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**
- 1.1 Ausgangssituation . . . 1
- 1.2 Motivation . . . 1
- 1.3 Vorgehen . . . 1
- 1.4 Dokumentationsstruktur . . . 2

2 Grundlagen **3**
- 2.1 Framework . . . 3
- 2.2 IDE und Plugins . . . 3
- 2.3 Was sind Sprites? . . . 3
- 2.4 Was ist eine TileMap? . . . 3

3 Architektur **4**
- 3.1 Klassenübersicht . . . 4
- 3.2 Vererbung . . . 5
- 3.3 Speichersystem . . . 6

4 Implementierung **7**
- 4.1 Spieler Steuerung . . . 7
- 4.2 Placeholder . . . 8

5 Evaluierung **9**

6 Fazit und Ausblick **10**
- 6.1 Features for the Future . . . 10

7 Anhang A **11**

Verzeichnisse **12**

Literatur **14**

1 Einleitung

1.1 Ausgangssituation

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 weiß auf schwarz und ein wenig Textausgabe, das wars.

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 verteilt.

- Oleg Geier: Programmierung

- Daniel Glück: Grafikdesign
- Jonas Kaiser: Spieldesign
- Tobias Lediger: Storydesign
- Daniel Mügge: Audiodesign

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 Was sind Sprites?

2.4 Was ist eine TileMap?

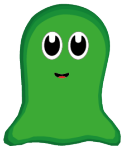


Abbildung 2.1: Das ist Josie

3 Architektur

3.1 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 unterscheided grundsätzlich zwischen einem normalen *Level*, einem automatisch generierten (*TMXEdit*) und dem Boss Kampf (mit vorgeschalteter *ShopScene*).

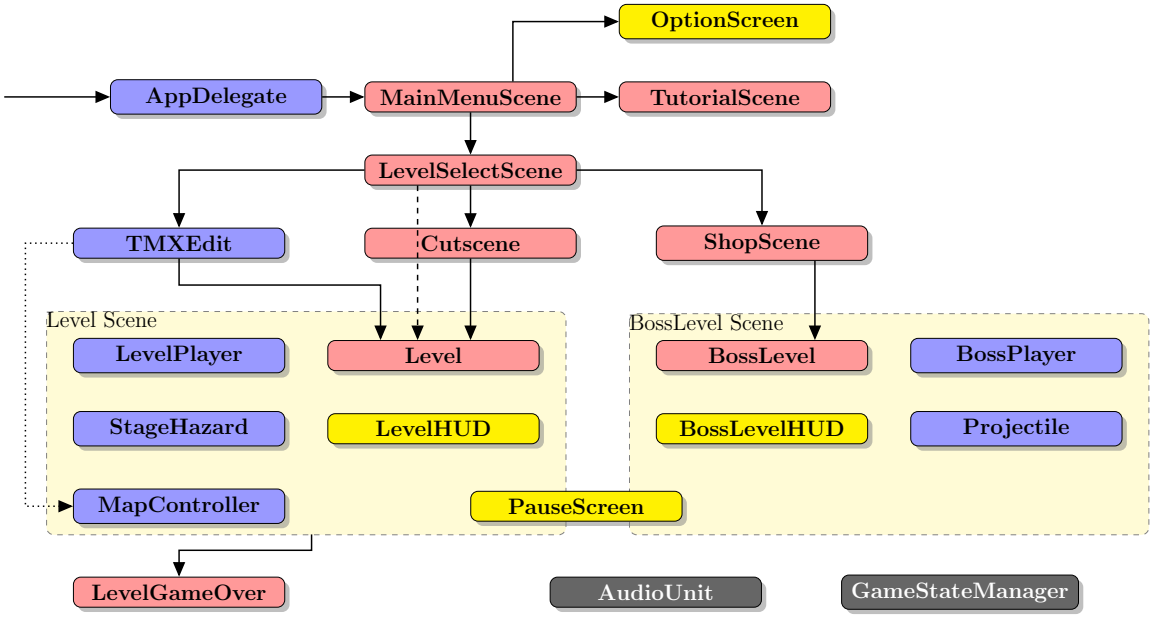


Abbildung 3.1: Aufruf und Abhängigkeiten der jeweiligen Screens

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 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 Levelïst 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*.

3.2 Vererbung

Wie bereits im Vorherigen Kapitel 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.

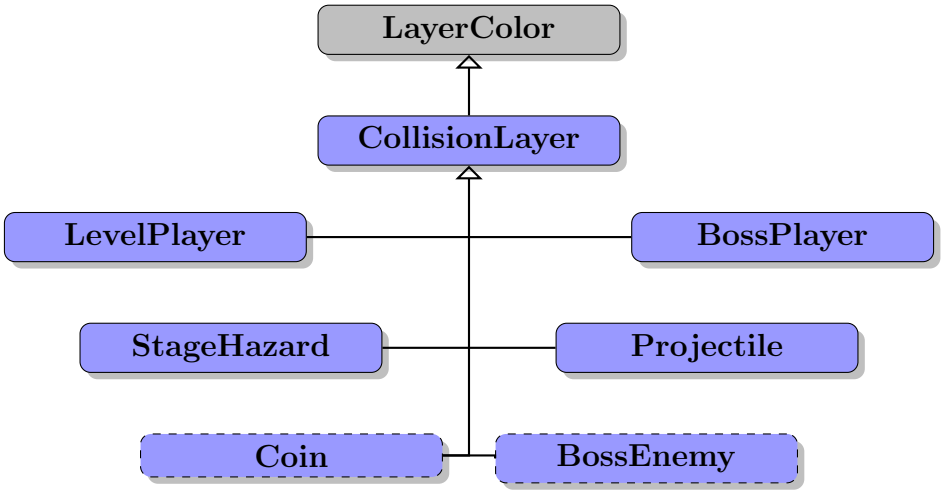
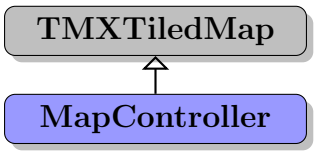


Abbildung 3.2: Vererbung der CollisionLayer Klasse

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 erkennen (*-intersectsRect()*), 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 [1] 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.

3.3 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 Placeholder

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 Levelebenen 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 weitern 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

2.1 Das ist Josie 3

3.1 Aufruf und Abhängigkeiten der jeweiligen Screens 4
3.2 Vererbung der CollisionLayer Klasse 5

Tabellenverzeichnis

Literatur

[1] Morgan McGuire. *2D OBB Intersection*. Brown University Department of Computer Science. Apr. 2003. URL: http://www.flipcode.com/archives/2D_OBB_Intersection.shtml.