FACHHOCHSCHULE **TRIER**
Hochschule für Technik, Wirtschaft und Gestaltung
University of Applied Sciences
Informatik

Bloody Plumber

Marco Hamann, Andreas Willmann, Nico Weskamp

Dokumentation zum Medineprojekt

Betreuer: Prof. Dr.Christof Rezk- Salama

Trier, 02.10.2013

Inhaltsverzeichnis

1

Vorwort

Abb. 1.1. Bloody Plumber Menü

Bloody Plumber ist ein 2D Jump and Run mit Shooter Elementen. Mit diesem Spiel möchten wir die Super Mario Reihe von Nintendo parodieren. Unsere Idee war es, ein möglichst gewissenloses Bild von Mario zu zeichnen und trotzdem den Charme vom Original zu erhalten. In der angehängten Version ist ein Demolevel implementiert, welches alle grundlegenden Elemente des Spiels enthält.
Der Spieler kann laufen, springen und schießen, jedoch ist es ihm nicht möglich die Gegner per Sprung zu töten. Zudem sollte er im Verlauf eines Level´s Kisten und Coins einsammeln, durch die sein Highscore erhöht wird. Dieser wird auch durch das Töten von Gegnern und die gebrauchte Zeit für das Level erhöht.
In einer finalen Version wäre die Welt in unterschiedliche Gebiete unterteilt, die aus mehreren Leveln bestehen und unterschiedliche Szenarien liefern. Jedes Level muss nach und nach freigeschaltet werden. Außerdem wartet am Ende jeden Gebietes ein Endgegner, den es zu besiegen gilt.
Das Spiel wurde für Windows 8 programmiert und läuft sowohl auf Desktop PC´s als auch auf Windows 8 Tablets. Dementsprechend haben wir in C-Sharp pro-

grammiert und XNA 4.0 genutzt. Um auch eine Möglichkeit zu haben das Spiel auf anderen Plattformen wie Android oder iOS umzusetzen, nutzen wir zusätzlich das Opensource Framework Monogame.
Im folgenden werden wir erläutern, wie wir bei der Implementierung und dem Gamedesign vorgegangen sind, vor welche Probleme das Spiel uns gestellt hat und wie wir diese gelöst haben. Teil unseres Spiels ist darüberhinaus ein Leveleditor, der es uns ermöglicht, Levels mit allen Elementen zu erstellen und diese zu speichern. Da dieser gleichzeitig als Leistungsnachweis für das Modul Tool- und Pliginprogrammierung dient, werden wir diesem ein extra Kapitel in der Dokumentation widmen.

2

Gameart

2.1 Texturen und Animationen

In unserer urpsrünglichen Idee, in Bezug auf den Grafikstil des Spiels, war vorgesehen einen einfachen Look zu wählen. Wir wollten einen Retro Stil wählen, wie ihn Mario ganz zu Beginn seiner Lebenszeit auszeichnete. Nachdem wir einige Grafiken erstellt hatten, kamen wir zu dem Entschluss, dass dieser Stil auf hochauflösenden Bildschirmen fehl am Platze ist. Der fehlende Farbreichtum sowie der geringe Detailgrad, passten nicht zu unserem Konzept.
Also entschlossen wir uns, einen weitaus zeitaufwendigeren Stil zu wählen. Es ist eine Art Cell-Shading Look. Diesen zeichnen kantige und fette Linien, extreme Übergänge in der Farbgebung sowie knallige Farben aus. Zudem wird durch abwedeln und nachbelichten ein plastischer Effekt erzeugt. Zum erstellen der Grafiken haben wir Adobe Photoshop verwendet.

2.2 Sound

Den Sound den wir in unserem Spiel verwenden wurde ebenfalls zum Großteil von uns selbst produziert. So haben wir die Hintergrundmusik selbst komponiert und mit Gitarre aufgenommen. Am PC haben wir es nochmals geschnitten und mit einem virtuellen AMP nachbearbeitet, um verschiedene Klänge und Effekte zu erzeugen. Lediglich die Drumloops und Schußsounds sind aus externer Quelle. Zur Aufnahme nutzten wir das Programm *Reaper* und für den virtuellen AMP das Programm *Guitar Rig*. Für die Vertonung des Mario lieh uns ein guter Freund seine Stimme und sprach einige Sätze ein, die wir noch gar nicht alle in das Spiel einbauen konnten.

3

Gamestate- Management

3.1 Einleitung

Bei dem Gamestate- Management handelt es sich um die Basis des Spiels. Mit ihm ist es möglich unterschiedliche „Arten“ von Screens zu erstellen. Dieses Klassenpaket wurden von Microsoft für XNA 4 erstellt. Wir haben dieses Paket an unsere Bedürfnisse angepasst. Einige Funktionalitäten ,wie die Möglichkeit mehrere Spieler anzumelden oder das Skalieren des Spiels, wurden von uns übernommen aber nicht angewendet.

3.2 ScreenManager

Zunächst gibt es einen generellen *ScreenManager*, welcher alle Arten von *GameScreens* verwaltet. Eine Instanz des *ScreenManagers* wird bereits beim Starten des Spiels erstellt.
Mit ihm werden alle Assets geladen und das Touchpanel bei Bedarf aktiviert. Dazu gehören die Sounds und Grafiken, zudem wird die Font für das Spiel geladen. Die Hauptaufgabe des *ScreenManagers* besteht jedoch darin, neue GameScreens zu verwaltet und wenn nötig auch wieder zu entfernen. Die angelegten GameScreens werden in einer Liste organisiert und verarbeitet.

3.3 GameScreen

Der *GameScreen* stellt den Rohling dar, aus dem später alle anderen Screens abgeleitet werden. Dieser enthält folgende Funktionalitäten:

- ScreenState
- ob der abgeleitet Screen ein PopUp ist
- welche Übergangszeit die Screens haben
- ob ein Screen Aktiv ist

Darüber hinaus werden die beiden Funktionen der Kerninteraktionsschleife *update* und *draw* erstellt und mit *virtual* überschreibbar gemacht. Diese beiden Funktionen werden im eigentlichen GameplayScreen jeden Tick aktualisiert. Hierauf werden wir im weiteren verlauf der Dokumentation näher eingehen.
Zuletzt lässt sich noch sagen, dass der *GameScreen* das Interface IDisposeable implementiert. Diese Schnittstelle wird von uns auch in vielen anderen Klassen verwendet. Mit diesem Interface lässt sich der Garbage-Collector(GC) verwalten und bestimmte Elemente können so zur Löschung makiert werden. Dies geschieht mit dem Befehl *GC.SupressFinalize*. Der GC sammelt diese dann eigenständig ein. Es ist möglich den GC zum aufräumen zu zwingen. Mit dem Befehl *GC.Collect()* sollte allerdings sparsam umgegangen werden, da er viele Resourcen verbraucht.

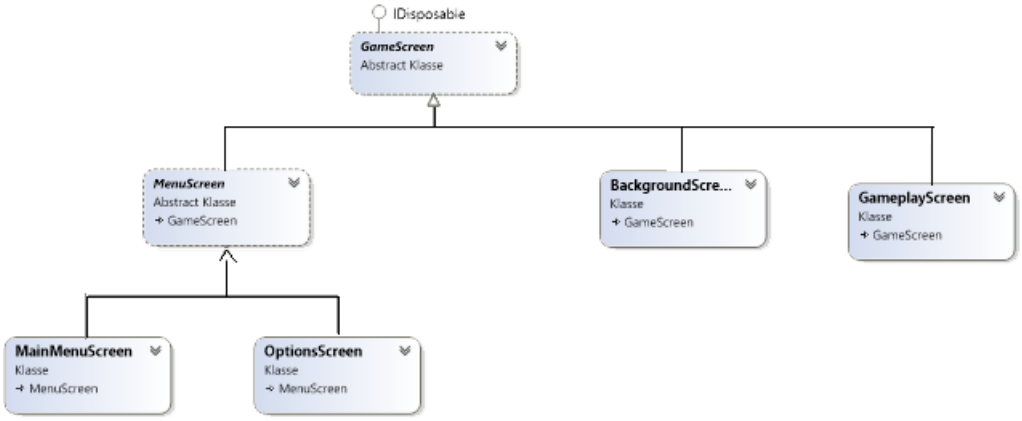

Abb. 3.1. Aufbau GameScreens

Das UML Diagramm zeigt eine vereinfachte Herachie zwischen einigen Komponenten des Gamestatemanagements.
Zunächst leitet sich von GameScreen die Klasse *MenuScreen* ab. Wie der Name schon sagt, handelt es sich um ein Screen der irgendwelche Optionen oder Einstellungen verwaltet. Er ist von einem GameScreen abgeleitet und enthält somit alle Elemente dieser Klasse.
Der *MenuScreen* verwaltet widerum Objekte der Klasse *MenuEntry*. Diese Klasse repräsentiert einen Menüeintrag mit Farbe und Text. In der Klasse *MenuScreen* werden die Menuentries in Listen abgelegt, aktualisiert und gezeichnet. Damit bei entprechendem Input auch eine Reaktion stattfindet, werden in der Funktion *handleInput* Eingaben über den TouchScreen, die Tastatur und die Maus aufgefangen und entsprechend verarbeitet.
Nun wird widerum die Klasse *MenuScreen* als Basis für alle tatsächlichen Menüs des Spiels gewählt. Wie das Diagramm zeigt, ist der *MainMenuScreen* ein Menü, der das Hauptmenü darstellt. Von hier lässt sich das Spiel starten, beenden, die Highscores einsehen und die Optionen ändern. Diese bilden dann den zweiten Menüscreen, nämlich den *OptionsScreen*. Hier ist es möglich die Toucheingabe zu aktivieren bzw. zu deaktivieren und die Tastatur zu individualisieren.

Zwei weitere Elemente im Bild oben sind der *BackgroundScreen* und der *GameplayScreen*. Zum *GameplayScreen* möchten wir an dieser Stelle nicht viel sagen, da er das Herzstück des Spiels ist und somit im nächsten Kapitel ausführlich behandelt wird. Der BackgroundScreen ist, wie der Name schon sagt, der Hintergrund für die Menüs. Er ist ebenfalls ein GameScreen, da sich auf ihm in unserem Fall „Spielgeschehen“ abspielt. Er ist sozusagen ein „kleiner“ *GameplayScreen*.
Zusammenfassend lässt sich sagen, wie auch das Schaubild zeigt, der GameScreen die Vaterklasse aller anderen Screens ist.

3.4 InputState

Mit dem InputState ist es möglich alle Arten von Eingaben zu verarbeiten. Die Klasse ermöglicht es das Keyboard, ein Gamepad, die Maus und auch den TouchScreen zu benutzen. Um auf Elemente dieser Klasse zugreifen zu können, muss nur die in der Klasse *GameScreen* definerte Funktion handleInput überschrieben werden. Diese erwartet ein Objekt von Typ InputState. Somit ist ein Zugriff auf alle Elemente des *InputStates* machbar. Die Klasse erbt nichts und ihr wird auch nichts vererbt.

3.5 Überblick

Das Spiel startet in der Klasse *Program*. Diese Klasse wird beim Erstellen eines XNA Projektes automatisch miterstellt. Hier wird festgelegt, welche Klasse sozusagen den Einstieg in das eigentlich Spiel liefert. Von dort wird die Klasse *BloodyPlumber* aufgerufen. Hier wird die Framerate, die Contentdirectory und die Auflösung definiert. Zudem wird der *Screen-* und *GraphicsDeviceManager* angelegt. Nachdem diese Befehle ausgeführt wurden, legt die Klasse den *Background-* und *MainMenuScreen* an und der Benutzer sieht das Hauptmenü. Ab hier werden, je nach Eingabe des Nutzers, Screens entfernt und hinzugefügt. Alle folgenden Interaktionen liegen somit in der Hand des Users.

4

Implementierung der Spiellogik

4.1 GameplayScreen

Um Objekte in ein Spiel, das auf XNA basiert, zu laden und zu aktualisieren sind vier Funktionen unerlässlich, die im *GameplayScreen* zu finden sind.
Es gibt zwei Funktionen die einmalig beim Starten des *GameplayScreens*aufgerufen werden. Zum einen der Konstruktur, in dem alle Objekte des Spiels angelegt werden und zum anderen die Funktion *LoadContent()*. In dieser Funktion werden alle Objekte initialisiert, zudem werden die LoadContent Funktionen der Objekte aufgerufen. In Abbildung 4.1 ist der Aufbau für das Objekt *Player*, welches unseren BloodyPlumber repräsentiert, zu sehen.

Diese Vorgehensweise wird für alle Objekte angewendet, die innerhalb eines bzw. aller Level nötig sind. Durch die vermerkte ID im Konstruktor können später einzelne Objekte vom Ladevorgang ausgeschlossen werden, falls sie für das zu ladene Level nicht benötigt werden.

Wie oben erwähnt, benötigt ein reibungsloser Spielablauf vier Funktionen. *Update()* und *Draw()* machen die vier komplett. Diese beiden Funktionen werden jeden Tick aktualisiert. In unserem Fall sind das 30 Aktualisierungen pro Sekunden, sprich 30fps. Nach der Aktualisierung werden die nun „neu“ positionierten Objekte mit Hilfe der Draw- Funktion gezeichnet. Dieser Vorgang wiederholt sich bis das Level beendet ist. Aber nicht nur updates bezüglich der Bewegung werden in dieser Funktion ausgeführt, sondern auch updates, die den Punktestand, die Eingaben, die GameTime, die Kollisionen oder die Sound betreffen, werden hier verarbeitet. Letztendlich bilden diese vier Funktionen das Herz des Spiels und befinden sich im GameplayScreen, welcher beim Drücken des „Play“- Buttons angelegt wird.
Die Funktion GameOver() behandelt das Beenden des Levels durch Tod. In diesem Fall greift wieder die Schnittstelle IDisposeable und alle Objekte die nicht weiter benötigt werden, werden zum Löschen freigegeben. Wenn der Spieler das Level erfolgreich beendet, werden ebenfalls alle überflüssigen Objekte entfernt, allerdings wird ein anderer screen angezeigt.

```
public GameplayScreen(ScreenManager manager, int id,bool touchState)
        {
            //screenManager wird übergeben
            // id = levelId;
            //if touchState = true -> TouchControl activ
            //.........................
            player = new Player();
            playerInput = new Input();
            //.........................
        }

        public override void LoadContent()
        {
            //.........................
            playerAnimation = screenManager.imageFileSystem.playerAnimation;
            playerAnimation.setAnimationActive(true);
            player.Initialize(100.0f, 700.0f, 0.0f, 0.0f, m_gameSpeed,1,
            playerAnimation, ScreenManager.Viewport.Width);
            playerInput.Initialize(player, levelEins);
            //.........................


        }
```

Abb. 4.1. Ausschnitt Konstruktor und LoadContent

4.2 Object- Klassen

Im folgenden wollen wir nun genauer auf die Object- Klassen des Spiels eingehen. Unter dieser Klasse sind Objekte wie Player, Enemy oder das Blut zusammengefasst.

Grafik 4.2 zeigt den Aufbau dieser Gruppe. Die Klasse *Object* ist eine Basisklasse die alle Attribute enthält, die auch in den abgeleiteten Klassen zu finden sind. Dazu gehören unter anderem die Position des Objektes, die Geschwindigkeit, eine Animation sowie Bool- Werte, welche kontrollieren ob ein Objekt in irgendeiner Kollision mit der Umgebung steht. Zudem implementiert die Klasse *Object* die Funktionen Initialize, loadContent, update und draw, welche alle von den abgeleiteten Klassen geerbt werden.
Diese Objekte verweden die Klasse Animation, in welcher die 2D- Animationen für das Spiel berechnet werden. Die Bilder werden wie ein Daumenkino nacheinander abgespielt. Dazu muss bloß angegeben werden, wie viele Bilder in dem Strip sind, wie breit und wie hoch der Strip ist und wie lang jedes Bild angezeigt werden soll. Mit diesen Werten aktualisiert die update- Funktion jeden Tick die Animation und ein bewegtes Bild entsteht. Die Klasse Animation wird von allen Objekten verwendet die irgendwie in Bewegung sind und nicht ausschließlich von den Klassen der *Object*-Gruppe.

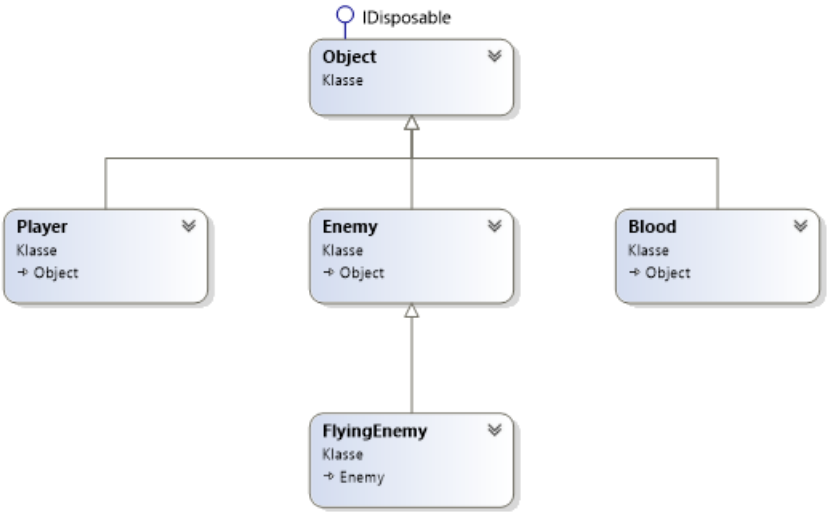

Abb. 4.2. Aufbau der Object- Gruppe

4.3 Schwierigkeiten

Wir wollen kurz auf ein paar Probleme eingehen, welche uns während der Programmierung begegnet sind und die entsprechenden Lösungen dazu erläutern.

4.3.1 Blood

Als wir in das Spiel unser Blut einfügen wollte, ist das Spiel bei vielen Treffen oft eingebrochen, da zu viele Blut- Objekte erstellt wurden und diese nicht so schnell in den Speicher gelegt werden konnten. Dieses Problem haben wir mit einer Liste von *PuddleOfBlood´*s gelöst. In dieser Liste befinden sich vier Puddles die immer wieder verwendet werden. Daraus haben wir gelernt und die gleiche Vorgehensweise bei den Projektilen angewandt.

4.3.2 Kollision

Ein weiteres Problem ergab sich bei der Kollision zwischen dem Spieler bzw. Gegner und der Welt. Zunächst war es so, dass der Player immer ein Stück in die Welt hinein lief oder gar beim Springen auf eine Plattform durch diese hindruch fiel. Da die Kollision mit Rectangles berechnet wird, mussten wir hier lange experementieren. Mit der Berechung des nächsten Zeitschrittes und einer dort möglichen Kollision und dem richtigen Platzieren von vier Rectangles um Gegner bzw. Spieler, konnte dieser Bug beseitigt werden.

4.3.3 TouchInput / Absturz

Auf der Zielgeraden erwischte uns ein weiterer Bug. Das Spiel stürtzte nach dem Tod des Spielers im Übergang zum Hauptmenü ab. Dieser Fehler war schwer zu lokalisieren. Letztendlich konnten wir ihn auf den Touchscreen zurückführen.

Der Grund für den Fehler ist uns jedoch weiterhin schleierhaft. Dazu müssen wir kurz auf die Funktionsweise des Touchpanels eingehen. Diese ist in der Lage vier Berührungen gleichzeitig in einer Liste zu speichern. Zudem wird das Touchpanel jeden Tick abgefragt. In diesem Fall war es nun so, dass das Programm zusammenbrach bei Tod des Spielers, weil die update- Funktion kein weiteres Mal durchlaufen wurde und die Liste mit den Touches nicht leer war.
Die Finale Lösung war der Einbau eines Stoppers, der das Touchpanel nochmals abfragte und aktualisierte ob noch eine Berührung aktiv ist. In diesem Stopper wurde das Touchpanel deaktiviert um eine erneute Berührung nicht möglich zu machen. Nachdem dieser Stopper den Input nochmals aktualisiert hatte, konnte das Spiel wie gewohnt weiter laufen. Diesen Stopper haben wir in beiden Screens, die nach beenden des Levels erscheinen,f versteckt.
Die Finale Lösung war der Einbau eines Stoppers, der das Touchpanel nochmals abfragte und aktualisierte ob noch eine Berührung aktiv ist. In diesem Stopper wurde das Touchpanel deaktiviert um eine erneute Berührung nicht möglich zu machen. Nachdem dieser Stopper den Input nochmals aktualisiert hatte, konnte das Spiel wie gewohnt weiter laufen. Diesen Stopper haben wir in beiden Screens, die nach beenden des Levels erscheinen,f versteckt.

5

Level Editor

5.1 Einleitung

Schon zu Beginn unseres Projektes war uns klar, dass wir für unsere Spiel Tiles verwenden wollten. Also ein Soruceimage, dass wie ein Baukasten aufgebaut ist und aus dem die Level erstellt werden können. Den großen Vorteil den wir bereits in der Konzeptphase sahen, lag in dem Sparen an Speicher und Rechenzeit. Denn es musste ja lediglich das entsprechende Sourceimage für das Level geladen werden. Wiederholende Elemente die häufig in Leveln vorkommen, konnten so leicht und resourcensparend erstellt werden. Wir bauen unser Level sozusagen aus Puzzlestücken zusammen.

5.2 Das Image

Um aus dem Image selbst möglichst viele Informationen beziehen zu können, haben wir die Reihen aufgeteilt. Das bedeutet, Tiles die in verschiedenen Reihen sind, haben verschiedene Eigenschaften. So sind die einen etwa tödlich, sie sind Kisten oder sie sind quasi ohne Kollisions-Rechteck, so dass der Spieler daneben her gehen kann. Dies war notwendig, da wir unsere Positionen in einem zweistelligen Vektor speichern und die z-Achse nicht berücksichtigen.Gespeichert werden diese Eigenschaften in separaten bool-Werten.

5.3 Der Editor selbst und seine Funktionsweise

Der Editor ist streng genommen ein Gamescreen und teilt sich mit dem Spiel Klassen wie *Enemy* oder *Tile*. Beim Start gelangt man direkt in seinen Bearbeitungsmodus. Das Interface besteht aus einem Kachel-Raster. Diese Kacheln sind die möglichen Standorte der Tiles, Enemys oder Coins. Per Mausklick wird in ein freies Feld dann ein solches Objekt abgelegt. Das Programm speichert dieses in einem zwei-dimensionalen Array. Dadurch kann sehr schnell bestimmt werden, ob ein ausgewählter Bereich frei ist. Welches Tile gerade ausgewähl ist, sieht man in einem kleinen Kästchen. Man kann diese Auswahl auf zwei Arten beeinflussen.

1. Man drückt die Tasten p(für previous) oder n(für next). 2. Man ruft die Tile Übersicht auf, welche das gesamte Sourceimage anzeigt, und klickt dann das gewünschte Tile an. Möchte man Gegner oder Münzen setzen, oder etwa vorhandene Objekte löschen, muss man den Modus ändern. Dies geschieht entweder durch die jeweiligen Kurztasten oder durch das Dropin-Menü, welches erscheint wenn man den Mauszeiger aus dem rechten Bild bewegt.
Unser Leveleditor funktioniert derzeit nur in der Auflösung von 1920 x 1080. Zwar wird er auch in kleineren Auflösungen angezeigt, jedoch haben wir unsere Berechnungen nicht in allgemeinen Abhängigkeiten geschrieben, sondern auf diese Auflösung programmiert, da jeder von uns diese nutzt.
Das Speichern eines Levels geschieht indem die jeweiligen Klassen serialisierbar sind. Das bedeutet, alle Objekte die zuvor als *XMLELEMENT* deklariert werden, werden beim Speichern in die erzeugte XML-Datei geschrieben. Abgelegt bzw. erzeugt wird diese im Download-Ordner des Systems.

5.4 Kurztasten

- C: Coin / Münze platzieren
- D: Delete / Objekte löschen
- E: Enemy / Gegner platzieren
- F: EndFlag / Boss oder letzen Gegner platzieren
- G: Grid / Raster anzeigen
- M: Minimap / Minimap anzeigen
- N: Next / nächstes Tile oder Gegner
- O: Speichern
- P: Previous / Vorheriges Tile oder Gegner
- S: Select Tile / Tile aus Sourceimage wählen
- T: Tile / Tile platzieren
- -¿: nach rechts scrollen
- ¡-: nach links scrollen

5.5 Probleme, Lösung und Fazit

Wir hatten dabei Probleme mit den Filestreams und mussten da lange rumprobieren bis wir eine Variante gefunden hatten. Leider ist es uns bisher noch nicht gelungen, dass der Speicherort durch den User selbst bestimmt werden kann.
Des Weiteren sollte man vor dem Speichern die Kamera zum Anfang des Levels bewegen, da es sonst zu Fehlern bei der Positionierung der Gegner kommen kann. Leider ist uns bis dato schleierhaft warum dies so ist.
Ein weiteres Problem welches wir zu Beginn hatten, war das wir alle Objekte einfach in einer Liste speichern wollten, dadurch die Abfrage ob ein Platz frei war, sich als sehr kompliziert gestaltet hat. Dies lösten wir, indem wir dazu alle Objekte in ein Array schrieben und sie somit einen festen Platz hatten.

Abschließend möchten wir betonen, dass uns der Level-Editor sicherlich in manchen Phasen sehr gefordert hat, wir die Entscheidung einen zu schreiben aber nie bereut haben. Er ermöglicht uns große Level in verhältnismäßig kurzer Zeit zu schreiben. Würden wir die Infos per Hand in eine XML oder ähnliche Text-Datei schreiben, wäre das ein viel größere Aufwand. So konnten wir auch schnell viele Sachen testen und uns spezifisch für Testfälle Level bauen, da die XML vom Spiel ausgelesen wird und dann problemlos das gesamte Level erstellt.
Der Leveleditor ist momentan genau auf unsere Bedürfnisse zugeschnitten. Es ist lediglich möglich die Level für unser Spiel zu erstellen. Wenn dritte den Editor nutzen wollen, müssten sie entweder ihr Spiel dementsprechend mit den Klassen aufbauen oder den Leveleditor anpassen.

6

Reflexion

Diese Projekt hat uns Dreien sehr viel Spaß gemacht. Vor allem weil wir eine Idee umsetzen konnte die uns selbst begeisterte. Im Nachhinein würde wir so manche Dinge anders angehen als zuvor. Beispielsweise würde wir uns ein Klassendiagramm erstellen um zumindest schon mal einen groben Überblick über die Struktur des Programms zu bekommen. Außerdem würden wir die Object-Struktur wie wir sie im Kapitel *Implementierung der Spiellogik* beschrieben haben, beim nächsten Mal sofort anwenden. Auch würden wir nicht mehr ein Opensource Framework verwenden, da es doch an einigen Funktionalitäten mangelt die im Original vorhanden sind. Dies führte bei uns zwangsläufig zu Problemen, die dementsprechend auch beseitigt werden mussten. Dies sind nur einige Erkenntnisse aus unserer Arbeit.
Persönlich konnte jeder von uns aus diesem Projekt viel mitnehmen. Natürlich wurden unsere Programmierfähigkeiten und -erfahrungen verbessert, mit neuen Fehlern kann so schneller umgegangen werden. Der Umgang mit Grafiktools wie Adobe Photoshop konnte ebenfalls erlernt bzw. verbessert werden. Denn in unserem Fall sind alle Grafiken und auch Sounds, selbst erstellt. Nicht zu vergessen ist das Debuggen, das wir häufiger anwenden mussten. Je länger wir an dem Projekt saßen, umso besser und schneller wurden wir.
Wir haben mit diesem Projekt noch nicht abgeschlossen. Zu sehen ist ja nur ein Demolevel. Unser Ziel ist es, das Spiel fertig zustellen und dazu haben wir noch viele Ideen in unseren Köpfen. Allerdings zeigt dieses Demolevel schon alle Kernelemente die auch in einem fertigen Spiel enthalten sein werden. Unserer Auffassung nach ist der Code zu 75% fertiggestellt. Jedoch wird im Bereich des Gameart noch eine Menge Arbeit warten und vielleicht ergeben sich daraus auch noch neue Elemente die es zu programmieren gilt.