltfläche, instanziiert.

3.6.1 GUI-Element:

Als Standart-Attribut wird dem *GUI-Element*-Konstruktor immer eine Instanz der *Rectangel*-Klasse für die Positions- und Größenbestimmung des Elements sowie ein *GUI-Elementen*-Typ mit einem eindeutigen Namen übergeben. Letzt genannter Parameter ist nötig um wichtige Elemente, wie zum Beispiel die bereits zuvor genannte Settings-Schaltfläche, eindeutige zu identifizieren.
Das Textfeld wird über ein zusätzliches String-Attribut instanziiert, während das grafische Element analog dazu statt mit einem String mittels eines Texture2D-Parameters realisiert wird. Die Schaltfläche-Instanz bekommt bei der Erzeugung wiederum beide Attribute übergeben. Darüber hinaus besitzt die *GUI-Element*-Klasse ein *Color*- und ein *Enable*-Attribut, welche über *Properties* gesetzt werden können und standardmäßig den Wert *White* bzw. *true* haben.
Alle *GUI-Elemente* implementieren eine *Update*-Funktion um die Komponente zu Aktualisieren, sowie eine einfache *Draw*-Funktion auf Basis der *SpriteBatch*-Klasse. Handelt es sich bei dem *GUI-Element* um eine Schaltfläche, so verwaltetet die *Update*-Methode außerdem alles was mit der eigentlichen Interaktion durch den Nutzer zu tun hat. Dazu zählt unter anderem, das farbliche Hervorheben der Schaltfläche bei Auswahl. Außerdem wird wird beim Betätigen des Schalt-Elements eine interne Variable gesetzt, welche über eine einfache *GetPressedButton*-Funktion

abgefragt werden kann und den Element-Typ zurückliefert. Diese Art des Pollings dient später dazu auf der *GUI-Klassen*-Ebenen das Auslösen der verschiedenen Schaltflächen zu registrieren und zum Beispiel mit Änderung des Gamestates zu reagieren.
Weiter Funktionen die die Klasse bereitstellt, ermöglichen ein zeitgesteuertes Einblenden und Ausblenden. Hierbei wird in einem einstellbaren Zeitintervall der Alpha-Wert des Farb-Attributs linear zwischen den Werten 0 und 1 verkleinert oder vergrößert. Die letzte Funktionalität welche das *GUI-Element* mit sich bringt, ist die Möglichkeit einer Verschiebung entlang der X- bzw. Y-Achse. Dabei wird analog zu dem zuvor beschrieben Ein- und Ausblenden innerhalb eines Zeitintervalls von der ursprünglichen (Rechteck-)Position zu den, durch den Parameter angegebenen, Zielposition interpoliert. Auf diese Weise kommt das typische Ein- und Ausfahren der Element zustand, welcher sich im gesamten Spiel wiederfindet.

Zum besseren Verständnis soll Abbildung 6 den zuvor beschrieben Aufbau der *GUI-Element*-Klasse und Zusammenhang zu den *BasicGUI*-Klassen noch einmal verdeutlichen.

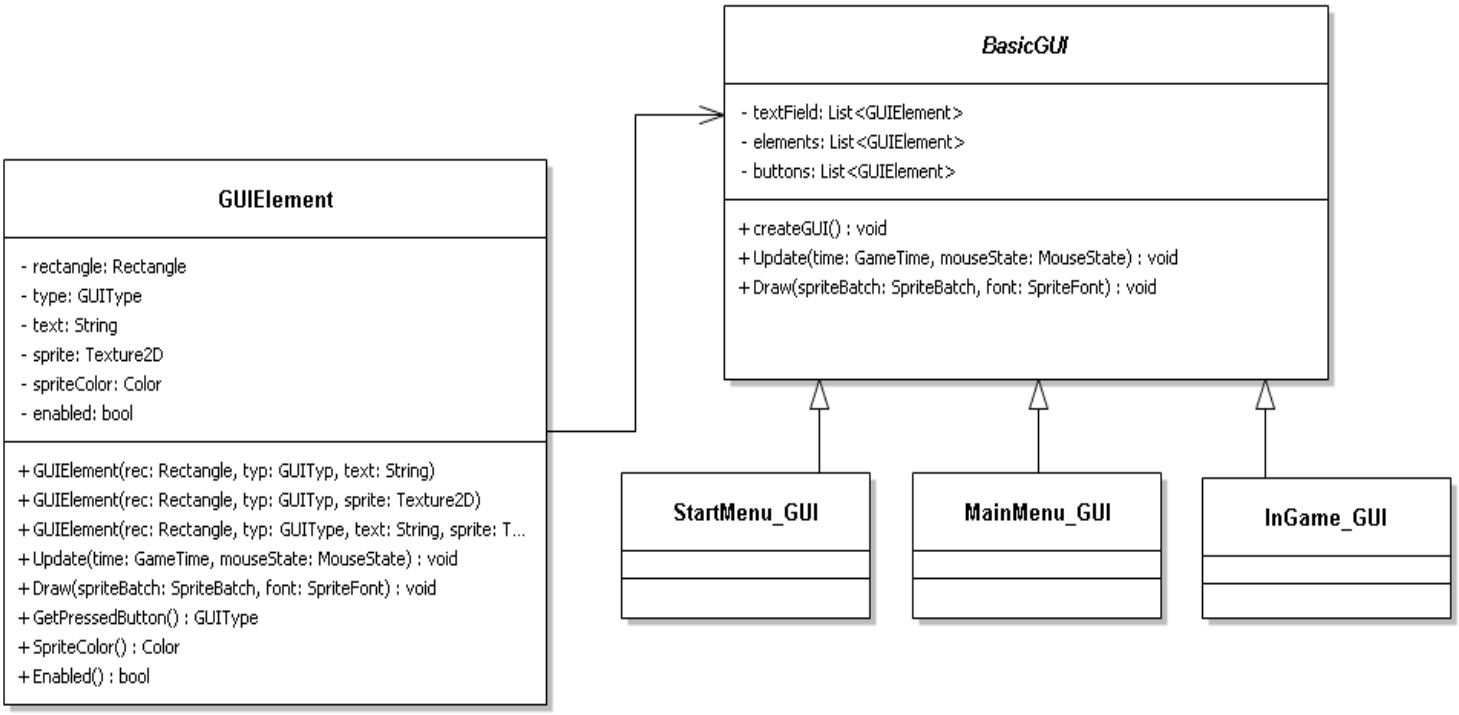


Abb. 6. Vereinfachtes *GUI-Element*- und *BasicGUI*-Klassendiagramm

3.6.2 BasicGUI:

Während die primäre Logik durch die eigentlichen *GUI-Elemente* realisiert wird, implementiert jede Ableitung der *BasicGUI*-Klasse ausschließlich die drei Funktionen *createGUI*, *Update* und *Draw*.
Dabei fungiert die *createGUI*-Funktion prinzipielle als *LoadContent*-Funktion und legt die verwendeten Komponenten, ihre Anfangsposition sowie den eigentlichen Aufbau der GUI fest.

Gleichzeitig spiegelt dabei die Instanziierung-Reihenfolge der Element die Abfolge wieder, in welcher die Komponenten gezeichnet werden. Das heißt die zuerst instanziierten Element befinden sich am weitesten hinten, da diese ebenfalls als erstes gezeichnet werden.
Die *Update*-Funktion koordiniert und steuert lediglich das Ein- und Ausfliegen sowie das Ein- und Ausblenden der Element und propagiert diese an die Logik der Element-Ebene weiter, wo anschließend die eigentliche Umsetzung des gewünschten Effekts stattfindet. Außerdem horcht die Update-Logik, durch ständiges Polling auf die *GetPressedButton*-Funktion, auf die Betätigung einer der Schaltflächen und löst gegebenfalls eine Gamestate-Änderung aus.
Die *Draw*-Funktion dagegen iterieren ausschließlich über die zuvor angelegten Elemente und ruft bei diesen die jeweilige benötigte *Draw*-Funktion auf.

3.6.3 Beispielhafter Aufbau:

Abschließend soll in Abbildung 7 kurz der schichtweise, grafische Aufbau und die Funktionsweise der *StartMenu*-GUI präsentiert und erläutert werden:

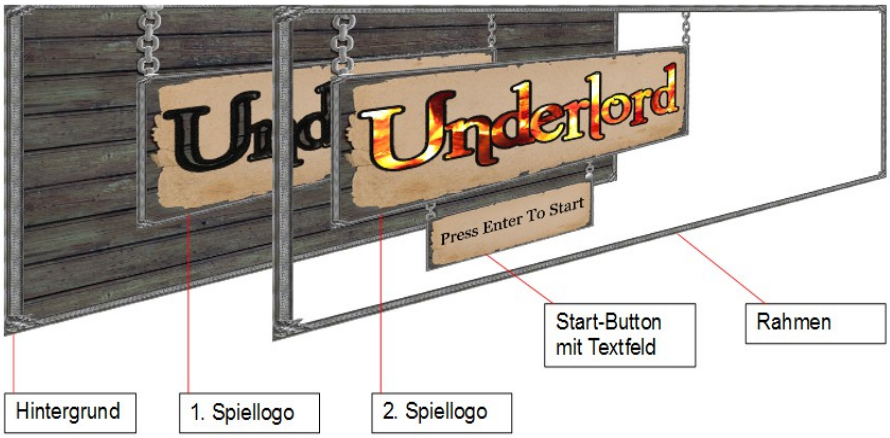


Abb. 7. Schichtweiser Aufbau der StartMenu-GUI

Wie der Abbildung zu entnehmen ist, stellt sich die *StartMenu*-GUI im Kern aus insgesamt 5 Komponenten zusammen: vier grafischen Elementen und einer Schaltfläche. Hierbei stellt der *Hintergrund* und der *Rahmen*, d.h. die beiden obersten und untersten Schicht der GUI, die einfachsten, grafischen Elemente dar, welche vollkommen ohne jegliche Logik auskommen. Die beiden Element *1.Spiellogo* und *2.Spiellogo* sind zwar auch nur einfache Grafiken, werden aber zunächst eingeflogen und anschließend geschickt übereinander geblendet, wodurch der Eindruck eines aufleuchtenden Logos entsteht. Einzig bei dem *Start-Button* handelt es sich, wie der Name bereits sagt, um eine Schaltfläche. Diese wird zu Beginn ebenfalls eingeflogen und löst bei Betätigung der Fläche mittels Maus oder Tastatur eine Gamestate-Änderung aus. Dabei werden zunächst alle ausgefahrenen Elemente wieder in ihre ursprüngliche Position gebracht, bevor tatsächliche von dem aktuellen *Start*-States auf den *MainMenu*-State umgeschaltet wird.

Der Aufbau der anderen GUI-Klassen ist dabei analog und besitzt grundsätzlich immer die beiden Hintergrund- und Rahmen-Komponenten als erste und letzte gezeichnete Grafik. Einzige Ausnahmen bilden die Ingame-GUI-Klassen, welche keinen Hintergrund benötigen da dieser durch das eigentliche Spiel ersetzt wird. Außerdem implementiert diese Klasse eine Schnittstelle für die Darstellung der Minimap um diese auf der richtigen Zeichen-Ebene zu setzten.

3.6.4 Minimap:

Die *Minimap*-Klasse wurde in ihren Grundzügen von Mareen erstellt als diese noch Mitglied des Teams war und anschließend von Patrick mehrfach überarbeitet und angepasst. In ihrer Endversion stellt die *Minimap* das gesamte Spielfeld über eine Farbkodierung dar. Dabei sind die Felder der *Minimap* zwar Quadrate, können aber dadurch das jede zweite Spalte um ein halbes Quadrat nach oben verschoben ist trotzdem den Hexagonaufbau des Spielfeldes gut approximieren.

4. Fazit

Abschließend kommen wir nun zum Fazit des Projekts.

Zu Beginn sollte an dieser Stelle gesagt werden, dass wir uns eindeutig mit dem ursprünglich geplanten Umfang des Projekts übernommen haben, da wir die für das Projekt benötigte Zeit sowie den Arbeitsaufwand für die anderen Semesterveranstaltungen unterschätzt haben. Zusätzlich kommt hinzu, dass das Projekt zunächst für vier Personen ausgelegt war und wir nach dem Ausscheiden eines Mitgliedes nicht Konsequent genug gekürzt haben, auch weil wir gewisse Aspekte des Spiel unbedingt realisiert sehen wollten.
Des Weiteren kam es zu gelegentlichen Missverständnissen zwischen den Mitgliedern im Bezug auf das Aussehen und das spielerische Verhalten des Projekts, was allerdings immer schnell und ohne große Probleme geklärt werden konnte.
Ein weiteres Problem war die Aufsetzung eines, für die Projektplanung sinnvollen, Redmine-Servers, die sich immer wieder aufgrund diverser technischer Schwierigkeiten verschob. Letztendlich gelang dies allerdings doch noch und so konnten die letzten Schritte des Projekts sauber durchgeplant werden. Diesem Umstand ist es auch zu verdanken, dass trotz eines krankheitsbedingten Ausfalls kurz vor der finalen Abgabe, weite Teile des geplanten Umfangs rechtzeitig fertiggestellt werden konnten.

Zusammengefasst kann gesagt werden, dass die Kernelemente des Spiel erfolgreich implementiert werden konnten, obwohl das Spiel sich mit Fortschreiten des Projekts immer weiter von dem Ursprungskonzept entfernt hat. Nichts desto trotz ist das gesamte Team mit dem entstanden Spiel sehr zufrieden, auch wenn es natürlich nicht ganz fehlerfrei ist. Und obwohl das gesamte Unterfangen eine höllische Arbeit war, die wirklich fast jeden Aspekt und jede Problemstellung eines richtigen Spiels abgedeckte, hatten doch alle Beteiligten sichtbaren Spaß bei der Umsetzung. Wir hoffe daher, sehr dass das Spiel auch anderen Leute Spaß macht und Freude bereitet.