

Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt  
Fakultät Informatik und Wirtschaftsinformatik

## **Projektdokumentation**

# **Entwicklung des 2D-Sidescrolling Spiels "Josie - A Jelly's Journey" auf Basis von C++**

**vorgelegt an der Hochschule für angewandte Wissenschaften  
Würzburg-Schweinfurt in der Fakultät Informatik und Wirtschaftsinformatik zum  
Abschluss des Programmierprojekts im vierten Studiensemester im Studiengang  
Informatik**

Oleg Geier      Daniel Glück      Jonas Kaiser  
Tobias Lediger      Daniel Mügge

Eingereicht am: 17.07.2015

Erstprüfer: Prof. Dr. Peter Braun  
Zweitprüfer: Prof. Dr. Steffen Heinzl

# **Zusammenfassung**

TODO

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Ausgangssituation . . . . .	1
1.2 Motivation . . . . .	1
1.3 Vorgehen . . . . .	2
1.4 Dokumentationsstruktur . . . . .	2
<b>2 Grundlagen</b>	<b>3</b>
2.1 Spielprinzip von Josie - A Jelly's Journey . . . . .	3
2.2 Einrichtung des verwendeten Frameworks . . . . .	3
2.3 Verwendete IDE und benötigte Plug-Ins . . . . .	5
2.4 Szenenprinzip . . . . .	6
2.5 Spriteprinzip . . . . .	7
2.5.1 cocos2d Spriteprinzip . . . . .	8
2.5.2 Möglichkeiten Sprites zu manipulieren . . . . .	9
2.6 Animationsprinzip . . . . .	10
2.6.1 Allgemeines und cocos2d Spriteprinzip . . . . .	10
2.6.2 Animate und Sequenze . . . . .	11
2.6.3 Spritesheet . . . . .	12
2.7 Datenkommunikation . . . . .	12
2.7.1 Callbackprinzip . . . . .	12
2.7.2 Observer Pattern . . . . .	12
2.8 Tilemaps . . . . .	13
2.8.1 Vorteile der Tilemap . . . . .	13
2.8.2 Funktionsweise von Tilemaps . . . . .	14
2.8.3 Tilemaps in cocos2d . . . . .	15
2.9 Musik und Sound-Effekte . . . . .	16
2.9.1 Möglichkeiten der cocos2d-x-Engine zur Audioverarbeitung . . . . .	16
2.9.2 Monofonie/Stereofonie und verwendete Dateiformate . . . . .	17
2.9.3 Audiotools . . . . .	17
<b>3 Architektur</b>	<b>19</b>
3.1 Spielstruktur . . . . .	19
3.2 Vererbung . . . . .	20
3.3 Speichersystem . . . . .	21

## Inhaltsverzeichnis

<b>4 Implementierung</b>	<b>22</b>
4.1 Spieler Steuerung . . . . .	22
4.2 Sprites . . . . .	23
4.2.1 Bewegen . . . . .	23
4.2.2 Animieren . . . . .	24
4.2.3 Fortsetzung des Programmablaufs . . . . .	25
4.3 AudioUnit . . . . .	26
4.3.1 Laden von Soundeffekten . . . . .	26
4.3.2 Abspielen von Soundeffekten und Musik . . . . .	27
4.4 CollisionLayer . . . . .	27
4.4.1 Debugging Optionen . . . . .	28
4.4.2 Listener registrieren . . . . .	29
4.4.3 Gegenseite Collision Notification . . . . .	29
4.5 Kollisionsabfrage zum Boden . . . . .	29
4.6 Automatisch erzeugte Tilemaps . . . . .	30
4.6.1 Generieren des Levels . . . . .	30
4.6.2 Platzieren der Münzen . . . . .	32
<b>5 Fazit und Ausblick</b>	<b>37</b>
5.1 Features for the Future . . . . .	37
<b>6 Anhang A</b>	<b>39</b>
<b>Verzeichnisse</b>	<b>40</b>
<b>Listings</b>	<b>41</b>
<b>Literatur</b>	<b>41</b>

# 1 Einleitung

## 1.1 Ausgangssituation Daniel Mügge

In der heutigen Zeit spielt man Videospiele nicht nur auf Computern oder Konsolen, sondern auch auf Mobiltelefonen. Der Markt von Spielen für mobile Geräte ist in den letzten Jahren rapide gewachsen und erfreut sich immer größerer Beliebtheit. Einer Studie von Bitkom [[bitkomgaming](#)] aus dem Jahr 2014 zufolge, in der die beliebtesten Spieleplattformen ermittelt wurden, führt das Smartphone bzw. Handy mit 78% um 9% gegenüber dem stationären PC und ist somit an erster Stelle. Dieselbe Studie hat sich auch mit den beliebtesten und meist gespielten Spieldenren beschäftigt. Strategie- und Denkspiele sind laut Bitcom am beliebtesten, gefolgt von Gelegenheitsspielen, Actionspielen, Social Games, Jump and Runs und Renn- und Sportspielen.

## 1.2 Motivation Daniel Mügge

Im bisherigen Verlauf unseres Informatik-Studiums hatten wir wenig mit GUI oder Grafik im allgemeinen Sinne zu tun. Die meiste Zeit sehen wir Konsolenausgaben in weiß auf schwarz, ein wenig Textausgabe und das war es dann auch schon.

Wir wollten etwas entwickeln, mit dem wir im späteren Leben höchstwahrscheinlich nur noch als Anwender zu tun haben. Ein Spiel.

Viele Informatik Studenten träumen oder haben davon geträumt ein Spieleentwickler zu werden. Doch meistens wird daraus nichts. Deshalb haben wir uns gedacht bevor wir ins wirkliche Berufsleben einsteigen, wollen wir einmal ein eigenes Spiel entwickeln und haben es **Josie–A Jelly’s Journey** getauft.

## 1.3 Vorgehen Daniel Mügge

Am Anfang war das Nichts.

Eine der schwierigsten Phasen in unserem Projektverlauf war das grobe Design. Wir wollten dass **Josie—A Jelly's Journey** jedem aus unserer Gruppe gefällt und jeder seine Ideen einbringen kann. Deshalb wurden die ersten zwei Wochen des Projekts dem Design gewidmet.

Nachdem wir wussten welche Komponenten für die Entwicklung benötigt werden, haben wir die Aufgabenbereiche auf die Team-Mitglieder wie folgt verteilt.

- Oleg Geier: Programmierung und Logik
- Daniel Glück: Grafikdesign und Spieldesign
- Jonas Kaiser: Spieldesign und Levelgenerierung
- Tobias Lediger: Storydesign und Zwischensequenzen
- Daniel Mügge: Audiodesign und Grafikdesign

## 1.4 Dokumentationsstruktur Daniel Mügge

Im nächsten Kapitel werden die Grundlagen erklärt. Diese beinhalten eine Anleitung zur Einrichtung der Entwicklungsumgebung und Einbindung der verwendeten Engine, eine grobe Einführung in die wichtigsten Bestandteile dieser und wie diese in unserem Programm eingesetzt wurden. Kapitel 3 zeigt die Struktur des Spiels und das Zusammenspiel der Klassen und deren Abhängigkeiten mit Hilfe von Diagrammen. In Kapitel 4 geben kleine Code-Beispiele einen Einblick über die Implementierung der zur Verfügung stehenden Klassen und Methoden in den Programmcode. Kapitel ?? beschäftigt sich mit der Evaluation des Projektes im Hinblick auf das was geschafft und wie gut es umgesetzt wurde. Sowie auf den Ablauf des Projektes und die Probleme die während der Entwicklung entstanden, gelöst oder nicht gelöst wurden. Kapitel 5 enthält Ausblicke für die Zukunft des Projektes und ein abschließendes Fazit.

## 2 Grundlagen

### 2.1 Spielprinzip von Josie - A Jelly's Journey Daniel Mügge

**Josie-A Jelly's Journey** erinnert an eine erweiterte Version des Spieleklassikers Super Mario World. In der Theorie sollte es mehrere Spielabschnitte geben, jedoch wurde aus zeitlichen Gründen nur ein Abschnitt implementiert. Ein Spielabschnitt besteht aus drei Jump and Run Levels und einem Boss Kampf Level.

**Jump and Run Level:** Der Spieler versucht ohne Zeiteinschränkung das Level abzuschließen und dabei so viele Münzen wie möglich zu sammeln. Bei einem perfekten Lauf kann man alle Münzen einsammeln, einige davon sind allerdings so plaziert dass sie nur schwer zu erreichen sind. Dem Spieler stehen die Steuerungsfunktionen "Stoppen", "Schrumpfen" und "Springen" zur Verfügung. Dass Josie von selbst in einer festgelegten Geschwindigkeit läuft, führt zu einer gewissen Schwierigkeit des Spiels und macht Steuerungsfunktionen für "Links" und "Rechts" überflüssig.

**Boss Level:** Im Bosskampf wird der Boss bekämpft. Allerdings ist man nicht mehr die gewohnte Figur aus dem Jump and Run Leveln sondern spielt Josie in Form eines Panzers. Dieser besitzt die folgenden Steuerungsfunktionen "Links" und "Rechts" sowie die Fähigkeit "Schießen". Eine weitere Neuerung bietet der Shop, welcher sich öffnet bevor man in das eigentliche Boss Level kommt. Hier kann man sich verschiedene Power Ups, mit vorher erspielten Münzen, kaufen. Im eigentlichen Bosskampf steht man, in Version 1.0, Sir Eichenborke gegenüber. Das Ziel ist ihm Leben abzuziehen bis die Lebensleiste leer ist. Dies passiert indem man seine Hände trifft. Allerdings muss man diesen auch ausweichen da er sonst Josie Leben abzieht.

### 2.2 Einrichtung des verwendeten Frameworks Tobias Lediger

Als die grobe Idee für das Spiel feststand, mussten wir uns für eine GameEngine entscheiden. Zur Auswahl standen unter anderem die AndEngine, Unity und cocos2d-x.

## 2 Grundlagen

Nach ausreichender Recherche über die Funktionalitäten der verschiedenen Engines haben wir uns einstimmig für cocos2d-x entschieden. Gründe hierfür waren unter anderem, dass die Engine immer weiterentwickelt wird, dass damit schon sehr viele mobile Spiele programmiert wurden, die Dokumentation sehr detailliert und ausgereift ist und da cocos2d-x sehr viele Funktionen zur Verfügung stellt, die auch wir im Spiel verwenden. Cocos2d-x unterstützt drei Programmiersprachen, C++, JavaScript und Lua. Hier haben wir C++ gewählt, da wir im Laufe unseres Studiums eine Vorlesung über C++ gehört haben. Die nächste Herausforderung war die Einrichtung von Cocos2d-x. Da wir drei verschiedene Betriebssysteme (Win8, MacOSX, Ubuntu) benutzen, hatten wir geringe Kompatibilitätsprobleme. Um Cocos2d-x einzurichten, werden folgende Pakete benötigt:

- *Cocos2d-x* - <http://www.cocos2d-x.org/download>
- *Android SDK* - <http://developer.android.com/sdk/index.html>
- *Android NDK* - <http://androids.zone/android-ndk/#.VBLHNPldWXY>
- *Apache Ant* - <http://ant.apache.org/bindownload.cgi>
- *Python* - <https://www.python.org/downloads/>
- *Eclipse* - <https://eclipse.org/downloads/>

Nachdem die Pakete heruntergeladen und ausgepackt sind, verschiebt man diese in den selben Ordner. Apache Ant und Python müssen installiert werden. Bei Python muss darauf geachtet werden, dass es sich um Version 2.x handelt. Unter Ubuntu muss Python nicht installiert werden, da dies schon in den Paketquellen vorhanden ist. Hat man die zwei Programme installiert, öffnet man den cocos2d-x Ordner und startet die setup.py Datei (über das Terminal). Daraufhin öffnet sich ein Terminalfenster, in welchem die Umgebungsvariablen angepasst werden, dies bedeutet man wird nach den Pfaden von der Android SDK/NDK, Apache Ant und Python gefragt. Entweder man gibt diese per Hand ein oder zieht die jeweiligen Ordner in das Terminalfenster. Vorsicht: Bei manchen Betriebssystemen ist die maximale Länge der angegebenen Pfade auf 32 Zeichen begrenzt.

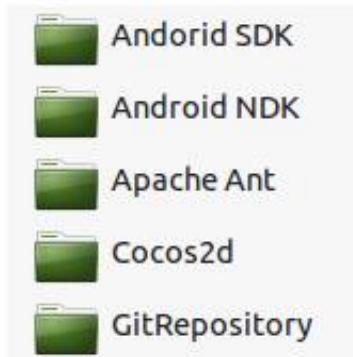


Abbildung 2.1: Ordnerstruktur

## 2.3 Verwendete IDE und benötigte Plug-Ins Tobias Lediger

Nachdem die Umgebungsvariablen eingetragen sind, muss das Android Developer Tool Plugin für Eclipse installiert werden. Hat man das erledigt öffnet man den im Plugin enthaltenen Android SDK-Manager und lädt sich hier nochmal die passende API herunter. In unserem Fall war das die API für Androidversion 4.4.2(API 19). Ist dieser Schritt geschafft, wird es Zeit das erste Projekt zu erstellen. Hierzu öffnet man einen Terminal und wechselt in das Verzeichnis in dem auch der cocos2d-x Ordner liegt. Hier gibt man folgenden Befehl ein:

```
$cocos new MyGame -p com.MyCompany.MyGame -l cpp -d ~/MyCompany
```

- *MyGame* gibt den Namen des Projekts an
- *com.MyCompany.MyGame* ist der Paketname für Android
- *cpp* steht für die Programmiersprache. cpp für C++ und lua für Lua
- */MyCompany* gibt das Verzeichnis an in dem das Projekt gespeichert wird

Der somit erstelle Ordner wird ins Git-Repository hochgeladen. Nun öffnet man wieder Eclipse und importiert das Git-Repository sowie die Cocos2d-x Bibliotheken.

*Git-Repository:* Eclipse → File → Import... → Git → 'Projects from Git'

Hier wählt man das vorher erstellte Projekt im Git-Repository aus.

*Cocos2d-x Bibliotheken:* Eclipse → Import... → Android → 'Existing Android Code into Workspace'

Hier dirigiert man in das Verzeichnis in dem der cocos2d-x Ordner liegt und wählt diesen komplett aus. Im nachfolgenden Fenster wählt man alle vorgeschlagenen Punkte ab bis auf libcocos2dx Nachdem beide *Projekte* importiert wurden, wählt man sein Projekt aus dem Git-Repository aus und wechselt in die Einstellungen. Unter dem Reiter *Android* ist darauf zu achten, dass als Library das Projekt libcocos2dx ausgewählt ist. Ist es nicht in der Liste enthalten, fügt man es mit *Add* hinzu.

Es ist darauf zu achten, dass sich alle Ordner inkl. dem Git-Repository in einem Ordner befinden.

Zu guter Letzt, öffnet man die *AndroidManifest.xml* in Eclipse aus und ändert hier den Wert auf die Entsprechende API.

```
<uses-sdk android:minSdkVersion="19"  
         android:targetSdkVersion="19"/>
```

Nun kann man anfangen zu arbeiten!

## 2.4 Szenenprinzip Tobias Lediger

Eine Szene ist im Grunde genommen nichts anderes als ein Container, welcher Sprites, Labels, Nodes und andere Objekte beinhaltet die ein Spiel benötigt. Eine Szene ist für die laufende Spiellogik und Darstellung des Inhaltes auf einer “per-frame basis” verantwortlich. Es wird mindestens eine Szene benötigt damit man das Spiel starten kann. Man kann beliebig viele Szenen-Objekte in einem Spiel verwenden und leicht zwischen diesen überleiten. Der Vorteil von Szenen liegt darin, dass man nicht jedes Objekt einzeln laden muss. An eine Szene lassen sich diverse Sprites, Labels und Nodes mit der von cocos2d-x gegebenen Funktion *addChild* anhängen(siehe Abbildung 2.2). Sobald das Szenen-Objekt geladen wird, werden die angefügten Kinder mitgeladen. Dies spart Zeit und entlastet den Speicher.

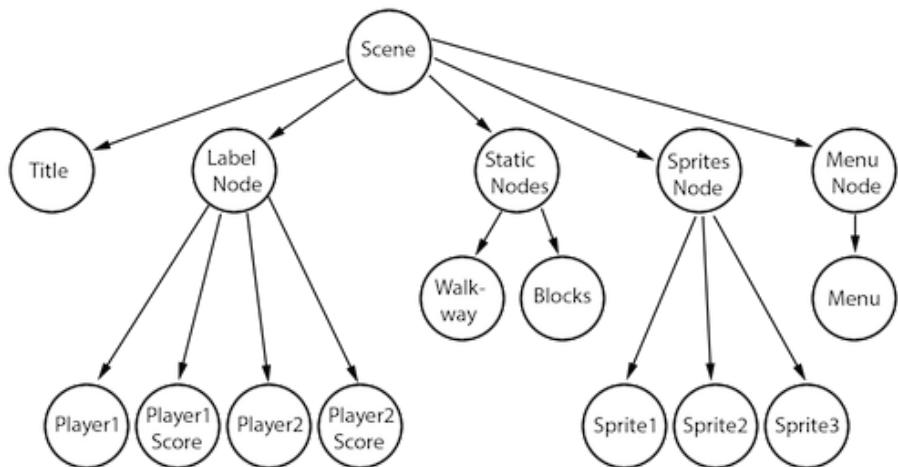


Abbildung 2.2: Szenengraph

Cocos bietet Funktionen an um Szenen zu erstellen und zwischen diesen zu wechseln. Um einen Szenenwechsel durchzuführen, muss zuerst eine Szene erstellt werden.

```
Szene* Cutscene = Szene::create();
```

Dies erstellt ein Objekt des Typ Szene mit dem Namen Cutscene. Im Laufe des Spiels ist es notwendig zwischen den verschiedenen Szenen zu wechseln. Dies wird deutlich, wenn man z.B. ein neues Spiel starten oder ein anderes Level auswählen möchte. Hierzu stellt Cocos2d-x verschiedene Funktionen bereit eine Szene zu wechseln.

- *replaceScene()* ersetzt eine Szene vollständig durch eine Andere
- *pushScene()* unterbricht die Ausführung der aktuellen Szene und verschiebt diese auf den Stack. Der Stack ist eine Art Warteschlange welche nach dem 'Last in, First out – Prinzip', dort wartet die Szene auf weitere Anweisungen. Diese Funktion darf nur aufgerufen werden, wenn bereits eine Szene aktiv ist.
- *popScene()* wiederum ersetzt die aktuelle Szene und löscht diese komplett. Diese Funktion darf nur aufgerufen werden, wenn bereits eine Szene aktiv ist.

## 2.5 Spriteprinzip Daniel Glück

Allgemein betrachtet kann man sagen, dass ein Sprite(engl. Kobold, Geistwesen) ein Grafikobjekt ist, welches über den Hintergrund gelegt wird und von der Grafikhardware platziert wird. Der Name röhrt daher, dass ein Sprite auf dem Bildschirm umherspäht

und im Grafikspeicher nicht zu finden ist. Heutzutage bezeichnet der Begriff “Sprite” jedoch alle Objekte die so aussehen wie ein solches Grafikobjekt, jedoch eigentlich von einer Software erzeugt werden und im Grafikspeicher vorliegen. Solche softwareerzeugten Sprites sind streng genommen “Shapes”, für deren Erzeugung überwiegend die CPU zuständig ist.

Für Computerspiele sind mit Sprites einige Vereinfachungen verbunden. So werden zum Beispiel in vielen 2D Jump and Run Spielen so genannte Tiles oder auch Kachelgrafiken verwendet, welche ebenfalls kleine Grafikelemente sind die zusammengesetzt eine größere Grafik ergeben. Ihr Anwendungsbereich findet sich unter anderem im Aufbau eines Levels, wobei aus ihnen die Spielwelt zusammengesetzt wird.

### 2.5.1 cocos2d Spriteprinzip

In der von uns verwendeten Gameengine cocos2d-x, ist ein Sprite ein “Bild”, welches durch Veränderung seiner Werte manipuliert werden kann. Es gibt verschiedene Wege ein Sprite zu erstellen, je nach dem wozu es benutzt werden soll. Beziiglich Dateiformaten werden von cocos2d-x PNG, JPEG, TIFF, etc. unterstützt. Wir haben uns für das Dateiformat PNG entschieden, da es eine gute Komprimierung, gute Qualität, Darstellung von Halbtransparenzen, also 50% Deckkraft, vorweist und außerdem ein sehr weit verbreiteter Datentyp ist.

Es gibt unterschiedliche Methoden ein Sprite zu erstellen. Eine Möglichkeit ist das Sprite aus einem Bild zu laden, wobei das in cocos2d-x erstellte Sprite Objekt die selben Abmessungen wie das benutzte Bild vorweist.

```
Sprite* mySprite = Sprite::create("mysprite.png");
```

Eine weitere Methode ist das Erschaffen eines Sprites durch Angabe eines Ausschnittes des benutzten Bildes. Dabei wird im Erstellungsprozess ein so genanntes `Rect` angegeben, welches die Position als auch die Dimension auf dem Bildschirm darstellt.

```
Sprite* mySprite = Sprite::create("mysprite.png", Rect(0,0,40,40));
```

Die Möglichkeit ein Sprite aus einem Spritesheet zu erstellen ist besonders empfehlenswert. Ein Spritesheet ist eine Bilddatei in der mehrere Sprites beliebig aneinander gereiht gespeichert werden können. Dies birgt den Vorteil, dass nur eine Datei geladen werden muss anstatt vieler einzelner Bilder, was die Ladezeiten erheblich verringert und zudem eine Speicherreduktion mit sich bringt. Außerdem reduzieren die Spritesheets die benötigten Aufrufe an OpenGL ES etwas zu zeichnen und zu rendern. Beim erstmaligen verwenden eines Spritesheets, wird dieses in den `cocos2d::SpriteFrameCache` geladen. Dies ist eine Klasse, welche ein `cocos2d::SpriteFrame` Objekt, für zukünftigen Schnellzu-

griff speichert. SpriteFrame-Objekte beinhalten den Bildnamen des Sprites und ein Rect um die Größe des Sprites zu spezifizieren. Aus dem SpriteFrameCache, in welchem das Spritesheet geladen wurde, kann nun ein Sprite erstellt werden. Spritesheets stellen in unserem Projekt speziell für Animationen und die Umgebung der Spielwelt eine optimale Lösung dar.

### 2.5.2 Möglichkeiten Sprites zu manipulieren

Der Ankerpunkt eines Sprites ist ein Punkt, welcher zur Orientierung bei der Positionsbestimmung eines Sprites dienen soll. Der Ankerpunkt benutzt ein Koordinaten System das von 0 bis 1 geht, wobei 0,0 unten links und 1,1 oben rechts vom Sprite ist.

```
Sprite* mySprite->setAnchorPoint(0,0); // (0,1), (1,0), (0.5,0.5)
```

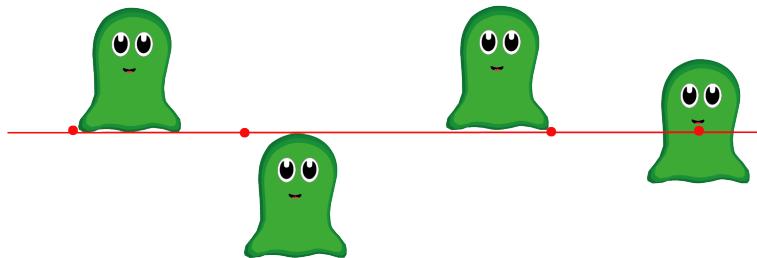


Abbildung 2.3: Plazierung von Josie bei verschiedenen Ankerpunkten (rot)

Möglichkeiten ein Sprite zu manipulieren sind unter anderem Skalieren, Rotieren und Verzerren. Weiterhin kann man die Farbe und Sichtbarkeit eines Sprites verändern. Die in unserem Projekt am häufigsten verwendete Manipulationsmethode ist das Skalieren. Sie ermöglicht es die Größe eines Sprites beliebig zu verändern.

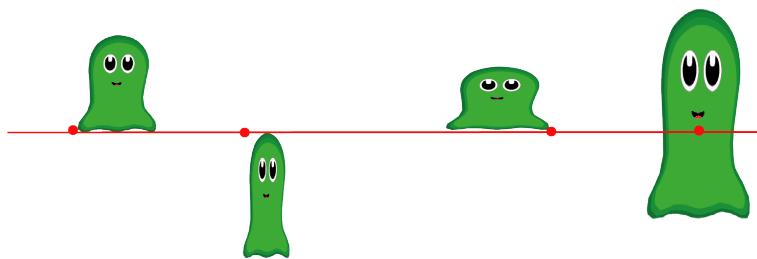


Abbildung 2.4: Josie nach dem Skalieren

## 2.6 Animationsprinzip Daniel Glück

### 2.6.1 Allgemeines und cocos2d Spriteprinzip

Im Verlauf unseres Projektes wurden verschiedenste Animationen benutzt. Im folgenden soll ein kurzer Einblick in das Prinzip der Animationen in cocos2d-x, als auch allgemein gewährt werden. Einfach gesagt sind Animationen nichts weiter als eine sehr schnell abgespielte Aneinanderreihung von Bildern. In der Computergrafik oder bei Computerspielen werden diese verwendet um den Spielecharakter zum Leben zu erwecken und das Spiel dynamischer zu machen. Ohne sie wären Spiele nicht grafisch darstellbar.

In cocos2d-x werden solche Animationen durch *cocos2d::Action* Objekte realisiert. Diese haben die Fähigkeit die Werte von *cocos2d::Node* Objekte, in Echtzeit zu transformieren. Dazu zählen ebenfalls Instanzen der Klassen die von einem solchen Objekt erben. Werte eines Nodes, die verändert werden können sind z.b. die Sichtbarkeit, Farbe, Position oder auch die Größe des Nodes. Zum Beispiel kann ein Sprite von einer Position zur anderen bewegt werden.

Grundsätzlich können zwei Versionen von Actions unterschieden werden, diese sind By und To. Wobei Letztere absolut sind und im Gegensatz zu By Actions d.h. sie berücksichtigen die aktuelle Position des Node nicht. Ein mögliches Beispiel für eine MoveTo Animation soll im folgenden kurz beschrieben werden. Man erstellt ein MoveBy Objekt mit einer Angabe über die Dauer der Animation in Sekunden sowie ein Vektor der die Ziel Koordinaten auf der X als auch die der Y Achse beinhaltet. Anschließend wird dieses Objekt auf einem Node z.b. ein Sprite durch die Methode **runAction()** ausgeführt.

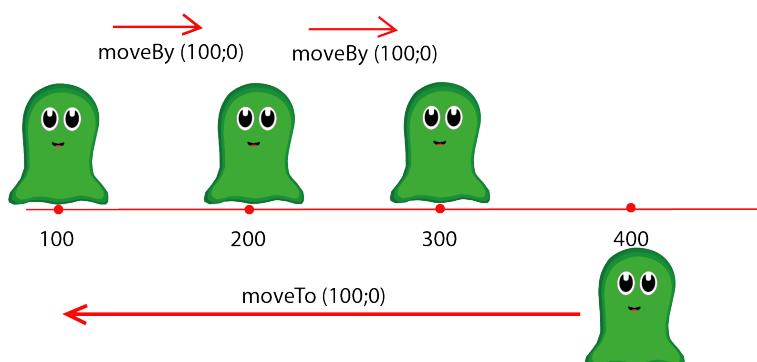


Abbildung 2.5: Unterschied zwischen MoveBy und MoveTo

Die von cocos2d-x bereitgestellten Funktionen zur Veränderung von *cocos2d::Nodes* sind *Move*, *Rotate*, *Scale*, *Fade In/Out* und *Tint*.



Abbildung 2.6: Beispiel für Fade In/Out bei Josie

## 2.6.2 Animate und Sequenzen

Im vorangegangenen Abschnitt wurden ein Überblick einiger cocos2d-x bereitgestellter Animationen geschaffen. Im Verlauf eines Spieleprojektes wäre es jedoch sinnvoll eigene Animationen erstellen zu können. Eine Möglichkeit eigene Animationen zu erstellen wie zum Beispiel eine Art simples Daumenkino, bietet die Klasse *cocos2d::Animate*. Ein *cocos2d::Animate*-Objekt enthält eine Animation bzw. wird aus dieser erstellt. Animationen sind Container, welche durch die SpriteFrame Verzögerungszeit zwischen den Frames als auch die Dauer der Action bestimmt werden. Wenn ein *cocos2d::Animate*-Objekt ausgeführt wird, werden bestimmte Frames auf dem Display durch die in dem Animationsobjekt enthaltenen, ersetzt. So kann zum Beispiel ein SpriteFrame durch ein Set von SpriteFrames ersetzt werden.

Um Komplexe Abläufe von Animationen zu definieren ist es sinnvoll eine Klasse wie *cocos2d::Sequence* zu benutzen. Die Instanz einer Sequence ermöglicht es verschiedene Action, Function und sogar Sequence Objekte hintereinander zu reihen und in einem Sequence-Objekt zusammenzufassen. So können Abläufe von Animationen zusammengefasst werden.



Abbildung 2.7: Beispiel für Sequenzen

Ein Zusammenspiel von Grundfunktionen und Sequenzen fand in unserem Projekt bei den Angriffspattern des Boss Gegners Anwendung. Hierbei war ein Problem, dass sich bei zu kurzem Puffer und zu langen Aktionszeiten die einzelnen Aktionen vermischen.

Eine weitere Vereinfachung durch cocos2d-x bietet die Reverse Methode. Diese dient dazu Animationen rückwärts ablaufen zu lassen. Diese ist in unserem Projekt unter anderem bei der Sprung Animation zum Einsatz gekommen.

### 2.6.3 Spritesheet

Eine häufige Methode zur Spriteerstellung ist es das Spritesheet bei der Erzeugung in den Cache zu laden. Hierzu benötigt man nicht nur das Spritesheet sondern auch eine beschreibende “.plist” Datei, die eine Zuweisung von SpriteFrame zu Bildern und deren Position im Spritesheet enthält. Nun kann man einerseits durch ein Zusammenspiel zwischen Spritesheet und .plist Datei die Animation bzw. den Ablauf im eigenen Programm definieren. Hierbei werden die SpriteFrame und die .plist Datei geladen, um im Anschluss die SpriteFrames in der .plist Datei mit den Sprites aus dem Spritesheet zu koppeln. Andererseits kann man den Ablauf einer Animation in der .plist Datei festlegen und im Anschluss die fertige Animation laden.

Wir entschieden uns für die Spritesheet .plist Methode, da wir ebenfalls bei der Realisierung des Levels mit Spritesheets arbeiteten und die Arbeit mit diesen daher kennen. Bei der Suche nach einem Programm für die “Datei Methode” fanden wir weiterhin nur schlechte Tutorials.

## 2.7 Datenkommunikation Oleg Geier

### 2.7.1 Callbackprinzip

In vielen Teilen des Spieles wird **CC\_CALLBACK\_0()** verwendet - unter anderem für Menü-Buttons. Ein Callback wird verwendet um einer Funktion, eine andere Funktion als Parameter zu übergeben. Im Besonderen wenn vorher nicht klar ist, wann diese Funktion ausgeführt wird (bsp. Klick auf einen Button). Genau genommen ist es nur ein Makro für die C++ Funktion **std::bind**. Anstatt der Null könnte man auch eine Eins angeben um zusätzlich den Sender als Parameter zu übergeben.

Hier sei noch kurz das **CallFuncN** erwähnt. Das Callback wird in eine *cocos2d::Action* gekapselt, um die Funktion innerhalb einer Animations Sequenz auszuführen (S: 25).

### 2.7.2 Observer Pattern

Für die Spielersteuerung (Kapitel 4.1) wird ein Observer Pattern verwendet. Auch hier ist der Hintergrund die asynchrone Abarbeitung von Benutzer Ereignissen. Im Unterschied zum Callback geht es beim Observer immer um mindestens zwei Objekte, dem der die Nachricht sendet und dem der sie empfängt. Es können aber sowohl mehrere Empfänger (bsp. Radio), als auch mehrere Sender (bsp. Tempomat und Fahrer beeinflussen die Geschwindigkeit) beteiligt sein.

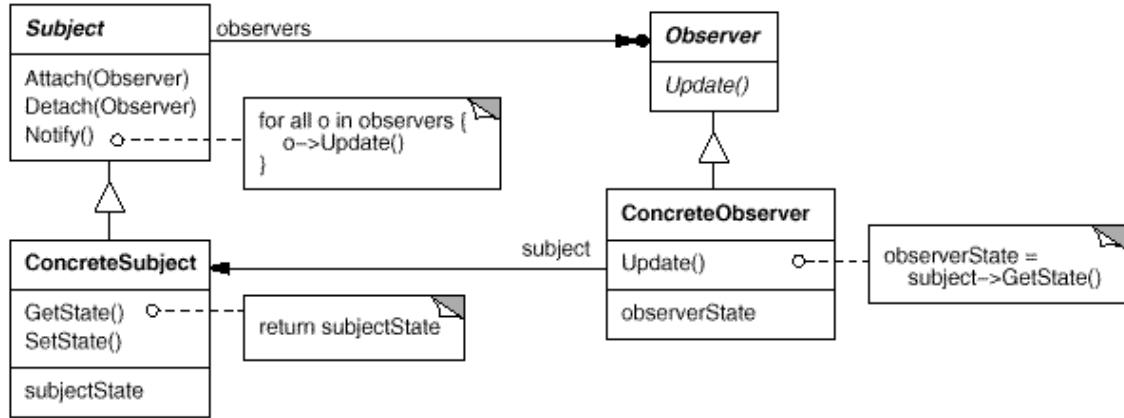


Abbildung 2.8: Observer Patter aus [gamma2011patterns]

Der Empfänger trägt sich selbst in eine Liste von Empfängern für ein bestimmtes Event/Namen ein und meldet somit sein Interesse für das Ereignis. Bei Eintreffen des Ereignisses geht der Sender die Liste der Empfänger durch und informiert jeden über das neue Ereignis.

## 2.8 Tilemaps Jonas Kaiser

Tilemaps sind spezielle Dateiformate, welche die Landschaft im Spiel, sowie die dazugehörigen Daten beschreiben. Charakteristisch für sie ist eine Untergliederung der Spielwelt in rechteckige oder Hexagon-förmige Felder (Wobei in seltenen Fällen auch andere Formen verwendet werden). Durch Tilemaps wird für diese Schachbrettartige Landschaft die grafische Darstellung sowie darüber hinaus Eigenschaften der Umwelt (Kollisionen, Spawn-Punkte usw.) festgelegt.

### 2.8.1 Vorteile der Tilemap

Als die Videospiel-Branche noch in den Kinderschuhen steckte, mussten Entwickler vor allem eines berücksichtigen: Der nur sehr begrenzt verfügbare RAM musste möglichst effektiv genutzt werden. Eine Idee um diesen Speicher effizienter zu nutzen war die Tilemap. Landschaften in Spielen sind oft in einem Stil gehalten – d.h. oft wiederholen sich Teile der Umgebung wieder und wieder. Anstatt also die gesamte Spielwelt in großen Bilddateien zu speichern die viel RAM beanspruchen, werden stattdessen nur die Komponenten aus denen die Umwelt besteht, in einer kleineren Datei/vielen kleinen Dateien

gespeichert. Die Tilemap beschreibt dann, wie diese Komponenten angeordnet werden müssen, um die Welt zu erschaffen.

Auch bei der Gestaltung der Level bieten Tilemaps einen großen Vorteil. Da man sich die Welt quasi zusammenpuzzelt, können Änderungen an Teilen der Landschaft leicht durchgeführt werden, ohne die gesamte Welt neu erschaffen zu müssen. Je nachdem wie genau das verwendete Tilemap-Format aufgebaut ist, kann theoretisch sogar ein Texteditor verwendet werden um schnell ein Level aus bestehenden Komponenten zu gestalten. Für gewöhnlich benutzt man jedoch Tools, die den Prozess visuell unterstützen. Level-Designer können sich so vollständig auf das Gestalten der Spielwelt konzentrieren, ohne sich Sorgen um den Code zu machen.

### 2.8.2 Funktionsweise von Tilemaps

Allgemein benötigt die Tilemap zwei Dinge um eine Umgebung im Spiel vollständig beschreiben zu können. Einerseits den „Map-part“ - ein 2-dimensionales Feld, welches Verweise auf den Inhalt der jeweiligen Kachelkoordinate beinhaltet. Andererseits wird noch ein Tileset benötigt. Die einzelnen Tiles sind dabei Objekte, die das Aussehen und die Eigenschaften beschreiben, welche einer Kachel zugewiesen werden können.

Eigenschaften sind frei wählbar, da sie in Form von Key-Value-Pairs gespeichert werden, die im Code des Spiels abgefragt werden können. Häufig genutzte Eigenschaften sind beispielsweise Kollisionen, die Auswirkungen des Felds auf den Spielercharakter usw. Auch Spawn-Positionen für Objekte in der Spielwelt können Eigenschaften von Kacheln sein, wobei dafür in vielen Fällen auch andere Funktionen des Tilemap-Formats genutzt werden können (diese sind jedoch nicht bei allen Formaten vorhanden).

Neben den Eigenschaften verweist jedes Tile auch auf die Grafik, die angezeigt werden soll wenn das Tile im Spiel dargestellt wird. Dafür wird ein Spritesheet verwendet, dessen einzelne Sprites in der Größe identisch mit der Kachelgröße der Tilemap sind. Die meisten Tile-Engines bieten noch weitere Features, wodurch die Tilemaps noch mächtiger werden. Beispielsweise ist es heute nahezu Standard, dass Tilemaps „Layer“ unterstützen. Durch den Aufbau aus verschiedenen Ebenen können so Tiles miteinander kombiniert werden. Dies geschieht sowohl grafisch - indem die Kacheln hintereinander Dargestellt werden - als auch auf Ebene der Eigenschaften. Oft wird eine Eigenschaftsebene angelegt (sie ist später im Spiel für gewöhnlich unsichtbar), auf der Tiles platziert werden die jeweils eine Eigenschaft bzw. eine Kombination aus Eigenschaften repräsentieren. Indem man die Eigenschaften von der Grafischen Ebene löst, bietet man dem Designer der Level einen größeren Grad an Freiheit. Sinnvoll ist das jedoch bloß, wenn Kacheln mit derselben Grafik nicht zwangsweise auch dieselben Eigenschaften haben sollen.

Ein weiteres Feature, das nicht immer vorhanden ist, sind Objekte. Geometrische For-

men, die unabhängig vom Kachelmuster sind erlauben es dem Entwickler Punkte und Bereiche in der Tilemap festzulegen, die abseits des regelmäßigen Rasters liegen. Genau wie Tiles, können auch sie mit Eigenschaften versehen werden.

### 2.8.3 Tilemaps in cocos2d

Cocos2d unterstützt ein einziges Tilemap-Format: TMX (Tile Map XML). In diesem Format kann eine Tilemap mit beliebig vielen Ebenen gespeichert werden. Jede Ebene kann dabei nur ein Tileset verwenden, das gleiche Tileset kann jedoch auf verschiedenen Ebenen genutzt werden. TMX unterstützt die beiden häufigsten Formen von Tiles: Rechtecke und Hexagone. Zusätzlich zu den Kachelebenen können auch Objektebenen erstellt werden. Hier können die zuvor erwähnten Objekte platziert werden, um auch unabhängig vom festgelegten Raster arbeiten zu können (Tiles können hier jedoch nicht platziert werden). Dateien im TMX Format werden mit dem Tiled Mapeditor erstellt. In der Online Dokumentation des Editors findet man auch eine umfangreiche Erklärung über den Aufbau des TMX-Formats [[TiledDocFormat](#)].

Cocos2d liefert durch die Klassen des Tilemap Moduls Funktionen, um den Umgang mit Tilemaps im Programmcode einfacher zu gestalten. Im Folgenden soll kurz gezeigt werden, wie man eine Tilemap im Spiel anzeigt und sich Informationen über diese beziehen kann bzw. wie man diese bearbeitet.

Um eine Tilemap anzuzeigen muss lediglich ein Objekt vom Typ *cocos2d::TMXTiledMap* erstellt werden und als Child in der aktuellen Szene angefügt werden. In unserem Code ist der Prozess über verschiedene Funktionen verteilt, weshalb hier nur ein einfaches allgemeines Beispiel gezeigt wird:

```
TMXTiledMap map = create("Path/Map.tmx");
currentScene.addChild(map);
```

Um einzelne Ebenen in Form von TMXLayer Objekten abzufragen nutzt man folgende Funktion:

```
TMXLayer* ebene = map.getLayer("NameDerEbene");
```

Ähnlich erhält man ein Tile Objekt mit:

```
Sprite* kachel = ebene->tileAt(Vec2(x, y));
```

X und y sind hier die Koordinaten des Felds auf der Ebene. Um zu bestimmen welches Tile an einer Position liegt (und nicht bloß den Sprite zu erhalten) benutzt man:

```
unsigned int gid = ebene->getTileGIDAt(Vec2(x,y));
```

## 2 Grundlagen

Anhand der GID kann man auch zur Laufzeit ein Tile an einer beliebigen Position platzieren bzw. ersetzen. Dabei erhält die Kachel selbstverständlich auch alle Eigenschaften der gewählten GID

```
ebene->setTileGID(gid, Vec2(x,y));
```

Die Attribute eines Feldes erhält man ebenfalls über einen simplen Funktionsaufruf:

```
Value werte = map.getPropertiesForGID(gid);  
ValueMap eigenschaften = werte.asValueMap();
```

Nun kann man die Eigenschaften über die vergebenen Namen aus der ValueMap ermitteln. Oft ist dies jedoch nicht nötig, da man eventuell bereits weiß, welche GID über welche Eigenschaften verfügt.

Cocos2d stellt noch hunderte weitere Funktionen bereit um Tilemaps zu handhaben, jedoch reichen die hier genannten aus, um mit den Kacheln der Tilemap zu arbeiten. Objekte als Elemente der Tilemaps werden in diesem Projekt nicht verwendet, jedoch ist auch der Zugriff auf sie vollständig durch Getter und Setter abgedeckt.

## 2.9 Musik und Sound-Effekte Daniel Mügge

Jegliche Musik und Soundeffekte die in unserem Spiel **Josie—A Jelly's Journey** zu hören sind wurden selbst geschrieben, aufgenommen und bearbeitet. Dazu gehören die **Hintergrundmusik** im Hauptmenü, in der Levelauswahl, in den Jump and Run Levels und im Boss Kampf. Außerdem **Soundeffekte** für das Springen, Schrumpfen, Stoppen und Schießen, sowie das kaufen von Aufwertungen und das Treffen des Endgegners.

### 2.9.1 Möglichkeiten der cocos2d-x-Engine zur Audioverarbeitung

Cocos2d-x bietet mit der *cocos2d::SimpleAudioEngine* eine relative einfache Möglichkeit Audiodateien, sei es die Hintergrundmusik oder ein Sound-Effekt, zu laden, abzuspielen, zu pausieren und wieder zu entfernen/entladen. Hierzu ein kurzes Beispiel wie man auf einfache Art und Weise eine Audiodatei abspielt.

```
SimpleAudioEngine::getInstance()->playBackgroundMusic("song.mp3",true);
```

Auf die Implementierung und die Verwendung der *cocos2d::SimpleAudioEngine* innerhalb unseres Codes wird im Kapitel 4.3 genauer eingegangen. Vorweg sei gesagt dass wir alle Funktionalitäten welche die *cocos2d::SimpleAudioEngine* betreffen in eine eigene statische Klasse *AudioUnit* ausgelagert haben.

### 2.9.2 Monofonie/Stereofonie und verwendete Dateiformate

Es ist möglich sowohl monofone als auch stereofone Audiodateien zu verwenden. Falls man möchte dass Sounds zum Beispiel aus bestimmten Richtungen kommen, um dem Spieler ein gewisses “Mittendrin-Gefühl“ zu vermitteln, sollten die Audiodateien stereofon sein. Das ist allerdings erst richtig sinnvoll wenn das Spiel mit Kopfhörern oder mit Anschluss an ein Soundsystem gespielt wird.

Im Fall von **Josie—A Jelly’s Journey** wurden stereofone Audiodateien ohne oder geringem Panning (mischen von einzelnen Instrumenten/Klängen nach links oder rechts) verwendet, da das Spiel hauptsächlich für mobile Geräte gedacht ist und diese meist nur über einen Lautsprecher verfügen. Zudem werden in der Realität Spiele auf mobilen Geräten meist ohne Ton oder Kopfhörer gespielt, sodass wir deshalb keinen speziellen Wert auf räumlichen Klang gelegt haben.

Es wurde ausschließlich “mp3“ als Format für Audiodateien verwendet, da dieses Dateiformat in Bezug auf cocos2d-x von den meisten Geräten unterstützt wird. Ein weiterer Vorteil von “mp3“ gegenüber anderen Formaten wie “wav“ ist die durch die verschiedenen Abtastraten entstehende Dateigröße, worauf hier allerdings nicht im Detail eingegangen wird. Dieser Dateigrößenunterschied ist im Bezug auf mobile Geräte ein sehr wichtiger Faktor.

### 2.9.3 Audiobearbeitungsprogramme

Auf dem Softwaremarkt gibt es unzählige Audiobearbeitungsprogramme. Wenn man sich mit dem Thema Audiobearbeitung noch nie beschäftigt hat, ist es sehr schwer eines zu finden das die nötigen Funktionen liefert um einen guten Resultat zu erzielen. Zudem kosten die meisten guten Programme viel Geld. Im Folgenden soll eine Auflistung von einigen kostenpflichtigen und kostenlosen Programmen einen groben Überblick verschaffen. Bei den kostenpflichtigen Programmen ist die Preisspanne sehr groß, da es auch auf Version und Ausführung ankommt, deshalb sind an dieser Stelle auch keine Preise genannt.

- *Cubase* (Steinberg, kostenpflichtig)
- *Pro Tools* (Avid, kostenpflichtig)
- *Logic Pro* (Apple, kostenpflichtig)
- *Musik Maker* (MAGIX, kostenpflichtig)

## 2 Grundlagen

- *Goldwave* (Goldwave Inc., teilweise kostenlos)
- *Audacity* (AudacityTeam, kostenlos)

Bei **Josie—A Jelly’s Journey** wurde Logic Pro X von Apple verwendet.

# 3 Architektur

## 3.1 Spielstruktur Oleg Geier

Beim Start des Spiels wird das AppDelegate aufgerufen, was wiederum augenblicklich die *MainMenuScene* lädt. Dieser Bildschirm dient zum Einen (a) die *Optionen* aufzurufen, (b) ein kurzes *Tutorial* zur Erklärung der Steuerung und Hindernissen im Level, sowie (c) der eigentlichen Level Auswahl (*LevelSelectScene*). Die Level Auswahl unterscheidet grundsätzlich zwischen einem normalen *Level*, einem automatisch generierten (*TMXEdit*) und dem Boss Kampf (mit vorgesetzter *ShopScene*).

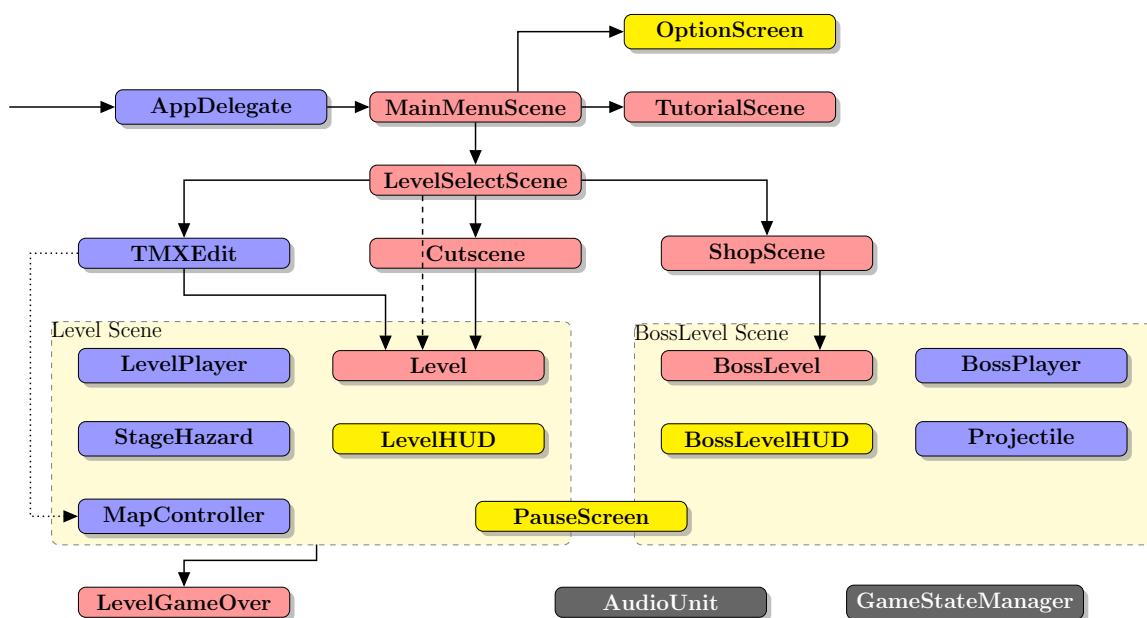


Abbildung 3.1: Aufruf und Abhängigkeiten der jeweiligen Screens

### Info zur Farbvergabe:

Rote Klassen stammen von der `cocos2d::Scene` Klasse ab

Gelbe Klassen sind `cocos2d::Layer` die über einer Scene eingeblendet werden

Blaue Klassen sind Objekte mit unterschiedlicher Basis-Klasse (siehe Kapitel 3.2)

Graue Objekte bezeichnen statische Klassen

Wird das Spiel zum Ersten Mal gespielt wird vor dem eigentlichen Level eine *Cutscene* geladen und abgespielt. Im späteren Verlauf wird das Level direkt geladen (gestrichelte Linie). Für das automatisch generierte "Random Level" ist die *TMXEdit* Klasse zuständig. Dabei wird der *MapController* mit der generierten Karte gefüllt und anschließend ein "normales" *Level* gestartet.

Beim Ende eines Levels wird das *LevelGameOver* angezeigt. Dabei spielt es keine Rolle ob das Level mit Erfolg absolviert wurde oder nicht. Die Übergabe erfolgt über einen Parameter bei der Instanz-Erstellung.

Es sei noch angemerkt, dass die beiden Klassen *AudioUnit* und *GameStateManager* nur statische Funktionen enthalten und somit nie eine Instanz gespeichert wird. Der Aufruf erfolgt an den entsprechenden Stellen. Auch der *PauseScreen* wird sowohl von der *Level* Scene, als auch vom *BossLevel* gleichermaßen benutzt und auf der jeweiligen HUD hinzugefügt. Die *LevelHUD* und *BossLevelHUD* steuern außerdem die Bewegungen des *LevelPlayer* bzw. *BossPlayer*.

Für einen detaillierteren Überblick ist im Anhang A ein Klassendiagramm angehängt.

## 3.2 Vererbung Oleg Geier

Wie bereits im Kapitel 3.1 erwähnt, stammen nicht alle Klassen von *cocos2d::Scene* bzw. *cocos2d::Layer* ab. Die Grafik 3.2 illustriert den Nutzen der *CollisionLayer* Klasse. Es wurde bewusst *cocos2d::LayerColor* gewählt um, für Debugging Zwecke, den Kollisions Rahmen anzeigen zu können (siehe Kapitel 4.4.1).

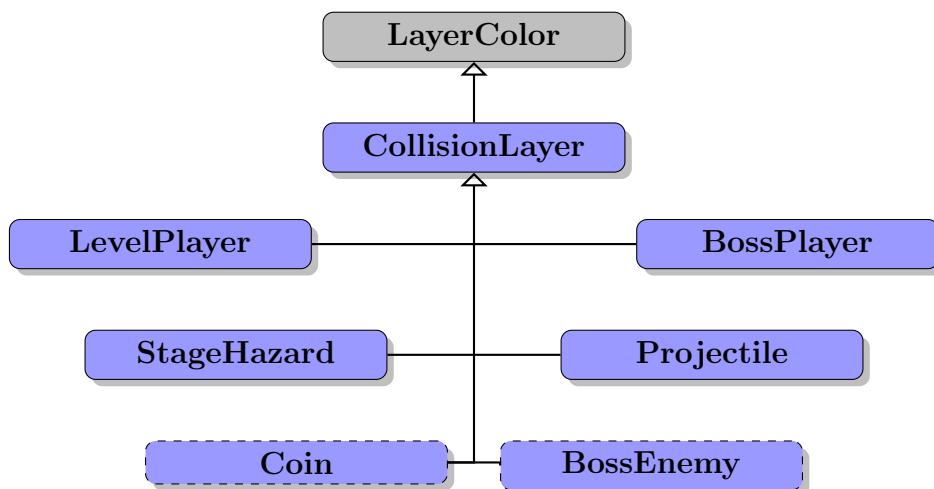
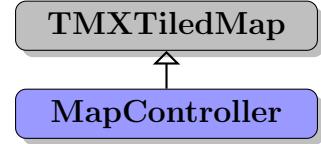


Abbildung 3.2: Vererbung der CollisionLayer Klasse

Die beiden Objekte *Coin* und *BossEnemy* werden direkt in der *CollisionLayer* Klasse bzw. im *BossLevel* erstellt und haben somit keine echte Klassenzugehörigkeit. Der Grund für diese Vererbung liegt auf der Hand, Objekte können unabhängig auf ihre Kollision hin überprüft werden. Die standard Funktionalität der *cocos2d::Rect* Klasse kann zwar eine Kollision mit **intersectsRect()** erkennen, dies funktioniert jedoch nicht mit rotierten Nodes wie es beim Boss Kampf der Fall ist. Hierfür wurde die 2D Oriented Bounding Box Intersection von Morgan McGuire [2DOBB] implementiert und für *cocos2d* umgeschrieben.

Der *MapController* erweitert die Funktionalität der *cocos2d* Klasse um die Erkennung der Kollision zum Boden, der Erkennung von tödlichen Objekten, sowie der Plazierung der Münzen im Level.



Die Klasse *TMXEdit* kommt ohne Eltern Klasse aus, das sie nur für das Generieren des Random Level zuständig ist. Sie holt sich dafür eine Instanz des MapControllers und erstellt zufällige Kartenelemente.

## 3.3 Speichersystem Oleg Geier

Bei der Speicherung des App Zustandes, also der Einstellungen und des Spielstandes, haben wir uns für die *cocos2d::UserDefaults* entschieden. Der Zugriff erfolgt einfach und es benötigt keiner speziellen zusätzlichen Klassen oder 3rd Party Libraries. Die Münzen und die benötigte Zeit für die Level werden codiert in zwei Strings gespeichert. Dabei gibt das Byte an der x. Stelle die Münzen/Zeit für das Level x wieder. Jede Dauer die darüber hinausgeht, wird mit der Maximalzeit von 255 Sekunden, also 4:15 Min gespeichert.

Die Hintergründe und Level Karten liegen einer bestimmten Struktur zugrunde. Wenn beispielsweise das Level 1.2 aufgerufen wird, so lädt das Level den Hintergrund „backgrounds/bg\_1.2.png“ und die Karte „tilemaps/1.2.tmx“.

# 4 Implementierung

## 4.1 Spieler Steuerung Oleg Geier

Die Spieler Steuerung wird mithilfe eines Observer Patterns realisiert. Beim Laden der *BossPlayer* Klasse wird der Spieler als Observer eingetragen:

Listing 4.1: BossPlayer als Observer eintragen ( BossPlayer.cpp )

```
103 EventDispatcher *ed = ←  
    Director::getInstance()->getEventDispatcher();  
104 if (reg) {  
105     ed->addCustomEventListener("BOSS_PLAYER_LEFT", ←  
        CC_CALLBACK_0(BossPlayer::moveLeft, this));
```

Die Methode **addCustomEventListener()** erwartet zwei Parameter. Den Namen auf den der Observer hören soll, und das Callback, also die Funktion die ausgeführt werden soll beim Eintreffen einer solchen Nachricht, in diesem Fall **moveLeft()**. Gleichbedeutend muss der Spieler auch wieder aus der Liste der Observer entfernt werden, sobald die Instanz gelöscht wird. Beides passiert über dieselbe Methode, die mit dem Parameter **false** die Einträge wieder entfernt.

Die Steuerung wird über das HUD bewerkstelligt. Um genauer zu sein in der **update()** Methode der *BossLevelHUD*.

Listing 4.2: Drücken des Laufen-Buttons ( BossLevelHUD.cpp )

```
181 void BossLevelHUD::update(float dt)  
182 {  
183     EventDispatcher *ed = ←  
    Director::getInstance()->getEventDispatcher();  
184     if (_key_left || _left->isSelected())  
185         ed->dispatchCustomEvent("BOSS_PLAYER_LEFT");
```

Die update Methode wird kontinuierlich aufgerufen, deshalb ist vor jedem Aufruf die Abfrage auf **isSelected()** ob der aktuelle Button gedrückt ist. Das Aktivieren des Observers ist denkbar einfach über **dispatchCustomEvent()**.

## 4.2 Sprites Daniel Glück

Die grundlegende Nutzung von Sprites wird im folgenden anhand der *Tutorial* erklärt. Es werden die Bereiche, die sehr viel Anwendung in unserem Projekt fanden, erklärt. Diese sind unter anderem: das Laden von Sprites, Verwendung des Ankerpunktes als auch die Verwendung grundsätzlicher Funktionen wie **moveBy()**.

Listing 4.3: Sprite laden ( TutorialScene.cpp )

```
54 Sprite* josie = Sprite::create("josie/josie_static.png");
55 josie->setAnchorPoint(Vec2::ANCHOR_BOTTOM_LEFT);
56 josie->setPosition(200,45);
57 this->addChild(josie);
```

Die erste Zeile erstellt ein Sprite Object und gibt in der create Methode den Pfad zum zu ladenden Sprite an. Als nächstes wird der Ankerpunkt des Sprites mit dem Befehl **setAnchorPoint()** festgelegt. Durch setPosition wird die Startposition festgelegt und das Sprite im Anschluss an die Scene hinzugefügt.

### 4.2.1 Bewegen

Im Anschluss soll ein Beispiel für die Verwendung von Grundfunktion zur Spritemanipulation als auch Sequenzen erläutert werden. Hierbei wird ein Beispiel aus der Klasse *BossPlayer* verwendet.

Listing 4.4: Sequence erstellen ( BossLevel.cpp )

```
219 auto right_rotate = RotateTo::create(2.0, 30.0f);
220 auto right_rotate_back = RotateTo::create(0.2, 0.0f);
221 auto right_down = MoveTo::create(0.2,Vec2(400,100));
222 auto right_up = MoveTo::create(1.0,Vec2(1520,600));
223 auto sequence = Sequence::create(right_rotate, right_down, ←
    right_up,right_rotate_back, nullptr);
224 right->runAction(sequence);
```

Im ersten Schritt werden zwei RotateTo Objekte als auch zwei MoveTo Objekte erstellt. Ein Objekt ist für eine Aktion zuständig die etwas verändert, das andere stellt den Ursprungszustand wieder her. Beim erstellen des RotateTo Objektes werden die Zeitdauer der Rotierung und der Winkel mitgegeben. Dem MoveTo Objekt wird bei der Erzeugung die Zeitdauer der Aktion als auch ein Vector der die Zielkoordinaten beschreibt übergeben. Im Anschluss wird eine Sequenz mit dem gewünschten Ablauf

## 4 Implementierung

der Animation erstellt und danach ausgeführt. Folgendes, Bildhaftes Beispiel soll dies veranschaulichen.

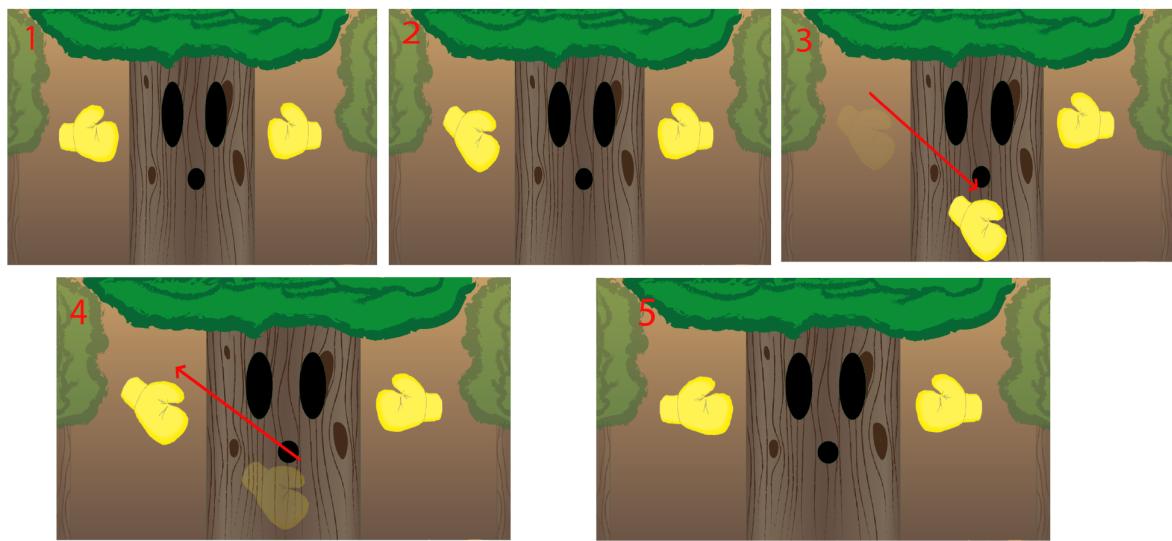


Abbildung 4.1: Ablauf einer Sequenz

### 4.2.2 Animieren

Eine Erklärung für den Aufbau einer Animation durch ein Spritesheet und eine .plist Datei soll durch das folgende Beispiel erbracht werden.

Listing 4.5: Animation erstellen ( CollisionLayer.cpp )

```
53 Vector<SpriteFrame*> frames;
54 SpriteFrameCache *frameCache = SpriteFrameCache::getInstance();
55
56 char file[9] = { 0 };
57
58 for (int i = 0; i < 6; i++) {
59     sprintf(file, "coin%04d", i);
60     SpriteFrame *frame = frameCache->getSpriteFrameByName(file);
61     frames.pushBack(frame);
62 }
63
64 Animation *animation = Animation::createWithSpriteFrames(frames, ←
65     0.1);
```

Zu Beginn erstellen wir einen String mit Hilfe eines Char Arrays und einem Vector zur

Erstellung von Frames. In der SpriteFrameCache Instanz sind bereits alle Spritesheets geladen. Die nachfolgende Schleife iteriert durch die benötigten Bildnamen um diese zum Array hinzuzufügen. Hierbei ist zu beachten dass Für eine Animation mehrere Bilder nacheinander geladen werden müssen, diese haben die Namenskonvention "coin0000", "coin0001" etc. Der SpriteFrame mit dem generierten Bildnamen wird aus dem SpriteFrameCache geladen und in den anfangs definierten Vektor eingefügt. Sobald die For-Schleife abgearbeitet ist, wird eine Animation mithilfe **createWithSpriteFrames()** erstellt. Dieser wird der Vektor und eine Verzögerung zwischen den Frames übergeben.

Listing 4.6: Sequence Umkehren ( LevelPlayer.cpp )

```
96 startRunningAfterAnimation(animationWithFrame("josiejump", 6, ←
    0.0001)->reverse())
```

Die Methode animationWithFrame liefert eine Animation zurück. Darauf wird die Methode **reverse()** aufgerufen, welche die Animation rückwärts ablaufen lässt.

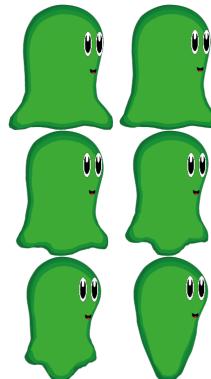


Abbildung 4.2: Die Sprung-Animation in einzelnen Sprites

### 4.2.3 Fortsetzung des Programmablaufs Oleg Geier

Manchmal ist es sinnvoll eine Funktion auszuführen sobald eine Animation beendet ist. Hierfür wird ein Callback erstellt und in eine *cocos2d::CallFuncN* gekapselt.

Listing 4.7: startRunningAfterAnimation ( LevelPlayer.cpp )

```
116 CallFuncN *call = CallFuncN::create(
117     CC_CALLBACK_0(LevelPlayer::startRunningCallback, this));
118 Sequence *seq = Sequence::createWithTwoActions(animation, call);
119 spriteImage->runAction(seq);
```

In diesem Beispiel wird die Animation für das Stoppen von Josie und anschließend die Methode **startRunningCallback()** ausgeführt. 'this' ist die aktuelle Player Instanz.

## 4.3 AudioUnit Daniel Mügge

Die Klasse *AudioUnit* kümmert sich um das Laden, Abspielen, Pausieren und Entfernen der Audiodateien. Hierbei handelt es sich durchgehend um statische Funktionen, sodass man die Klasse nicht erst instanziieren muss, sondern die Funktionen einfach von Außerhalb mit *AudioUnit::* aufrufen kann.

### 4.3.1 Laden von Soundeffekten

Ein wichtiger zu beachtender Aspekt bei Spielen mit vielen Soundeffekten ist die Notwendigkeit des vorangehenden Ladens dieser. Falls man dies nicht tut, kann es zu Performance-Problemen kommen. Der Grund dafür ist, dass zum Beispiel beim Drücken des “Sprung”-Buttons jedesmal beim Ausführen, der Sound erst geladen, abgespielt und anschließend wieder aus dem Speicher entfernt werden würde.

Deshalb wird zum Beispiel beim Erstellen des *BossLevel* im Konstruktor die Funktion *AudioUnit::preloadBossSounds()* aufgerufen.

Listing 4.8: BossLevel-Sounds laden ( *AudioUnit.cpp* )

```

30 void AudioUnit::preloadBossSounds()
31 {
32     SimpleAudioEngine* engine = SimpleAudioEngine::getInstance();
33     engine->setEffectsVolume(UserDefault::getInstance()
34                             ->getIntegerForKey("sfx_volume")/200.0);
35     engine->preloadEffect("audio/boss_sounds/boss_hit1.mp3");
36     //Weitere Preloads
37 }
```

An dieser Stelle sei erwähnt dass ganze Musiktitel, wie zum Beispiel die Hintergrundmusik, nicht zwingend geladen werden müssen da diese nicht öfter hintereinander abgespielt, sondern wenn nötig geloopt werden.

Die Methode *unloadBossSounds* gleicht der Methode zum Laden der Sounds. Der einzige Unterschied ist, dass **unloadEffect** anstelle von **preloadEffect** verwendet wird.

Gleichzeitig mit dem Laden der Sounds wird die Lautstärke der Effekte auf den in *cocos2d::UserDefault* gespeicherten integer-Wert mit dem Key “sfx\_volume” gesetzt. Dieser wird durch einen individuellen double-Wert (in diesem Fall 200.0) geteilt, da die **setEffectsVolume** Methode nur double-Werte akzeptiert. Die Möglichkeit Einstellungen für Musik- und SFX-Lautstärke im Spiel selbst vorzunehmen, ist durch die Klasse *OptionScreen* realisiert.

Innerhalb der *AudioUnit* wird auf die Singleton-Instanz der *cocos2d::SimpleAudioEngine*, die alle nötigen Funktionen liefert, zugegriffen. So gesehen ist die *AudioUnit* ein Wrapper der die Verwendung von Audio im Code der anderen Klassen erheblich erleichtert und zusätzlich zu “saubererem“ Code führt.

### 4.3.2 Abspielen von Soundeffekten und Musik

Um nun einen Sound-Effekt abzuspielen, ruft man an passender Stelle die gewünschte Methode auf. Als Beispiel ist das Abspielen des Sounds gegeben, den man hört wenn der *BossPlayer* getroffen wird.

```
AudioUnit::playJosieHitSound();
```

Die Logik hinter der Funktion ist relativ simpel. Es existieren drei verschiedene Sound-Effekte die mit Hilfe eines String-Ersetzers und einer Zufallszahl zwischen eins und drei zufällig ausgewählt und abgespielt werden. Die übergebenen Parameter an die Methode **playEffect** sind der Pfad der Audiodatei, Loop, Pitch, Pan, Gain.

Listing 4.9: BossLevel Shoot Sound abspielen ( *AudioUnit.cpp* )

```
30 void AudioUnit::playJosieHitSound()
31 {
32     std::ostringstream s;
33     s << "audio/josie_sounds/josie_hit" << (rand()%3)+1 << ".mp3";
34
35     SimpleAudioEngine* engine = SimpleAudioEngine::getInstance();
36     engine->playEffect(s.str().c_str(), false, 1.0, 1.0, 0.7);
37
38 }
```

Analog kann das ganze auf das Abspielen der Hintergrundmusik übertragen werden, wobei dann auf den String-Ersetzer verzichtet und der Pfad hard decoded wird. Außerdem wird die Zufallsfunktion überflüssig da es die Hintergrundsongs nur einmal gibt und die Parameter Pitch,Pan und Gain fallen weg.

## 4.4 CollisionLayer Oleg Geier

Das *CollisionLayer* kümmert sich ausschließlich um die Kollision von Objekten. Die Kollision zum Boden wird im nächsten Kapitel erläutert. Alle Interaktiven Elemente im

## 4 Implementierung

Spiel besitzen Kollision, dazu zählt der Spieler, Boss Gegner, Münzen, Geschoss Kugeln und herabfallende Stachelbeeren.

### 4.4.1 Debugging Optionen

Zu Debugging Zwecken kann die *CollisionLayer* Klasse den Bereich der Kollision grafisch hervorheben.

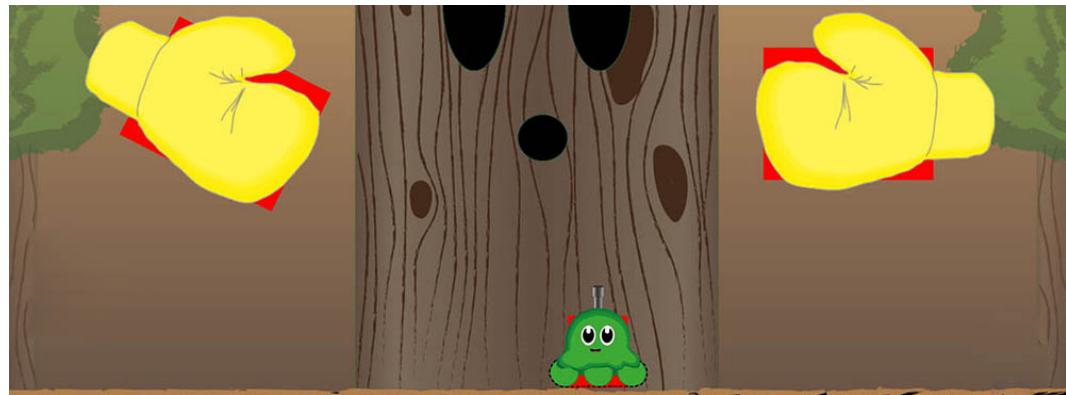


Abbildung 4.3: CollisionLayer Debug im Boss Kampf

Wenn man genau hinsieht erkennt man, dass auch Münzen über eine Kollision verfügen. Tödliche Objekte in der Karte (bsp. Dornen) jedoch nicht.

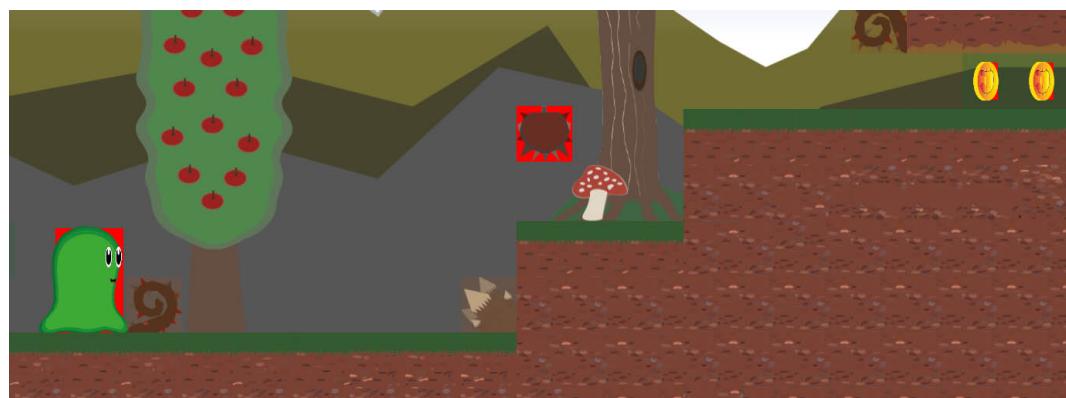


Abbildung 4.4: CollisionLayer Debug im Level

#### 4.4.2 Listener registrieren

Die Klasse verfügt über eine Funktion **setCollisionListener(CollisionLayer\*)** die ein anderes Collision Layer als Parameter erwartet. Dabei wird das übergebene Objekt in einer internen Variable gespeichert und in der **update()** Methode kontinuierlich auf Kollision überprüft.

#### 4.4.3 Gegenseite Collision Notification

Sobald eine Kollision festgestellt wird, werden beide Objekte darüber informiert. Die Methode **hitByCollision(CollisionLayer\*)** ist in der *CollisionLayer* Klasse nicht implementiert und muss von den einzelnen Subklassen durch Logik ergänzt werden.

So wird bei einem *StageHazard* - im Falle einer Collision mit dem Spieler - das tödliche Objekt wieder auf Anfang positioniert.

Listing 4.10: Collision Notification ( StageHazard.cpp )

```
32 void StageHazard::hitByCollision(CollisionLayer* other)
33 {
34     if (other->collisionType == CollisionLayerTypeLevelPlayer) {
35         this->fallDown();
36     }
37 }
```

### 4.5 Kollisionsabfrage zum Boden Oleg Geier

Die Kollisionsabfrage ist auf den ersten Blick nicht sofort einleuchtend. Prinzipiell wird für die komplette Karte ein Array mit ganzzahligen Werten angelegt, also für jede Spalte (72px breite, vertikale Linie auf dem Bildschirm) wird ein **long** Wert gespeichert. Die Karte ist 15 Tiles hoch. Für jedes Tile wird ein Bitwert gesetzt ob Kollision besteht.

Listing 4.11: Collision Column abfragen ( MapController.cpp )

```
271 long MapController::getColumnBitmapForGID(int x, int tile_gid)
272 {
273     TMXLayer *meta = getLayer("Meta_layer");
274     long col=0;
275     for (int i=_mapSize.height; i>0; i--) {
276         col<<=1;
```

```

277     int gid = meta->getTileGIDAt(Vec2(x,i-1));
278     col |= (gid==tile_gid);
279 }
280 return col;
281 }
```

Die Schleife durchläuft - von unten angefangen - alle Tiles einer Spalte und fragt ab, ob das Kollisions Attribut gesetzt ist. Bei jedem Schleifendurchlauf wird der Bit-Shift-Operator (`<<`) angewandt, sodass das höher liegende Tile hinten angefügt wird. Das Anfügen geschieht mit dem Oder-Operator und der gleichzeitigen Zuweisung (`|=`).

Der abschließende `long` Wert weißt an dem höchstwertigen Bit die Kollision für das unterste Teil auf und am niedrig wertigsten Bit die Kollision für das Tile am oberen Bildschirmrand.

Dieselbe Bitmap wird auch für tödliche Kollision in einem separaten Array erstellt. Beides geschieht nur beim Laden der Karte. Für die tatsächliche Kollisionsabfrage wird nur noch auf diese Bitmap zugegriffen.

## 4.6 Automatisch erzeugte Tilemaps Jonas Kaiser

Die `TMXEdit` Klasse hat die Aufgabe, abhängig von einem übergebenen Schwierigkeitsgrad, automatisch einen Level zu generieren. Drei Aspekte waren dabei besonders wichtig: Ein erzeugter Level solltefordernd sein, er sollte Abwechslungsreich sein und er sollte – unabhängig davon, wie die Zufallsparameter ausfallen – keine Stellen beinhalten, die unmöglich zu bewältigen sind. Zusätzlich war es wünschenswert, dass der Spieler alle Münzen, welche im Level platziert sind erreichen kann.

### 4.6.1 Generieren des Levels

Das Grundgerüst des Zufallslevels ist immer identisch. Die ersten 20 Tiles stellen nie ein Hindernis dar, die letzten 38 sind das Levelende, welches in jedem Level gleich aussieht. Aus diesem Grund laden alle automatisch erzeugten Level die gleiche Tilemap, welche diesen Aufbau bereits umsetzt, und befüllen diese dann zur Laufzeit. Zum Befüllen des Levels wird dieses von Anfang bis Ende durchlaufen. Dabei werden in zufälliger Reihenfolge Funktionen aufgerufen, welche Abschnitte des Levels nach einem jeweils eigenen Muster erzeugen. Das tatsächliche platzieren der Tiles erfolgt durch das zuweisen der GID für die zu bearbeitenden Tilemap-Koordinaten. Dir folgenden Funktionen beispiele-

## 4 Implementierung

weise füllen eine ausgewählte Spalte ab einer gewissen Höhe nach unten komplett mit Boden-Tiles und einem Gras-Tile an oberster Stelle.

Listing 4.12: Platzieren von Boden-Tiles ( TMXEdit.cpp )

```
127 void TMXEdit::placeGround(int x, int y) {  
  
129     _backgroundLayer->setTileGID(TOPDIRT[arc4random() % 4], ←  
130         Vec2(x, y));  
131     _metaLayer->setTileGID(COLLIDE, Vec2(x, y));  
132     if (y < _minheight)  
133         placeDirt(x, y + 1);  
134 }  
  
135 void TMXEdit::placeDirt(int x, int y) {  
136     _backgroundLayer->setTileGID(DIRT[arc4random() % 4], Vec2(x, ←  
137         y));  
138     _metaLayer->setTileGID(COLLIDE, Vec2(x, y));  
139     if (y < _minheight)  
140         placeDirt(x, y + 1);  
141 }  
  
142 }
```

Ein einfaches Beispiel um zu erklären, wie Abschnitte nach einem gewissen Muster erzeugt werden findet man in der folgenden Funktion, welche im Level eine oder mehrere aufeinander folgende schwebenden Plattformen erzeugt. Wie alle Funktionen die gesamte Abschnitte erzeugen, verfügt sie über einen ganzzahligen Rückgabewert. Dieser gibt an, bis zu welcher Spalte die Funktion die Tilemap bearbeitet hat.

Listing 4.13: Abschnitt mit schwebenden Plattformen erzeugen( TMXEdit.cpp )

```
171 int TMXEdit::makeFloating(int x, int height) {  
172     x += _minJumpDistance + arc4random() % _maxJumpDistance-1;  
173     int done = x + _partlength * (1 + arc4random() % 2);  
174     while (x <= done) {  
175         x = FloatingPlatform(x, height, 3+arc4random()%3);  
176         x += _minJumpDistance + arc4random() % _maxJumpDistance;  
177     }  
178     return x;  
179 }
```

Zu Beginn der Funktion wird eine gewisse Anzahl an Spalten übersprungen. Die so entstehende Lücke wird in ihrer Weite durch den Schwierigkeitsgrad definiert (dieser beeinflusst die variablen `_minJumpDistance` und `_maxJumpDistance`). Anschließend wird

bestimmt, wie lang der aktuelle Abschnitt sein soll. Die Variable „done“ legt fest, bis zu welcher x-Koordinate ein neues Element(hier: eine einzelne schwebende Plattform) dieses Abschnitts angefangen werden kann. Die Plattform selbst wird hier durch eine andere Funktion erzeugt. Diese weißt der Reihe nach den Kacheln auf der übergebenen Höhe die jeweiligen GIDs zu. Die Länge der Plattform ist ebenfalls ein Parameter, welcher hier durch einen Zufallswert geliefert wird. Die Wahl der einzelnen Levelbausteine erfolgt in Form eines Switch-Statements. Der übergebene Ausdruck ist dabei selbstverständlich ein Zufallswert, durch ihn wird bestimmt, welcher Abschnitt als nächstes generiert werden soll. Durch das wiederholen dieses switch Befehls in einer Schleife kann so der gesamte Level zufällig aufgefüllt werden.

#### 4.6.2 Platzieren der Münzen

Neben dem Erstellen des Levels gibt es noch ein zweites Problem das bewältigt werden musste. Die Münzen mussten im Level verteilt werden – in erreichbaren Positionen, abhängig vom Aufbau des Levels. Dafür zuständig ist folgende Funktion (sowie die von ihr aufgerufenen):

Listing 4.14: Automatisches Platzieren von Münzen( TMXEdit.cpp )

```

171 void TMXEdit::placeCoins(int number) {
172     int counter = 0;
173     int distance = (int) map->getMapSize().width/number;
174     int x = 50;
175     while (counter < number) {
176         int y = isSafe(x);
177         if (y > 0) {
178             placeCoin(x, y);
179             counter++;
180             x = x - (x % distance) + distance;
181             if(x > map->getMapSize().width -50) x= 50;
182         } else {
183             x += 1;
184         }
185     }
186 }
```

Die Funktion stellt sicher, dass die gewünschte Anzahl an Münzen in regelmäßigen Abständen zueinander platziert wird. Durch die isSafe Funktion wird zuerst die Höhe der Umgebung an der betrachteten Position ermittelt und dann geprüft, ob die Position für den Spieler erreichbar ist, ohne zu verlieren. Ist dies nicht der Fall, so wird Spalte für Spalte überprüft, bis die Münze platziert wurde. Sollte das Ende der Karte

#### *4 Implementierung*

erreicht werden, so wird die nächste Münze wieder am Anfang platziert. Ein Problem tritt dabei höchstens auf, wenn die gewünschte Anzahl an Münzen die Anzahl an sicheren Positionen übertrifft. Das Spiel würde dann eine Endlosschleife betreten. Da eine so hohe Anzahl an Münzen jedoch ohnehin nicht wünschenswert ist- auf manchen Geräten kann dies zu Einbrüchen der Framerate führen- darf dieser Sonderfall ignoriert werden (Außerdem wäre es aus Sicht eines Game-Designers keine gute Idee, so viele Münzen verfügbar zu machen, da so das Erfolgserlebnis beim Einsammeln verloren geht).

# 5 Evaluierung Daniel Mügge

Im Großen und Ganzen betrachtet ist **Josie—A Jelly's Journey**, umgangssprachlich ausgedrückt, ein rundes Ding. Es gibt einige Dinge die sehr gut glaufen sind, einige Punkte auf die wir nicht unser Hauptaugenmerk gelegt haben und einige die wir einfach vergessen haben. Das kann vielleicht daran liegen das uns viele nützliche Features erst zu Ende des Projektes aufgefallen sind und uns schlicht und ergreifend die Zeit davon gelaufen ist. Zudem wurde JIRA eher schlecht als recht zum managen der Vorgänge und Sprints eingesetzt, wohingegen Git ein nützliches und kontinuierlich verwendetes Tool war.

Die Portierung des Spiels auf andere Plattformen außer Android, wie iOS und OS X, hat bis auf einige kleine Hürden problemlos funktioniert. Speziell für die plattformübergreifende Entwicklung ist cocos2d-x äußerst praktisch, weil die Engine den C++-Code selbstständig nativ übersetzt. Das größte Problem war die unterschiedliche, erlaubte Pixelbreite von Bildern, was aber relativ schnell behoben war.

Auf die integrierte Physiks Engine von cocos2d-x haben wir verzichtet da unser Spiel nur von der Kollision zum Boden Gebrauch macht. Eine normale Physik Engine bezieht dafür noch viel mehr Parameter (bsp. Masse, Momentum, Gravitation) mit ein, was bei uns nicht gegeben ist oder mit einem unnötig hohen Mehraufwand verbunden.

Hinsichtlich der Performance haben wir uns auch wenig Gedanken gemacht. Hier bleibt noch offen Tests anzulegen um bestimmte Kriterien einzuhalten. Oder selbst einfache Tests zwischen zwei Funktionen durchzuführen. Beispielsweise ob es effektiver ist viele kleine Bilder nach Bedarf zu laden oder ein großes Bild ständig im Speicher zu halten. Des weiteren wäre zu klären ob die implementierte Kollisionsabfrage effektiv ist.

Die recht langen Ladezeit beim Starten eines Levels und besonders beim Starten des Zufallslevels sollte mit einem Lade-Bildschirm überdeckt werden, damit für den Endanwender ersichtlich ist, dass etwas im Gange ist.

Als Verbesserung des Ablaufs/Aufbaus der Animationen hätte man die Spawn Klasse von cocos2d verwenden können. Dadurch wäre eine variablere Verwendung von Sequenzen und Aktionen möglich. Diese hätte besonders bei den Angriffspattern der Boss Klasse Anwendung gefunden. Dadurch ist die komplette Boss Klasse momentan auf einen einzigen Endgegner fixiert.

## 5 Evaluierung

Im Hinblick auf die Schwierigkeit und die Balance des Spiels war es für uns als Entwickler besonders schwer das Spiel neutral zu betrachten. Wenn dann haben wir es selbst getestet und nach einer gewissen Zeit weiß man eben wie die Levels aussehen und wie die Steuerung einzusetzen ist. An dieser Stelle hätte man gezielte Langzeittests mit Unbeteiligten (sozusagen den Endanwendern) durchführen sollen. Wenn wir es Außenstehenden vorgestellt haben, dann nur kurz nebenbei um zu zeigen wie es aussieht. Jedoch war bei den kurzen Tests schon zu sehen, dass unser Produkt ein relativ hohes, nennen wir es an dieser Stelle, "Suchtpotential" hat. Alle Tester **wollten** die Levels schaffen und waren aufgrund von schwierigeren Passagen nicht gleich demotiviert.

Im Bezug auf den Schwierigkeitsgrad ist der Vorteil der "festen" Levels gegenüber dem zufällig generiertem Level, dass man diese einfach über den Editor, an Stellen die zu schwer sind, entschärfen kann. Jedoch muss man erwähnen das die Umsetzung des Zufallslevels sehr gut gelungen ist und außerdem wird man natürlich je nach Schwierigkeitsgrad auch mit einer entsprechenden Anzahl an Münzen belohnt(1: 10, 2: 20, 3: 80).



Abbildung 5.1: Schwierige Passage bei Schwierigkeitsgrad 1

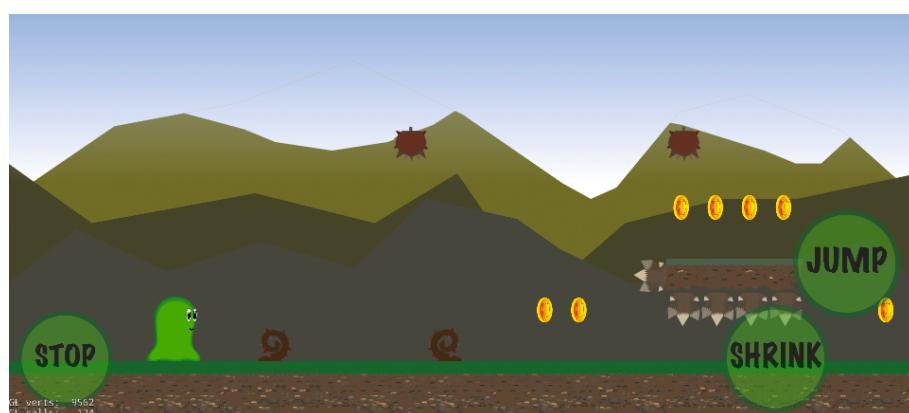


Abbildung 5.2: Schwierige Passage bei Schwierigkeitsgrad 3

Die Implementierung des Sprungs bei **Josie - A Jelly's Journey** ist für Erstanwen-

## 5 Evaluierung

der möglicherweise etwas gewöhnungsbedürftig, da es keine Mindestsprunghöhe gibt. Es wäre möglich diese zu implementieren jedoch haben wir während der Entwicklung festgestellt, dass das Spiel dadurch eine gewisse Individualität gegenüber anderen Spielen auf dem Markt erhält. Die restlichen Funktionalitäten der Steuerung weisen keine großen Besonderheiten auf und sind selbsterklärend, was sich weder positiv noch negativ auswirkt.

Ein Highlight des Spiels sind die Grafiken. Auch wenn es sich nicht hochkarätige 3D-Grafiken, sondern einfache 2D-Grafiken handelt, harmonieren sie mit dem Gesamtkonzept des Spiels. Sie wurden mit sehr viel Liebe zum Detail gestaltet und strahlen einen gewissen “niedlichen“ Charme aus. Diese Niedlichkeit ist genau das was wir bei **Josie – A Jelly’s Journey** haben wollten. Ein gutes Beispiel dafür ist der Shop, in den man gelangt wenn man das Bosslevel startet.



Abbildung 5.3: Shop zum Erwerb von Aufwertungen für das Bosslevel

# 6 Fazit und Ausblick

Kritisch betrachtet lässt sich sagen, dass unser Spiel ein vollständig funktionierendes 2D-Sidescroll-Game darstellt. Mit komplett individuellem Audio, Animations und Grafikdesign sowie eigener Steuerung.

## 6.1 Features for the Future Daniel Mügge

Aus zeitlichen Gründen konnten wir einige Features nicht umsetzen, dazu gehören:

**Mehr Level** Die momentane Levelstruktur 1.1–1.2–1.3–Boss könnte in die vertikale Ebene erweitert werden. In Zukunft wären mehrere Levelebenen (2.1–2.2–2.3–Boss, etc) wünschenswert. Durch eine höhere Anzahl von Levels ist auch eine interessante Story mit Cutscenes und neuen Umgebungen besser umsetzbar.

**Erweiterung der Levels** Eine Bereicherung des Spiels wären außerdem neue Mechaniken und Gegner im Jump and Run Level. Mit Mechaniken sind zum Beispiel Plattformen gemeint auf denen man stehen bleiben muss um von Punkt A nach Punkt B zu gelangen. Zusätzlich dazu könnte man Gegner einführen, bei deren Berührung der Spieler stirbt. Sozusagen sich bewegende, tödliche Hindernisse.

**Mehr Endgegner** Neue Bosse mit neuen Angriffspatterns und Bewegungsabläufen, neuen Designs und interessanteren Mechaniken wären eine weitere große Ergänzung die das Spiel noch besser machen würde. Hier war es angedacht die BossLevel Klasse so umzuschreiben, damit man neue Gegner durch Subklassen erstellen kann und nur noch die Grafiken und Angriffspattern implementiert muss.

**Neue Aufwertungsmöglichkeiten** Die bestehenden Aufwertungen im Shop könnte man beispielsweise mit Element-Projektilen erweitern, die über verschiedene Schadensarten verfügen und in Abhängigkeit zum Element des Boss Gegners mehr oder weniger

## *6 Fazit und Ausblick*

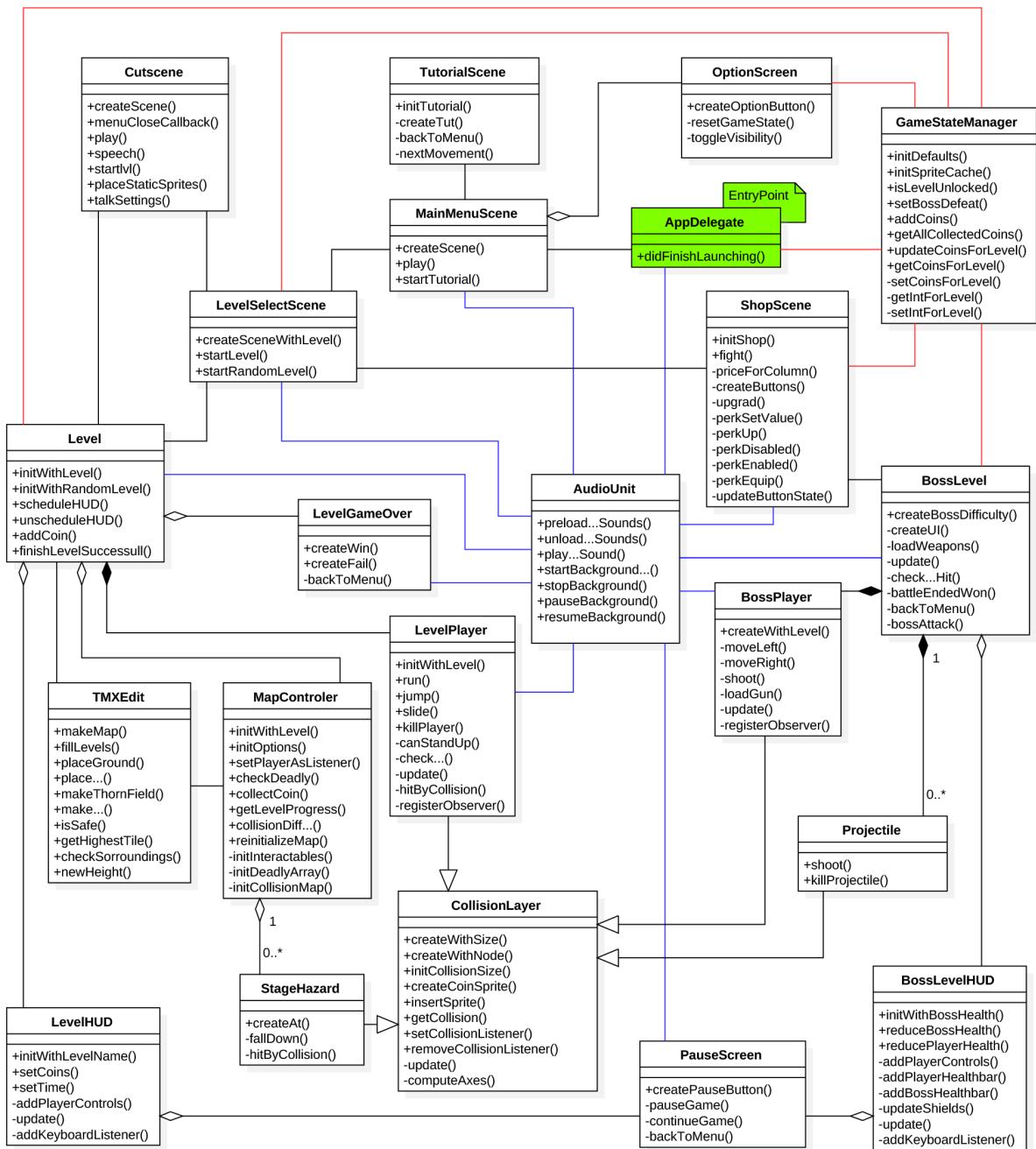
Schaden verursachen. Hierfür wären auch neue Projektil-Grafiken erforderlich was das Spiel fürs Auge interessanter machen würde.

**Lade Bildschirm** Das Initialisieren eines Levels soll im Hintergrund geschen und dem Anwender ein Lade Bildschirm angezeigt. Das steigert die gefühlte Ausführungszeit und dem Benutzer wird immer etwas geboten.

**Double-Jump-Gliding** Die Erweiterung der Sprungfunktion könnte nach einem weiteren Klick in der Luft dazu führen, dass Josie ein kleines Stück in der Luft gleitet. Das hätte zur Folge dass man auch Levelabschnitte mit größeren Sprungabständen einbauen könnte.

**Neue Kampfmodi** Damit ist gemeint, dass Josie sich in einen Hubschrauber verwandeln und den Boss von oben bekämpfen kann oder in einen Mech/Roboter, der anstatt einer Links-Rechts-Bewegung lediglich einen Sprung ausführt und den Boss von der Seite bekämpft.

# 7 Anhang A Oleg Geier, Daniel Mügge



# Abbildungsverzeichnis

2.1	Ordnerstruktur . . . . .	5
2.2	Szenengraph . . . . .	7
2.3	Plazierung von Josie bei verschiedenen Ankerpunkten (rot) . . . . .	9
2.4	Josie nach dem Skalieren . . . . .	9
2.5	Unterschied zwischen MoveBy und MoveTo . . . . .	10
2.6	Beispiel für Fade In/Out bei Josie . . . . .	11
2.7	Beispiel für Sequenzen . . . . .	11
2.8	Observer Patter aus [gamma2011patterns] . . . . .	13
3.1	Aufruf und Abhängigkeiten der jeweiligen Screens . . . . .	19
3.2	Vererbung der CollisionLayer Klasse . . . . .	20
4.1	Ablauf einer Sequenz . . . . .	24
4.2	Die Sprung-Animation in einzelnen Sprites . . . . .	25
4.3	CollisionLayer Debug im Boss Kampf . . . . .	28
4.4	CollisionLayer Debug im Level . . . . .	28
4.5	Schwierige Passage bei Schwierigkeitsgrad 1 . . . . .	35
4.6	Schwierige Passage bei Schwierigkeitsgrad 3 . . . . .	35
4.7	Shop zum Erwerb von Aufwertungen für das Bosslevel . . . . .	36

# Listings

4.1	BossPlayer als Observer eintragen ( BossPlayer.cpp ) . . . . .	22
4.2	Drücken des Laufen-Buttons ( BossLevelHUD.cpp ) . . . . .	22
4.3	Sprite laden ( TutorialScene.cpp ) . . . . .	23
4.4	Sequence erstellen ( BossLevel.cpp ) . . . . .	23
4.5	Animation erstellen ( CollisionLayer.cpp ) . . . . .	24
4.6	Sequence Umkehren ( LevelPlayer.cpp ) . . . . .	25
4.7	startRunningAfterAnimation ( LevelPlayer.cpp ) . . . . .	25
4.8	BossLevel-Sounds laden ( AudioUnit.cpp ) . . . . .	26
4.9	BossLevel Shoot Sound abspielen ( AudioUnit.cpp ) . . . . .	27
4.10	Collision Notification ( StageHazard.cpp ) . . . . .	29
4.11	Collision Column abfragen ( MapController.cpp ) . . . . .	29
4.12	Platzieren von Boden-Tiles ( TMXEdit.cpp ) . . . . .	31
4.13	Abschnitt mit schwebenden Plattformen erzeugen( TMXEdit.cpp ) . . . . .	31
4.14	Automatisches Platzieren von Münzen( TMXEdit.cpp ) . . . . .	32