

Hochschule für angewandte Wissenschaften Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

Projektdokumentation

Entwicklung eines Spiels 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

1 Einleitung

1.1 Ausgangssituation

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 Bitcom 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 Spielegenres beschäftigt. Strategie- und Denkspiele sind laut Bitcom am beliebtesten, gefolgt von Gelegenheitsspielen, Actionspielen, Social Games, Jump n' Runs und Renn- und Sportspielen.

1.2 Motivation

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 getauft.

1.3 Vorgehen

Am Anfang war das Nichts.

Eine der schwierigsten Phasen in unserem Projektverlauf war das grobe Design. Wir wollten dass JOSIE jedem aus unserer Gruppe gefällt und jeder seine Ideen einbringen kann.

Nachdem wir in etwa wussten welche Komponenten wir benötigen, haben wir die Aufgabenbereiche auf die Team-Mitglieder wie folgt verteilt.

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

1.4 Dokumentationsstruktur

2 Grundlagen

2.1 Framework

Cocos2d-3.4 Engine

Für unser Projekt haben wir die cocos2d-Engine verwendet, da sie am meisten Möglichkeiten bietet und flexibel ist.

2.2 IDE und Plugins

2.3 Szenenprinzip

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 X.X). Sobald das Szenen-Objekt geladen wird, werden die angefügten Kinder mitgeladen. Dies spart Zeit und entlastet den Speicher.

TODO: Szenengraph, zeigt Szene mit „Kindern“

2.3.1 Szenenfunktionen von Cocos2d-x

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

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

Dies erstellt ein Objekt des Typ Szene mit dem Namen Cutszene. 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.

2.4 Spriteprinzip

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 rührt 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-Spielen wie Jump'n Runs so genannte Tiles oder auch Kachelgrafiken, welche ebenfalls kleine Grafikelemente die zusammengesetzt eine größere Grafik ergeben sind, verwendet. Ihr Anwendungsbereich findet sich unter anderem im Aufbau eines Level's, wobei aus ihnen die Spielwelt zusammengesetzt wird.

2.4.1 Cocos2dx's Spriteprinzip

In der von uns verwendeten Gameengine Cocos2dx, 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. Bezüglich Dateiformaten werden von Cocos2dx 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 Erste ist es ein Sprite aus einem Bild zu laden, wobei das in cocos2dx 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 dafür benutzen Bild. 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 viele einzelne Bilder, was die Ladezeiten erheblich verringert und zudem eine Speicherreduktion mit sich bringt. Beim erstmaligen verwenden eines Spritesheets, wird dieses in den SpriteFrameCache geladen. Dies ist eine Klasse, welche ein SpriteFrame-Objekt, für zukünftigen Schnellzugriff 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(verlinkung Animationen) und die Beschreibung der Spielwelt eine optimale Lösung dar.

2.4.2 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 in der unteren Linken Ecke startet.

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

Weitere Möglichkeiten den Ankerpunkt zu setzen werden hier veranschaulicht. Rote Punkte::Ankerpunkte

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. Im folgenden wird dies Bildlich dargestellt.

```
mySprite->setScale(2.0);
```

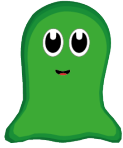


Abbildung 2.1: Das ist Josie

2.5 Callbackprinzip

In vielen Teilen des Spieles wird **CC_CALLBACK_0()** verwendet. Es handelt sich dabei um eine Referenz, die auf eine Methode einer Instanz verweist. Somit kann man eine Methode oder Funktion dynamisch im Spiel ausführen, wie das Drücken eines Buttons.

```
CC_CALLBACK_0(Director::popScene, Director::getInstance());
```

Der Director (**Director::getInstance()**) soll die Funktion **popScene()** ausführen. Zur Unterscheidung ist es wichtig die Klasse (**Director::**) zu nennen die ausgeführt werden soll. Denkbar wäre hier eine Kind-Klasse die den selben Methoden Namen aufweist. Weitere Parameter der Funktion können Komma getrennt hinten angehängt werden.

2.6 Tilemaps

2.7 Musik und Sound-Effekte

Musik sowie alle Sounds die in unserem Spiel “Josie“ zu hören sind wurden selbst geschrieben, aufgenommen und bearbeitet. Dazu gehören:

- Hintergrundmusik im Hauptmenü, in der Levelauswahl, in den Jump and Run Levels und im Boss Kampf
- Effektsounds für Sprung-, Schrumpf-, Stop und Schuss-Sounds von Josie, Shop-Sound, Bosstreffer-Sound

2.7.1 Möglichkeiten der cocos2d-Engine zur Audioverarbeitung

Cocos2d bietet mit der **SimpleAudioEngine** eine relative einfache Möglichkeit Audio-dateien, sei es die Hintergrundmusik oder ein Sound-Effekte, zu laden, abzuspielen, zu pausieren und wieder zu entfernen. Hierzu ein kurzes Beispiel wie man auf einfache Art und Weise eine Audiodatei abspielt.

```
SimpleAudioEngine::getInstance()--\textgreater; ←
    playBackgroundMusic("'song.mp3"',true);
```

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

2.7.2 Mono-/Stereo-Kanäle und Dateiformate

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

In unserem Fall wurden Stereo-Audiodateien ohne Panning (mischen von Spuren nach links oder rechts) verwendet, da **Josie-A Jelly's Journey** hauptsächlich für mobile Geräte gedacht ist und diese meist nur über einen Lautsprecher verfügen. In der Realität ist es außerdem meistens so, dass man bei Handyspielen den Ton ausschaltet bzw. ohne Kopfhörer spielt.

Wir haben ausschließlich .mp3 verwendet, da dieses Dateiformat in Bezug auf cocos2d von den meisten Geräten unterstützt wird. Ein weiterer Vorteil von .mp3 gegenüber anderen Formaten wie .wav ist die Dateigröße, was im Bezug auf mobile Geräte ein sehr wichtiger Faktor ist.

2.7.3 Audibearbeitungsprogramme

Auf dem Softwaremarkt gibt es unzählige Audibearbeitungsprogramme. Wenn man sich mit dem Thema Audibearbeitung noch nie beschäftigt hat, ist es sehr schwer eines zu finden das die nötigen Funktionen liefert um einen gutes Resultat zu erzielen.

2 Grundlagen

Zudem kosten die meisten guten Programme viel Geld. Im Folgenden soll eine Auflistung von einigen kostenpflichtigen und kostenlosen Programmen einen groben Überblick verschaffen:

- Cubase (Steinberg, kostenpflichtig)
- Pro Tools (Avid, kostenpflichtig)
- Logic Pro (Apple, kostenpflichtig)
- Audacity (AudacityTeam, kostenlos)
- Goldwave (Goldwave Inc., teilweise kostenlos)

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

3 Architektur

3.1 Spielstruktur

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 platziert 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.

3.2 Klassenübersicht

Beim Start des Spieles 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 vorgeschalteter *ShopScene*).

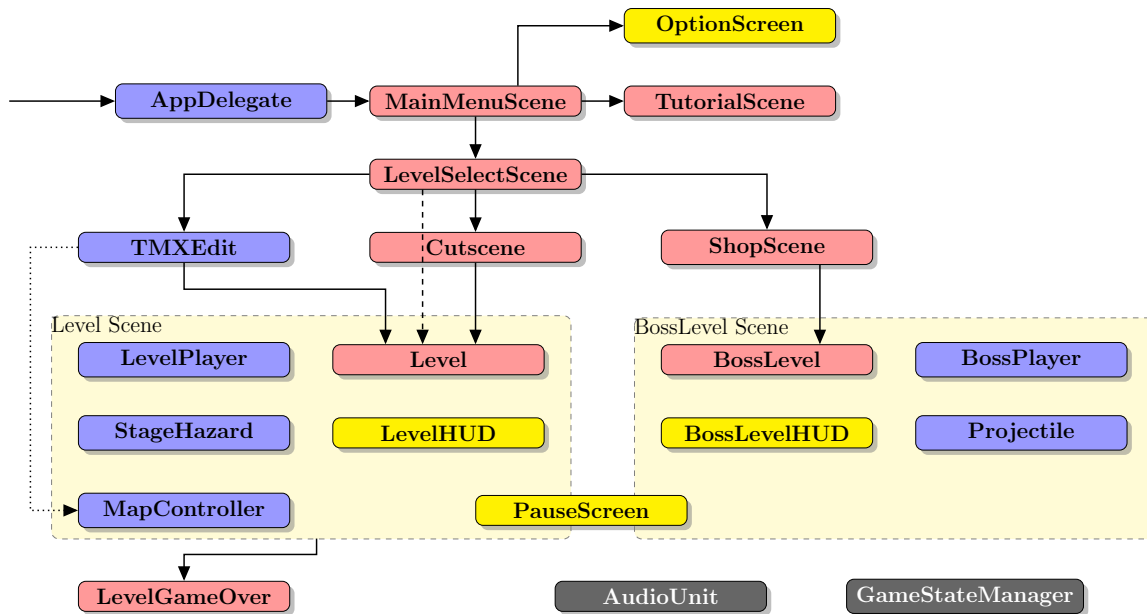


Abbildung 3.1: Aufruf und Abhängigkeiten der jeweiligen Screens

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 ??)

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 Levelist 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*.

3.3 Vererbung

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

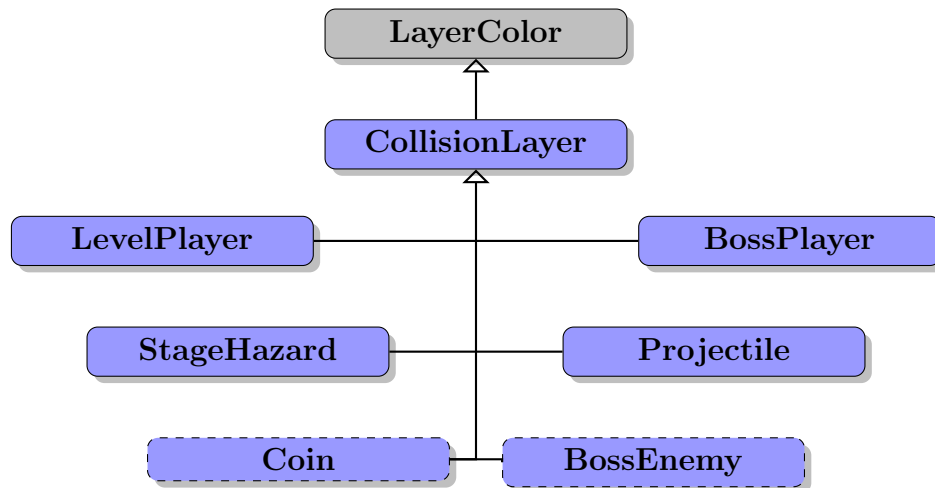
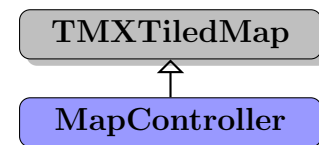


Abbildung 3.2: Vererbung der CollisionLayer Klasse

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 *cocos* 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.

3.4 Speichersystem

Bei der Speicherung des App Zustandes, also der Einstellungen und des Spielstandes, haben wir uns für die *cocos2d::UserDefault* 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

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 AudioUnit

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 instanziiieren muss, sondern die Funktionen einfach von Außerhalb aufrufen kann.

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

Listing 4.3: 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     engine->preloadEffect("audio/josie_sounds/josie_hit3.mp3");
38 }
```

Wir greifen innerhalb der *AudioUnit* auf die Singleton-Instanz der **SimpleAudioEngine**, die alle nötigen Funktionen liefert, zu. So gesehen ist unsere *AudioUnit* nur ein Wrapper um im Code der anderen Klassen nicht ständig **SimpleAudioEngine::getInstance()** aufrufen zu müssen, was zu saubererem Code führt.

Hierbei sei erwähnt dass ganze Musiktitel, wie die Hintergrundmusik, nicht zwingend geladen werden müssen, da diese nicht öfter hintereinander abgespielt werden.

Abspielen von Soundeffekten und Musik: Um nun

4.3 Kollisionsabfrage zum Boden

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 4.4: 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.

4.4 CollisionLayer

4.4.1 Debugging Optionen

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

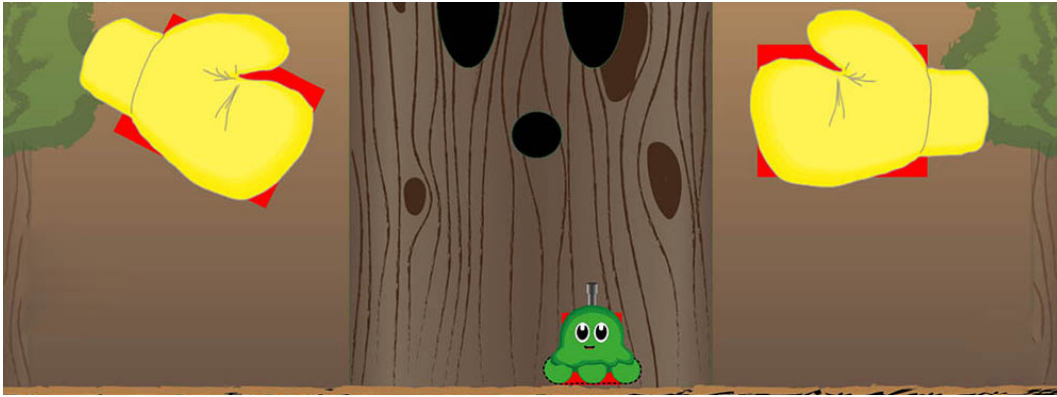


Abbildung 4.1: CollisionLayer Debug im Boss Kampf

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

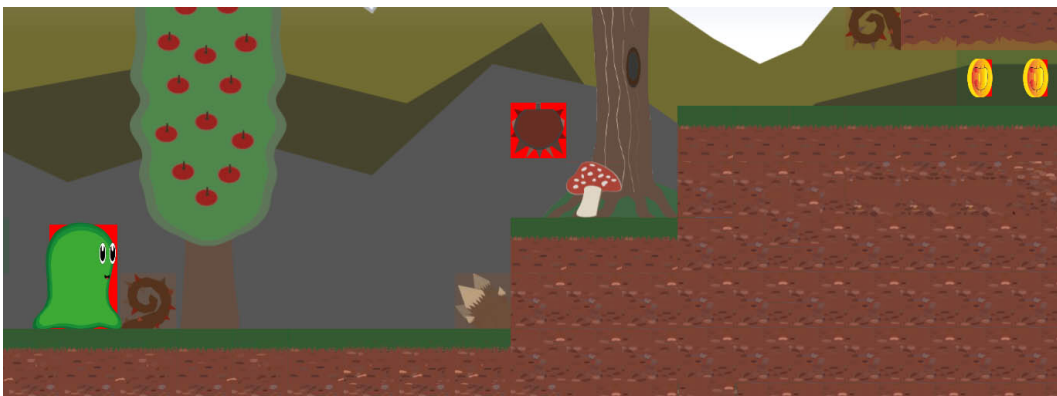


Abbildung 4.2: CollisionLayer Debug im Level

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 im-

plementiert 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 4.5: Collision Notification (StageHazard.cpp)

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

5 Evaluierung

6 Fazit und Ausblick

6.1 Features for the Future

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

- Mehr Levels

Die momentane Levelstruktur 1.1–1.2–1.3–Boss könnte in die vertikale Ebene erweitert werden. In Zukunft wären mehrere Levelbenen wünschenswert. Damit ist gemeint: 2.1–2.2 ... 3.1–3.2

- Neue Kampfmodi mit neuen Bossen

Mit neuen Kampfmodi ist gemeint dass Josie sich zum Beispiel anstatt in einen Panzer, in einen Hubschrauber verwandelt und den Boss von oben bekämpft. Eine weitere Möglichkeit ist die Verwandlung in einen Mech, der anstatt einer Links-Rechts-Bewegung lediglich einen Sprung ausführt.

Neue Bosse mit neuen Angriffspatterns, neuen Designs und interessanteren Mechaniken wären eine weitere große Ergänzung die das Spiel noch besser machen würde.

- 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.

7 Anhang A

Abbildungsverzeichnis

Tabellenverzeichnis