MIB-Projekt SS 2012:

LastMan Dokumentation

Betreuer:

Prof. Dr. Jens-Uwe Hahn

Norman Pool

Studenten:

Jan Horak (22946)

Marius Oehler (22961)

Stanislaw Wolf (22972)

Inhaltsverzeichnis

Motivation ... 1

Projektdefinition ... 1

Projektziele ... 1

Technologien ... 2

Konzeption ... 2

Planungen ... 2

Spielmodus ... 2

Obefläche ... 3

Architektur ... 3

Manager ... 3

Interfaces ... 5

GameObject ... 6

QuadTree ... 6

Config-Files ... 7

Die Karte ... 7

Wegfindung der Gegner ... 8

GUI ... 9

Die Hauptansicht ... 9

Inventar ... 10

Konstruktor ... 11

Ergebnis ... 12

Erzeugung der Gegner ... 12

Aufgetretene Probleme und deren Lösung ... 13

Erfahrungen ... 14

Ausblick ... 14

Anhang ... 16

Abbildungsverzeichnis ... 16

Motivation

Im Rahmen unseres Studiums der Medieninformatik wird uns die Möglichkeit gegeben, Projekte als Studienleistung durchzuführen. Diese Möglichkeit wollten wir nutzen, in dem wir ein Computerspiel entwickeln.

Im dritten Semester besuchten wir ein Wahlmodul “Entwicklung von Computerspielen”. Als Prüfungsleistung in diesem stand zur Auswahl, entweder ein Mini-Game zu entwickeln, oder ein Game-Design-Script zu erstellen. Wir entschieden uns für das Letztere und legten auch gleichzeitig fest, dieses Spiel im vierten Semester im Rahmen eines Projektes umzusetzen. Diese Kombination ergab sich, nachdem wir festgestellt haben, dass wir einerseits das während der Vorlesung erlangte Theoriewissen unbedingt in Praxis umsetzen wollten, das Mini-Game dafür aber zu klein wäre, andererseits wird die gesamte Arbeit so in zwei Schritten erledigt und wir nach dem Erstellen des Skriptes eine gute Grundlage für die Implementierung des Spiels haben.

Projektdefinition

Projektziele

Das Hauptziel für das erfolgreich abgeschlossene Projekt war, einen Prototyp des Spiels an der MediaNight der HdM präsentieren zu können. Die dafür notwendigen Unterpunkte waren:

- Auslesen der Karte aus einer Textdatei
- Selbständige Bewegung der Gegner auf der Karte
- Bewegung des Spieler und dessen Kollision mit den Wänden
- Der Spieler und die Gegner müssen schießen können und der dadurch entstandene Schaden muss mit den jeweiligen Lebenspunkten verrechnet werden
- Der Spieler muss auf der Karte verteilte Gegenstände sammeln können

Zusätzlich wurden folgende Ziele bis zum Ende erreicht:

- Gegner suchen auf der gesamten Karte nach dem Spieler, um ihn anzugreifen
- Gegenstände erscheinen an unterschiedlichen Orten
- Verschiedene Gegenstände sind zu neuen Waffen kombinierbar

- Einlesen von Gegner-, Itemattributen aus einer XML-Datei
- Speichern und Laden der Spielkonfiguration in/aus einer ini-Datei

Technologien

Als Programmierumgebung für die Entwicklung des Spieles wurde Visual Studio 2010 von Microsoft, in Kombination mit dem Spieleentwicklungs-Framework XNA (XNA's Not Acronymed), welches auf der Programmiersprache C# basiert, verwendet.

Für die Speicherung der Spielobjekte (Gegner, Items,..) wurde die Markup Language XML genutzt.

Bei der Entwicklung der Grafiken kam die Adobe Creative Suite CS5 mit Photoshop und Illustrator zum Einsatz.

Damit alle Projektdateien bei allen beteiligten Personen immer auf dem neusten Stand waren und eine regelmäßige Sicherung durchgeführt wurde, ist die Versionsverwaltungssoftware Git zum Einsatz gekommen.

Konzeption

Planungen

Ein großer Teil der Planung und ersten Orientierung für das Ziel des Projekts geschah schon im bereits genannten Game-Design-Script aus dem Modul 'Entwicklung von Computerspielen'.

Diese sehr ausführliche Grundlage bot uns die Möglichkeit direkt mit der gezielten Planung zu beginnen. Im ersten Schritt der Planung befassten wir uns mit dem Aufbau der Architektur des Spiels. Es wurde festgelegt, welche Design-Patterns zum Einsatz kommen, welche Klassen wir benötigen und wie diese miteinander interagieren. Darauf basierend wurde ein ausführliches UML Diagramm gefertigt, welches im weiteren Projektverlauf angepasst wurde.

Spielmodus

Im Game-Script wurde das Spiel in einzelne Missionen aufgeteilt. Im Gegensatz dazu läuft der Prototyp rundenbasiert, d.h. es gibt keine linearen Missionen, sondern es erscheinen in regelmäßigen Abständen Wellen an Gegnern, welche man besiegen muss. Diese Änderung wurde aus Zeitmangel eingeführt.

Das Ziel des Spiels ist es, so lange wie möglich zu überleben und dabei möglichst viele Punkte zu sammeln.

Oberfläche

Die Oberflächen, die in dem Game-Script geplant waren, wurden in dem Spiel mit kleinen Änderungen, implementiert. Das Spiel umfasst folgende Oberflächen, die im Einzelnen im Abschnitt UI beschrieben werden:

- Hauptansicht
- Inventar
- Konstruktor

Aufgrund der Spielmodusänderung wurde die Missionsansicht nicht implementiert.

Architektur

Manager

Ein auf dem XNA-Framework basierendes Projekt, besteht von Anfang an aus zwei großen Schleifen, die durch das Framework kontrolliert und regelmäßig durchlaufen werden. Bei einem Durchlauf werden dabei die Methoden Update() und Draw(), der Spiel-Hauptklasse, ausgeführt.

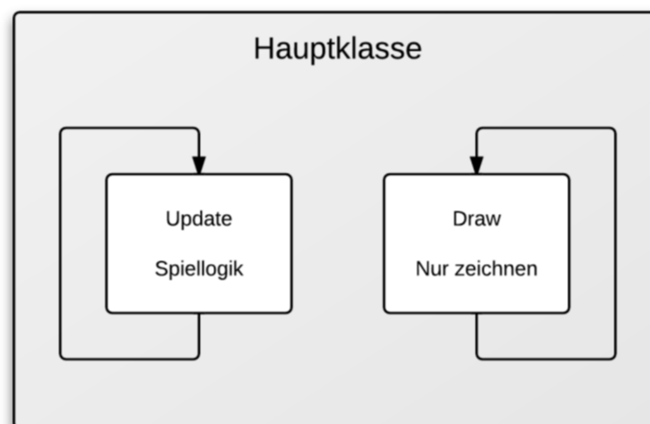

Abbildung 1: Hauptschleifen eines XNA-Projekts

Bei der Umsetzung des Spiels entschieden wir uns dafür die einzelnen, großen Komponenten wie das Menü, das eigentliche Spiel, etc. in einzelne Klassen zu trennen. Jede dieser Klassen ist für die

Verwaltung und das Zeichnen ihres Teils zuständig. Für jede dieser Komponenten wurde nun eine Klasse angelegt, die ein Interface "Manager" implementiert, welches wie die Hauptklasse zwei Methoden Update() und Draw() besitzt.

Die Hauptklasse besitzt nun jeweils ein Objekt dieser Klassen und führt bei dem Methodenaufruf Update() und Draw() jeweils die zugehörige Methode des "Aktuellen Managers" aus, welcher die ihm zugeteilten Aufgaben verwaltet. Am Beispiel des GameManagers wäre dies: Updaten des Spielers, der Gegner, Prüfung der Kollisionen, etc. und das Zeichnen der Karte, der Gegner, des Spielers und der Benutzeroberfläche, wie Inventar

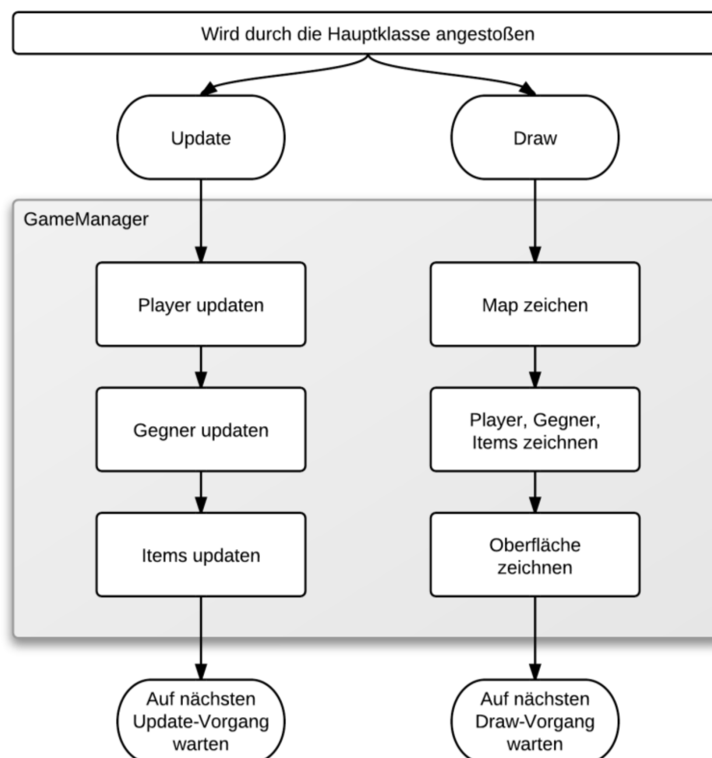

Abbildung 2: Ablauf des GameManagers

Da jeder Manager für einen Teil des Spiels zuständig ist, können wir zwischen diesen Teilen problemlos hin- und herwechseln, in dem wir den aktuellen Manager durch einen anderen ersetzen.

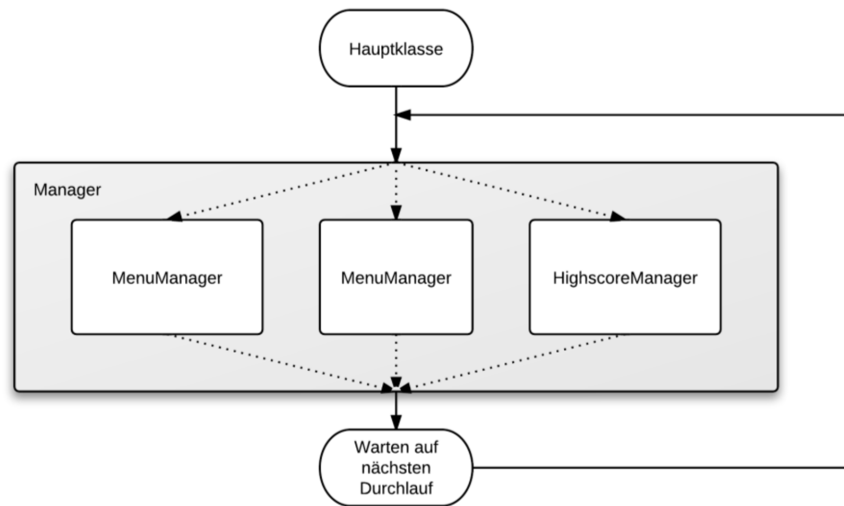

Abbildung 3: Manager

Interfaces

Da wir unterschiedlichste Funktionen zum Zeichnen brauchten (Animationen, statische Grafiken...), entschieden wir uns dafür, alle unsere Renderer-Klassen ein einziges Interface (IRenderBehavior) implementieren zu lassen. Die Klassen (z.B. StaticRenderer oder AnimationRenderer) bedienen sich dadurch alle derselben Schnittstelle und implementieren diese so, wie es für die jeweilige Klasse notwendig ist. Dadurch ist gewährleistet, dass einerseits die eigentliche Logik, die für das Zeichnen verantwortlich ist, gekapselt wird und andererseits ist es möglich die Renderer dynamisch auszutauschen.

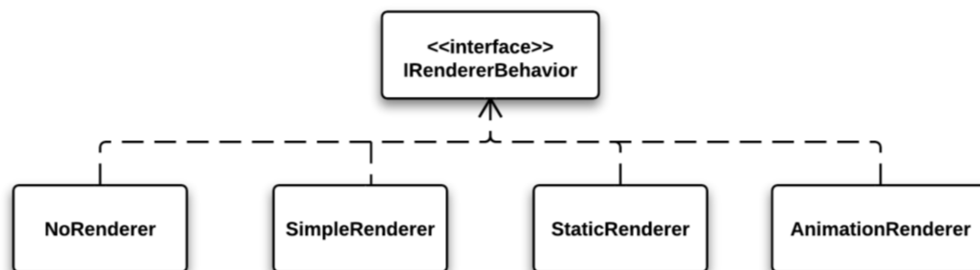

Abbildung 4: Struktur der Renderer

GameObject

GameObjekt ist unsere zentrale Klasse, von der alle Klasse erben, welche auf das Spielgeschehen Einfluss haben bzw. in irgendeiner Weise im Spiel erscheinen. Sie stellt die Attribute bzw. Methoden für das Zeichnen, Positionierung und die Einordnung in ein QuadTree bereit.

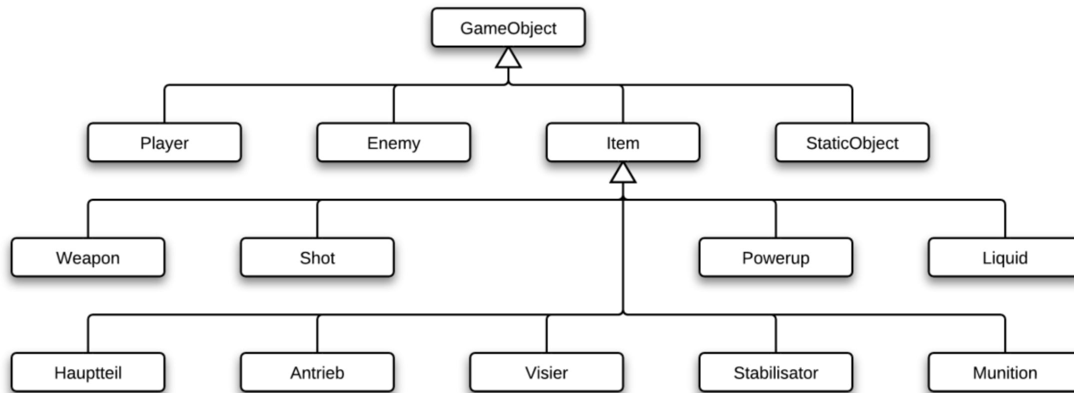

Abbildung 5: Von GameObject erbende Klassen

QuadTree

Der Quadtree ist eine beliebte Datenstruktur in der Informatik und der Computeranimation, mit der es möglich ist, Daten zu strukturieren und performant verarbeiten zu können. Dabei wird eine Baumstruktur angenommen, dessen Wurzel durch eine zweidimensionale Fläche representiert wird. Diese kann dann rekursiv in kleine Quadrate geteilt werden, um Mengen (Daten) des Baumes darzustellen.

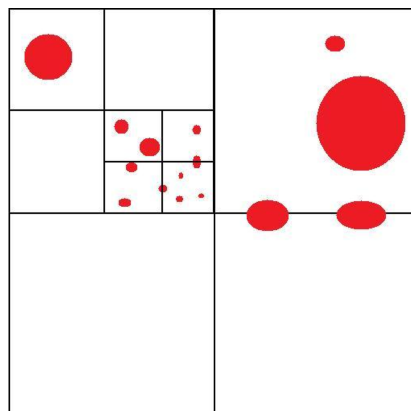

Abbildung 6: eines QuadTree[1]

[1] http://www.piko3d.com/wp-content/uploads/2010/11/SimpleQuadTreeExample.jpg

Zur Laufzeit müssen viele Objekte gezeichnet werden. Diese Objekte erweitern zwar alle die Klasse GameObjekt, werden aber auf unterschiedliche Weise weiter verarbeitet. Aus diesem Grund haben wir unsere Objekte auf drei QuadTrees aufgeteilt. So gibt es einen Baum für die Gegner, einen für die Items und einen für die Objekte, die zum Hintergrund gehören (z.B. Möbel, Karte usw.). Diese Aufteilung erspart uns die Vergleiche zur Feststellung des Typs der jeweiligen Objekte, was uns einen deutlichen Performancegewinn bring.

Config-Files

Wir benutzen Konfigurationsdateien für LAST MAN, um die Spielkonfigurationen gebündelt laden und speichern zu können. Noch bevor der GameManager seine Arbeit antritt, wird die *config.ini* von dem Hauptprogramm ausgelesen und entsprechend an die Setter für z.B. die Bildschirmauflösung weitergegeben, damit diese gesetzt werden kann.

```
resolutionWidth=1024
resolutionHeight=576
isFullscreen=false
antiAliasing=true
mapDir=Content/maps/
itemDir=Content/items/
keyMoveUp=W
keyMoveDown=S
keyMoveLeft=A
keyMoveRight=D
```

Die Karte

Die Karte des Spiels ist modular aufgebaut. Die Bestandteile der Karte (Wände, Möbel usw.) werden durch Angabe ihres Typs, der Koordinaten (X, Y) und der Ausmaßen (Breite, Höhe) in einer Textdatei gespeichert. Des Weiteren beinhaltet die Karten-Datei die Wegpunkte für die Wegberechnung der Gegner und ihre Verbindungen, und die Startpunkte des Spielers und der Gegner.

R, 4769, 2846, 254, 20 — Typ = Wand, X,Y, Breite, Höhe

M, monitor, 807, 3925, 1, 180 — Typ = Objekt, Elementname X,Y, Breite, Höhe

S, 132, 3085 — Typ = Startpunkt, X,Y

W, 1, 501, 3137 — Typ = Wegpunkt, ID, X,Y

C, 1, 4 — Typ = Verbindung, ID Wegpunkt, ID Wegpunkt

Abbildung 7: Beschreibung der Karten-Datei

Durch diese Art der Speicherung ist es möglich, verschiedene Karten zu erstellen und diese ohne Änderungen im Quellcode dynamisch zu laden.

Wegfindung der Gegner

Für die Wegfindung der Gegner verwenden wir den Dijkstra-Algorithmus, welcher in einem Graphen, den kürzesten Weg zwischen zwei Punkten herausfinden kann. Da wir in unserem Spiel pixelgenaue Koordinaten verwenden, kann man das Pixelgitter als Graphen sehen, wobei jedes Pixel als Knoten und benachbarte Pixel als Kante repräsentiert werden können. Dies würde jedoch einen sehr großen Graphen ergeben, was bei mehreren Wegberechnungen, über die komplette Karte, aus zeittechnischen Gründen nicht sehr effektiv wäre.

Somit entschieden wir uns ein Wegpunktnetz über die eigentliche Karte zu legen, welches einen sehr vereinfachten Graphen der Karte darstellt und die Wegberechnung beschleunigt. Es wird somit ein Weg von dem Knoten, der am nächsten bei dem Monster liegt bis zu dem Wegpunkt berechnet, welcher am nächsten beim Spieler liegt. Sollte sich der Spieler in direktem Sichtkontakt mit einem Monster befinden, werden die Wegpunkte ignoriert und direkt in Richtung des Spielers gegangen bzw. an die Position, an der er zuletzt von dem Monster gesehen wurde. Anschließend wird wieder auf die Wegfindung durch die Wegpunkte gewechselt.

Die Berechnung der Wege, vor allem bei mehreren Gegnern, nimmt sehr viel Zeit in Anspruch. Da die Wege der Monster nicht zwangsläufig in Echtzeit aktualisiert werden müssen, wurden diese Berechnungen in einen separaten Thread ausgelagert, der die Berechnung in unterschiedlichen Zeitintervallen ausführt. Damit wird die eigentliche Spiellogik nicht verzögert.

GUI

Das XNA-Framework stellt keine vorgefertigte grafische Oberflächenelemte bereit, deswegen haben wir beschlossen, dafür ein eigenes kleines Framework zu bauen. Die Entwicklung eines vollständigen GUI-Frameworks würde die Rahmen unseren Projektes sprengen, deswegen haben wir nur die Komponenten implementiert, die wir auch benötigt haben und versuchten, diese möglichst vielseitig zu verwenden. In dem Namespace UI werden hauptsächlich zwei Design-Patterns verwendet: Listener Pattern und Composite Pattern.

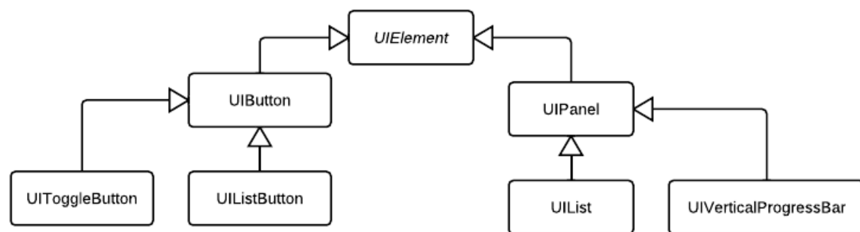

Abbildung 8: UML-Diagramm der UIElemente

Aus diesen Basisklassen werden die weiteren UI-Klassen abgeleitet. Um die Benutzerinteraktion zu ermöglichen, wurden zwei Listener-Klassen implementiert, die auf Maus-Events reagieren.

Die Hauptansicht

Abbildung 9: Hauptansicht des Spiels

In der Mitte der Hauptansicht ist die Spielerfigur zu sehen (1). Die Gegner (2) bewegen sich auf den Spieler zu und schießen auf ihn, sobald sie in Reichweite sind. An unterschiedlichen Stellen der Karte

sind Gegenstände platziert (3), die von dem Spieler aufgesammelt werden können. Unten links wird die Anzahl der dem Spieler zu Verfügung stehenden Schüsse angezeigt (4). Wird der Spieler von den gegnerischen Schüssen getroffen, werden seine Lebenspunkte reduziert. Den aktuellen Stand der Lebenspunkte sieht man an dem Lebensbalken (5). Um dem Spieler die Orientierung in der Spielwelt zu erleichtern, wird im linken unteren Eck ein verkleinerter Ausschnitt der Karte angezeigt (6).

Inventar

Die Übersicht aller, während des Spiels, gesammelter Objekte, kann in der Ansicht "Inventar" betrachtet werden.

Abbildung 10: Ansicht des Inventars

Alle Gegenstände, die der Spieler gesammelt hat, werden in einer Liste dargestellt (1). Unter der Liste erscheint das Gesamtgewicht aller Gegenstände im Inventar (2). Im Spiel existieren sechs unterschiedliche Typen von Gegenständen: Waffe, Hauptteil, Stabilisator, Munition, Visier, Antrieb und Sonstiges. Der Schaltflächenbereich "Filter" (3) ermöglicht schnelleres Anzeigen der Gegenstände eines bestimmten Typs. Links neben der Liste befindet sich das sog. Shortcut-Panel. Hier kann der Spieler die Waffen mit vier Schnelltasten verknüpfen. Eine Schnellzugriffstaste (4) kann mit einer Waffe (5) und mit einer bestimmten Munitionsart (6) verknüpft werden. Die Beschreibung der Attribute des in der Liste ausgewählten Gegenstandes in dem unteren linken Bereich des Inventars betrachtet werden (7). Der X-Button (8) unter der Liste ist dafür vorgesehen, um die Gegenstände aus dem Inventar zu entfernen. Manche Gegenstände können von dem Spieler "gegessen" werden, dafür ist der V-Button (8) zuständig.

Konstruktor

Manche von dem Spieler aufgesammelte Gegenstände können miteinander kombiniert werden, um so eine neue Waffe zu bekommen. Dies geschieht in dem Konstruktionsmenü.

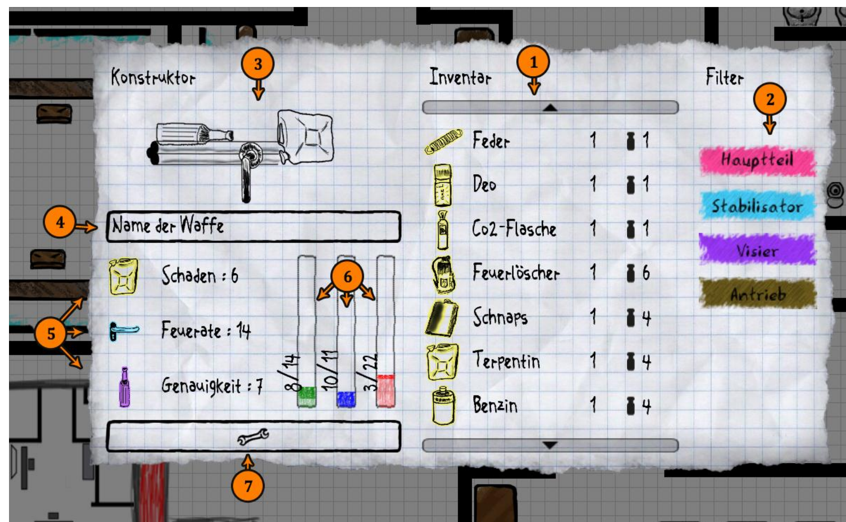

Abbildung 11: Ansicht des Konstruktionsmenü

Der Spieler wählt aus der Inventarliste (1) vier Bestandteile einer Waffe (Hauptteil, Stabilisator, Visier und Antrieb) aus. Die Liste kann nach unterschiedlichen Typen gefiltert werden (2). Diese Gegenstände werden auf die entsprechenden Stellen in den linken Bereich gelegt (5). Die einzelnen Attribute der Waffenbestandteile werden neben den Gegenstand-Icons angezeigt. Je nachdem, welcher Gegenstand ausgewählt ist, verändert sich die Grafik der fertigen Waffe (3). Unterhalb dieser Grafik (4) muss der Spieler die neue Waffe benennen, bevor er diese erstellen kann. Der Spieler sammelt im Verlauf des Spiels die von den Gegner “fallen gelassene” Flüssigkeiten. Diese Flüssigkeiten werden für das Erstellen der Waffen benötigt. Den Füllstand der einzelnen Flüssigkeitstanks kann in der Konstruktor-Ansicht betrachtet werden (6). Zusätzlich kann dort die zur Erstellung der Waffe benötigte Menge abgelesen werden. Um eine Waffe fertigzustellen, muss der Spieler auf den Button links unten drücken (7).

Score

Damit der Spieler den Anreiz hat das Spiel mehrmals zu spielen, wurde eine Highscore-Liste eingeführt, in der er sich mit anderen Spielern vergleichen kann.

Die erreichte Punktzahl berechnet sich aus mehreren Faktoren:

- Anzahl der getöteten Monster
- Benötigte Rundenzeiten

- Schaden der verursacht wurde
- Schaden den der Spieler bekommen hat
- Anzahl der abgegebenen Schüsse
- Anzahl der abgegebenen Schüsse

Ergebnis

Erzeugung der Gegner

Ein wichtiger Punkt für den Survival-Modus ist das geregelte Erscheinen der Gegner. Die Startpunkte (Spawnpunkte) der Gegner sind dabei in der Map-Datei angegeben und werden beim Laden der Karte eingelesen.

Nach einigem Überlegen entschieden wir uns dafür, das Erzeugen der Gegner durch Wahrscheinlichkeiten zu regulieren. Bei einem Berechnungsdurchgang wird somit für jeden Startpunkt ein Wert zwischen 0 und 1 berechnet, der die Wahrscheinlichkeit angibt, dass ein Monster erzeugt wird.

Hierbei waren die Sollkriterien:

- Je weiter ein Startpunkt entfernt ist, desto niedriger ist die Wahrscheinlichkeit ein Gegner zu erzeugen
- Gegner dürfen nicht innerhalb des Bildschirms bzw. sichtbaren Bereichs erscheinen
- Gegner dürfen nur in einem bestimmten Abstand zum Spieler erscheinen
- Es gibt ein Limit für die Anzahl an Gegner auf der Karte

Die Wahrscheinlichkeit wird wie folgt berechnet:

$$p(spawn) = distanz * istInBildschirm * modifier * limit$$

Dabei sind die Variablen, welche auf den Bereich zwischen 0 und 1 begrenzt sind, wie folgt belegt:

distanz: Ein linearer Verlauf zwischen 0 und 1. Ist die Distanz zwischen dem Startpunkt und des Bildschirmrandes 0, ist diese Variable 1. Je näher sich die Distanz der maximal erlaubten "Spawn-Distanz" nähert, geht die Variable gegen 0.

istInBildschirm: Befindet sich der Startpunkt in dem sichtbaren Bereich, ist die Variable 0, ansonsten 1. Somit wird ausgeschlossen, dass ein Gegner auf dem Bildschirm erscheint.

modifier: Mit diesem Wert, kann die Wahrscheinlichkeit manuell manipuliert werden.

limit: Eine Gerade, welche gegen 0 geht, je mehr sich die aktuelle Gegneranzahl der maximalen Gegneranzahl nähert. Diese Gerade ist jedoch vertikal verschoben, damit sich die Wahrscheinlichkeit, erst nach Erreichen von 75% der Gegneranzahl, verringert.

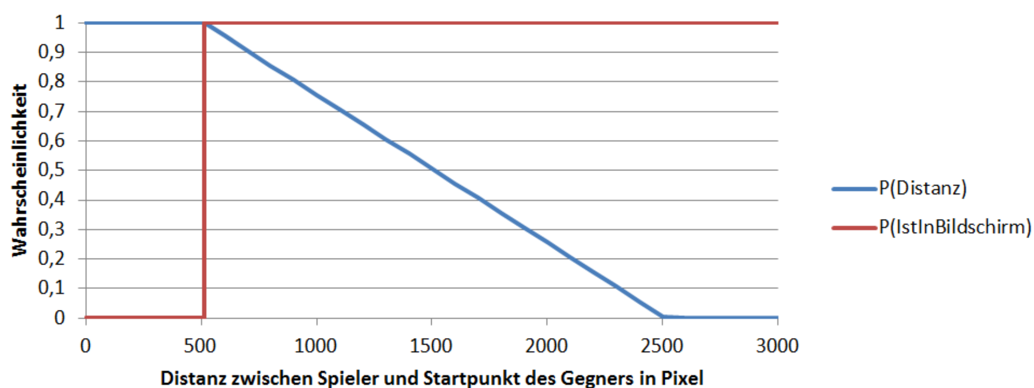

Abbildung 12: Einzelne "Spawn"-Wahrscheinlichkeit in Abhängigkeit der Distanz

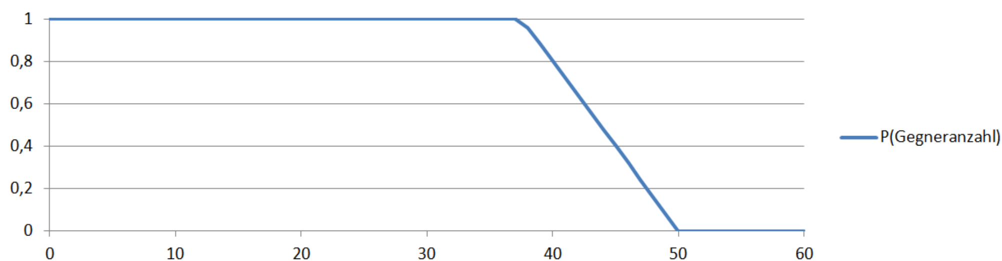

Abbildung 13: Wahrscheinlichkeit in Abhängigkeit der Gegneranzahl

Aufgetretene Probleme und deren Lösung

Eines der Hauptprobleme des Projektes war der Zeitmangel, den wir durch ein falsches Einschätzen des Umfangs bekamen. Zu Beginn der Planung waren wir der Überzeugung, dass das Projekt relativ ‘schnell gemacht’ sei. Allerdings stellten wir bald fest, dass das nicht so ist und mussten aus diesem Grund viele Einschnitte in unseren angestrebten Features hinnehmen.

Dazu kam, dass wir bei der Planung zu Beginn des Projektes viele Ungewissheiten über Ablauf, Umsetzung und Machbarkeit einiger Spielelemente hatten. Lösungen, die wir daraufhin entwickelten, stellten sich erst im Laufe des Projekts als ungünstig heraus. Sie lösten zwar das

aktuelle Problem, verursachten aber an anderer Stelle Fehlverhalten, die wir auf Grund der geringen Erfahrung mit Projekten dieser Art nicht erkennen konnten.

Für die Wegsuche implementierten wir zuerst den A*-Algorithmus. Allerdings funktionierte dieser nicht ganz korrekt und führte bei einer bestimmten Menge an Gegnern auf dem Spielfeld zu ungewünschtem Verhalten. Kurz vor der MediaNight ersetzen wir die fehlerhafte Implementierung durch den Dijkstra-Algorithmus. Dieser ist zwar langsamer, reichte uns aber für den Moment.

Ein weiteres Problem stellt die Skalierung der Schrift des Spiels dar. Dadurch, dass wir dem Spieler die Möglichkeit geben, beliebige Auflösungen des Spiels zu nutzen, bekamen wir das Problem, dass unsere Schriftart nicht mitskaliert wird. Die Anpassung der Schriftgröße muss immer manuell vorgenommen werden.

Erfahrungen

Die meisten Themen, die wir während der Vorlesung "Entwicklung von Computerspielen" behandelt haben, fanden in unserem Projekt ihren praktischen Einsatz. Auch die Design-Patterns und Algorithmen, die wir bis jetzt nur aus den Vorlesungen kannten, halfen uns in großem Maße bei der Entwicklung unseres Spiels.

Am Anfang des Projekts war keiner von mit der Sprache C# und der Entwicklungsumgebung Microsoft Visual Studio besonders vertraut. Während der Arbeit an unserem Spiel lernten wir die Vorzüge dieser Sprache kennen, besonders das XNA Framework empfanden wir als einen gut durchdachten Baukasten mit vielen Werkzeugen, die einem Spieleentwickler das Programmieren erleichtert.

Außerdem konnten wir uns durch das gemeinsame Projekt fachlich wie auch menschlich weiterentwickeln: wir haben unsere Teamfähigkeit ausbauen können und vor Allem am eigenen Leib erlebt, was es bedeutet ein eigenes, kleines Software-Projekt zu stemmen.

Da uns das Projekt aber großen Spaß bereitet hat und wir durch das vorwiegend positive Feedback auf der MediaNight motiviert sind unser Spiel weiter zu verfeinern, planen wir auch in der Freizeit weiter daran zu arbeiten.

Ausblick

Für die nächsten Versionen bzw. für den weiteren Entwicklungsvorgang haben wir verschiedene Fixes und etliche neue Features eingeplant. Unter anderem stellen wir den Suchalgorithmus wieder auf

den A*-Algorithmus um. Des Weiteren haben wir vor eine neue, detailreichere Karte für unser Spiel zu erstellen.

Zu den Features gehören unter anderem neue Modifikationen (‘Buffs’) für Spieler und Gegner oder Effekte der Karte (z.B. Springbrunnenanimation), wir wollen auch die Musik während des Spiels neu anpassen bzw. einfügen und den Nahkampf zwischen Gegnern und Spieler umsetzen. Ein weiteres Feature, welches bereits weitestgehend implementiert wurde, aber noch keinerlei Auswirkung hat, ist das Gewicht der sich im Inventar befindlichen Items, das auf die Bewegungsgeschwindigkeit des Charakters Einfluss haben sollte.

Wir haben uns auch vorgenommen unseren ‘Level-Modus’ aus dem Game-Design-Script zu implementieren, dass der Spieler zu Beginn des Spiels die Wahl des Modus’ selbst entscheiden kann.

Natürlich werden wir die Bugs, die wir während der Tests und während der MediaNight entdeckt haben in den nächsten Versionen Großteils versuchen zu beheben. Dazu zählt z.B. der zufällig auftretende Fehler, dass ein Gegner in einer Welle zwar als ‘noch zu besiegen’ gezählt wird, aber nie erstellt wurde. Die Minikarte dient im Moment nur als Platzhalter und wird in der nächsten Version vollkommen implementiert.

Unter dem Punkt ‘nice-to-have’ für die nächsten Versionen haben wir uns vorgenommen, zerstörbare Einrichtungsgegenstände zu implementieren oder das Einfügen von Explosionsmunition.

Anhang

Abbildungsverzeichnis

Abbildung 1: Hauptschleifen eines XNA-Projekts 3
Abbildung 2: Ablauf des GameManagers 4
Abbildung 3: Manager 5
Abbildung 4: Struktur der Renderer 5
Abbildung 5: Von GameObject erbende Klassen 6
Abbildung 6: eines QuadTree 6
Abbildung 7: Beschreibung der Karten-Datei 8
Abbildung 8: UML-Diagramm der UIElemente 9
Abbildung 9: Hauptansicht des Spiels 9
Abbildung 10: Ansicht des Inventars 10
Abbildung 11: Ansicht des Konstruktionsmenü 11
Abbildung 12: Einzelne "Spawn"-Wahrscheinlichkeit in Abhängigkeit der Distanz 13
Abbildung 13: Wahrscheinlichkeit in Abhängigkeit der Gegneranzahl 13