Abstract

Diese Arbeit befasst sich mit dem Erstellen eines 2D-Action-Adventures mit Rollenspielelementen. Sie soll einen Überblick über verschiedene Aspekte der Realisierung geben. Zuerst wird die Spielidee und die verwendeten Spielmechanismen beschrieben und daraus eine Anforderungsdefinition erstellt. Nachdem das verwendete Framework kurz erläutert wurde, wird das softwaretechnische Konzept zur Realisierung vorgestellt. Die Umsetzung der Komponenten Steuerung, Spieleditor, Sound und Grafik wird aufgezeigt. Bei der grafischen Umsetzung wird ein besonderes Augenmerk auf die Abstraktion von Licht und Schatten in die 2D-Spielewelt gelegt.

Abstract

This work is concerned with creating a 2D action-adventure with roleplay elements. It provides an overview over various tasks of the implementation. First, the game idea and the used gamemechanism are verified and a definition of requirements is created. After introducing the used framework, the software engineering concept for realization is presented. The implementation of control components, game editor, sound and graphics is shown. The graphical implementation pays special attention to the abstraction of light and shadow into the 2D game world.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

....

(Ort, Datum) (Unterschrift)

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit
Christian Schmitt
(Matr.-Nr. 209 210 238)

Thema: **Entwicklung eines interaktiven 2D Multiplayer-Adventures mittels XNA 4.0 Framework in Visual Studio mit C#**

Die Entwicklung eines Computerspiels ist für mich ein persönlicher Traum und unter anderem einer der Gründe, warum ich mich für das CV-Studium entschieden habe. Diverse Techniken der Computergrafik und verschiedenste Programmieransätze sind nötig um solch ein Spiel zu realisieren.

Hauptziel: In dieser Arbeit soll ein interaktives Adventure vergleichbar mit dem Spielklassiker Zelda entwickelt werden: Man zieht mit seinem Helden in einer 2D Landschaft umher, tötet Monster und erledigt diverse Aufgaben und Rätsel. Darüber hinaus Entwicklung eines leistungsfähigen Spieleditors, sowie Abstraktion von Licht und Schatten in die 2D Welt. Das fertige Spiel soll zunächst für Windows realisiert werden. Als Entwicklungsumgebung sehe ich VisualStudio, XNA framework und C# vor.

Schwerpunkte dieser Arbeit sind:

1. Einarbeiten in die Entwicklungsumgebung
2. Recherche zum Thema
3. Erstellen eines softwaretechnischen Konzepts um die Spielwelt/Spieleditor abzubilden
4. Implementierung
5. Weiterentwicklung
6. Dokumentation und Evaluierung der Ergebnisse

Koblenz, den 22.1.2013

- Christian Schmitt -   - Prof. Dr. Stefan Müller-

Contents

1 Einleitung **6**
1.1 Vorwort 6
1.2 Motivation 6

2 Systemvision **7**

3 Das Spiel **8**
3.1 Titel und Handlung 8
3.2 Regelwerk 8
3.3 Verwendete Spielmechaniken 9
3.3.1 Standard-Spielmechaniken 9
3.3.2 Spielmechaniken im Multiplayer-Modus 10

4 Anforderungsdefinition **11**

5 XNA-Framework **12**
5.1 Programmiersprache C# 12

6 Softwaretechnisches Konzept **13**
6.1 Gesamtaufbau 13
6.2 Aufbau der Spielelemente 14
6.2.1 Weitere Vererbung 17
6.2.2 Beispiele für Spielelemente 18
6.3 Factoryklassen 19
6.4 Datenflusssicht 20

7 Grafik **22**
7.1 Spriteanimation 22
7.1.1 Kombination mit 2D-Translationen 22
7.2 Texturatlas 23
7.3 Tiefenwertanpassung 24
7.4 Abstraktion von Licht und Schatten 26
7.4.1 Kreisrunde Lichtkegel 26
7.4.2 Schattenfühler 28
7.4.3 2D-Texturschatten 33
7.4.4 Finale Komposition 37

8 Sound- und Musikenviroment **38**
8.1 SoundAtlas 38
8.2 Angepasste Hintergrundmusik 38
8.3 Simpler 2D Ambient Sound 38

9 Steuerung **39**
9.1 Allgemein 39
9.2 Kollisionserkennung 39
9.3 Bewegungskorrektur 42

10 Editor **45**
10.1 Grundüberlegung zum Design 45
10.2 Komponenten 45
10.3 Benutzeroberfläche 46
10.4 Genauere Beschreibung 47

11 Evaluation **48**
11.1 Evaluation über Anforderungsliste 48
11.2 Evaluation über Benutzertests 49
11.2.1 Aufbau des Benutzertests 49
11.2.2 Auswertung des Benutzertests 50

12 Fazit **53**
12.1 Ausblick 53

13 Danksagung **54**

1 Einleitung

1.1 Vorwort

Die Grafik von Computerspielen verbessert sich fortwährend mit der verfügbaren Hardware. Grafik ist bei der Spieleentwicklung aber nur einer von vielen Faktoren, die das Gesamtkonzept eines Spiels ausmachen. Elemente wie Steuerung, Spielidee und die Geschichte die im Spiel erzählt wird, sind wichtige Elemente. Spiele wie Tetris oder Zelda zeigen, dass ein bestehendes "altes" Spielprinzip auch heute noch erfolgreich sein kann. Tatsächlich nutzen viele 3D-Spiele nicht alle Möglichkeiten der dreidimesionalen Welt aus, sondern bleiben in ihrer Spielmechanik auf der zweidimensionalen Ebene. Manche Spiele wie *New Super Mario Bros.* nutzen die 3D-Grafik wirklich nur für die Darstellung. Das Spielprinzip unterscheidet sich dabei im Wesentlichen nicht von dem älteren 2D-Genrevertreter.

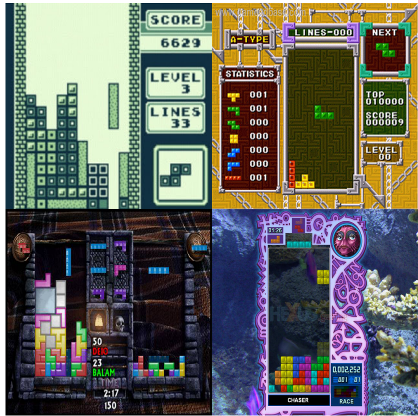


Figure 1: l.o. Tetris NES, r.o. Tetris SNES, l.u. Tetris N64, r.u. Tetris X-BoX

Figure 2: l.o. Zelda NES, r.o. Zelda SNES, Zelda The Minish Cap, Zelda X-BoX

1.2 Motivation

Die Weiterentwicklung von 2D-Grafik ging mit der Einführung der ersten 64 Bit-Spielkonsole in Europa [4] ab 1997 zurück. Nach wie vor kamen weitere Titel aus diesem Genre heraus, doch an grafischen Neuerungen im Bereich der 2D-Grafik mangelte es immer mehr.
In dieser Arbeit werden Erfolgskriterien des 2D-RPG-Genres analysiert, um ein archetypisches Schema zu erstellen. Der softwaretechnische Aufbau wird gezeigt. Grafische Neuerungen werden aufgezeigt ohne den Schritt in die dritte Dimension zu unternehmen.

2 Systemvision

Es wurden verschiedene Spiele aus dem Genre des 2D-RPG analysiert, um daraus Spielidee, Regelwerk, Spielmechanismen und Anforderungsdefinitionen abzuleiten.
Folgende Spiele lagen der Ideenfindung zugrunde:
Zelda: A link to the past, Zelda: The minish Cap, Terranigma, Secret of mana, Seiken Densetsu, Zelda: The Spirit Track.

Spielidee:
Bei diesem Projekt handelt es sich um ein 2-dimensionales Action-Adventure. Man läuft mit bis zu drei Spielern durch eine Welt und erledigt in ihr diverse Aufgaben. Die Spielfigur wird mittels einem Gamepad gesteuert. Hauptziel ist es bestimmte spielrelevante Gegenstände, sogenannte Tokens, einzusammeln. Dazu müssen etliche Gegner besiegt oder umgangen werden und verschiedene Rätsel gelöst werden.
Hierzu stehen eine Fülle von Gegenständen und Hilfsmitteln zur Verfügung. Die "Tokens" sind an speziellen Orten in einer Spielwelt versteckt.

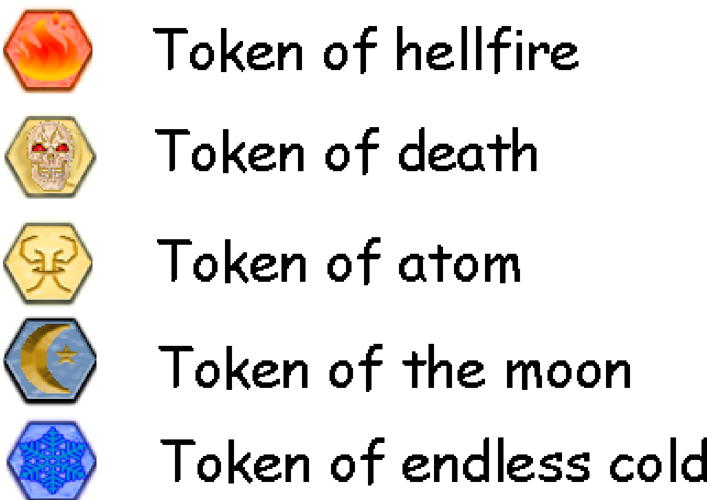


Figure 3: Tokens zu sammeln ist das zentrale Ziel des Spiels

Anforderungen an den Spieler:
- Geschicklichkeit
- Kombinatorisches Denken
- Orientierungsfähigkeit
- Ressourcenmanagement
- Gegebenenfalls Teamwork
- Timing

3 Das Spiel

3.1 Titel und Handlung

Das Spiel trägt den Namen "Infinite Worlds". In den großen Götterkriegen vernichteten sich die Götter gegenseitig, ihre Überreste regneten in Form von Sternstaub auf die Welten hernieder. Die finale Schlacht setzte soviel Macht frei, dass ein Riss im Universum entstand und alle Welten und Zeiten ineinander gerissen wurden.
Die alten Götter waren verschwunden, doch ihre Macht existierte in dem verteilten Sternstaub weiter. Die Menschen begriffen erst nach und nach seine Bedeutung und so kam es zur Fertigung der ersten "Tokens". Aus den Resten der Götter geschmiedet, bergen sie unglaubliche Kräfte.
Es wurde sich bewußt für diese Art der Handlung entschieden, da sie die größtmöglichen Freiheiten zulässt, was die Wahl der Schauplätze, Gegner und gefundener Gegenstände betrifft. Von der Steinzeit bis zur fernsten Zukunft kann alles handlungstechnisch begründet werden. Auch abstrakte Szenarien, wie z.B. eine Welt aus Feuer sind denkbar.

3.2 Regelwerk

- Jeder Spieler steuert eine Spielfigur.
- Jeder Spielfigur hat eine bestimme Anzahl an Lebenspunkten, eine Kollision mit einem Gegner oder einer feindlichen Attacke verringert diese Lebenspunkte. Lebenspunkte können durch Sammeln von Gegenständen zum Beispiel Heiltränken wieder aufgefüllt und durch Finden von sogenannten Herzcontainern kann die Anzahl der maximalen Lebenspunkte erhöht werden.
- Ziel des Spiels ist es einen bestimmten Gegenstand (Token) zu finden.
- Die Spielfigur verfügt über eine begrenzte Anzahl von Gegenständen, von denen er immer jeweils drei ausrüsten kann. Nur ausgerüstete Gegenstände können auch benutzt werden. Neue Gegenstände können im Laufe des Spiels gefunden oder für die im Spiel verfügbaren Währungen (Diamanten oder Sternstaub) bei Händlern gekauft werden.
- Diamanten können unter Spielelementen im Level versteckt, bei besiegten Gegnern oder in Truhen gefunden werden.
- Um Sternstaub zu erhalten ist meist das Lösen eines Rätsels notwendig.
- Jede Spielfigur verfügt über eine bestimmte Anzahl an Manapunkten. Das Verwenden bestimmter Gegenstände kostet Manapunkte, sie können ähnlich wie Lebenspunkte, durch Tränke aufgefüllt werden. Durch sogenannte Manacontainer können die maximalen Manapunkte erhöht werden.
- Durch das Sammeln von Sternstaub können neue Fähigkeiten der Spielfigur freigeschaltet werden. Es wurde sich bewußt gegen das gängige Erfahrungspunktesystem entschieden, um das wiederholte Töten von Gegnern zu minimieren und den Anreiz zum Lösen von Rätseln zu erhöhen.

3.3 Verwendete Spielmechaniken

3.3.1 Standard-Spielmechaniken

- **Post-Hit-Invincibillity**

  Nach einem Treffer wird das getroffene Objekt für einen kurzen Zeitraum (hier: 500 Millisekunden) unverwundbar. Dadurch wird verhindert, das Gegner in kürzester Zeit das komplette Leben des Spielers abziehen können. Dadurch hat der Spieler Zeit zu reagieren und es erzeugt ein konstantes Verhalten der Software bei verschiedener Rechenleistung des Systems. Ohne Einbeziehung dieses Zeitraums würde der erhaltene Schaden um so schneller das Leben verringern, je leistungsfähiger das System wäre. Schaden wird durch Kollision ermittelt, somit für den gesamten Zeitraum, den der Spieler in der Kollision mit einer Schadensquelle steht und somit mit jedem Durchlauf der Main-loop Schaden erzeugt werden würde.

- **Vulnerabillities/Resistances**
  In "Infinite Worlds" werden 4 verschiedene Arten von Schaden unterschieden: Feuer-, Eis-, Blitz- und physikalischer Schaden. Die zerstörbaren Objekte im Spiel sind besonders anfällig oder resistent gegen bestimmte Typen.

- **Melee-Combat-Knockback**
  Englisch für "Nahkampf-Rückstoß"
  Alle Nahkampfwaffen stoßen getroffene Ziele eine bestimme Strecke zurück. Einerseits hilft dies Gegner auf Abstand zu halten, andererseits kann es genutzt werden um Gegner durch strategisches Verschieben z.B. in Lava zu besiegen.

- **Enemy-Respawn**
  Mit Ausnahme von Endgegnern werden besiegte Gegner mit einer bestimmten Wahrscheinlichkeit wiederbelebt, wenn der Spieler nach dem Verlassen des Raumes in diesen zurückkehrt. Dabei werden sie auf ihre Startposition zurückgesetzt. Dadurch wird vermieden, dass ein Gegner einen Eingang blockiert bzw. den Spielern sofort Schaden zufügt, ohne dass sie reagieren können.
  Als ein geeigneter Richtwert für die Respawn-Wahrscheinlichkeit hat sich ein Wert von 50 % erwiesen. Da Ressourcenmanagement ein wichtiger Bestandteil des Spiels ist, gibt die oben genannte Wahrscheinlichkeit die Möglichkeit in dafür günstigen Räumen mehrmals alle Gegner zu besiegen. Hierdurch ist es möglich Ressourcen aufzustocken, jedoch zwingt es den Spieler nach $log_2(Gegneranzahl)$ Durchläufen weiterzuziehen. Da auch Lebenspunkte eine wichtige Ressource sind, welche im Laufe des Spiels verbraucht werden, kann das Senken der Gegneranzahl sehr nützlich sein. Insbesondere bei weit verzweigten Kampagnen, die es notwendig machen, einzelne Räume mehrmals aufzusuchen.

3.3.2 Spielmechaniken im Multiplayer-Modus

- **Shared Inventory**
  Da "Infinite Worlds" ein kooperatives Spiel ist, werden gesammelte Ressourcen geteilt, ansonsten würden die Spieler für das Vorankommen im Spiel um benötigte Ressourcen konkurrieren. Ein geteiltes Inventar bietet die Option strategischen Austauschens von Gegenständen.

- **Collective Game-Over**
  Erst wenn alle Spieler gestorben sind, endet das Spiel.

- **Save the dying**
  Sinken die Lebenspunkte eines Spielers auf 0 oder weniger, liegt dieser am Boden und kann nicht mehr handeln.
  Wenn sich ein anderer, noch lebender Spieler 4 Sekunden zu ihm positioniert, wird der am Boden liegende Spieler wiederbelebt und kann, wenn auch nur mit wenigen Lebenspunkten, wieder handeln.
  Am Boden liegende Spieler können von lebenden Spielern verschoben werden, so ist es z.B. möglich, jemanden aus einem Gefahrenbereich zu entfernen.

4 Anforderungsdefinition

1.) Gameplay

(a) Standard & Multiplayerspielmechanismen laut Kap. 3.3.

(b) verschiedene Gegner unterschiedlicher Art

(c) Enemy Drop-, Dying- and Respawnverhalten

(d) verschiedene Gegenstände (Items)
-Item-Shops
-verschiedenes Verhalten der Gegenstände, multiple Anwendungen für ein und denselben Gegenstand

(e) Rätsel mit unterschiedlichen Lösungsansätzen

(f) interaktive Spielelemente

(g) Ressourcenmanagement (Mana, Trefferpunkte, Geld)

(h) Kampagne
-Tutorial
-Learning by doing
-Backtracking for advancing
-Story
-Cut-Scenes

(i) Collision Detection (Rectangle und Circle)

(j) Laden/Speichern von Spielständen

2.) Grafik

(a) Spritegrafik

(b) Spriteanimation

(c) Simple Effekte wie z.B. Glow / Transparenz

(d) Darstellung von Schatten und Licht

3.) Editor

(a) Auswahlboxen für Spielelemente

(b) Setzen/Löschen von Spielelementen

(c) Rotation von Spielelementen

(d) Verschieben von Spielelementen

(e) Erstellen/Löschen von Räumen

(f) Steuervariablen von Spielelementen setzen

(g) Laden/Speichern bestehender Levels

(h) Übersichtskarte

(i) Sicht (mit Zoomverhalten)

(j) Verschiedene Texturen auswählbar

3.) Sonstige Anforderungen

(a) Möglichkeit zur Verwendung von einem X-Box-Controller

(b) Angepasste Musik an jeweilige Spielsituation

(c) Soundeffektlautstärke relativ zu Position des Spielers

(d) Mehrspielermodus

(e) Optionsmenü um Sound, Musik, Tastenbelegungen einzustellen

5 XNA-Framework

XNA ist eine Technologie zur Spieleentwicklung für Microsoft Windows, Xbox 360, sowie Windows Phone 7. XNA vereint verschiedene Spiele-Entwicklungs-Programmierschnittstellen, unter anderem Direct3D aus DirectX Version 9.0c für die Darstellung von 2D- und 3D-Grafiken, XACT als plattformübergreifende Schnittstelle für Ausgabe von Audiodaten und XInput zur Kommunikation mit allen nötigen Peripheriegeräten, in einem gemeinsamen Framework.

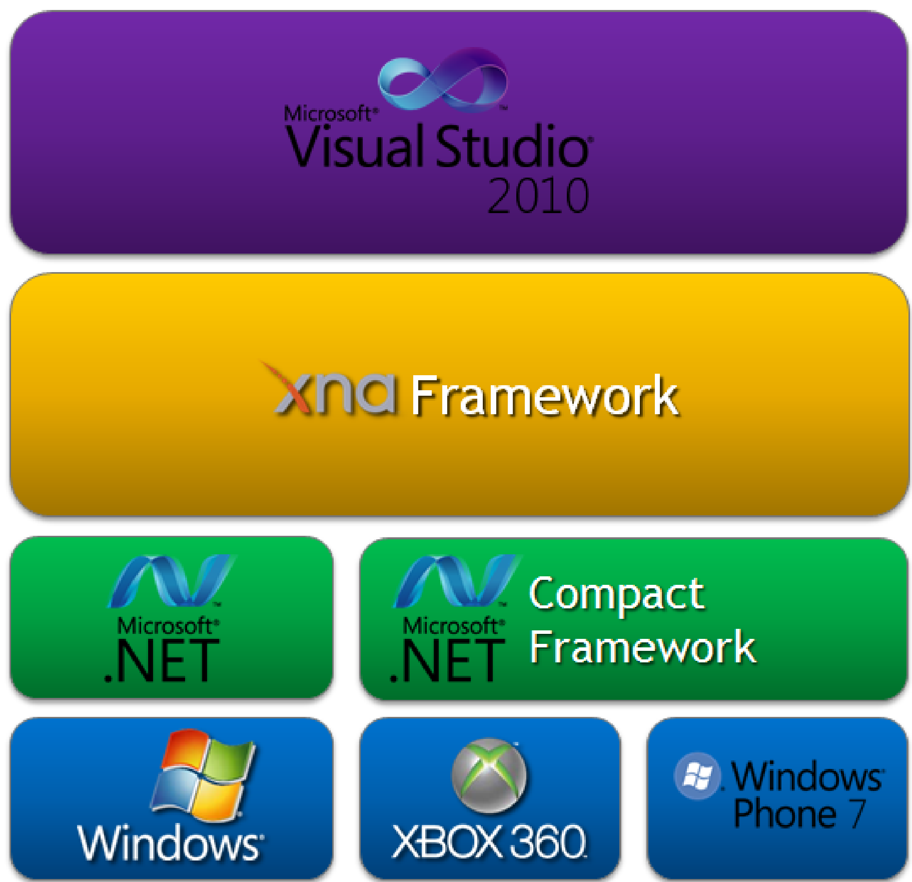


Figure 4: Arbeitsprinzip des Gesamtframeworks

5.1 Programmiersprache C#

C# greift Konzepte der Programmiersprachen Java, Haskell, C, C++ sowie Delphi auf. C# zählt zu den objektorientierten Programmiersprachen und unterstützt die Entwicklung von sprachunabhängigen .NET-Komponenten und bietet vielseitige Anwendungsmöglichkeiten.

6 Softwaretechnisches Konzept

6.1 Gesamtaufbau

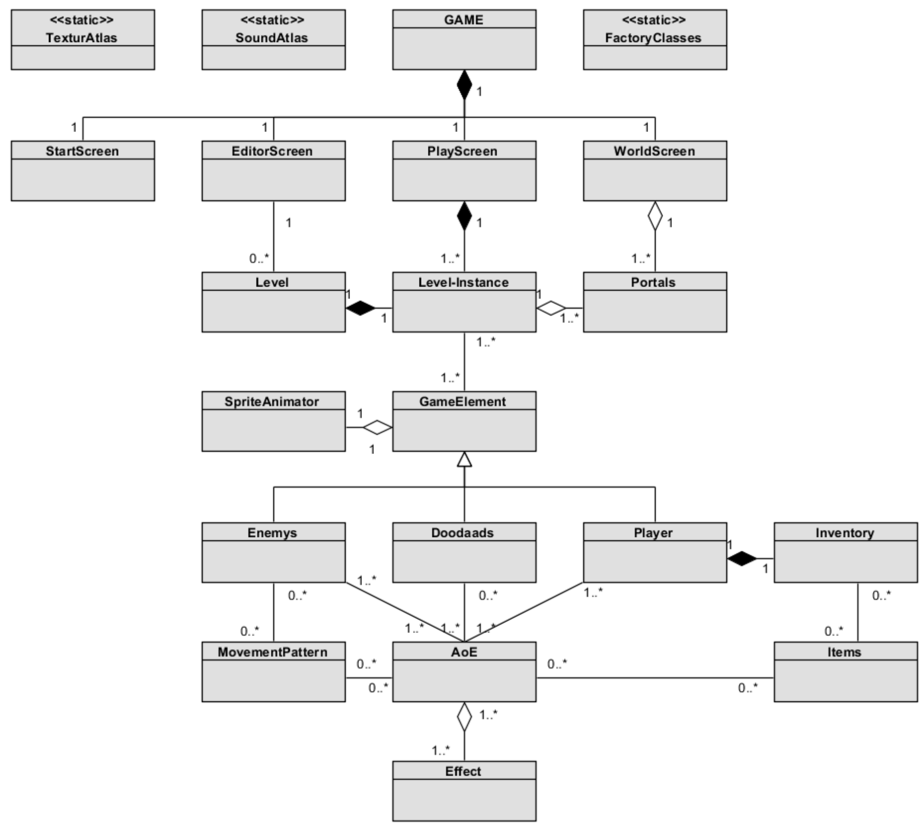


Figure 5: Arbeitsprinzip des Gesamtframeworks

Das Spiel an sich wird in vier verschiedene Bildschirme unterteilt:

1. **Startscreen**
   Hier befindet sich das Startmenü, von dem aus alle anderen Screens erreichbar sind.

2. **Editorscreen**
   Hier befindet sich der Spieleditor mitsamt Benutzeroberfläche. Von hier aus kann nur zum Playscreen gewechselt werden. Wenn das Spiel im Playscreen endet und es vom Editor aus gestartet wurde, wird zum Editor zurückgekehrt.

3. **Playscreen**
   Hier findet das eigentliche Spielen statt. Wurde das Spiel über das Startmenü gestartet, wird nach Spielende zum Startmenü zurückgekehrt. Durch die Verwendung von Portalen ist es möglich zwischen Play- und Worldscreen zu wechseln.

4. **Worldscreen**
   Hier befindet sich eine Landkarte der Spielwelt.

6.2 Aufbau der Spielelemente

Als oberste Elternklasse wird die Klasse Gameelement verwendet.Die Klasse Gameelement verfügt über eine Liste von Attributen. Diese bilden die Grundlage bei der Interaktion mit anderen Spielelementen. So kann z.B. ein Magnet jedes Spielelement anziehen auf das die Attribute *istMagnet* oder *istMetallisch* zutreffen.

Übersicht über ein paar Attribute

istMagnet	hatSchatten	besitztArme
istPassierbar	ignoriertLevelgrenzen	hatBlut
istLichtquelle	istVerschiebbar	istMetallisch
istAmSterben	istAmLeben	erzeugtGegner
istUnzerstörbar	wurdeGetroffen	immunGegenDoodaadAoE
istVerschiebbar	bewegtSich	reflektiertStrahlen
wurdeAktiviert	kannAngreifen	kannSprechen

Die Klasse Gameelement besitzt eine Menge nützlicher Funktionen, von denen hier exemplarisch die drei wichtigsten kurz erläutert werden:

public virtual void loadContent(...)...

LoadContent wird genau einmal bei Initialisierung des Levels ausgeführt, hier werden Identifier für Texturen und Sound für das betreffende Spielelement festgelegt.

public virtual void update(GameTime gameTime,...)...

In der Update-Methode wird die Spiellogik des Spielelements ausgeführt, dazu gehören z.B. Kollisionserkennung, Daten sammeln, Sound abspielen, Bewegungen ausführen, Interaktion mit anderen Spielelementen etc. Bevor die Spiellogik ausgeführt wird, wird zuallerst getestet, ob das Updaten überhaupt nötig ist. Sollte dieser Test negativ ausfallen, wird der "wasUpdated"-Flag auf false gesetzt und die Ausführung wird sofort beendet.
Einer der wichtigsten Parameter dieser Methode ist gameTime. Er enthält einen Zeitstempel, mit dem systemunabhängig ein konstantes Verhalten für Bewegungen, Spriteanimation, Kollisionserkennung, Angriffsverhalten und verschiedene andere Funktionalitäten gewährleistet werden kann.

public virtual void draw(...)...

Draw wird aufgerufen, wenn das Spielelement gezeichnet werden soll. Ist der "wasUpdated"-Flag auf false gesetzt, wird draw nicht ausgeführt.
Jedes Spielelement enthält immer eine Memberklasse namens spriteAnimator (vgl. Kap.7.1), so kann jede beliebige Textur bei Bedarf animiert werden.

Auf der ersten Vererbungsebene werden 4 Klassen abgeleitet, mit denen alle Spielelemente abgebildet werden können:

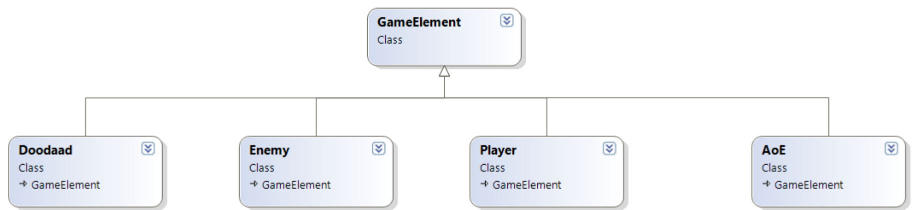


Figure 6: Klassenstruktur auf oberster Ebene

Im Folgenden werden die abgeleiteten Klassen und ihre Erweiterungsfunktionen erläutert:

Enemy (engl. Gegner)

Besitzt Bewegungsmuster (engl. Movement-Pattern), Angriffsverhalten, Dying-, Drop- und RespawnBehavior.

public virtual bool checkRespawn()...

Getötete Gegner werden nicht komplett aus dem Spiel entfernt, sondern kommen in eine Warteliste. Beim Betreten des Raumes werden für alle Elemente dieser Liste die *checkRespawn*-Methode ausgeführt.
Über eine Wahrscheinlichkeit wird entschieden, ob der Gegner wieder auftaucht. In diesem Fall werden alle Werte auf ihre Initialwerte gesetzt. Der Gegner wird von der Warteliste entfernt und der Liste der aktuellen Gameelemente hinzugefügt. Taucht er nicht wieder auf, wird er aus der Warteliste entfernt und kann im aktuellen Spiel nicht mehr auftauchen.

public virtual bool checkDropping()...

Diese Methode wird aufgerufen, wenn ein Gegner stirbt. Über eine Wahrscheinlichkeit wird entschieden, ob der Gegner etwas zurücklässt. In diesem Fall wird eine Membervariable namens *droppedItem* gesetzt.

Player (engl. Spieler)

Player ist ein dynamisches Objekt und besitzt Schnittstellen zum Einlesen und Verarbeiten von Eingaben, in denen die Steuerung der Spielfigur definiert wird. Außerdem besitzt es ein Revivebehavior und eine Memberklasse namens Inventory. In dieser wird die jeweilige Itemlogik der Spielfigur verwaltet.

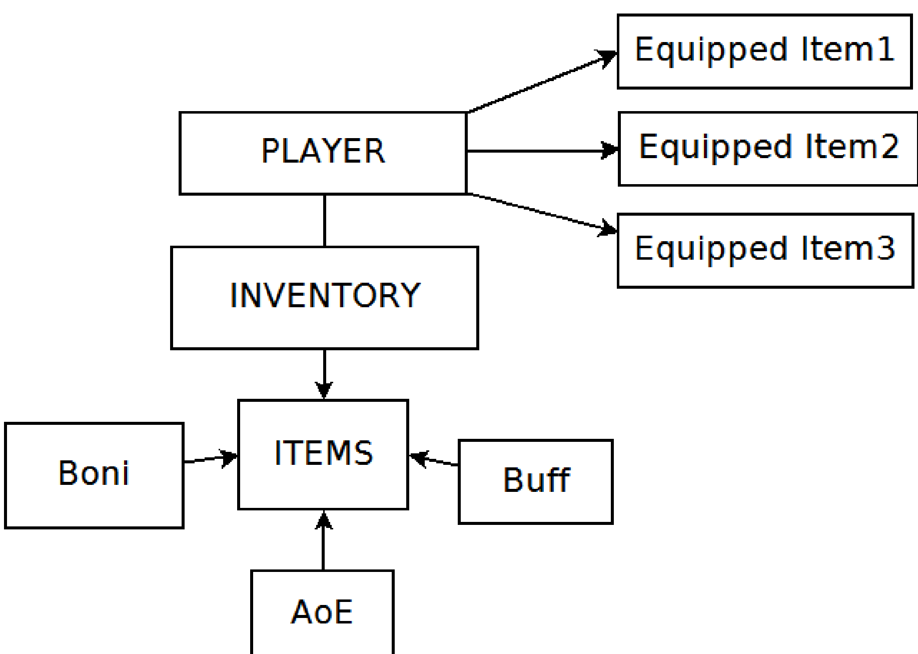


Figure 7: Über die Memberklasse Inventory werden die getragenen Items und deren Wirkung auf den Spieler verwaltet

AoE ("Area of Effect", engl. Wirkungsbereich)

AoE ist ein dynamisches Objekt. AoE sind Gebiete die bei Kollision mit GameElementen Effekte auf diese ausüben. Kann entweder über ein eigenständiges Movement-Pattern bewegt werden oder über ein Bewegungsmuster relativ zu dem Erzeuger des AoE.
Es werden zwei Sorten von AoE unterschieden:

1. Zeitlich-gebundene: AoE verschwindet nach einer vorgegebenen Zeitspanne.
2. Kausal-gebundene: AoE verschwindet nach Eintreten eines Ereignisses, z.B. eine Kugel die auf ein Hindernis trifft und dann dort zerschellt.

public virtual void refresh()...

Diese Methode wird benutzt um den AoE in seinen Ausgangszustand zurückzuversetzen. Das macht die Klasse wiederverwendbar und vermeidet das unnötige Erstellen eines neuen Objekts.

public virtual AoE getCopy()...

Bestimmte Kontexte machten es notwendig zwei voneinander unabhängige Instanzen eines aktuellen AoE zu verwenden (z.B. der Gegner Copy-Cat, vgl. Fig.9). Diese Methode erzeugt eine weitere Instanz im selben Zustand.

Doodaad (engl. Ding)

Doodaads sind statische (z.B. Wand) oder dynamische Objekte (z.B. Feuer), sie können ihr komplettes Verhalten je nach eingehendem "Reiz" (meist Kontakt mit AoE) ändern können.
Statische Doodaads enthalten noch eine zusätzliche Datenstrukur, welche ihre Nachbarschaften (oberer, unterer, linker, rechter) zu anderen statischen Doodaads erfasst. Dies ist nützlich für Move-Correction (vgl. Kap. 8.3), Schattenberechnung (vgl. Kap. 7.4.3) und bestimmte Movement-Pattern (z.B. wenn ein Gegner sich immer entlang einer Wand bewegen soll). Sollte ein statisches Doodaad verschwinden oder verschoben werden, wird die Nachbarschaftstruktur neu berechnet.

6.2.1 Weitere Vererbung

Mit den gegebenen Grundklassen werden alle vorkommenden Gameelemente in der zweiten Vererbungsebene modelliert. Der besondere Vorteil ist, dass neue Gameelemente nur ihre Erweiterungslogik enthalten müssen und die restliche benötigte Funktionalität von den Elternklassen übernommen werden kann. So kann aus einem Movementpattern und einem AoE mittels weniger Codezeilen ein neuer Gegner erstellt werden.

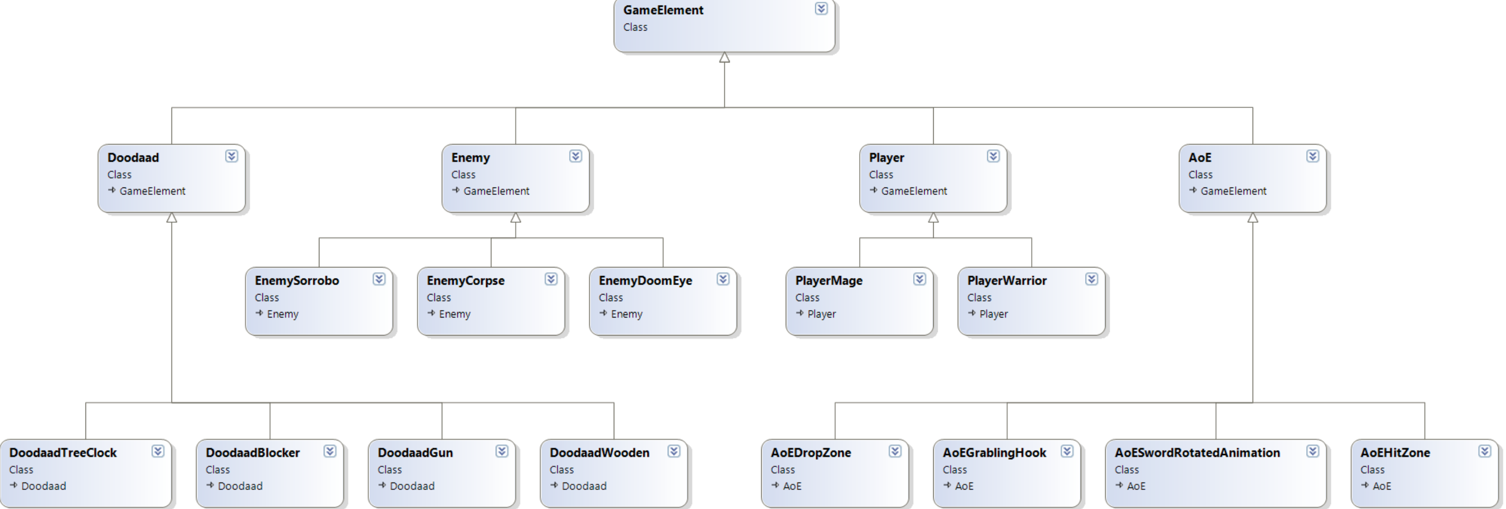


Figure 8: text

Ab der zweiten Vererbungsstufe bietet es sich für bestimmte Spielelemente an, aus diesen weitere Kinder zu bilden. Siehe hierzu Fig. 10. "Magic Box" und "Magic Magnet Box".

6.2.2 Beispiele für Spielelemente

Im Rahmen dieses Projektes wurden 52 verschiedene Doodaad- und 45 Enemyklassen erstellt. Nachfolgend werden exemplarisch ein paar davon erläutert, um die Möglichkeiten der gewählten Klassenstruktur aufzuzeigen.

Felsenmonster:
zerstört Wände und schleudert diese Wand dann nach dem Spieler

Zombie:
läuft planlos umher, bis Spieler zu nahe kommt, dann stürmt er zum Spieler

Staubwurm:
bewegt sich abwechselnd ober- und unterirdisch, kann sich an Spieler festbeißen

Copy-Cat:
kopiert die Attacken des Spielers und greift mit diesen an

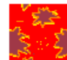
Changeling:
kann sein Äußeres in alles verwandeln was sich in seiner Nähe befindet

Magic Box:
kann verschoben werden, aktiviert Schalter wenn sie auf dem Schalter steht, kann Strahlen umleiten

Magic Magnet-Box:
gleiche Eigenschaften wie Magic Box und kann zusätzlich metalische Dinge und Gegner verschieben

Strahlgenerator:
erzeugt gefährlichen Laserstrahl, kann durch Blitzschaden an und aus geschaltet werden

Orb des Blutes:
kann genutzt werden um Lebenspunkte wieder aufzufüllen, Orb wird gefüllt durch das Töten von Gegnern in seiner Nähe



Lava:
kann mittels Eisschaden kurzzeitig passiert werden ohne Schaden zu bekommen

Figure 9:

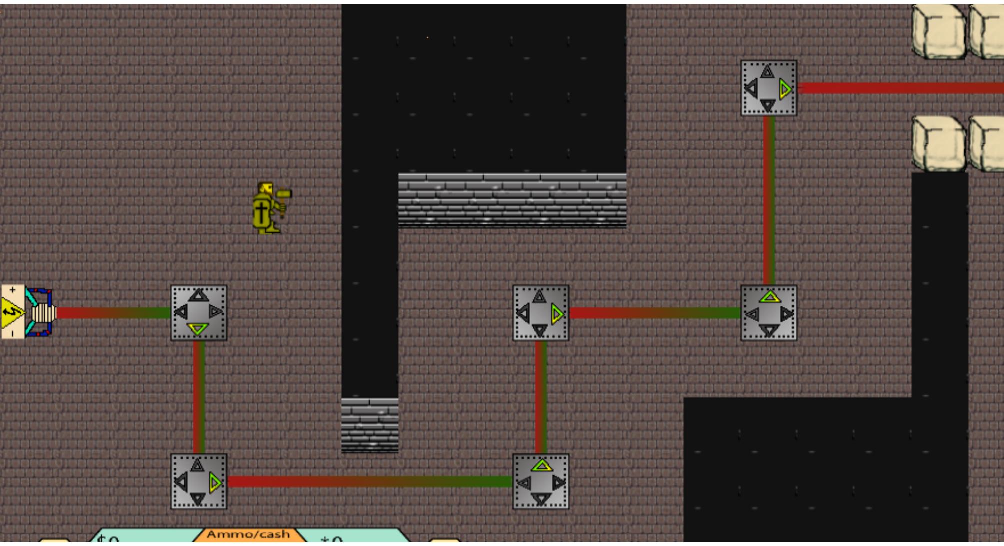


Figure 10: Zusammenspiel verschiedener Spielelemente:
Durch geschicktes Platzieren verschiedener "Magic Boxen" kann der Laserstrahl des Strahlengenerators links so umgeleitet werden, dass er die Eisblöcke rechts oben schmilzt

6.3 Factoryklassen

Die Anzahl der vorhandenen Gameelemente eines Levels kann sich im Spielverlauf erhöhen. Um unnötige Mengen an Konstruktorcode in den Klassen zu vermeiden, wurden im Rahmen dieser Arbeit sogenannte Factoryklassen verwendet.
Factoryklassen sind statische Klassen und im Entwurfsmuster angelehnt an dem Factory-Pattern [11]. Sie besitzen load-Methoden die mittels einer Spielelement-ID oder dem Spielelementnamen direkt eine Instanz des Elements liefern. Zu jeder Instanz ist durch Namen oder ID auch ein gegebenes Set an Parametern vorhanden. So kann z.B. ein bestimmter Gegnertypus in verschiedenen Ausführungen erzeugt werden. Das vereinfacht das Balancing[1] der geladenen Spielelemente. Änderungen, die in der Loaderklasse vorgenommen wurden, beziehen sich dann auf alle Instanzen dieser Klasse.

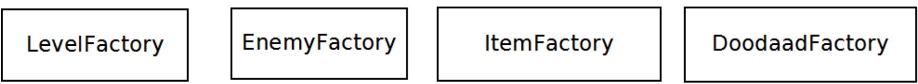


Figure 11: Verschiedene Factoryklassen

Die Factoryklassen könnten noch um einen XML-Reader erweitert werden. Somit kann eine Schnittstelle für Gamedesigner geschaffen werden, welches es ermöglicht, Werte im Spiel zu beeinflussen, ohne direkt am Code arbeiten zu müssen.

[1](engl. für Ausbalancieren) Beschreibt den Prozess, ein faires und anspruchvolles Spiel zu erzeugen, Balancing verhindert z.B. das Gegner zu viele Trefferpunkte haben oder zuwenige

6.4 Datenflusssicht

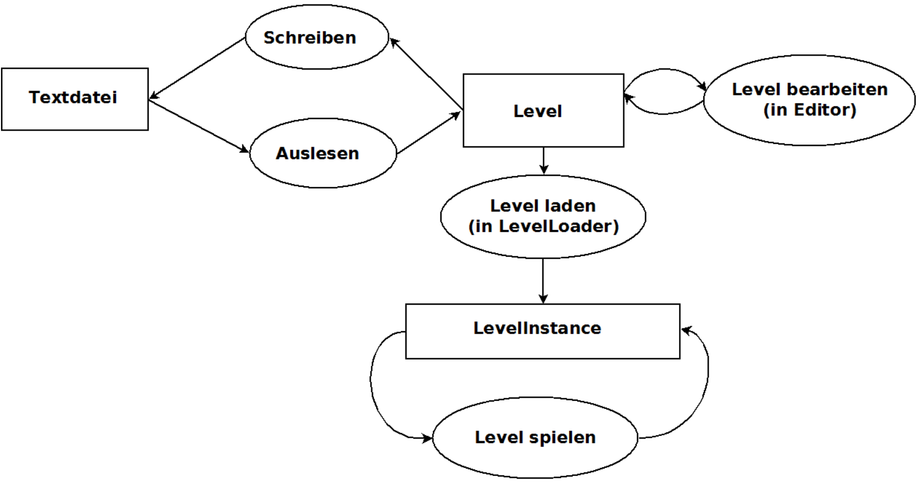


Figure 12: Datenflusssicht

Eine Kampagne wird dargestellt als eine Menge verknüpfter Level (Räume). In der Klassenstruktur existieren drei Abstraktionen von Typ Level, welche im Folgenden kurz erläutert werden:

Das Level als Textdatei

Diese Repräsentation wird benutzt, um die Level zu speichern und bei Initialisierung des Spiels wieder laden zu können. Den einzelnen Spielelementen im Level werden IDs in Form von Strings zugeordnet. Aus der Position der IDs in den Files (Matrixaufbau) und der ID selbst lassen sich die Position im Level und die Art des Objekts somit rekonstruieren. Die ID besteht aus einem Objektidentifier und einem angehängten Informationstag in dem sich Eigenschaften des Objektes speichern lassen, z.B. ob eine Tür offen oder geschlossen ist. Die Auswertung des Informationstags erfolgt in der zugehörigen Factoryklasse (vgl. Kap. 6.3), in der die Art der Auswertung frei implementierbar ist. Somit können beliebige Arten von Zusatzinformationen gespeichert werden.

```
X -----x0000 X -----x0000 X B0460x0000 X -----x0000 X -----x0000 X -----
X -----x0000 X -----x0000 X M0308x0000 X -----x0000 X -----x0000 X -----
X -----x0000 X -----x0000 X B0460x0000 X -----x0000 X -----x0000 X -----
B 04003x0002 B 04003x0002 B 04003x0002 B 04002x0002 B 05002x0002
B 04003x0002 B 04003x0002 B 04003x0002 B 04003x0002 B 05003x0002
B 04003x0002 B 04003x0002 B 04003x0002 B 04004x0002 B 05004x0002
X E----x0000 X E----x0000 X E----x0000 X E----x0000 X E0600x0000 X E---
X E----x0000 X E----x0000 X E----x0000 X E----x0000 X E----x0000 X E010
X E----x0000 X E----x0000 X E----x0000 X E----x0000 X E----x0000 X E----
B -----x0000 B G----x0000 B G----x0000 B G----x0000 B G----x0000 B G---
B G----x0000 B G----x0000 B G----x0000 B G----x0000 B G----x0000 B G--
B G0000x0000 B G----x0000 B G----x0000 B G----x0000 B G----x0000 B G-
I IIIIx0000 B +0000x0000 L +0000x0000 X +0000x0000 Y +0000x0000 W
```

Figure 13: Level als Textdatei

Das Level als editierbares Objekt

Diese Repräsentation stellt alle Informationen dar und besitzt Schnittstellen für die Bearbeitung im Editor. Die ausgelesenen IDs aus der Textdatei sind hier in Matrizen abgelegt. Durch die Verwendung mehrerer Matrizen ist das Stapeln verschiedener Spielelemente möglich.

Figure 14: Level im Bearbeitungsmodus

Das Level als spielbare Instanz

Alle Spielelemente werden bei Initialisierung der Levelinstanz geladen, d.h. ihre loadContent()-Methode wird ausgeführt. Im Gegensatz zu den anderen beiden Repräsentationen werden hier auch alle Zustände der Spielelemente mit verwaltet. Das bedeutet, dass der aktuelle Zustand aller Objekte eines Levels nach dem Verlassen solange gespeichert wird, bis es wieder betreten wird.

Figure 15: Spielbare Levelinstanz

7 Grafik

7.1 Spriteanimation

Um einen Bewegungseindruck bei einem Sprite (besonders bei Gegnern) zu erzeugen, wird eine Bildsequenz bestehend aus mehreren Sprites verwendet. Um dies effizient zu realisieren, wird eine Spriteanimationsklasse (hier SpriteAnimator genannt) verwendet. In der Update-Methode erhält sie die Systemzeit, welche zum Bestimmen des aktuellen Bildes verwendet wird. Zur besseren Bildindizierung und Animationssteuerung verwendet man eine Indizierungsmatrix. Über die Zeilen können dann einzelne Unteranimationen in einem SpriteSheet angesteuert werden.
Der SpriteAnimator besitzt einen Texturidentifier. Die Animationsgeschwindigkeit wird in Bildern pro Sekunde angegeben. Wahlweise kann zeilen-, spaltenweise, aufsteigend oder absteigend animiert werden.

Figure 16: Verschiedene Spritesheets und ihre Unterteilungen

7.1.1 Kombination mit 2D-Translationen

Für bestimmte Bewegungsabläufe bietet sich die Realisierung der Animation über Translationen an, z.B. für das Schwingen eines Schwertes, ein Auge welches immer in Richtung des Spielers ausgerichtet sein soll, oder Lava, die sich ausbreitet. Gegebenenfalls ist die Animation dadurch flüssiger und als Nebeneffekt wird Speicherplatz gespart, da man anstatt einer Abfolge von Einzelbildern nur ein Bild speichern muss.
Im Rahmen dieses Projekts wurde eine Kombination aus Spriteanimation und 2D-Translationen gewählt.

7.2 Texturatlas

Besonders bei größerer Anzahl an Spielelementen ist es sehr ineffizient für jede Instanz eines Objektes eine eigene Instanz der Textur zu laden. Daher wurde in diesem Projekt ein sogenannter Texturatlas benutzt. Die einzelnen Texturen werden in einer großen Textur zusammengefasst. Jeder spezifischen Einzeltextur wird ein Identifier in Form eines Strings und ein Rechteck, welches seine Position, Höhe und Breite im Texturatlas beschreibt, zugewiesen. Der String wird mittels eines Verzeichnisses (Dictionary) dem Rechteck zugeordnet.
Diverse Zeichenmethoden im Texturatlas können über folgende Parameter beeinflusst werden können:

- Position
- Bounding Box
- Transparenz
- Zeichenfarbe
- Rotation
- Horizontale/Vertikale Spiegelung
- Skalierung
- Tiefenwert (für Verdeckungen und Überlappungen)

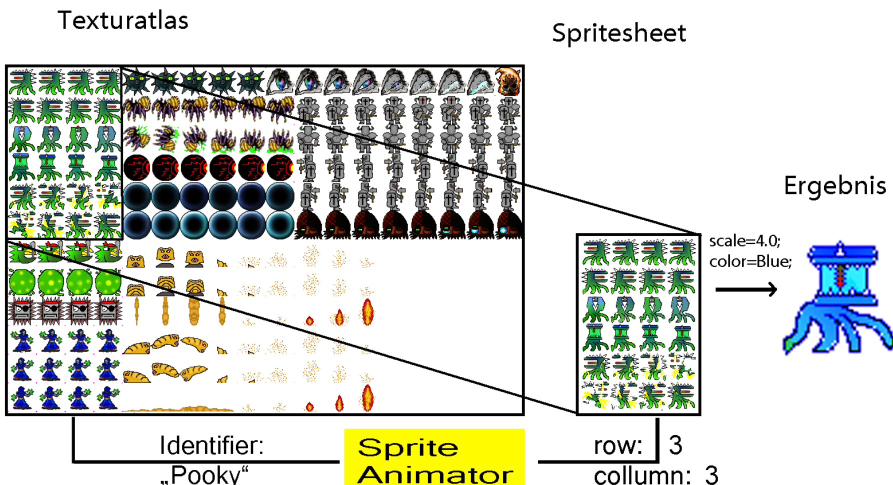


Figure 17: Beispielaufruf: *SpriteAnimator* übergibt Texturidentifier und Werte für die Indizierungsmatrix, in der *draw*-Methode der Klasse werden Farbe und Skalierung hinzugefügt

7.3 Tiefenwertanpassung

Um in der flachen 2D-Welt die Illusion von Tiefe zu erzeugen und unschönes Flackern bei sich überlagernden Texturen zu vermeiden, wurde ein Verfahren entwickelt um eine koordinatenbezogene Tiefenwertsanpassung vorzunehmen.
Hierbei handelt es sich um eine Abbildung von der X- und Y-Position im Bild auf den Tiefenwert.
Durch verschiedene Tiefenwerte ist es möglich, Verdeckungen zu erzeugen, um damit Tiefe zu simulieren.
Über das Spielfeld werden verschiedene Ebenen E $\in$ {0.1,...,1.0} gelegt, welche verschiedene Tiefenebenen repräsentieren. Der Kerngedanke ist, dass es keine zwei Spielelemente in einer Ebene geben soll, welche den gleichen Tiefenwert haben. So stehen dem Designer bis zu 10 Ebenen zur Verfügung, innerhalb derer Spielelemente dargestellt werden können.
Je weiter unten im Bild sich das Objekt befindet, desto näher erscheint es uns auch. Objekte mit geringerem Y-Wert sind folglich hinter diesem Objekt. Die verwendete Position entspricht hierbei dem linken oberen Eckpunkt des Kollisionsrechtecks des Spielelements (vgl. Kap. 8.2.). Die X-Komponente geht mit geringerer Gewichtung als die Y-Komponente in den Tiefenwert mit ein.

$$Tiefenwert = E - \frac{Y}{10^3 * W_z} - \frac{X}{10^5 * W_z}$$

W_z: Die Seitenlänge einer Spielzelle in Pixeln
X,Y: Koordinaten der Spielzelle

Zum Entwickeln dieser Formel wurden zwei Fakten beachtet. Der Tiefenwert ist vom Typ float und hat damit eine Genauigkeit von 7 bis 8 Stellen. Daher arbeitet diese Anpassung optimal für W_z<100. Die Anordnung der Spielelemente ist tilebasiert (vgl. Kap. 10.4) mit maximaler Tileanzahl von 100 auf 100.
Damit ergeben sich folgende Abschätzungen:

$0 \leq \frac{Y}{10^3 * W_z} < 0.1$
$0 \leq \frac{X}{10^5 * W_z} < 0.001$

Mit dieser Anpassung wird der Tiefenwert in ein Intervall $I = (E - 0.1, E]$ abgebildet.
In "Infinite Worlds" wird E=0.3 als Ebene für alle Spielelemente auf dem Boden verwendet und E=0.2 für fliegende Spielelemente.

Figure 18: Beispielszene, $W_z = 50$: Alle gezeigten Spielelemente, außer die fliegenden Untertassen links oben, befinden sich auf E=0.3, der Kopf des Gegners in der Mitte erscheint uns vor den Bäumen; vom Gegner ganz oben ist nur der Kopf zu sehen, da er sich hinter dem Baum befindet. Nebeneinanderstehende Bäume bilden eine einheitliche Verdeckung durch Gewichtung der X-Koordinate. Die fliegenden Untertasse sind über den Bäumen dargestellt, da sich in der Ebene E=0.2 befinden.

7.4 Abstraktion von Licht und Schatten

7.4.1 Kreisrunde Lichtkegel

Bei dieser Variante kommt eine Lichtmaske zum Einsatz. Sie repräsentiert den von einer Punktlichtquelle beleuchteten Bereich. Als Grundform wurde ein Kreis auf schwarzem Hintergrund gewählt.

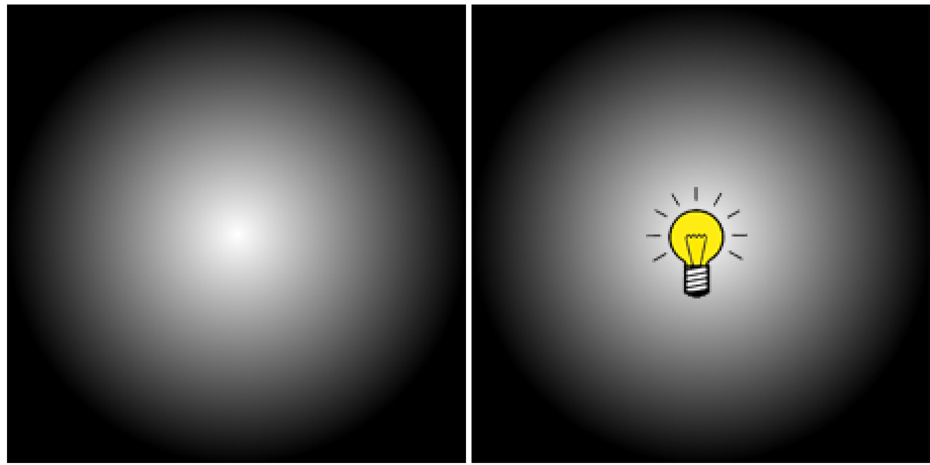


Figure 19: Lichtmaske

Figure 20: Punktlichtquelle

Verwendeter Algorithmus

1. Es wird auf einem Rendertarget gearbeitet, dass die Größe des sichtbaren Bereichs hat.
2. Dann wird eine schwarze, bildschirmfüllende Textur gezeichnet.
3. Es wird über jede gegebene Lichtquelle im sichtbaren Bereich die Lichtmaske gezeichnet, wobei der Mittelpunkt des Lichtkreises dem Mittelpunkt der Lichtquelle entspricht. Somit erhält man eine Lightmap der sichtbaren Szene.
4. Nun wird in ein anderes Rendertarget die Hauptszene gezeichnet, d.h. das Level an sich und all seine Spielelemente im sichtbaren Bereich.

5. Im letzten Schritt wird die Textur aus dem Rendertarget der Hauptszene gezeichnet unter Anwendung des folgenden Pixelshaders (der sogenannte Lightmapshader) mit der Textur der Lightmap als Parameter:

```
sampler s0;
texture lightMask;
sampler lightSampler = sampler_state{Texture = lightMask;};

float4 PixelShaderFunction(float2 coords: TEXCOORD0) : COLOR0
{
    float4 color = tex2D(s0, coords);
    float4 lightColor = tex2D(lightSampler, coords);
    return color * lightColor;
}
```

Figure 21: "Lightmapshader"

Figure 22: Beispielszene: die beiden Fackeln rechts oben bilden dank halbtransparenter Lichtmaske einen helleren Bereich, als die einzelne Fackel oben links

7.4.2 Schattenfühler

Im Rahmen dieser Arbeit wurde ein Verfahren zum Simulieren von Schatten entwickelt. Bei dieser Variante werden Schattenfühler an die Kanten der opaquen Spielelemente gezeichnet. Grundsätzlich werden dabei 2 verschiedene Typen von Fühlern unterschieden:

1. **Schattenfühler mit Schattenübergang:**
   hier wird der Halbschattenraum (Penumbra) angenommen.
2. **Schattenfühler im Kernschatten:**
   hier wird der Kernschattenraum (Umbra) angenommen.

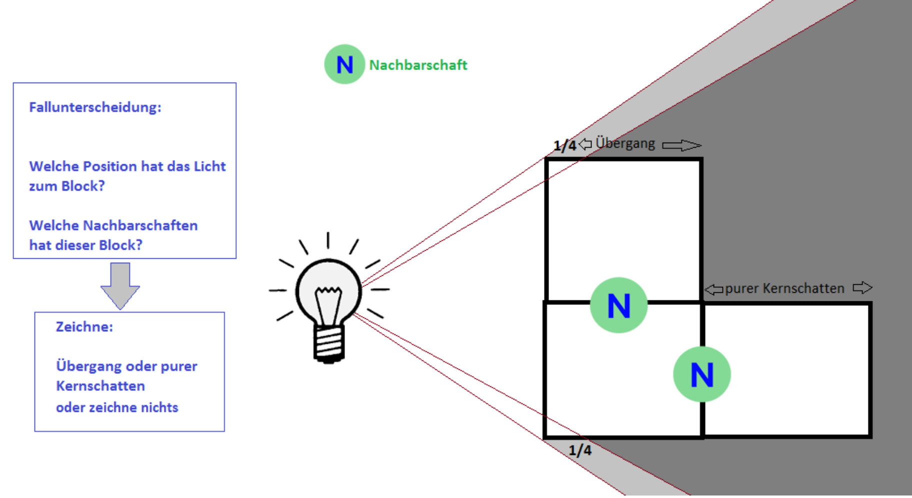


Figure 23: Schattenbündel

Für jedes sichtbare opaque Spielelement im Zeichenbereich werden folgende Schritte ausgeführt:

1. Erfassen der Abtastseiten
2. Verifizieren der Abtastseiten
3. Schattenfühler generieren

Erfassen der Abtastseiten

Die Position des Lichtes wird in einen von 8 Sektoren relativ zum opaquen Objekt eingeordnet. Je nach zugeordnetem Sektor entstehen eine Menge an Abtastseiten mit zugehöriger Abtastrichtung. Sollte die Lichtquelle in einem geradzahligen Sektor (vgl. Fig. 24) landen entsteht immer eine Seite im Kernschatten. Die Abtastrichtung spielt beim Generieren der Schattenfühler mit Schattenübergang eine wichtige Rolle. Im Folgenden beschreibt Abtastseite AB, dass die Abtastrichtung von A nach B verläuft.
Im Falle, dass sich die Lichtquelle innerhalb des opaquen Objekts befindet, werden keinerlei Abtastseiten generiert und der Algorithmus stoppt nach diesem Bearbeitungsschritt.

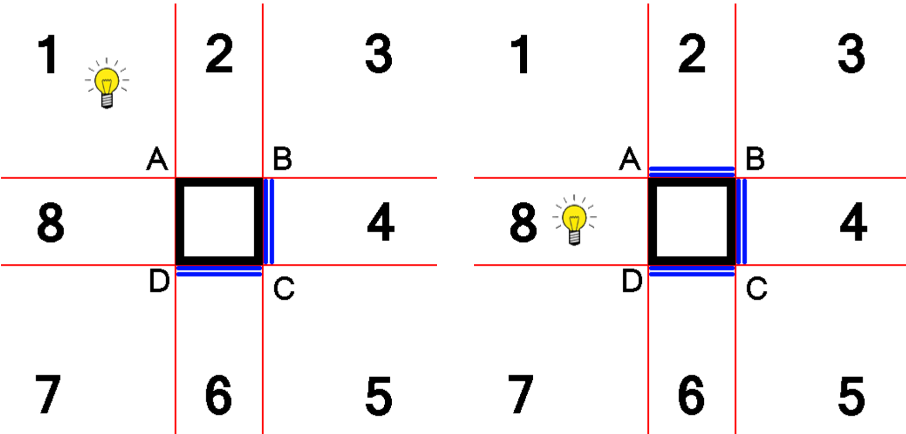


Figure 24: Licht befindet sich in Sektor 1, Abtastseiten sind BC und DC

Figure 25: Licht befindet sich in Sektor 8, Abtastseiten sind AB, DC und BC

Verifizieren der Abtastseiten

Jede der erfassten Kanten ist nun einen Kandidat für einen Schattenfühler. Hat die Kante einen direkten Nachbarn, scheidet die Kante als Kandidat aus. An dieser Kante wird keinerlei Schatten gezeichnet.
Hat sie keinen direkten Nachbarn, wird entschieden, welche Art von Schattenfühler generiert werden soll: Schattenübergang oder Kernschatten.

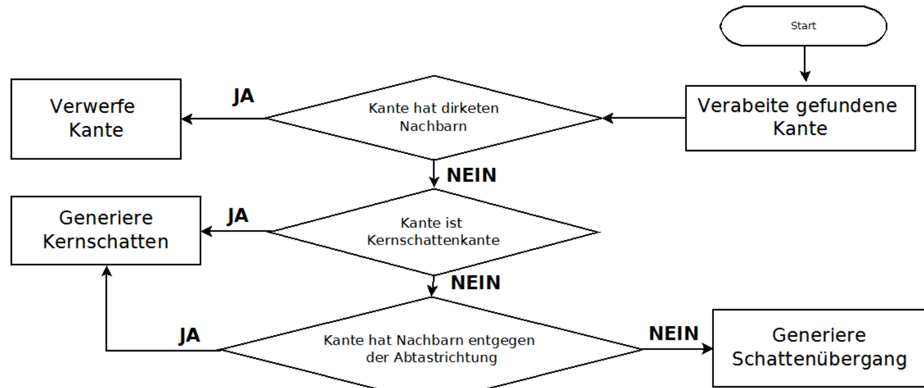


Figure 26: Kandidatenverifizierungsprozess

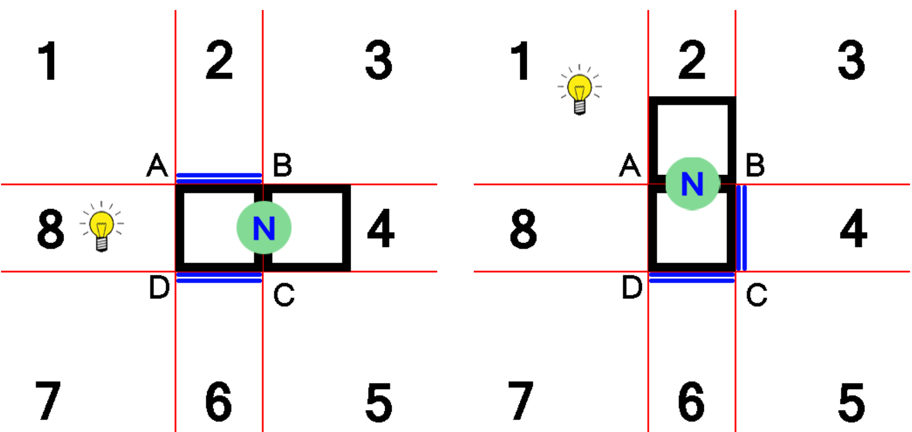


Figure 27: Kandidat BC wurde entfernt, da er einen direkten Nachbarn hat

Figure 28: Kandidat BC bleibt, da er keinen direkten Nachbarn hat, wird aber von Schattenfühler mit Übergang zu Schattenfühler im Kernschatten, da er einen oberen Nachbarn hat, welcher sich in gegengesetzter Laufrichtung zu der Kante BC befindet

Erzeugung eines Schattenfühlers
Für die einzelnen Kanten eines Spielelements wird eine Abtastung entlang der Kante durchgeführt, bei der für jeden abgetasteten Punkt eine schwarze Linie von der Lichtquelle aus durch diesen Punkt gezeichnet wird, beginnend ab diesem Punkt.
Handelt es sich um einen Schattenfühler mit Übergang zum Licht, wird der Alphawert der Zeichenfarbe an dem Punkt, der dem Licht zugeneigt ist, auf 0 gesetzt und linear zur 255 interpoliert, sodass er bei einem Schwellenwert (S) an der Gesamtstrecke den Wert 255 annimmt. Im Kernschatten findet diese Interpolation nicht statt; alle Fühler werden mit maximalem Alphawert gezeichnet.

Entscheidendes Kriterium für die Dichte des Schattens ist die Abtastrate (ATR). Sie beschreibt die Anzahl an Abtastungen pro Kante.
Anmerkung: Bei "Infinite Worlds" entspricht die Zellenbreite 50 Pixel, eine Abtastung von 150 entspräche dann einer Abtastungsschrittweite von 0.333 Pixel.

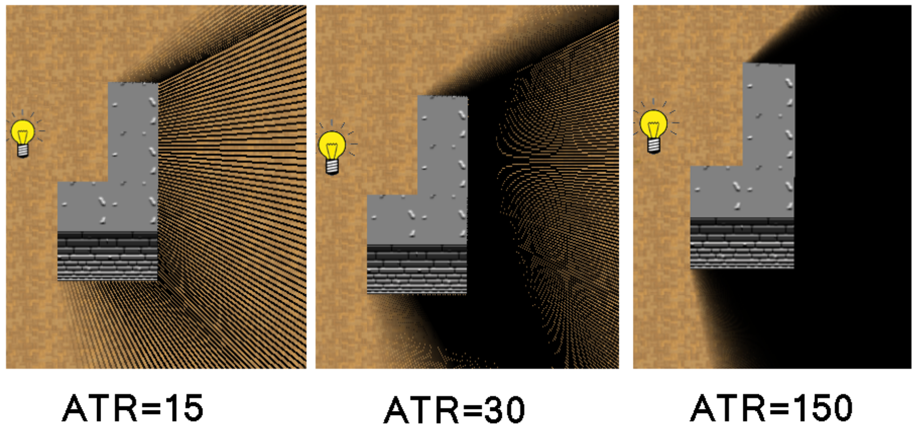


Figure 29: Auswirkungen verschiedener Abtastraten

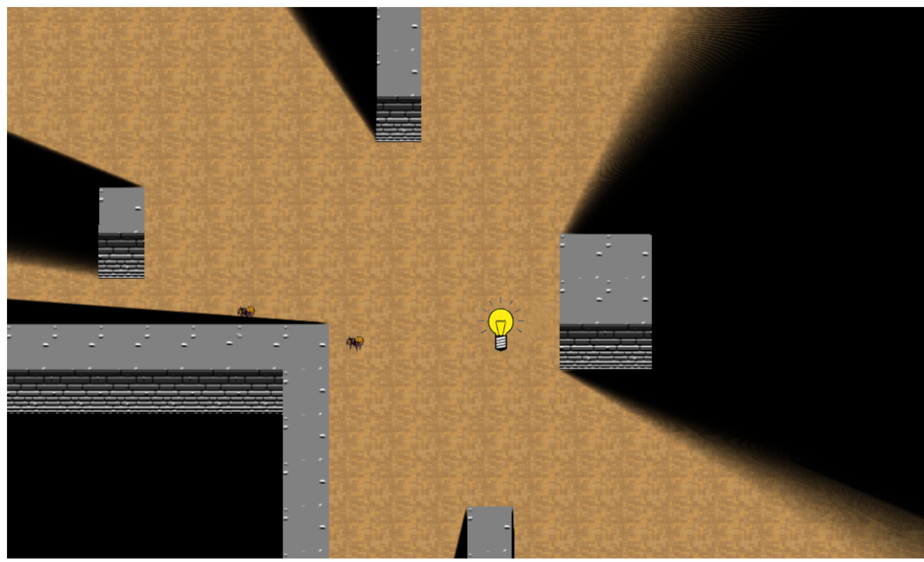


Figure 30: Beispielszene mit einer Lichtquelle

Um das Verfahren auf mehrere Lichtquellen zu erweitern, wird für jede Lichtquelle eine eigene Shadowmap erstellt. Alle so generierten Shadowmaps werden zu einer gesamten Shadowmap unter Verwendung des Lightmapshaders aus Abb. vereint und diese durch erneute Anwendung auf die gerenderte Gesamtszene angewandt.
In Tests zeigte sich die Erweiterung für multiple Lichtquellen nur für eine maximale Anzahl von 5 Lichtquellen als praktikabel, da das zusätzliche Abtasten und die mehrmalige Anwendung des Shaders einen zusätzlichen Rechenaufwand zur Folge hat. Im Rahmen dieses Projekts werden Schattenfühler mit einer maximalen Anzahl von 3 verwendet (vgl. Kap. 7.4.4 Finale Komposition).

Figure 31: Kombination zweier Lichtquellen

7.4.3 2D-Texturschatten

Im Rahmen dieser Arbeit wurde ein Verfahren zur Simulation von Schatten einer gegebenen Textur in der 2D-Welt entwickelt. Die Grundidee ist, mit einer erstellten Schattenschablone, die Schattentextur in die Welt mittels Translationen abzubilden.

Erstellen der Schattenschablone

Aus dem Texturatlas wird unter Verwendung des folgenden Pixelshaders ein Texturatlas für Schatten erzeugt:

```
sampler textureSample;

float4 PS_COLOR_ALLSAMEGRAY (float2 texCoord: TEXCOORD0) : COLOR
{

float4 xcolor;
xcolor = tex2D(textureSample, texCoord);

if(xcolor.a==0)return xcolor; //if pixel is transparent return pixel-color

xcolor=float4(0.46f,0.46f,0.46f,1.0f); // else color pixel gray
return xcolor;
```

Figure 32: Shader zum Erzeugen der Schattenschablone

Es ist ausreichend diesen einmal bei Initialisierung des Spiels zu erstellen.

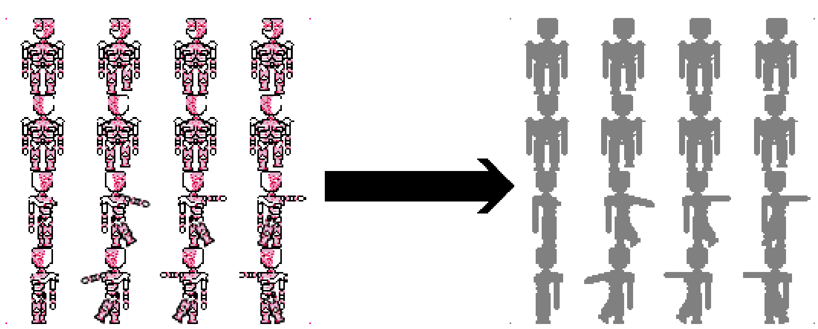


Figure 33: Spritesheet und zugehörige Schattenschablone (ShadowSheet)

Begriffsklärung:
O : Schattenobjekt
D : Rotationspunkt für die Drehung in O
L : betrachtete Lichtquelle
dist(L) : Distanz zwischen O und L
maxLightDistance(L) : Reichweite von L
W(L) : Winkel zwischen O und L

Algorithmus

1.) Berechnen des Lichtraumes

Es wird eine Liste von Lichtquellen erstellt, in deren Reichweite sich der Mittelpunkt der BoundingBox der Textur befindet. Diese Menge von Lichtquellen wird im Folgenden als Lichtraum ψ bezeichnet. Analog dazu definieren wir ψ(O) als den Lichtraum eines Objektes O.

Figure 34: Die beiden Fackeln und die untere Glühbirne bilden hier den Lichtraum für die Textur des Gegners in der Mitte, die Glühbirne rechts oben ist zu weit entfernt

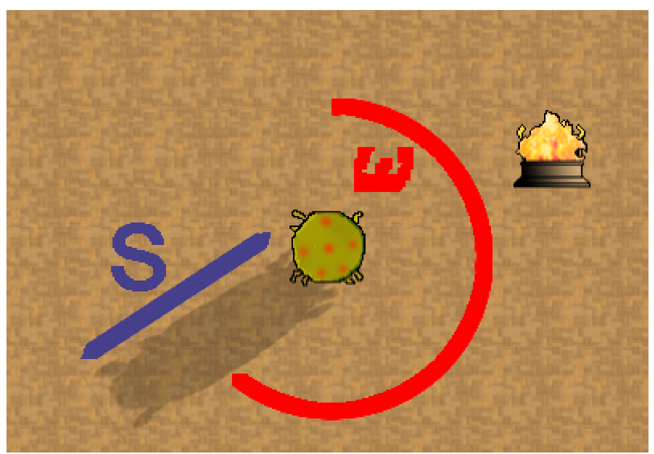


Figure 35: Schattenparameter

2.) Kummulieren der Schattenparameter

ω: Winkel der Drehung ˜ Winkel zur Primärlichtquelle

$\omega = \arctan \frac{L.x - D.x}{L.y - D.y} - \frac{3}{2} * \pi$

α Alphawert der Schattenfarbe (Transparenz des Schattens) Abstand zur Primärlichtquelle, Anzahl und Abstand zu Zweitlichtquellen

$\alpha = (\alpha_s - \frac{dist(L)}{maxLightDistance(L)} * \alpha_m) * \frac{1}{\delta * M}, \alpha_S => \alpha_M$
$M = \prod_{i=0}^{n} (2 - \frac{dist(L_i)}{maxLightDistance(L_i)}), \forall L_i \in \psi(O) \wedge L_i \neq L \wedge |W(L_i) - \omega| < 90$

S: Skalierung ˜ Abstand zur Primärlichtquelle

$S = S_{max} * (1 - \frac{dist(L)}{maxLightDistance(L)})$

$\alpha_s \in [0, 255]$: maximaler Alphawert des Schattens
$\alpha_m \in [0, 255]$: Alpha-Abstandsmodifikator
$S_{max} > 0$: Maximale Schattenlänge
$\delta > 1$: Sekundärlichtsgewichtung

3.) Zeichnen:

Da die Anordnung der Sprites im Texturatlas der Anordnung im Schattenatlas entspricht, kann der Schatten analog zum normalen Animationsablauf animiert werden. Die Indizes aus der Indizierungsmatrix des SpriteAnimators werden an den Zeichenaufruf des Sprites und an den Zeichenaufruf des Schattens übergeben.

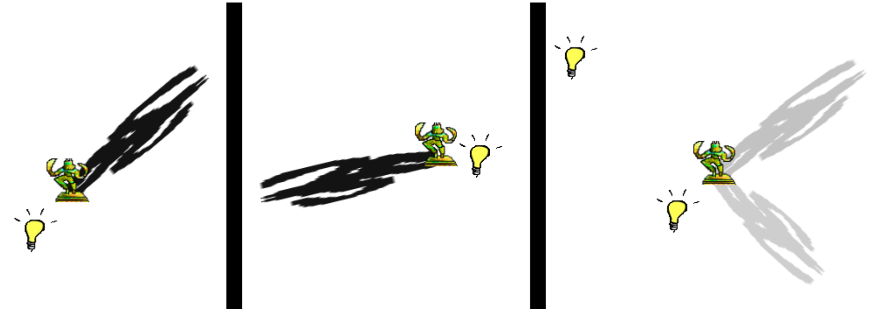


Figure 36: $\alpha_s = 255, \alpha_m = 155, \delta = 5$: Im rechten Bild ist deutlich die Aufhellung des Schattens durch die Sekundärlichtquellen zu erkennen

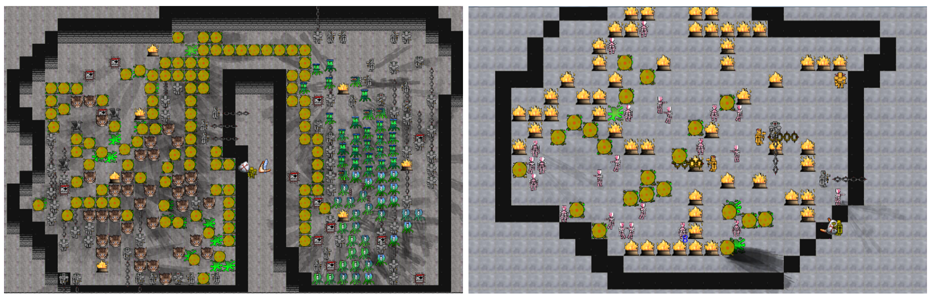


Figure 37: 308 Gameelemente, 7 Lichtquellen

Figure 38: 42 Gameelemente, 53 Lichtquellen

Das Verfahren zeigt sich auch für eine größere Anzahl an Lichtquellen und Spielelementen pro Bildschirm performant. Um die Performanz zu steigern, könnten Look-Up-Tabellen für die Berechnung trigonometrischer Funktionen verwendet werden.
Insbesondere bei einer größeren Anzahl an Lichtquellen tritt das Verschattungsproblem auf. Obwohl durch den Einfluss vieler Sekundärlichtquellen der Schatten aufgehellt wird, sorgt das Überlappen zuvieler Schatten für eine signifikante Abdunklung (vgl. Fig. 38). Um diesem Effekt entgegenzuwirken, kann ab einem gewissen Schwellwert an Lichtquellen im Lichtraum δ angepasst werden.

7.4.4 Finale Komposition

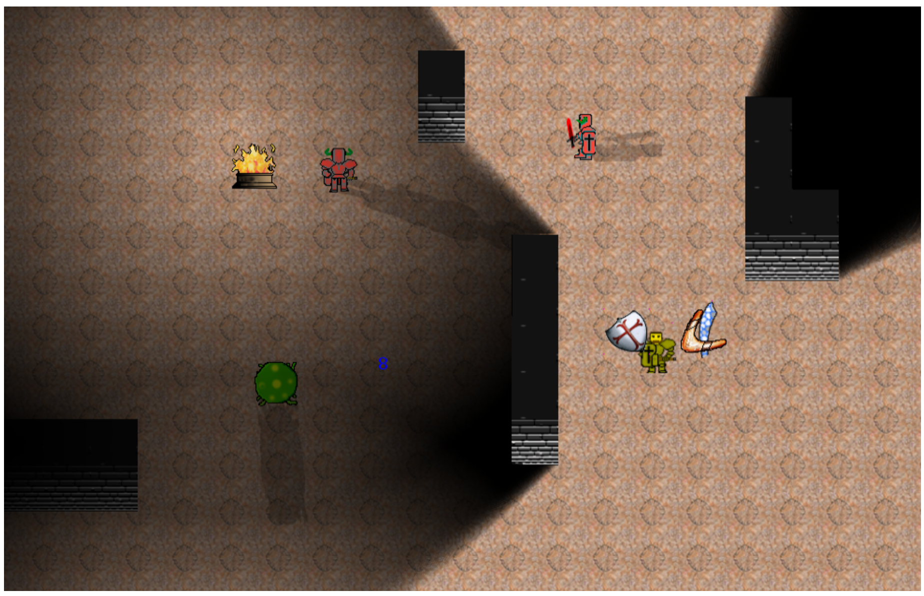


Figure 39: Beispielszene

Schattenfühler werden eingesetzt, um im Spiel die Sichtweite der Spieler zu visualisieren. Durch Lichtquellen kann der Sichtbereich erweitert werden. Schattentexturen werden sowohl zu dekorativen Zwecken als auch als relevante Spielelemente eingesetzt. (siehe Fig. 40).

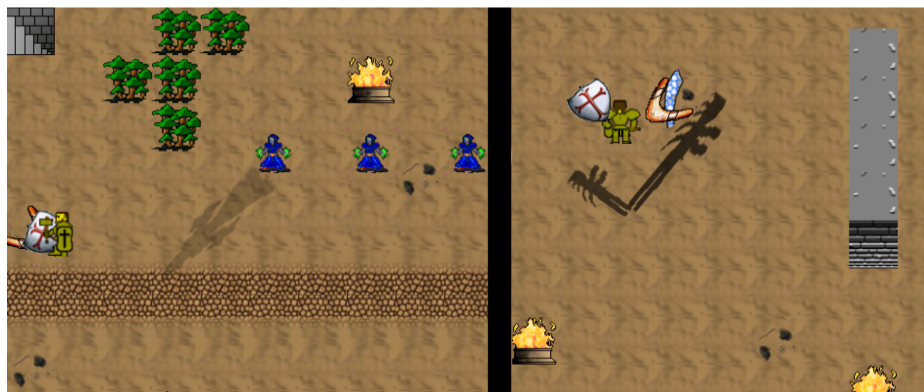


Figure 40: links: echter Gegner kann über Schatten von Doppelgängern unterschieden werden; rechts: die Position von unsichtbaren Gegnern ist an ihrem Schatten zu erkennen

8 Sound- und Musikenviroment

8.1 SoundAtlas

Analog zum Texturatlas wird ein Soundatlas verwendet. Hier werden die bei Initialisierung des Spiels geladenen Sounds mittels eines Verzeichnisses (C# - Dictionary) einem String zugeordnet. Für jeden Sound muss zusätzlich ein Maximalpegel angegeben werden. Per default liegt dieser bei 0.5, was 50 % Lautstärke entspricht.
Der Soundatlas stellt sicher, dass ein aufgerufener Sound nicht zu häufig oder zu laut abgespielt wird.
Über Priorisierung können bestimmte Sounds andere überlagern oder deren Ausgabe verhindern, z.B. stoppt die Ansprache des Endgegners kurzzeitig die Hintergrundmusik.

8.2 Angepasste Hintergrundmusik

Um die Musik passender zum Geschehen im Spiel zu gestalten, stehen für jedes Level immer mindestens 2 Musiksamples zur Verfügung, ein friedliches und ein actionreicheres. Das System erfasst die gegebene Anzahl an Gegnern im Level. Ist sie kleiner oder gleich einem bestimmten Schwellenwert, wird das friedlichere Stück gespielt, ansonsten das andere.
Um die Anpassung weiter zu Verbessern, sollten weitere Kriterien herangezogen werden, wie z.B. der Abstand der Gegner zum Spieler sowie der Bedrohungsgrad der Gegner.
Besonders wichtig ist auch eine Art Übergang von einem Musiksample in das andere miteinzubauen, da der Wechsel ansonsten immer zu abrupt stattfindet. Eine Möglichkeit hierzu wäre, das eine Sample linear leiser zu machen, während das andere linear dazu lauter wird ("Crossfading").

8.3 Simpler 2D Ambient Sound

Die Klasse Gameelement besitzt eine Membervariable namens soundLevel $\in$ [0,1]; ihr Wert wird entsprechend zum Abstand zwischen Gameelement und Spieler auf dem Bildschirm skaliert. Je näher man dem Gameelement (in diesem Fall der Schallquelle) steht, desto lauter wird auch der entsprechende Sound abgespielt.

$SoundLevel = (1 - \frac{distance(Player)}{maxSoundRange})$
IF($soundLevel$<0) soundLevel = 0;

Sollten mehrere Spieler vorhanden sein, wird der kleinste Abstand zur Berechnung des soundLevels herangezogen.

9 Steuerung

9.1 Allgemein

Ziel war es, die Steuerung so einfach und so fehlerquellenminimierend wie möglich zu gestalten, um die Dauer der Steuerungslernphase zu minimieren.

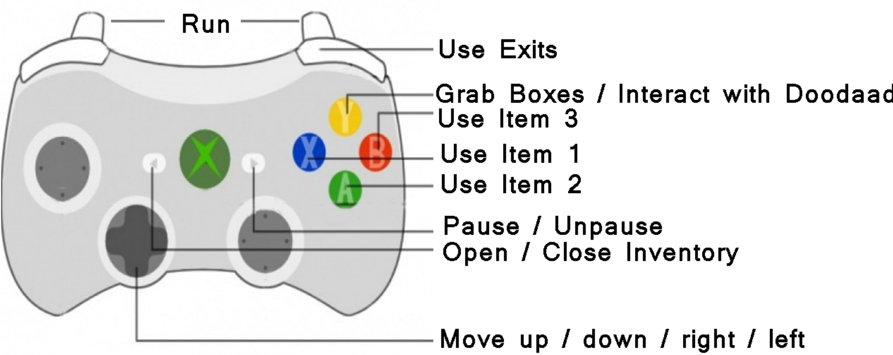


Figure 41: Control-Scheme für Infinite Worlds

Das Primär- und Sekundäritem (hier: Item 1 und Item 2) sind die im Spiel am häufigsten benutzten Items, daher wurden der Knopf X und A für diese gewählt, da diese aus ergonomischer Sicht am leichtesten zu erreichen sind.
Das Benutzen eines Ausgangs (Exit) sollte immer eine bewusste Entscheidung sein. Die Möglichkeit es unbewusst zu tun wurde dadurch mimimiert, das hierfür der hintere Knopf gewählt wurde, welcher mit dem Zeigefinger bedient wird.
Das Öffnen des Inventars pausiert das Spiel, da es das Spielfeld verdeckt und dadurch effizientes Spielen unmöglich machen würde. Der Pause-Knopf ist, während das Inventar geöffnet ist, deaktiviert; ein weiterer Druck auf den Inventory-Knopf schliesst das Inventar und lässt das Spiel weiterlaufen.
Der Pause-Knopf funktioniert erwartungskonform außerhalb der Inventarumgebung. Damit kennt das Spiel zwei verschiedene Pause-Zustände, einen fehlerminimierenden und einen, welcher genutzt werden kann, um sich in Ruhe Übersicht über das Spielgeschehen schaffen zu können.

9.2 Kollisionserkennung

Die Kollisionserkenneung (engl. Collision-Detection) findet über den Schnitt geometrischer Primitive statt. In dieser Arbeit wurden hierfür hauptsächlich Rechtecke verwendet.
Der Begriff der Bounding-Box wird hierbei strikt von dem Begriff der Kollisions-Box bzw. des Kollisionsrechtecks unterschieden. Die Bounding-Box definiert den Bereich in dem die zugehörige Textur des Gameelements gezeichnet wird,

auf die Kollisions-Box wird Collision-Detection angewandt. Wenn sich zwei Kollisions-Boxen schneiden findet eine Kollision statt.
Für die Kollisions-Box müssen folgende Eigenschaften gelten:

1. **Enthaltensein**
   Die Kollisions-Box ist komplett in der Bounding Box enthalten.

2. **Positionsbestimmtheit**
   Die Position der Kollsions-Box ist in der Bounding Box festgelegt.

3. **Updatebarkeit**
   Bei Verschiebung, Rotation und Skalierung des Objekts muss das Kollisionsvolume die Eigenschaft 1 und 2 behalten.

Die Bounding-Box kann auch zur Kollisionserkennung verwendet werden, hat aber einen entscheidenden Nachteil: Es könnten Kollisionen erkannt werden, obwohl optisch gesehen die gezeichneten Texturen sich in keinem Punkt überlappen. Gamedesigntechnisch gesehen ist es besser, obwohl sich ein paar Pixel der beiden Texturen schneiden nicht getroffen zu werden, als getroffen zu werden obwohl optisch gesehen keine Überschneidung stattfindet.
Anmerkung: AoE haben eine eigene BoundingBox, es wird daher AoE welcher sich losgelöst von dem Gegner bewegen kann und AoE welcher eine Zone am Gegner darstellt unterschieden. Letztere werden nicht gezeichnet, daher ist der Kollisionsbereich und die Bounding Box in diesem Fall identisch.

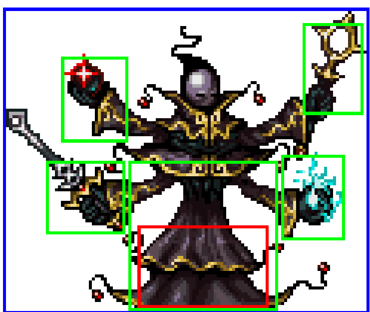


Figure 42: Blau: Bounding Box, Grün: AoE(Area of Effect) am Gegner, Rot: Kollisionsbereich

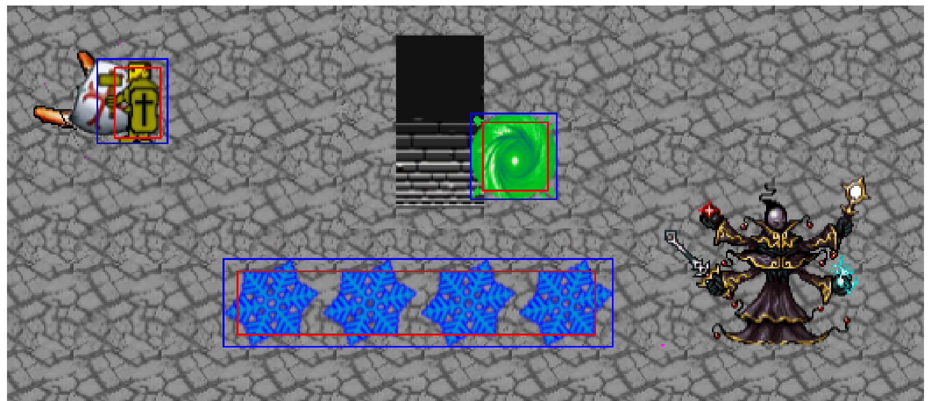


Figure 43: Blau: Bounding Box, Rot: Kollisionszonen, hier 2 AoE, welche sich selbstständig fortbewegen

Der Kollisionstest kann mit geringem Aufwand durchgeführt werden:

```
public static bool checkForIntersection(Rectangle a, Rectangle b)
{
int top = Math.Max(a.Top, b.Top);
int bottom = Math.Min(a.Bottom, b.Bottom);
int left = Math.Max(a.Left, b.Left);
int right = Math.Min(a.Right, b.Right);

if (top >= bottom II left >= right)
return false;

return true;
}

    public static bool checkForIntersection(Circle a, Circle b)
{
Vector2 distVector =a.pos-b.pos;

    if (distVector.Length <= a.radius+b.radius)
return true;

return false;
}
```

Um den Kollisionstest weiter zu verbessern könnten Quattuorvigintiecke [12] verwendet werden, da diese eine bessere Anpassung an die Textur darstellen.

9.3 Bewegungskorrektur

Definition: KMSI
Kleinstmögliches Steuerintervall: sie beschreibt die Genauigkeit, mit der die Spielfigur gesteuert werden kann bei in einem Schwellenwert minimalen Input (MI). Hier wird dieser in Pixeln angegeben.

Vorbedingung:
Als Kollisionsvolumes werden Rechtecke benutzt. Jedes Kollisionsrechteck in unserer Spielwelt hat die Eigenschaft, dass seine Seitenlängen ein Vielfaches des KMSI ist.

Problemstellung:
Das Navigieren der Spielfigur unter Zeitdruck durch eine schmale Lücke, gerade so groß wie die Figur selbst. Um diese Lücke perfekt zu treffen, ist eine korrekte Abfolge an MI notwendig. Man spricht in diesem Zusammenhang auch von dem "Speed-Accuracy-Trade-Off" [1] . Da Zeitmanagement und daher Zeitdruck relevante Spielfaktoren sind, kann dies eine sehr frustierende Aufgabe sein. Wie ist es möglich, eine möglichst genaue und gleichzeitig schnelle Steuerung zu realisieren?

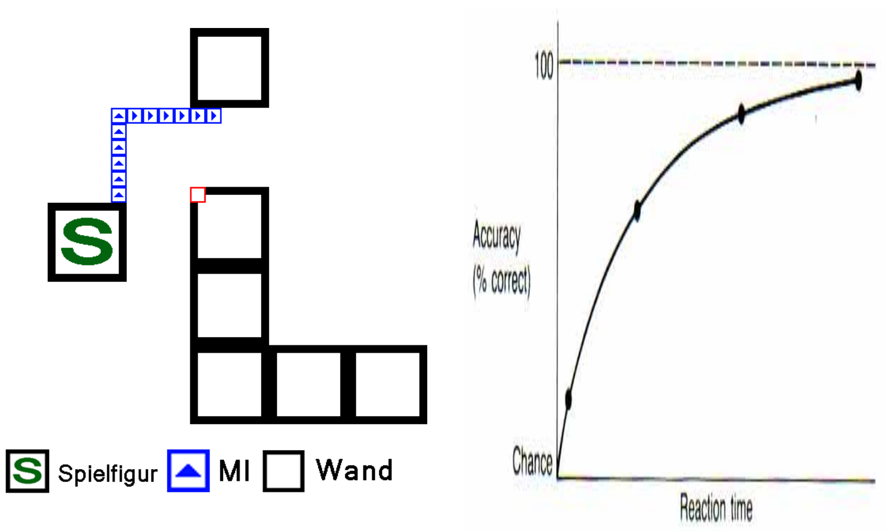
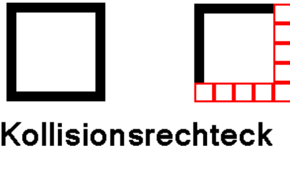


Figure 44: Problemstellung

Figure 45: Speed-Accuracy-Trade-Off



KMSI

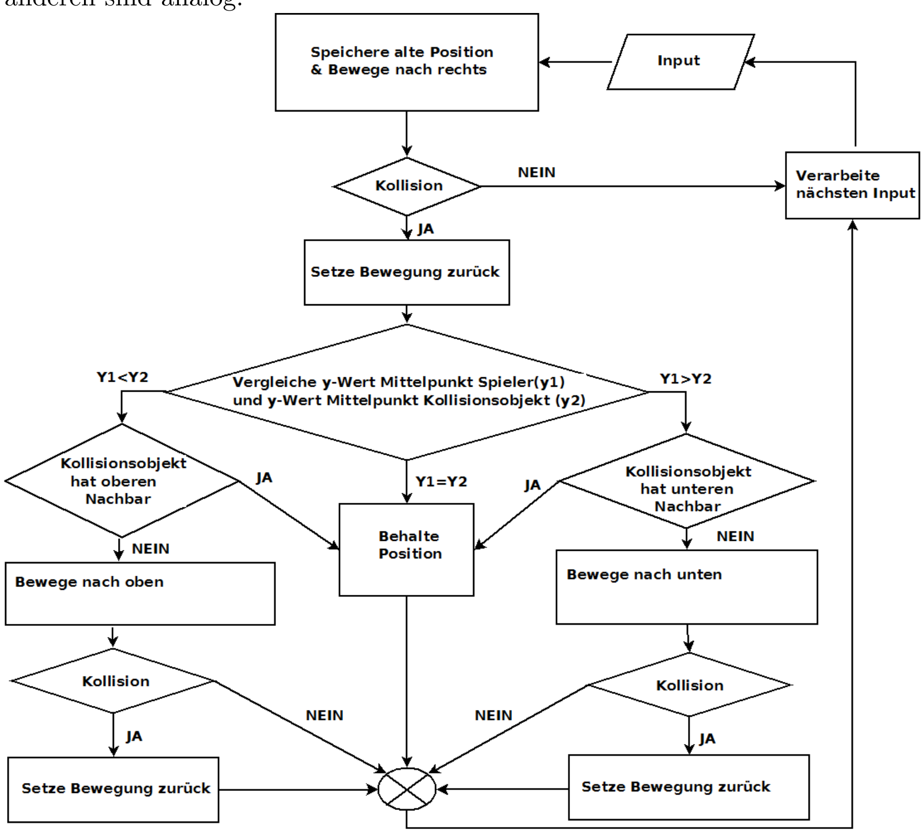
Figure 46:

Bewegungskorrektursystem

Idee: Bei einer Kollision erkennt das System, wohin der Spieler die Spielfigur bewegen will und korrigiert die Bewegung in die richtige Richtung. So sind flüssige Bewegungsabläufe um kantige Wandsegmente und durch schmale Lücken möglich.

Algorithmus:

Der Algorithmus wird exemplarisch für eine Bewegungsrichtung erläutert, alle anderen sind analog.



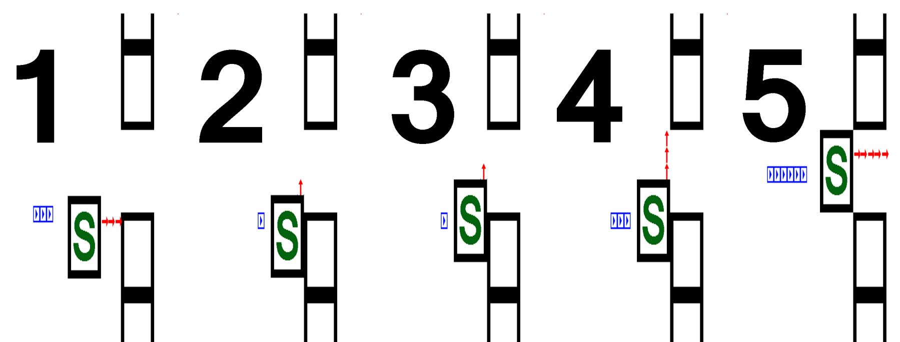


Figure 47: Anwendungsbeispiel:
1) Ohne Kollision bewegt sich das Kollisionsrechteck dem Input entsprechend.
2) Es findet eine Kollision statt. Der Mittelpunktsvergleich, liefert die Korrektur nach oben.
3+4) Solange die Kollision weiterstattfindet, wird korrigiert.
5) Ohne Kollision bewegt sich das Kollisionsrechteck dem Input entsprechend.

Der Algorithmus korrigiert die Bewegung entsprechend und erkennt über die Nachbarschaftsabfrage, ob eine Korrektur überhaupt notwendig ist. Dadurch wird verhindert, dass man an einer Wand entlang rutscht, ohne dass eine Ecke in der Nähe ist. Da Move-Correction nur für Spieler angewandt wird, entsteht ein Aufwand von O(n).

10 Editor

10.1 Grundüberlegung zum Design

Für das Design des Editors wurde ein Kompromiss aus softwareergonomischen Ansätzen und Gamedesignentscheidungen gewählt. Einerseits soll alles übersichtlich und effizient angeordnet sein, aber andererseits soll das Interface auch einer Art bunter Spielzeugkiste ähneln.

10.2 Komponenten

Da keine offizielle funktionale GUI-Library für XNA 4.0 existiert, mussten für dieses Projekt die benötigten Funktionalitäten selbst geschrieben werden.
Der Editor wurde aus zwei Funktionalitäten erstellt, welche im Folgenden kurz vorgestellt werden:

Button (engl.: Knopf)
Die Klasse Button erhält in ihrer Update-Methode den Mausstatus, also Position der Maus und Tastenevents. Ist der Mauszeiger im Rechteck des Buttons und wird gleichzeitig eine Taste gedrückt, sendet die Buttonklasse ein Signal, welches im System dann verwendet wird, um eine Aktion (Event) auszulösen.

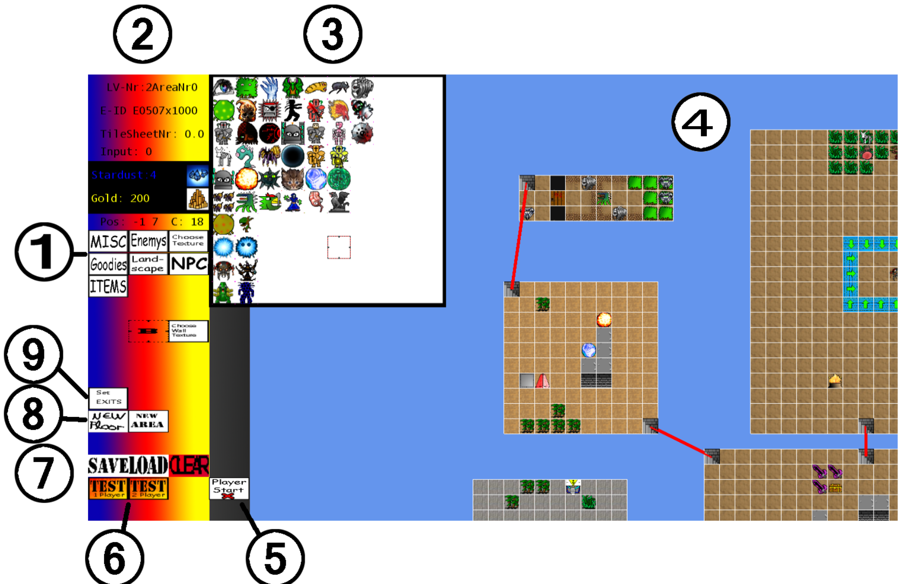
SelectionBox (engl.: Auswahlkiste)
Die Klasse SelectionBox besteht aus einem X*Y Felder großen Raster ($X, Y \in [0,1,2,...]$), in dem jedes Feld systemintern auf einen 2-dimensionalen Punkt abgebildet wird. Jeder Box ist eine Box-ID zugeordnet, welche in Kombination mit dem Punkt die GameelementID ergibt, siehe Fig. 48 und Fig. 49.

Figure 48: SelectionBox Verschiedenes: Box-ID ist M, Position im Raster ist (3,1), daraus ergibt sich die GameelementID: M0301

Figure 49: SelectionBox Gegner: Box-ID ist E, Position im Raster ist (7,7), daraus ergibt sich die GameelementID: E0707

Durch Kombination aus Buttons und SelectionBoxes können verschiedenste Zusammenhänge konstruiert werden.

10.3 Benutzeroberfläche



1. Buttons zum Wählen der verschiedenen Selection-Boxen
2. Infos zur aktuellen Selection-Box
3. Anzeige der aktuellen Auswahl
4. Level-Work-Bench
5. Startpunkt
6. Levels testen
7. Gebiet laden/speichern
8. neues Level in Gebiet erstellen/neues Gebiet
9. Erstellen von Verbindungen zwischen Leveln

10.4 Genauere Beschreibung

Der Levelaufbau ist im Entwurfsmodus tilebasiert (felderbasiert), dies vereinfacht Levelplannung und Platzierung der einzelnen Gameelemente. Im Sinne von Übersichtlichkeit werden Spielelemente, die im Spiel viel größer sind, immer nur in einem Tile dargestellt.
Die Verdeckungshierarchie wird im Editor umgekehrt: im Spiel wird der Diamant unter dem Busch gezeichnet, da er dort versteckt sein soll - im Editor, da man das Level planen will, muss der Diamant sichtbar sein und der Busch ist nun der Hintergrund.
Zusätzliche Informationen zu bestimmten Gameelementen werden in Form von einfachen Symbolen über ihnen angezeigt, z.B. die Laufrichtung eines Fließbandes mit einem Pfeil.

Figure 50: Direkter Vergleich von Editoransicht (links) und Spielansicht (rechts). In der Editoransicht ist der rote Trank (l.o.) und die Gegner in Form von blauen Händen (r.u.) zu sehen. In der Spielansicht sind diese nur teilweise oder gar nicht zu sehen. Die beiden Gegner in der Mitte sind in der Editorsicht deutlich kleiner dargestellt, als in der Spielansicht. Der Inhalt der Truhe wird in der Editoransicht angezeigt, in der Spielansicht erst nachdem die Truhe geöffnet wurde.

11 Evaluation

11.1 Evaluation über Anforderungsliste

1.) Gameplay

- (a) Standard & Multiplayerspielmechaniken laut 3.2 √
- (b) verschiedene Gegner (Enemies), z.B. am Boden, fliegend, Endgegner etc. √
- (c) Enemy Drop-, Dying- and RespawnBehavior √
- (d) verschiedene Gegenstände (Items) √
  -Item-Shops
  -verschiedenes Itembehavior, multiple Anwendungen für ein und denselben Gegenstand √
- (e) Rätsel mit multiplen Lösungsansätzen √
- (f) interaktive Gameelemente √
- (g) Ressourcenmanagement √ (Mana, Trefferpunkte, Geld, Komponenten)
- (h) Kampagne
  -Tutorial
  -Learning by Advancing
  -Backtracking for Advancing
  -Storyline
  -Cut-Scenes
- (i) Collision Detection (Rectangle, Circle Collision) √
- (j) Laden/Speichern von Spielständen

2.) Grafik

- (a) Spritegrafik √
- (b) Spriteanimation √
- (c) Simple Effekte wie z.B. Glow / Transparenz √
- (d) Darstellung von Schatten und Licht √

3.) Editor

- (a) Auswahlboxen für Spielelemente √
- (b) Setzen/Löschen von Spielelementen √
- (c) Rotation von Spielelementen √
- (d) Verschieben von Spielelementen √
- (e) Erstellen/Löschen von Räumen √
- (f) Steuervariablen von Spielelementen setzen √
- (g) Laden/Speichern bestehender Levels √
- (h) Übersichtkarte √
- (i) Ansicht rein- und rauszoombar √
- (j) Verschiedene Texturen verwend- und anwählbar √

3.) Sonstige Anforderungen

- (a) X-Box Controller Usabillity √
- (b) Musik angepasst an Spielsituation √
- (c) Soundeffektlautstärke relativ zu Position des Spielers √
- (d) Mehrspielermodus √
- (e) Optionsmenü um Sound, Musik, Tastenbelegung einzustellen

11.2 Evaluation über Benutzertests

11.2.1 Aufbau des Benutzertests

Probanden wurden 3 Testszenarien zum Spielen angeboten:

1. Sehr leichtes Szenario:
   Größe: 12 Räume
   linear aufgebaut, geringe Gegneranzahl, das Lösen der vorhandenen Rätsel ist nicht zum Erreichen des Spielziels notwendig, sondern nur zum Erhalten zusätzlicher Punkte, kein Backtracking notwendig

2. Mittelschweres S zenario:
   Größe: 20 Räume
   linear aufgebaut, mittlere Gegneranzahl, das Lösen der vorhandenen Rätsel ist zum Erreichen des Spielziels notwendig, kein Backtracking notwendig

3. Schweres Szenario:
   Größe: 36 Räume
   nicht linear aufgebaut, es gibt Sackgassen und multiple Abzweigungen, hohe Gegneranzahl, das Lösen der vorhandenen Rätsel ist zum Erreichen des Spielziels notwendig, Backtracking notwendig

In jedem Szenario startet der Spieler mit 250 Lebenspunkten, 250 Manapunkten und einer Grundausrüstung (Schwert, Fernkampfwaffe und Schild).

Das Spiel sammelt während des Spielens der Probanden zusätzliche Information wie Spieldauer, gesammelte Ressourcen, gefundene Geheimnisse, Spielziel erreicht, wieviel HP/Mana wurden verbraucht. Diese werden dann in Textdateien zur späteren Auswertung gespeichert.

Zusätzlich wurde den Probanden nach dem Spielen folgender Fragebogen vorgelegt:

Alter:
Geschlecht:

Erfahrung mit vergleichbaren Spielen:
(Skala 1-10, 1 keine, 10 sehr hoch)
Bewertung der Steuerung:
(Skala 1-10, 1 unspielbar, 10 supergenau)
Spielspaß:
(Skala 1-10, 1 langweilig, 10 spaßig)
War erkennbar was die einzelnen Dinge im Spiel darstellen sollen?
(Skala 1-10, 1 alles unerkennbar, 3 teilweise, 6 größtenteils, 10 absolut)
Wie gut fanden sie die 2D-Grafik allgemein?
(Skala 1-10, 1 schlecht, 10 sehr gut)
Wie schwierig fanden sie es vom Anspruch her?
(Skala 1-10, 1 zu leicht, 10 unschaffbar)
Welches Level haben sie gespielt?
Hatten sie Mitspieler?

Sonstige Bemerkungen/Kritik/Verbesserungsvorschläge:

11.2.2 Auswertung des Benutzertests

ø-Alter der Teilnehmer : 21,793 Jahre
Jüngster Teilnehmer : 12 Jahre
Ältester Teilnehmer : 32 Jahre

Figure 51:

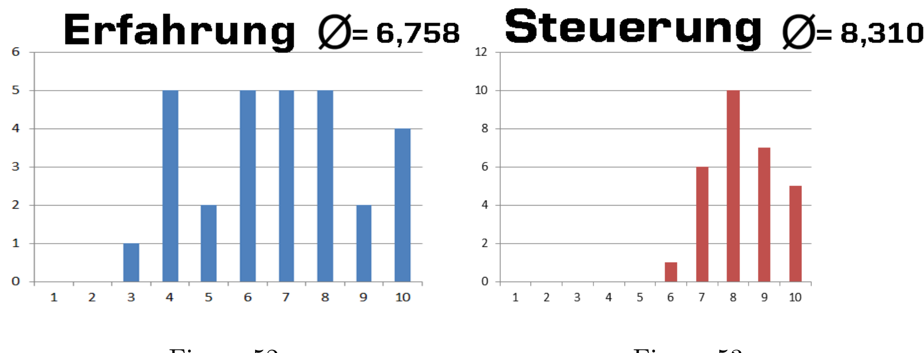


Figure 52: Figure 53:

Obwohl Leute mit wenig Erfahrung (4 und weniger) gespielt haben, kamen alle Probanden mit der Steuerung gut zurecht (ø=8.310), fehlerminimierendes Bewegungskorrektursystem und einfaches Kontrollersetup wurden erfolgreich angenommen und angewandt. Viele der erfahreneren Spieler empfanden die Entscheidung, einen Extraknopf zum interagieren mit Doodaads zu haben (der Y-Button vgl. Kap. 8.1) zuerst gewöhnungsbedürftig, fanden es aber im Laufe des Spiels vor allem beim genauen Verschieben von Kisten sehr nützlich.

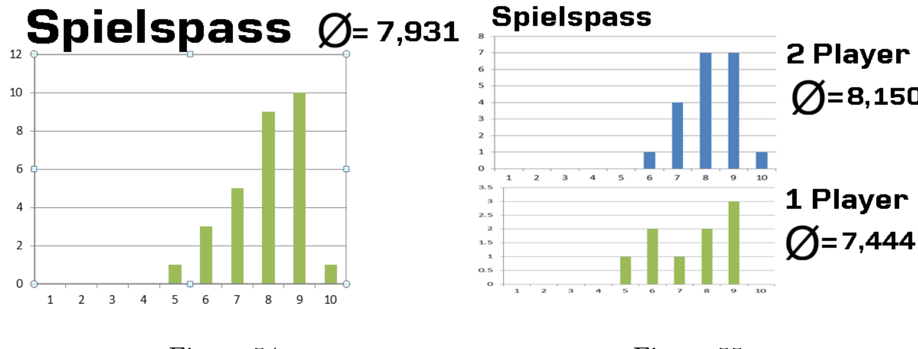


Figure 54:

Figure 55:

Die Probanden empfanden das Spiel größtenteils als sehr unterhaltsam. "Infinite Worlds" war als kooperatives Spiel geplant. Der signifikante Unterschied in der getrennten Spielspaßwertung in Fig. 55 spricht für die erfolgreiche Umsetzung dieser Idee.
Besonders gut angekommen ist die Idee, dass wenn jemand im Team zu Boden geht, man ihm wieder aufhelfen kann (vgl. Kap. 3.2.2.). Bemängelt wurde die Tatsache, wenn man ein neues Item findet, dies nicht zwingend besser ist als die bereits gefundenen.

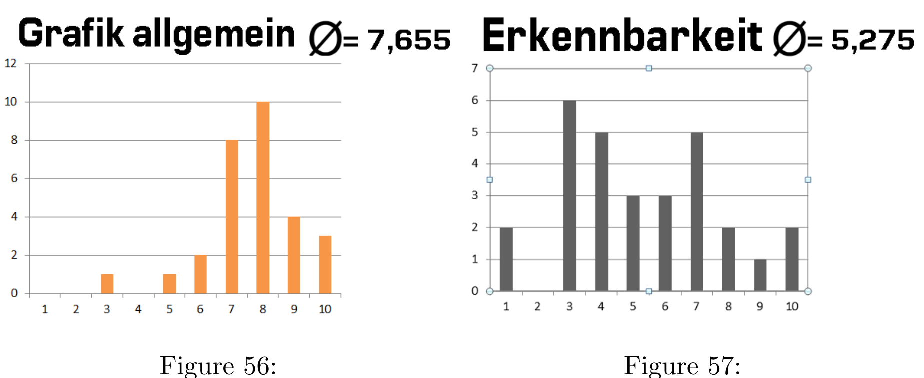


Figure 56:

Figure 57:

Das Designkonzept für "Infinite Worlds" war eine bewußte Entscheidung für einfache animierte 2D-Texturen. Mit ø=7.655 wurde zwar der Geschmack der Zielgruppe getroffen, dennoch kristallisierte sich im Laufe des Testens besonders bei Spielanfängern die Erkennbarkeit als ein relevantes Problem heraus.

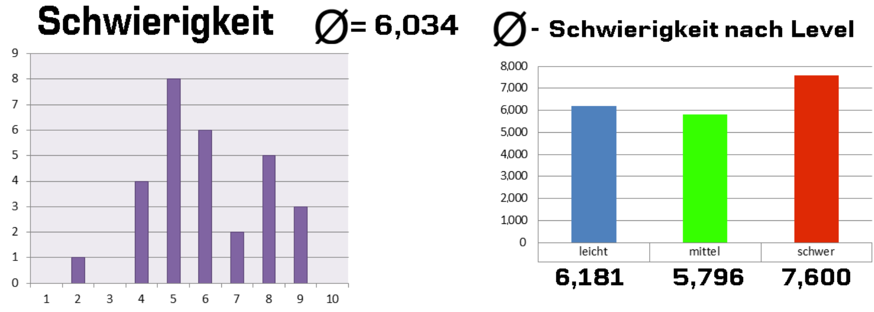


Figure 58:

Figure 59:

Um an verschiedenen erfahrenen Spielern die Schwierigkeit des Spiels zu testen, wurden verschiedene Schwierigkeitsgrade angeboten. Im Idealfall sollte ein Wert von 5,5 erreicht werden. Der globale ø-Wert mit 6,034 liegt noch relativ nahe daran, aber die lokale Betrachtung liefert eindeutigen Verbesserungsbedarf was das Setzen der Schwierigkeitsgrade angeht. Der erhöhte Wert bei dem leichten Szenario spricht für das Erkennbarkeitsproblem. Selbst für fortgeschrittene Spieler mit viel Erfahrung stellt das Verstehen der schieren Menge an verschiedenen Spielelementen eine Herausforderung dar.

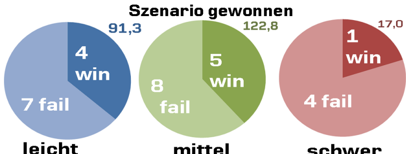


Figure 60: Die Zahl links oben gibt an, wenn das Szenario gewonnen wurde mit wieviel Hitpoints die Spieler es beendet hatten. Das schwere Szenario wurde nur von einer einzigen Person geschafft.

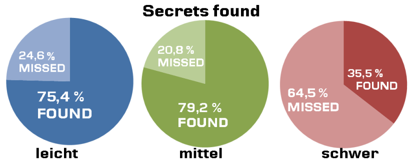


Figure 61: Nachdem die einzelnen Rätselelemente um die Secrets zu finden verstanden wurden, konnten im leichten und mittleren Szenario ein Großteil der Aufgabenstellungen gelöst werden.

12 Fazit

Die Spielmechaniken aus Kap. 3.2. wurden alle wie beschrieben umgesetzt, die gegebene Klassenstruktur der Gameelemente funktioniert, um beliebige Spiellogiken abzubilden. Im graphischen Bereich funktioniert die Spriteanimation wie gewünscht, doch das Schattenverfahren der Schattenfühler weist noch deutlichen Verbesserungsbedarf auf.
Gesamtkonzept und Steuerung kamen bei den Probanden im Benutzertest sehr gut an. Erwartungskonform wurde eine höhere Spielspaßwertung im Multiplayer-Modus erzielt.
Die Editorumgebung stellte sich zum Erstellen der Testszenarien mehr als ausreichend heraus.

12.1 Ausblick

"Infinite Worlds" soll nach Abschließen dieser Arbeit weiterentwickelt werden, um es anderen Menschen zum Spielen anzubieten. Die im Laufe des Benutztertestes aufgezeigten Defizite sollen wie folgt behandelt werden:
-Erstellen eines Tutorials.
-Erstellen einer Hilfefunktion in Form eines Fragecursors, welcher im Pausemodus eine Kurzbeschreibung des Spielelements anzeigt, über welchem er gerade steht.

Der Editor soll durch ein einfaches Malprogramm erweitert werden. Dadurch können Texturen selbst gezeichnet und bestehende Texturen bearbeitet werden.
Die Spriteanimation könnte mit 2D-Skelett-Animation [3] erweitert werden.

13 Danksagung

Ich bedanke mich bei Stefan Müller für die Betreung dieses Projekts und in erster Linie für die Befürwortung einer solchen Idee.
Viel von meinem Dank geht auch an Etiene Miller, welcher mir als Texturartist viel geholfen hat.

References

[1] http://macs2.psychologie.hu-berlin.de/aio/index.php/psychologische-grundlagen/wahlreaktion/konzepte-wahlreaktion/44-reaktionszeit
Stand: 21.07.2013,

[2] David Perry,Rusel deMaria: *David Perry on GAME DESIGN, A Brainstorming Toolbox*, Course technology (2009)

[3] Catalin: *2D Skelletal Animations*, http://www.catalinzima.com/2011/06/2d-skeletal-animations/, Stand:30.07.2013

[4] www.nintendo.de: http://www.nintendo.de/Unternehmen/Unternehmensgeschichte/Nintendo Geschichte-625945.html, Stand:30.07.2013

[5] Square,Nintendo: *Secret of Mana*, 24. November 1994

[6] Square Electronic Arts, L.L.C. (USA): *Seiken Densetsu*, 7. Juni 2000

[7] Nintendo,Capcom: *Zelda: The minish cap*, 10. Januar 2005

[8] Nintendo,Quintet: *Terranigma*, 19. Dezember 1996

[9] Nintendo,Quintet: *Illusion of time*, 27. April 1995

[10] Nintendo EAD: *The Legend of Zelda: Spirit Tracks*, 11. Dezember 2009

[11] http://msdn.microsoft.com/en-us/library/ee817667.aspx
Stand: 21.07.2013

[12] Lutz Priese: *Verarbeitung und Analyse digitaler Bilder*, 2012