Sophie-Hedwig-Gymnasium Diez

BLL im Fach Informatik

Entwicklung von 60-Sekunden, ein Jump and Run-Spiel mit LibGDX

mit Hilfe der Box2D Bibliothek und Tiled

Verfasser	Florian Sprenger
Betreuende Lehrkraft	Frau Wolf
Datum der Abgabe	18.09.2020

Inhaltsverzeichnis

1 Einleitung

1.1 Aufbau der Arbeit

Diese Arbeit beschäftigt sich primär mit der Box2D Library innerhalb des LibGDX Frameworks. Hierfür fange ich jedoch erst an die grundlegende Struktur des Projektes, da dies nötig ist um die Abläufe verstehen zu können zu erklären und spezialisiere mich dann im folgenden Teil (Kapitel 3) auf die physikalische Berechnung und den eigentlichen Box2D Teil.

Zum besseren Verständnis sind im Folgenden wichtige Begriffe in blau geschrieben. Klassen und Methoden die von LibGDX stammen sind in rot geschrieben und eigene Klassen, Objekte und Methoden sind in grün geschrieben.

Da der Code mehrere tausend Zeilen umfasst und damit zu groß ist um ihn hier anzuhängen, befindet sich in den Quellen ein Link zum Source Code, welcher auf GitHub hochgeladen wurde. Dort befinden sich auch ausführbare Dateien für Android (jumpandrun.apk) und Windows (jumpandrun.jar).

1.2 Anforderungen an das Spiel

Ziel dieser Arbeit ist es, ein Jump and Run zu entwickeln auf Basis der Box2D Library innerhalb des LibGDX Frameworks. Dieses soll aus unterschiedlichsten Leveln bestehen, welche sowohl statische Elemente wie normale Wege oder zerstörbare Blöcke beinhalten als auch dynamische Elemente wie sich bewegende Plattformen oder Gegner.

Das Code soll möglichst modular gehalten werden. Dazu gehört, dass man ihn ohne großen Aufwand erweitern kann und man nicht für zum Beispiel nur einen neuen Gegner das bestehende (Gegner-) System überarbeiten oder komplett neue Mechaniken implementieren muss. Erweiterungen wie neue Items oder Gegner sollen auf der Basis bestehender Systeme und (Ober-)Klassen einfach implementierbar sein.

Zudem soll das Spiel plattformübergreifend spielbar sein, dies umfasst die Betriebssysteme Android, iOS und Windows. Hierzu muss man besonders auf die Skalierbarkeit achten, da es vor allem viele Handys und Handymarken mit verschiedenen Auflösungen gibt. Des variieren die Eingabemedien. Hier gibt es sowohl Maus und Tastatur als auch den Touchscreen am Handy.

1.3 Die Wahl des Frameworks / der Engine

Um ein Projekt mit den oben beschrieben Anforderungen als einzelne Person in einem angemessenen Zeitraum realisieren zu können, brauch man eine Engine oder ein Framework zur Hilfe. Meine Wahl ist hier auf das LibGDX Framework gefallen, da es ein vorgefertigtes System bietet um Level in einer XML-Datei zu speichern oder zu laden. Des Weiteren kann man die Bibliothek Box2D mit LibGDX nutzen, welche eine einfache Physikberechnung ermöglicht. Mit LibGDX kann man ein Spiel einfach, ohne großen Aufwand von Android zu iOS oder zu Windows übersetzen, was exakt den Anforderungen entspricht.

Ich hatte vor Beginn der Arbeit auch alternativen wie Unity in Betracht gezogen, gegen die ich mich letztendlich jedoch vor allem wegen der Programmiersprache entschieden habe (Unity unterstützt kein Java). Außerdem ist LibGDX leistungsstärker.

2 Die Spielidee

Der Spieler steuert einen Dino, der sich durch nach XML-Dateien erstellte Level aufgeteilt auf mehrere Welten kämpfen muss (diese Arbeit umfasst ausschließlich die Grasfelder, also die erste Welt). Hierbei muss er grüne oder rote Kristalle einsammeln, um seine Leben zu erhöhen. Wenn er einen Gegner berührt, runterfällt oder ihm die Zeit ausgeht verliert er ein Leben.

Jede Welt besitzt normale statische Böden, sowie unzerstörbare statische Blöcke. Zudem besitzt jede Welt „Fragezeichen-Blöcke“, gegen die der Spieler springen kann um Booster zu erhalten. Diese Booster umfassen in der ersten Welt Leben sowie 1-50 Coins die aus dem Block kommen können. Damit der Spieler Leben verlieren kann, laufen Gegner in der Welt rum. Hier besitzt jede Welt drei verschiedene Typen. In der ersten sind das die Schleime, die fliegende Drachen und die Grashüpfer. Um die Level abwechslungsreicher zu machen enthält auch jede Welt zwei verschiedene Arten von kinematischen (Objekte die sich bewegen lassen, aber nicht von Kräften wie der Schwerkraft beeinflusst werden) Objekten. In den Grasfeldern gibt es zum einem sich horizontal bewegende Plattformen und zum anderen Plattformen, die abstürzen, sobald sie berührt werden. Um jede Welt etwas einzigartig zu machen, kann sie bis zu 2 besondere Elemente enthalten. In der ersten Welt befindet sich nur eins, das Trampolin.

Nachdem man ein Level abgeschlossen hat, kann man einen bis drei Sterne für dieses bekommen Diese Sterne sind ein rein optisches Element im Level-Auswahlbildschirm und haben keinen spielerischen Einfluss. Das Spiel heißt 60 Sekunden, da man nur solange Zeit hat um ein Level abzuschließen mit 3 Sternen, da dies jedoch sehr schwer werden kann, gibt es auch einen einfacheren Modus, für den man aber weniger Sterne bekommt.

3 Struktur

3.1 Struktur LibGDX

Ein LibGDX Projekt besteht immer aus einem Core Modul, welches den eigentlichen Code des Spiels beinhaltet. Jede Plattform, in meinem Fall Android, iOS, Windows, hat ebenfalls ein eigenes Modul, in dem plattformspezifische Eigenschaften wie Fenstername (Windows) oder Bildschirmdrehung (Android) festgelegt werden. Am Ende wird in jedem Plattform Modul die Hauptklasse des Core Moduls aufgerufen (siehe 3.3 Zyklus des Spiels).

Jedes Plattform Modul besitzt einen Asset Ordner, dieser beinhaltet alle Sounds, Texturen und Level-Dateien, die in dem Spiel verwendet werden. Damit kann man verschiedene Versionen des Spiels später exportieren. zum Beispiel kann man für Windows Texturen mit einer höheren Auflösung verwenden, da Computer in der Regel leistungsstärker und einen größeren Bildschirm besitzen sind als Handys. In meinem Fall verwende ich jedoch ausschließlich den Asset Ordner von Android. Auch um das Projekt später für Windows zu exportieren wird der Asset Ordner aus dem Android-Modul verwendet, da es nur eine Version der Texturen gibt und keine weitere höher auflösende Version.

3.2 Struktur des Spiels

Innerhalb des Core Moduls besitzt das Spiel eine Hauptklasse (JumpAndRun.java), welche von Game erbt und dementsprechend die onCreate() Methode enthält, die aufgerufen wird sobald das Spiel gestartet wird. In dieser Methode werden nur Objekte initialisiert, die sehr viel Grafikspeicher benötigen und nur einmal innerhalb des Spiels existieren sollten jedoch innerhalb des gesamten Spiels benötigt werden. Diese sind ein SpriteBatch, der Texturen zusammenfasst um sie effizienter zeichnen zu können (weswegen er sehr viel Grafikspeicher belegt), ein Assetmanager, mit dem Texturen vorgeladen werden können (mehr dazu unter Optimierung) und jeweils ein Objekt der eigenen Klasse PrefManager.java und LevelStarProgressSafer.java welche dazu dienen Werte wie die Anzahl an Leben permanent speichern zu können, was sehr viel Arbeitsspeicher kosten kann.

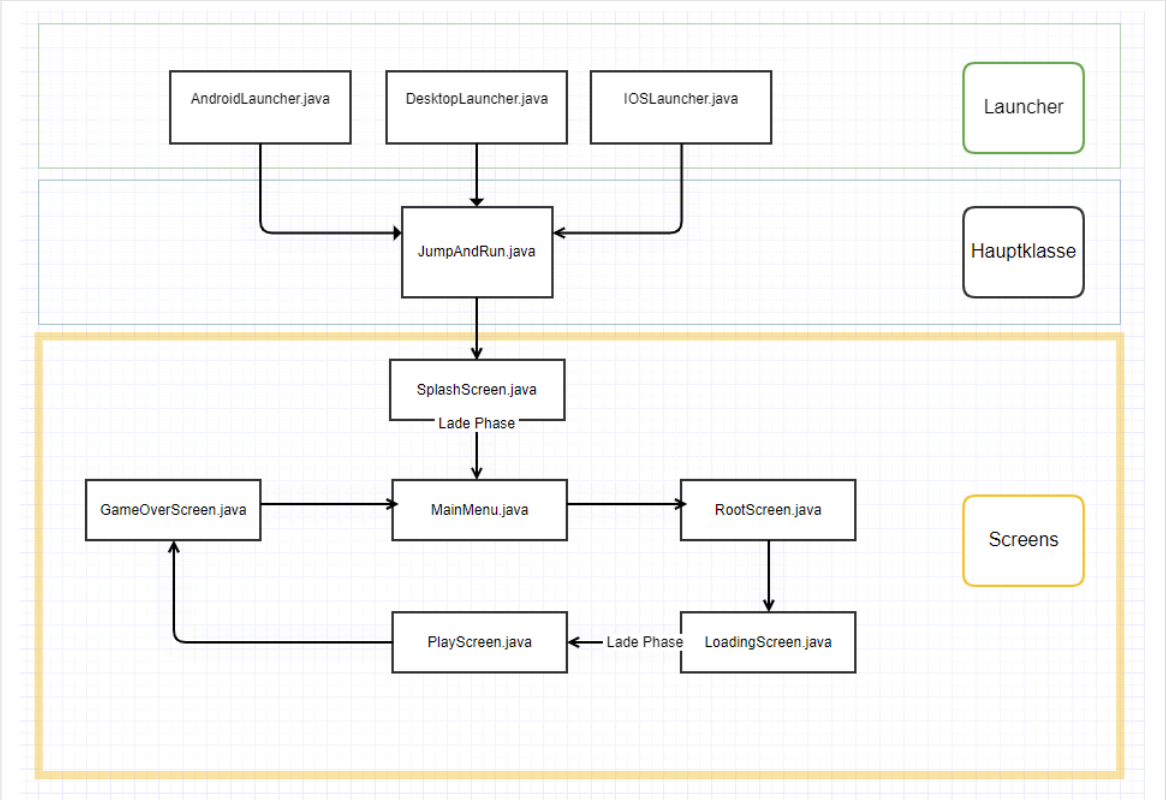
Diese Hauptklasse (JumpAndRun.java) gibt nun die Fähigkeit das Bild zu „zeichnen“ ab an eine Unterklasse, die Screen implementieren muss, dies geschieht mit der Methode setScreen() (innerhalb der onCreate() Methode), welche von der Oberklasse Game vererbt wurde. Während des Spielzyklus wechselt der Screen. Es gibt den Playscreen.java in dem das eigentliche Spiel abläuft, den GameOverScreen.java in welchem dem Spieler die Anzahl an gesammelten Kristallen und die Leben anzeigt werden nachdem man das Level abgeschlossen oder verloren hat, den MainMenu.java Screen in dem man die Schwierigkeit wählt, den LoadingScreen.java welcher das Level vorlädt, den SplashScreen.java der verschiedene Logos beim Spielstart anzeigt und den RootScreen.java in dem man das Level auswählt, was man spielen möchte.

Der PlayScreen ist die wichtigste Klasse in diesem Spiel, da hier das eigentliche Spiel abläuft. Alle Objekte, die während dem eigentlichen Spiel gebraucht werden kommen hier zusammen. So kennt der PlayScreen immer ein Objekt vom Typ Spieler, er kennt jeden Gegner innerhalb des Levels, jede Textur und so weiter. Hier wird die Bildentstehung abgehandelt, die wird Physik berechnet, der Input wird verarbeitet und die Welt wird erstellt.

Dazu sind verschiedene Funktionen auch aus dem PlayScreen heraus in eigene Klassen ausgelagert um ihn aufgeräumt und strukturiert zu halten. So hat der Spieler eine eigene Klasse und jede Art von Gegner oder Item hat seine eigene Klasse. Es gibt den MapCreator.java, in dem die Welt erstellt wird und den MyContactListener.java in dem die Kollisionsabfrage abgehandelt wird. Des Weiteren hat das Hud-Overlay seine eigene Klasse (Hud.java) und es gibt eine Klasse, mit allen statischen Variablen (Statics.java) wie die Bilder pro Sekunde. Mehr dazu im Folgenden.

3.3 Zyklus des Spiels

Wie bereits oben beschrieben ruft einer der Launcher (z.B.: DesktopLauncher.java) abhängig vom Betriebssystem die Hauptklasse des Core Moduls auf (JumpAndRun.java), diese gibt dann die Fähigkeit, das Bild zu zeichnen an verschiedene Screens ab, die wie im Bild beschrieben durchlaufen werden.



B1, Zyklus des Spiels, erstellt mit Gliffy

Während die Screens durchlaufen werden gibt es außerdem noch zwei Ladephasen. In der nach dem SplashScreen.java werden alle Texturen geladen, die für jedes Level oder Menü gebraucht werden, dies geschieht einmalig. In der zweiten Ladephase wird das Level, also die XML-Datei, vorgeladen.

4 Der PlayScreen

4.1 Aufbau eines Screens

Da die sechs Screens (Kapitel 3.2 und 3.3) die Klasse Screen, welche ein Interface ist, implementieren, müssen sie auch die Methoden des Interfaces überschreiben. Diese sind render(), show(), resize(), pause(), resume(), hide() und dispose().

Die render() Methode wird 60-mal pro Sekunde aufgerufen, sie wird genutzt um das eigentliche Bild inklusive der Physik zu berechnen. Die resize() Methode wird aufgerufen, wenn das Fenster die Größe ändert oder der Bildschirm sich dreht. Diese Methode wird verwendet, um das das Spiel der Auflösung entsprechend zu skalieren. In der dispose() Methode wird Grafikspeicher freigegeben, sie wird mit dem zerstören des aktuellen Screens aufgerufen (also sobald der Screen wechselt oder das Spiel geschlossen wird). Die Methoden pause(), resume(), show() und hide() werden in diesem Spiel nicht gebraucht, müssen aber trotzdem überschrieben werden (wegen des Interfaces).

4.2 Bildberechnung mit Box2D

Box2D ist eine Bibliothek, die ausschließlich zur Physikberechnung dient. Die Texturen werden im Nachhinein an den Positionen der Box2D Körper gezeichnet. Jeder Körper ist ein Objekt, wovon es 2 Typen gibt.

Statische, die in der Welt platziert werden, danach nicht mehr bewegt werden können und demensprechend nicht mehr verändert werden können und auch nicht von Kräften wie zum Beispiel der Schwerkraft beeinflusst werden können. Ein Statisches Objekt ist zum Beispiel der Boden über den man läuft.

Des Weiteren gibt es Dynamische Objekte. Ein solches Objekt kann bewegt werden, es wird von Kräften wie zum Beispiel der Schwerkraft beeinflusst. Jedes Dynamische Objekt besitzt eine update() Methode, die vielseitig eingesetzt wird. Zum Beispiel werden Plattformen damit bewegt, Coins drehen gelassen oder der Spieler wird bewegt.

Das Bild entsteht nun immer in 3 Schritten. Als erstes werden von der render-Methode im PlayScreen aus die update Methoden von allen Klassen ausgeführt, die so eine besitzen. Hierrüber werden Werte angepasst, die für den nächsten Schritt gebraucht werden. Zu diesen Werten zählen zum Beispiel Kräfte, die den Körper „angreifen“. So wird zum Beispiel in der update-Methode des Spielers, wenn er W drückt eine Kraft angelegt in positive y-Richtung oder die Koordinate der sich bewegenden Plattform wird in positive x-Richtung erhöht.

Im nächsten Schritt wird mit world.step() die Physik von Box2D berechnet. Hierzu zählen Kollisionen, Schwerkraft, wie welche Kraft auf welchen Körper wirkt, wie sich Körper gegenseitig beeinflussen, … Während world.step() darf kein Wert mehr angepasst werden, weswegen das System auch strikt in diese 3 Schritte getrennt ist. Zudem darf kein neues Objekt dazu kommen oder ein bestehendes Objekt entfernt werden, dies ist eine Vorgabe des Box2D Systems, da sonst das Spiel abstürzt.

Im Letzten Schritt wird das gesamte alte Bild geleert bzw. einmal weiß übermalt. Im Anschluss wird an den Positionen der Box2D Objekten die passende Textur gezeichnet. Dieser 3-schrittige Zyklus läuft 60-mal die Sekunde ab.

4.3 Einen Körper mit Box2D erstellen

Um einen Körper erstellen zu können wird zunächst eine Welt benötigt, in der sich der Körper befinden soll. Diese wird bei diesem Spiel im Konstruktor des PlayScreens erschaffen. Hierzu müssen 2 Parameter übergeben werden. Zum einem die Schwerkraft in Form eines Vektors, der 2 Koordinaten hält (Schwerkraft in x und y Richtung) und zum anderen ein boolean, der angibt ob die Welt Körper, die sich nicht verändern (also wenn keine neuen Kollisionen an diesen Körpern stattfinden und sie sich nicht bewegen) „einschlafen“ lassen kann. Dies führt dazu das die Physik der Objekte nicht erneut berechnet wird, was Leistung spart.

Jedes Box2D Objekt das an der Physikberechnung teilnimmt ist ein Körper. Im Code heißt diese Klasse nachdem die Objekte erschaffen werden Body (siehe Zeile 2). Um einen Body zu erschaffen wird eine Body Definition benötigt. Die dazugehörige Klasse heißt BodyDef (siehe Zeile 3). Diese Body Definition muss mindestens den Typ des Körpers beinhalten (siehe Zeile 4), also statisch oder dynamisch und die Koordinaten des Körpers innerhalb der Welt (siehe Zeile 5). Mit der Methode von World createBody(BodyDef) (siehe Zeile 6) wird schließlich der Körper erschaffen.

Um die physikalischen Eigenschaften des Körpers wie zum Beispiel das Gewicht oder die Dichte festlegen zu können und um dem Körper eine Form geben zu können wird außerdem eine Fixture Definition im Code FixtureDef benötigt (siehe Zeile 7). Dieser Fixture Definition muss man nun das Attribut shape setzen, also dem Körper eine Form geben. Hier gibt es verschiedene Möglichkeiten, die von einer Box (siehe Zeile 9) oder einem Kreis bis hin zu benutzerdefinierten Polygonen reichen. Nachdem man als Beispiel eine PolygonShape erstellt hat, muss man das Attribute shape der Fixture Definition auf den Wert der ertstellten PolygonShape (siehe Zeile 8) setzen (siehe Zeile 10). Dieses Attribut muss mindestens gesetzt werden, da es standartmäßig null also nicht definiert ist und damit das Spiel (sobald world.step() aufgerufen wird) abstürzen würde.

Sobald man eine Fixture Definition hat, die als Attribut eine Shape gesetzt bekommen hat, kann man damit dem Körper die Form und die festgelegten Eigenschaften mit der Methode createFixture() von Body geben (siehe Zeile 12). Hier als Beispiel die Erschaffung eines Bodenstücks [Einige Variablen wurden durch feste Werte zur Demonstration ersetzt, weil der Code sonst zu lang werden würde].

Ein Body kann mehrere Fixture und damit mehrere Shapes haben, muss aber mindestens eine besitzen. Die Eigenschaften die mit der Fixture Definition gesetzt werden betreffen nur die dazugehörige Shape. Als Beispiel wurde hier an eine Box Shape die Eigenschaft restitution gebunden (siehe Zeile 11), also das der Körper abprallt wie ein Flummi an dieser Shape, sobald diese Shape mit einer weiteren interagiert. Der Körper kann noch weitere Fixture und damit eine weitere Shapes haben, auf die diese Eigenschaft nicht zutrifft. Als Beispiel der Spieler (siehe unten Kapitel 4.7).

```
1
2  Body body;
3  BodyDef bdef = new BodyDef();
4  bdef.type = BodyDef.BodyType.StaticBody;
5  bdef.position.set(2f, 2f);
6  body = playscreen.getWorld().createBody(bdef);
7
7  FixtureDef fdef  = new FixtureDef();
8  PolygonShape shape = new PolygonShape();
9  shape.setAsBox(bounds.getWidth(), bounds.getHeight);
10 fdef.shape = shape;
11 fdef.restitution = 1f;
12 body.createFixture(fdef);
13
```

4.4 Von Tiled zu Box2D

Wie bereits oben beschrieben soll die Welt nun aus einer XML-Datei heraus erschaffen werden. Um die XML-Datei zu erstellen, nutze ich das Tiled System von LibGDX. Diese XML-Dateien haben die Endung .tmx und können entweder mit einem Editor erstellt werden (https://doc.mapeditor.org/en/stable/) oder per Hand geschrieben werden.

Innerhalb dieser Datei werden Objekte zu Gruppen zusammengefasst, so gibt es zum Beispiel eine Gruppe Hitbox die alle statischen Objekte enthält (wo der Spieler drüber läuft) oder eine Gruppe die Coins heißt, die alle Coins enthält. In so einer Gruppe (objectgroup) hat jedes Objekt (object) eine id, sowie eine x- und eine y-Koordinate. Außerdem hat jedes Objekt eine Höhe und eine Breite.

```
<objectgroup id="3" name="Hitbox">
  <object id="47" x="736" y="208" width="208" height="16"/>
  <object id="48" x="880" y="160" width="64" height="16"/>
  <object id="61" x="992" y="160" width="176" height="10.6667"/>
  <object id="62" x="1104" y="208" width="185.5" height="16"/>
  <object id="64" x="192.5" y="192" width="494.5" height="16"/>
  <object id="65" x="1328" y="160" width="96" height="16"/>
  <object id="66" x="1408" y="128" width="64" height="11"/>
  <object id="87" x="1504" y="96" width="96" height="16"/>
</objectgroup>
<objectgroup id="3" name="Coins">
  <object id="55" x="1104" y="176" width="16" height="16"/>
  <object id="56" x="1104" y="192" width="16" height="16"/>
  <object id="57" x="1120" y="192" width="16" height="16"/>
</objectgroup>
<objectgroup id="3" name="Enemy">
  <object id="107" x="2592" y="176" width="16" height="16"/>
  <object id="109" x="2928" y="176" width="16" height="16"/>
  <object id="193" x="1744" y="192" width="16" height="16"/>
  <object id="266" x="1168" y="192" width="16" height="16"/>
  <object id="267" x="1360" y="192" width="16" height="16"/>
</objectgroup>
```

Nun muss diese Datei in Box2D übersetzt werden. Dies geschieht in der der Klasse MapCreator.java, von welcher ein Objekt im PlayScreen existiert. Dazu wird im Code jede Objektgruppe/Ebene mit einer For-Schleife durchlaufen (siehe Zeile 2 und 10). Zu jeder Objektgruppe in der XML Datei gibt es im Code eine Passende Klasse. So gehört zu jeder HitBox aus der 2. Gruppe/Ebene die Klasse Ground.java, zu Coins die Klasse Coins.java, … Für jedes Objekt werden jetzt die Koordinaten sowie die Breite und Höhe geholt (siehe Zeile 5). Anschließend wird mit diesen 4 Werten dann ein neues Objekt vom Typ der Objektgruppe erschaffen. Handelt es sich um ein Statisches Objekt, wie zum Beispiel der normale Boden über den man läuft wird ein neues Objekt vom passenden Typen, in dem Fall Ground, erstellt (siehe Zeile 6) ohne Referenzattribut, da man auf diese Objekte nicht mehr zugreifen muss nachdem sie erstellt wurden.

Handelt es sich jedoch um ein dynamisches Objekt, das eine update Methode brauch, wird das Objekt nicht nur erschaffen, sondern auch in eine Arrayliste geschrieben, damit man ein Referenzattribut hat, über das man die update() Methode, die jedes dynamische Objekt besitzt aufrufen kann. Die Rectangle hier sind einfach nur Objekte die als Attribute x, y, width und height haben und den Code verkürzen. Hier ist enemys eine Arrayliste [Code im Beispiel verkürzt].

```
1
2  for (MapObject object : playscreen.getMap().getLayers().get(2)
3                        .getObjects().getByType(RectangleMapObject.class)){
4
5      Rectangle rect = ((RectangleMapObject) object).getRectangle();
6      new Ground(playscreen, rect);
7  }
8  for (MapObject object : playscreen.getMap().getLayers().get(4)
9                        .getObjects().getByType(RectangleMapObject.class)){
10
11     Rectangle rect = ((RectangleMapObject) object).getRectangle();
12     enemys.add(new Slime(playscreen, rect));
13 }
```

Im Spiel gibt es mehrere verschiedene Gegner, die jedoch viel gemeinsam haben. Alle Gegner besitzen eine x und y Koordinate und haben ähnliche Methoden.

Um den Code abzukürzen und das System modular zu halten, benutze ich hier eine Oberklasse Enemy, von der alle Gegner erben. Es gibt auch nicht mehrere Arraylisten (eine für die Schleims, eine für die Grashüpfer, …), sondern nur eine, die Enemy also die Oberklasse enthält. Dieses System mit einer Oberklasse wurde nun im Code auf alle Klassen angewendet. So erben alle Klasen, die Gegner erzeugen, also Dragon.java, Slime.java und Something.java von Enemy.java oder alle Objekte die aus Fragezeichen-Blöcken kommen, also Coin.java, DynmaicCoin.java und Life.java von Spawnable.java.

Durch diese Modularität und die Oberklassen lassen sich später neue Inhalte sehr einfach und schnell hinzufügen, was einer Anforderung (Kapitel 1.2) entspricht.

4.5 Körper bewegen

Um Körper bewegen zu können, muss man wie bereits oben erwähnt eine Kraft anlegen. Hierzu besitzt die Klasse Body die Methode applyLinearImpuls(). Dieser muss man als Parameter einen Vector2 (Objekt das 2 Koordinaten hält), der die Stärke der Kraft in x und y Richtung beinhaltet, einen Vector2 der den Angriffspunkt der Kraft enthält und einen Boolean der angibt ob der Körper dadurch aus dem Ruhemodus rausgeholt werden darf mitgeben. Diese Methode „stößt“ den Körper in die gewünschte Richtung und ändert unmittelbar die Geschwindigkeit des Körpers, ist jedoch eine einmalige Kraft, die durch zum Beispiel die Reibung beim laufen oder die Schwerkraft beim Springen wieder ausgeglichen werden kann.

Der Spieler wird nun min der Methode handleInput(float dt) (PlayScreen.java) bewegt. Wenn er sich noch langsamer als 1.5 (Newton) in positive x Richtung bewegt, wird mit der folgenden Zeile Code eine Kraft in Positive x-Richtung angelegt. Der Punkt wo die Kraft angreift ist der Mittelpunkt des Körpers, welchen man über die Methode getWorldCenter() von Body bekommt. [Code gekürzt].

```
1
2 .applyLinearImpulse(new Vector2(0.1f, 0), player.b2body.getWorldCenter(), true);
3
```

Genauso lässt man den Spieler auch springen, nur das der Vector2 statt 0.1f und 0 als Parameter die Werte 0f, 1f übergeben bekommt, da der Spieler um zu springen in positive y-Richtung „gestoßen“ werden muss. Hierbei entsteht jedoch das Problem, das der Spieler auch in der Luft springen kann. Um dies zu lösen muss man testen können, ob der Spieler den Boden berührt, nur dann darf er springen (Dies wird im nächsten Kapitel 4.6 gelöst).

Gegner und Plattformen werden dagegen anders bewegt. Hierfür nutze ich die Methode setLinearVelocity(). Dieser muss man lediglich einen Vector2 mitgeben, da diese Methode den Körper konstant in die über den Parameter übergebene Richtung bewegt, ungeachtet vom Angriffspunkt. Somit haben Körper, die über diese Methode bewegt werden dementsprechend auch eine konstante Geschwindigkeit. Als Beispiel wird hier ein Schleim in positive x-Richtung bewegt (ausschnitt aus Slime.java). [Code gekürzt]

```
1
2 .setLinearVelocity(new Vector2(1f, 0));
3
```

4.6 Benutzerdefinierte Kollisionsabfrage von Box2D Objekten

Normalerweise werden alle Kollisionen zwischen zwei Objekten von Box2D gleich abgehandelt, sie stoßen gegeneinander. Es wird also verhindert, dass 2 Körper ineinander geraten. Jedoch ist dies ab und zu nötig. Oder wie soll man unterscheiden ob der Spieler einen Gegner oder ein Item zum Einsammeln berührt. Je nachdem, ob es ein Gegner oder ein Item ist müssen unterschiedliche Aktionen folgen: Der Spieler verliert ein Leben oder bekommt Coins dazu. Dies lässt sich mit dem normalen Kollisionshandling nicht lösen.

Die Klasse World besitzt die Methode setContactListener(), mit welcher ein ContactListener an die Welt gebunden wird, um eine benutzerdefiniert Kollisionsabfrage zu ermöglichen. Diese benutzerdefiniert Kollisionsabfrage wird nun in die Klasse WorldContactListener.java abgehandelt, welche ContactListener implementiert.

Durch das Implementieren von ContactListener muss die Klasse die Methoden beginContact(), endContact(), preSolve() und endSolve() überschreiben. Damit lassen sich Aktionen ausführen, sobald 2 Körper aneinanderstoßen. Jetzt muss man nur noch zwischen den Objekten unterscheiden können.

Hierfür liefert LibGDX das Bit Filter System. Hier wird einer Shape ein categoryBit gesetzt. Dies lässt sich dann darauf ausweiten, das als Beispiel jeder Gegner eine Shape mit demselben categoryBit besitzt. Als Beispiel ein Ausschnitt aus Slime.java.

```
1
2 fixturedef.filter.categoryBits = BIT_FILTER.ENEMY_BIT;
3
```

Über diese Zahl kann man die Shape dieser Körper Gruppe dann eindeutig identifizieren. Um Fehler wie zum Beispiel verschiedene Bits für eigentliche gleiche Gegner zu vermeiden gibt es die Klasse BIT_FILTER.java, welche alle categoryBits enthält. [Code wurde gekürzt]

```
1
2 public class BIT_FILTER {
3
4    public static final short PLAYER_BIT = 2;
5    public static final short BRICK_BIT = 4;
6    public static final short COIN_BIT = 8;
7
8 }
9
```

Zudem muss man jeder Shape eines Körpers noch einen maskBit oder mehrere maskBits setzen. Dieser beschreibt, mit welchem Körper dieser kollidieren kann. Als Beispiel ein Ausschnitt aus Slime.java. [Code wurde gekürzt]

```
1
2 fdef.filter.maskBits = BIT_FILTER.BRICK_BIT | BIT_FILTER.DEFAULT_BIT;
3
```

Nun kann man im ContactListener zwischen diesen Bits unterscheiden. Hier als Beispiel eine Abfrage, ob der Spieler mit einem Gegner kollidiert.

```
1
2 if (fixA.getFilterData().categoryBits == BIT_FILTER.PLAYER_BIT &&
3 fixB.getFilterData().categoryBits == BIT_FILTER.ENEMY_BIT){}
4
```

Hier wird getestet, ob eine der beiden miteinander kollidierenden Shapes den categoryBit PLAYER_BIT (=2) gesetzt bekommen hat und die andere Shape den categoryBit ENEMY_BIT (32). Wenn dieses Statement wahr wird, ist der Spieler mit einem Gegner kollidiert.

Da man nicht unterscheiden kann, welche der beiden Shapes welche ist, muss man die Abfrage noch umdrehen.

```
1
2  if (fixB.getFilterData().categoryBits == BIT_FILTER.PLAYER_BIT &&
3  fixA.getFilterData().categoryBits == BIT_FILTER.ENEMY_BIT){}
4
```

Nun kann man dementsprechend eine dazu passende Aktion ausführen. Hier verliert der Spieler als Beispiel ein Leben.

```
1
2  playscreen.getPlayer().looselife();
3
```

Da diese Klasse (WorldContactListener.java) ContactListener implementiert, befinden sich hier auch die Methoden beginContact() und endContact(). Hiermit lässt sich das im vorherigen Kapitel beschriebene Problem, das der Spieler in der Luft springen kann, lösen.

Dazu muss eine neue Variable eingeführt werden, welche angibt, ob der Spieler den Boden berührt (playerIsOnGround) sowie eine Getter Methode für diese.

Hier ist es schlecht als Typ der Variable einen Boolean zu verwenden, der auf true gesetzt wird, sobald der Spieler einen Kontakt beginnt (beginContact()) und auf false gesetzt wird sobald der Kontakt endet (endContact()), da es möglich ist, das der Spieler mehrere Körper gleichzeitig berührt. Wenn der Spieler zum Beispiel über einen Hitboxübergang läuft (Zum Beispiel von 2 aneinander anschließenden Böden), wird die Variable zuerst auf true gesetzt, weil ein neuer Kontakt stattfindet, da der Spieler aber parallel noch die alte Hitbox berührt zu dem Zeitpunkt, als er die neue berührt wird die Variable kurz danach wieder auf false gesetzt, da eben dieser zur alten Hitbox Kontakt endet, nachdem der Kontakt zur neuen angefangen wurde.

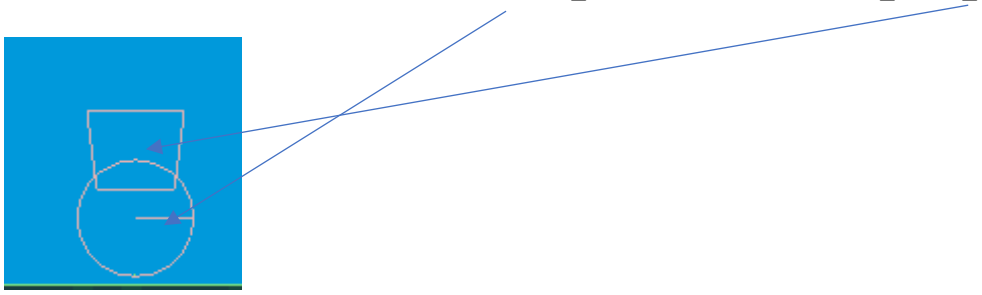
Um dies zu lösen, ist die Variable vom Typ Integer und die Getter-Methode isplayeronground() gibt true zurück sobald dieser Integer größer oder gleich 1 ist.

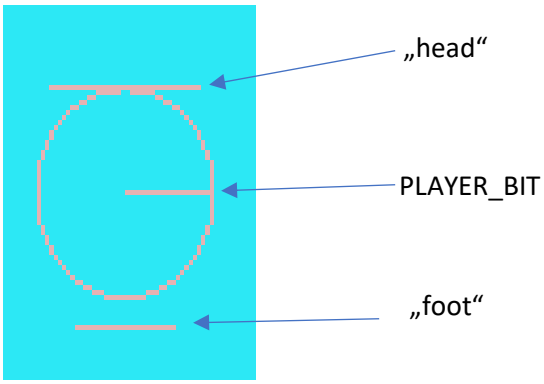
Nun kann der Spieler nicht mehr springen, wenn er in der der Luft ist, jedoch gibt es immer noch den Fall, dass er gegen eine Wand springt. Dann findet ja auch genau ein Kontakt statt und der Spieler kann springen obwohl er nicht den Boden berührt.

Um dies lösen zu können, muss man unterscheiden können, ob der Spieler einen anderen Körper mit dem unteren Teil oder der Seite berührt. Dies ist aktuell schwer umzusetzen, da der Spieler nur aus einer Shape besteht (einem Kreis) und man nur testen kann ob diese Shape eine andere berührt und nicht wo diese Shape eine andere berührt. Um dies zu lösen, muss ein Körper aus mehreren Shapes bestehen.

4.7 Mehrere Shapes an einem Körper

Wie bereits oben beschrieben gibt es das Problem, dass der Spieler springen kann, obwohl er in der Luft ist. Ein weiteres Problem ist, weswegen oftmals eine Shape pro Körper nicht ausreicht, dass zum Beispiel es einen Unterschied macht ob der der Spieler gegen einen Gegner läuft und damit ein Leben verliert oder ob der Spieler auf den Kopf des Gegners spring und ihn damit umbringt. Um diese Probleme zu lösen, kann ein Body mehrere Shapes besitzen. Als Beispiel besitzt ein Gegner 2 Shapes, denen jeweils andere categoryBits zugeordnet sind zwischen denen man dann im ContactListener unterscheiden kann. Einmal der normale ENEMY_BIT und dazu der ENEMY_HEAD_BIT.





So ähnlich ist auch der Spieler zusammengesetzt, nur dass er aus drei verschiedenen Shapes besteht, die normale Körper Shape mit dem categoryBit PLAYER_BIT, am Kopf die extra Shape der in dem Fall die eindeutige ID „head“ zugewiesen wurde und einer Extra Shape mit der eindeutigen ID „foot“.

Die beiden Shapes haben jeweils eine eindeutige ID und keinen categoryBit, da sie einzigartig sind und nur als „Sensor“ dienen.

Hiermit lässt sich nun ganz einfach überprüfen, ob der Spieler mit dem Kopf gegen einen Fragezeichen-Block springt oder sich nur darüber bewegt oder ob der Spieler auf einen Gegner springt oder gegen ihn läuft.

Zudem lässt sich hiermit das Problem lösen, dass der Spieler noch in der Luft springen kann, indem man die Variable isplayeronground nur erhöht oder verkleinert, in beginContact() und endContact() wenn nicht der Spieler, sondern nur die Shape mit der ID „foot“ einen anderen Körper berührt. Dies sieht dann so aus (Ausschnitt aus WorldContactListener.java) [Code gekürzt].

```
1
2  @Override
3  public void endContact(Contact contact) {
4      Fixture fixA = contact.getFixtureA();
5      Fixture fixB = contact.getFixtureB();
6
7      if ((fixB.getFilterData().categoryBits == BIT_FILTER.DEFAULT_BIT ||
8              fixB.getFilterData().categoryBits == BIT_FILTER.BRICK_BIT ||
9              fixB.getFilterData().categoryBits == BIT_FILTER.MOVING_BIT)
10              && (fixA.getUserData() == "foot")){
11          playerisongorund--;
12      }
13 }
14
```

4.8 Box2D Objekte erschaffen während des Spiels

Wie bereits in Kapitel 4.2 (Bildentstehung mit LibGDX) beschrieben, ist die Bildentstehung strikt in drei Phasen unterteilt, da während world.step(), also der eigentlichen Physikberechnung, keine Werte mehr geändert werden dürfen oder Körper erschaffen/zerstört werden dürfen. Dies war bis jetzt strikt durch die update Methoden in den jeweiligen Klassen, die vorher aufgerufen wurden getrennt. Mit dem ContactListener wird jetzt ein weiteres System benötigt, da es passieren kann, dass der Spieler zum Beispiel während world.step() gegen einen Fragezeichenblock springt und somit einen Coin spawnt oder auf einen Gegner springt um ihn umzubringen bzw. den Körper aus der Welt zu entfernen. Solche Interaktionen dürfen jedoch nicht während world.step() stattfinden.

Um einen Körper zu erschaffen, besitzt der PlayScreen ein System bestehend aus zwei Arraylisten, eine heißt tospawn, die andere spawned. Immer wenn ein Körper erschaffen werden soll, wird zunächst diese Information in der tospawn gespeichert. Vor world.step() wird nun diese tospawn Liste durchgelaufen und alle Objekte die hier drin eingereiht sind werden erschaffen und in die spawned ArrayListe geschrieben.

Dementsprechend wir eine Oberklasse Spawnable benötigt. Die Arrayliste spawned enthält Objekte die von Spawnable erben später.

Um ein neues Objekt erschaffen zu können werden mehr als nur eine einfache Information wie eine ID benötigt. Mindestens muss die tospawn Arrayliste also den Typen bzw die Klasse enthalten die als Bauplan genutzt werden soll für das Objekt und die Koordinaten, an denen das Objekt und damit der Körper erschaffen werden soll. Zudem haben manche Objekte noch Besonderheiten beim Spawnen, zum Beispiel ob ein Coin nach rechts oder links aus dem Fragezeichenblock rausgeflogen kommen soll. Dazu muss noch eine meta gespeichert werden.

Dazu gibt es die Klasse SpawnDef.java. Diese enthält eine Klasse unbekannten Typens (Class<?>), einen Vektor2 für die Position und einen int für die meta. Diese Werte müssen nun über den Konstruktor initialisiert werden.

Insgesamt wird also zuerst ein Objekt vom Typ SpawnDef in die Arrayliste die als Warteschlange dient, geschrieben, diese SpawnDef enthält Informationen über den Typ des Objektes, die Position und die Meta. Bevor world.step() aufgerufen wird, werden dann alle zu erschaffenden Objekte aus der Warteschlange entfernt und anhand des Bauplan Objektes (SpawnDef) in die Arrayliste mit den aktiven Objekten geschrieben. Durch dieses System ist es unmöglich, das ein Objekt während world.step() erschaffen wird.

Ein weiteres Problem stellt das Entfernen der Objekte dar. Dies darf ebenfalls nicht während world.step() geschehen. Diese Funktion wird jedoch gebraucht, um zum Beispiel einen Gegner umzubringen. Um dies zu lösen, erhalten alle Gegner durch ihre Obreklasse Enemy zwei neue booleans, nämlich toremove und removed. Hier wird nun dasselbe System wie mit den Arraylisten angewendet. Mit dem Unterschied, das keine Arraylisten gebraucht werden, da jeder Gegner eine update-Methode besitzt, die bereits vor world.step() aufgerufen wird. Sobald der Spieler nun auf einen Gegner springt wird das toremove-Attribute auf true gesetzt. Sobald der dreischrittige Bildentstehungszyklus nun das nächste Mal durchläuft und alle update Methoden aufruft, wird in diesen Update-Methoden der Gegner getestet, ob toremove auf true gesetzt wurde. Wenn dies der Fall ist wird der Körper zerstört. Damit ist es wieder unmöglich, das der Körper während world.step() aus der Welt entfernt wird.

Somit wurde das erschaffen und zerstören der Körper in das 3-Schritt-System (Kapitel 4.2 Bildberechnung mit Box2D) integriert.

4.9 Box2D und große Objekte, das PPM System

Nach der Dokumentation von Box2D wird empfohlen, dass man kleinere Werte zur Physikberechnung benutzen soll, da Box2D damit besser klarkommen würde. Es wird empfohlen eine Variable für PixelPerMeter zu benutzen, welche die Werte verkleinert.

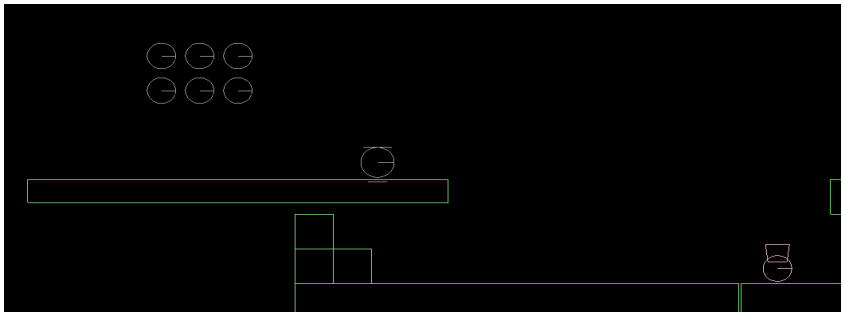
Nun man teilt einfach alle Werte, die die Physikberechnung betreffen durch den Wert für PixelPerMeter, in meinem Fall 100. Dies betrifft Positionen, Kräfte und so weiter. Dadurch läuft die Physikberechnung wie empfohlen mit kleinen Werten ab.

Wie bereits oben in 3.2 Bildentstehung beschrieben wird im dritten Schritt an den Positionen der Box2D Körpern dann Texturen gezeichnet. Da die Koordinaten durch 100 dividiert sind muss man sie an dieser Stelle wieder hochskalieren, also wieder mit 100 multiplizieren.

Da die Variable für PixelPerMeter PPM überall im gesamten Projekt verwendet wird, habe ich sie in der Statics.java Klasse hinzugefügt.

4.10 von Box2D zu den Texturen, der dritte Schritt

Bis hierher sind im Spiel nur die farbigen Linien der Box2D Körper, welche durch den DebugRenderer im PlayScreen gezeichnet werden. In der Abbildung stehen grüne Linien für statische Objekte und die roten Linien stehen für dynamische Objekte. Hier sieht man 6 Coins oben links, den Spieler und einen Gegner unten rechts.



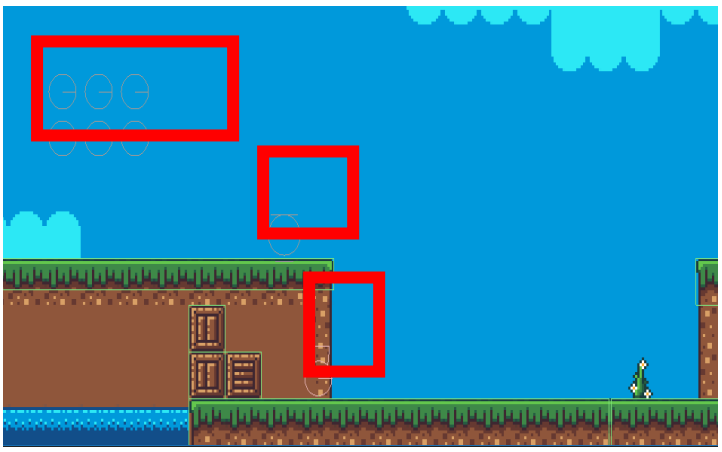
Der Strich in den Körpern steht für die Rotation der Körper.

Wie bereits oben beschrieben wird die Welt aus einer XML-Datei gebaut (das System heißt Tiled). Im Moment trifft das nur auf die Box2D Körper zu, wie man auch auf der Abbildung sieht. Nun hat Tiled noch eine weitere Funktion. Man kann mit dem Editor zudem feste, nicht mehr bewegbare Texturen in die Welt platzieren. Dazu brauch man ein Tileset. Dieses enthält alle statischen Texturen der Welt. Zudem gehört zu so einem Tileset eine weitere XML-Datei mit Informationen, wie das Bild zerlegt werden muss, um an die einzelnen Texturen zu kommen.

Als Beispiel enthält die XML-Datei zu diesem Tileset die Informationen, dass die eigentliche Textur 16x16 Pixel groß ist, die einzelnen Texturen einen Toleranzrand von 2px haben sowie das die Texturen jeweils 2px Abstand zu einanader haben.

Anschließend kann man im Editor die einzelnen, ausgeschnittenen 16x16 Pixel großen Texturen in die Welt platzieren, diese werden dann komprimiert in die XML-Datei geschrieben, so enthält die kryptische Kombination aus Zahlen und Buchstaben innerhalb des data Tags alle Informationen über die Position, Größe und Art der Texturen, die in die Welt platziert wurden.

```
<tileset firstgid="1" name="nature-paltformer-tileset-16x16" tilewidth="16" tileheight="16" tilecount="77" columns="7">
 <image source="nature-paltformer-tileset-16x16.png" width="112" height="176"/>
</tileset>
<layer id="1" name="Texture" width="240" height="13">
 <data encoding="base64">
  TgAAAE4AAABOAAAATgAAAE4AAABOAAAATgAAAE4AAABOAAAATgAAAE4AAABOAAAATgAAAE4AAABOAAAATgAAAE4AAABOAAAATgAAAE4AAABOAAAATgAAAE
 </data>
</layer>
```



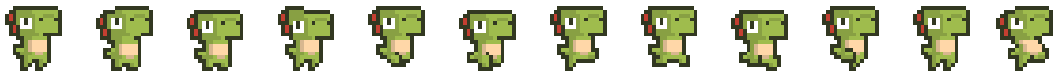
Diese können dann mithilfe der Klasse OrthogonalTiledMapRenderer gezeichnet werden. dazu muss man lediglich ein Objekt dieser Klasse erstellen, diesem Objekt die XML-Datei als Parameter übergeben und anschließend in der render() Methode des PlayScreens die Methode render() des Objektes aufrufen. Dadurch sieht das Spiel dann so wie in der Abbildung aus. Alle nicht bewegbaren Texturen sind nun gezeichnet. Es fehlen allerdings

noch die Texturen an den bewegbaren, dynamischen Objekten. Hier sind lediglich feste, statische Texturen gezeichnet. Nicht die bewegbaren, dynamischen Objekte, die hier rot umrahmt sind da man sie schlecht erkennt, gezeichnet. Diese Objekte können ihre Position und Rotation bis zu 60-mal die Sekunde ändern, also durch jeden Zyklus des 3-Schritt Systems (siehe 3.2 Bildberechnung mit Box2D). Somit ist es auch sinnvoll, das die Texturen dieser Körper immer nach world.step(), also nach der Physikberechnung neu an der Position des Box2D Körper gezeichnet werdem.

Um dies für die dynamischen Objekte umzusetzen, erben alle dynamischen Objekte, also der Spieler, alle Items, Coins und Gegner direkt von Sprite oder durch ihre Oberklasse von Sprite. Ein Sprite ist ein Objekt, dass ein „Bild“ enthält, sowie sämtliche Methoden und Eigenschaften zu diesem Bild wie zum Beispiel die Rotation, um die das Bild gedreht ist, die Breite und Höhe, die Koordinaten des Bildes, ….

Da als Beispiel die Lauf-Animation des Spielers jedoch aus zwölf Bildern besteht, wird für diese Animation ein Sprite Sheet benutzt. Ein Sprite Sheet ist ein Bild, das aus mehreren einzelnen Bildern zusammengeschnitten besteht. Dieses Sprite Sheet wird dann in der Spieler Klasse auseinandergeschnitten. Die einzelnen Bilder werden anschließend in einen zweidimensionalen Array geschrieben, der dann TexturRegions enthält.

Um die Textur nun an der Stelle des Box2D Körpers zu zeichnen, muss zunächst die passende Textur mit der Methode setRegion() von Sprite gesetzt werden. Dieser Methode muss man als Parameter eine TexturRegion übergeben, welche man aus dem zweidimensionalen Array bekommt. Hiermit lassen sich nun auch Animationen implementieren. Denn Animationen sind im Endeffekt einfach nur eine aneinander Reihung von leicht abgeänderten Bildern. So sieht zum Beispiel die Lauf-Animation des Spielers aus.



Innerhalb der Spieler Klasse gibt es nun eine Zählvariable, die immer wieder von 0 bis 11 zählt. Diese wird alle 0.15 Sekunden erhöht, womit dann auch gleich dem Spieler eine neue Textur gesetzt wird, nämlich die aus dem Array an der Stelle der Zählvariable (Ausschnitt aus Player.java) [Cod gekürzt].

```
1
2  setRegion(regions[0][counter]);
3
```

Nun muss nur noch die Position der Textur der Position des Box2D Körpers angepasst werden. Hierzu wird wieder die update-Methode genutzt. Problematisch ist daher, dass Box2D und Sprite unterschiedliche Koordinaten-Systeme nutzen. Wenn man sich die Koordinaten des Box2D Körper holt, bekommt man immer den Mittelpunkt des Körpers, wohingegen Sprite immer die untere linke Ecke benutzt. Durch eine Umrechnung der Koordinaten lässt sich das jedoch leicht beheben. (Ausschnitt aus Player.java) [Cod gekürzt].

```
1
2  setPosition(body.getPosition().x - getWidth() / 2, body.getPosition().y -
3                                                          getHeight() / 2);
4
```

Das Prinzip mit den Sprite Sheets die auseinandergeschnitten werden und der Zählvariable lässt sich nun auf jedes dynamische Objekt anwenden (also auf den Spieler, die Coins, die Gegner und auf die

Items). Jedes dynamische Objekt hat eine eigene Animation und damit auch ein eigenes Sprite Sheet mit variierenden Längen. Dadurch auch eine eigene Art, wie die Bilder hintereinander abgespielt werden.

Eine Besonderheit ergibt sich noch bei der Spieler-Textur und der Gegner-Textur. Mit setFlip() (Methode von Sprite) wird diese umgedreht, abhängig davon, ob sich der Körper in positive oder negative x-Richtung bewegt.

Das eigentliche Zeichnen der Textur geschieht in PlayScreen.java. Hier wird ein Objekt vom Typ SpriteBatch, das ganz am Anfang in der Hauptklasse des Spiels erstellt wurde, geöffnet. Im Anschluss werden alle Objekte, die von Sprite erben, gezeichnet (dazu wird die Methode draw() von Sprite aufgerufen), dies umfasst den Spieler, alle Coins, bewegbare Plattformen, Gegner und Items (Ausschnitt aus PlayScreen.java) [Code gekürzt].

```
1
2  jumpAndRun.batch.begin();
3
4  player.draw(jumpAndRun.batch);
5
6  jumpAndRun.batch.end();
7
```

4.11 Steuerung / ein HUD Overlay

Aktuell gibt es eine Kamera (und einen Viewport), die sich in x-Richtung an den Spieler gebunden durch die Spielwelt bewegt. Um nun ein Overlay mit Buttons zum nach rechts laufen, nach links laufen und zum Springen hinzuzufügen, werden ein neuer Viewport und eine neue Kamera gebraucht. Sonst müsste man die Buttons wie ein normales Objekt in die Welt platzieren und in x-Richtung mit bewegen. Dies bringt jedoch die Buttons zum „flimmern“.

Um nun das Hud zu implementieren besitzt die dafür erstellte Klasse Hud.java eine update Methode, einen eigenen Viewport und eine eigene Kamera. Zudem benutze ich hier eine Stage, welche im Grunde genommen eine 2D Szene ist, die dazu dient 2D Bedienelemente zu organisieren. Innerhalb dieser Stage sind die Bedienbuttons sowie die Lebens- und- Zeitanzeige in Tabellen (Table) geordnet.

Damit das Overlay über dem eigentlichen Spielgeschehen angezeigt und gezeichnet wird, muss, nachdem die Welt und alle dynamischen Objekte in der render() Methode gezeichnet wurden, an den Spritebatch (Container um Texturen zusammenhängend zu zeichnen) die Kamera des Huds gebunden werden. Im Anschluss wird dann die Stage und damit alle Elemente, die sich in dieser befinden, gezeichnet.

```
1
2  //neue Kamera binden
3  jumpAndRun.batch.setProjectionMatrix(hud.stage.getCamera().combined);
4
5  //Hud zeichnen
6  hud.stage.draw();
7
```

Hier wird also 60-mal pro Sekunde erst das Spielgeschehen neu auf den Bildschirm gezeichnet und im Anschluss wird über das berechnete und angezeigt Bild das Hud über eine neue Kamera und einen neuen Viewport darübergelegt. Sozusagen werden hier also 2 separate Bilder übereinandergelegt.

5 Optimierungen

5.1 Assetmanager

Durch das Hinzufügen von Texturen ist ein Problem aufgetreten: Sobald der Spieler eine Textur sieht, die er vorher noch nicht gesehen hat, hängt das gesamte Spiel kurz, da die Bilder pro Sekunde nicht mehr 60, sondern eine niedrigere Zahl betragen (auch Mikro Ruckler genannt).

Dies liegt daran, dass die Texturen erst in den Grafikspeicher geladen werden, sobald der Spieler diese sehen könnte. Um dies zu umgehen, besitzt das Spiel wie bereits ganz am Anfang beschrieben zwei Ladephasen. Hier werden mit Hilfe des Assetmanagers Texturen vorgeladen.

Um dies zu implementieren wird in der Hauptklasse ein neues Objekt vom Typ Assetmanager angelegt, auf das man dann vom ganzen Projekt aus drauf zugreifen kann. Um nun eine Textur zu laden benutzt man folgende Methode.

```
1
2  assetManager.load("Sprite/enemies-spritesheet.png", Texture.class);
3
```

Mit folgender Zeile wird dann die vorgeladene Textur verwendet.

```
1
2  assetManager.get("Sprite/spr_coin_strip4.png", Texture.class);
3
```

Dies hat einen weiteren Vorteil, man muss nicht ständig neue Texturn erstellen, da durch den Assetmanager zum Beispiel alle Coins das gleiche Objekt vom Typ Textur besitzen. Dies spart enorm Speicher.

5.2 Dispose

Normalerweise werden alle Objekte die im Grafikspeicher abgelegt werden, aus diesem nicht wieder von selbst herausgeworfen. Selbst wenn das eigentliche Objekt nicht mehr existiert, verbleibt zum Beispiel die Textur die innerhalb dieses Objektes verwendet wurde, im Grafikspeicher. Dies betrifft auch Shapes, die komplette Welt (world im Playscreen.java) und viele weitere Objekte. Hierfür gibt es die Methode dispose(), die durch das Implementieren des Interfaces Screen somit auch im Playscreen.java überschrieben werden muss. Hier werden nun alle Objekte wieder herausgeworfen. Hier als Beispiel ein Auszug aus Playscreen.java. [Code gekürzt].

```
1
2  @Override
3  public void dispose() {
4      world.dispose();
5      map.dispose();
6      renderer.dispose();
7      b2dr.dispose();
8
9      for (Enemy enemy : mapCreator.getEnemys()){
10         enemy.dispose();
11     }
12 }
13
```

5.3 Sleep und setActive()

Wie in Kapitel 4.3 beschrieben, wurde das Objekt vom Typ World mit true für den Parameter dosleep initialisiert. Dies sorgt dafür, dass ein Körper, wenn er sich gerade nicht bewegt und keine neue Berührung stattfindet, in einen Ruhemodus versetzt werden. Dies verbessert enorm die Performance.

Des Weiteren werden alle bewegbaren Objekte zu Beginn auf inaktiv gesetzt mit folgender Zeile Code (Auszug aus Enemy.java) [Code gekürzt].

```
1
2  body.setActive(false)
3
```

Nun wird in der update Methode getestet, ob der Spieler das Objekt sieht. Wenn dies der Fall ist wird es aktiv geschalten. Sobald der Spieler es nicht mehr sieht, wird es wieder auf inaktiv gesetzt (Auszug aus Playscreen.java) [Code gekürzt].

```
1
2  if (player.getX()+3f > enemy.getX()) {
3      if (!enemy.getdestroyed()){
4          if (!enemy.b2body.isActive()) {
5              enemy.b2body.setActive(true);
6          }
7      }
8  }
9
```

Dies hat einen sehr großen Effekt auf die Performance, da dieses System auf alle Gegner und Coins angewendet wird. Vor allem von letzteren kann es schonmal über 100 in einem Level geben. Diese Optimierung kommt vor allem älteren Android-Geräten zu Gute.

5.4 Plattformübergreifend

Das Spiel an sich wird wie oben beschrieben einmal im Core-Modul geschrieben, das von verschiedenen Launchern aus gestartet wird. An sich würde ein Spiel, das für Android entwickelt ist, direkt auch für Windows startbar sein. Hierbei treten jedoch zwei Probleme auf. Zum einem die Steuerung, zum anderen die Auflösung.

Bezüglich der Steuerung gibt es zwei Mögliche Arten von Input. Zum einem die Berührung des Displays, zum anderen die Eingabe mit Tastatur. Die Tastatureingabe lässt sich einfach über folgende Zeile Code realisieren. (Auszug aus Playscreen.java) [Code gekürzt].

```
1
2  if (Gdx.input.isKeyPressed(Input.Keys.D)){}
3
```

Für die Eingabe durch ein Display gibt es folgende Methode, welche allerdings dreimal pro Zyklus abgefragt werden muss, da es auch mehrere Berührungen (Zum Beispiel laufen und springen) gleichzeitig geben kann. (Auszug aus Playscreen.java) [Code gekürzt].

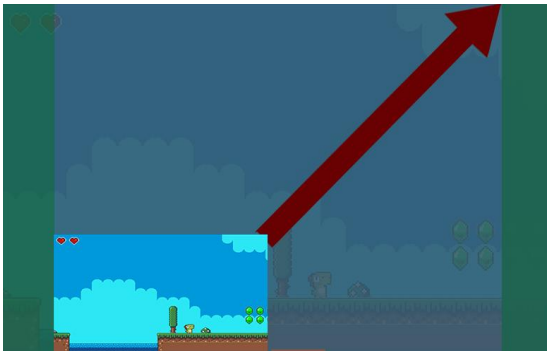
```
1
2  for (int i = 0; i < 3; i++){
3    if (Gdx.input.isTouched(i)){}
4  }
5
```

Der zweite Unterschied zwischen den Geräten ist die Auflösung. Hierfür stellt LibGDX den Viewport bereit. Genauer verwende ich in meinem Fall den ExtendViewport, da man mit diesem das Spiel auf fast jede Auflösung skaliert bekommt, ohne schwarze Balken an den Seiten zu haben.

Alle Texturen sind 16x16 Pixel groß. Dementsprechend macht es Sinn eine Auflösung zu wählen, die einem Vielfachen von 16 entspricht. Hier wird unterschieden zwischen der virtuellen Auflösung, in der das Bild berechnet wird, und der tatsächlichen Auflösung des Gerätes.

In meinem Fall ist die Virtuelle Auflösung 400x208 Pixel (25 Texturen breit und 13 Texturen hoch, da eine Textur 16 Pixeln entspricht). Dieses virtuelle Bild wird dann bei gleichbleibendem Seitenverhältnis hoch skaliert auf die reale Auflösung des Gerätes. Da man jedoch nicht jede Auflösung mit gleichbleibenden Seitenverhältnis durch hochskalieren bekommen kann, entstehen bei einem normalen Viewport dann schwarze Balken.

Um diese schwarzen Balken zu vermeiden, nutze ich den ExtendViewport und eine Karte, die nicht 13 Texturen hoch ist, sondern 15. Denn dieser Viewport nutzt den Bereich wo schwarze Balken entstehen würden und füllt diesen Bereich mit der Spiel Welt soweit es geht. Somit sieht man also, wenn man eine ungünstige Auflösung hat, etwas mehr von der Welt. Hier im Bild steht der rote Pfeil für das Hochskalieren mit einem Seitenverhältnis, das beibehalten wird. Da die Zielauflösung in dem Fall aber zu breit ist, sieht er mehr von der Spielwelt (grüne Balken) in x-Richtung. Durch dieses System wird nahezu jede Auflösung unterstützt, ohne das schwarze Balken entstehen.



5.5 Fehler innerhalb des Box2D Systems

Während der Spielentwicklung bin ich auf mehrere Fehler innerhalb des Box2D Systems gestoßen. Zum einem konnte der Spieler (zu dem Zeitpunkt war der Spieler noch ein Quadrat und nicht rund) an dem Übergang zweier Hitboxen hängen bleiben, obwohl diese auf der gleichen Höhe waren. Dies ging so weit, dass man sich gar nicht mehr bewegen konnte. Um dies zu umgehen habe ich die Hitbox des Spielers rund gemacht. Dadurch bewegt er sich nur leicht nach oben (kaum sichtbar) anstatt an einem Übergang hängen zu bleiben. Dies sorgt dafür, dass der Fehler fast gar nicht mehr auftritt.

Zudem kann es zu flimmernden Texturrändern kommen. Die Welt besteht ja aus 16x16 Pixel großen Texturen, die in einem Raster angeordnet sind. Es kann vorkommen, dass eine Textur fehlerhaft aus dem Sheet ausgeschnitten wird. Dadurch entstehen schwarze Balken. Um dies zu vermeiden, sind die Texturen auf dem Sheet nicht 16x16px groß, sondern haben eine Zone von 2px um die eigentliche Textur drum herum, wo diese weitergeführt ist. Durch diesen Toleranzrand entstehen selbst wenn die Textur falsch ausgeschnitten wird keine schwarzen Ränder.

5.6 Game Design

Wie bereits am Anfang erwähnt ist Box2D eine Physikberechnung, die an die echte Welt angelehnt ist. Im Laufe der Entwicklung habe ich immer mehr gemerkt, dass ein Jump and Run nicht allen physikalischen Regeln folgt. Zum Beispiel fällt einem bei näherem Betrachten von zum Beispiel Super Mario, das Mario sich mehr wie eine Feder verhält und weniger wie ein Körper mit Gewicht. Dies habe ich so gut wie es geht mit Funktionen innerhalb des Frameworks und Box2Ds auszugleichen. So habe ich dem Spieler zum Beispiel einen linearen Dämpf Effekt gesetzt.

Ein weiteres Problem stellt nach wie vor das Springen dar. Realistisch wäre es, das nachdem man eine größere Strecke fällt, durch den Aufprall Geschwindigkeit in x-Richtung durch die Reibung und die y-Geschwindigkeit verliert. Dies wird auch durch Box2D simuliert, jedoch passt dies nicht zu einem Jump and Run. Hier müsste es so sein, dass man die Geschwindigkeit in x-Richtung beibehält. Also müsste hier an einem Punkt die Physikberechnung geändert werden bzw. aussetzen. Da dies jedoch nicht möglich, mussten hier Tricks angewendet werden:

Sobald der Sensor mit der ID „foot“ (siehe 3.7 mehrere Shapes an einem Körper) den Boden berührt, wird die y-Geschwindigkeit direkt auf 0 gesetzt und die x-Geschwindigkeit wird beibehalten. Kurz bevor der Spieler also den Boden berührt wird ihm seine Geschwindigkeit in y-Richtung genommen. Damit wird er dann auch nicht mehr langsamer in x-Richtung, da es so ist, als ob er nur einen Pixel gefallen wäre. Somit entsteht nicht so eine starke Reibung und der Spieler wird nicht verlangsamt.

6 Schluss mit Fazit

Insgesamt ist es gut möglich, mit LibGDX ein Jump and Run zu erstellen. Funktionen wie das Tiled System und die einfache Übersetzung in Box2D machen ein Level basiertes Spiel einfach umsetzbar innerhalb des Frameworks.

Die Box2D Bibliothek bringt einige elementare Funktionen mit, auf deren Grundlage man ein Jump and Run implementieren kann. Dazu gehören die sehr effiziente Kollisionsabfrage, die einfache Möglichkeit, Körper mit unterschiedlichsten Eigenschaften zu erstellen oder schlicht weg überhaupt die Möglichkeit, über eine Plattform zu laufen ohne durchzufallen.

Box2D ist jedoch ein in sich geschlossenes System um physikbasierte Spiele umzusetzen. Da ein Jump and Run jedoch nicht allen Regeln der Physik folgt und man von außen nicht mehr in die Physikberechnung eingreifen kann, sind hier an einigen Stellen Tricks erforderlich.

Durch die Schnittstelle zu OpenGL und die Option, selbst entscheiden zu können, wann Texturen in dem Grafikspeicher landen oder wieder rausfliegen, kann man das Spiel durch Ladephasen so konzipieren, dass es selbst auf älteren Android-Geräten noch flüssig und einwandfrei läuft.

Die am Anfang beschriebenen Anforderungen konnten komplett umgesetzt werden. Sowohl die plattformübergreifende Entwicklung, welche unter anderen durch den Viewport realisiert wurde als auch die Modularität, welche durch das Vererbungskonzept implementiert wurde entsprechen exakt den Anforderungen. Es ist möglich, unterschiedlichste Level und sogar ein Weltensystem nur auf Basis der XML-Datei zu bauen, da sowohl die verwendeten Texturen als auch die Art der Objekte (durch Objektgruppen) in dieser XML-Datei gespeichert sind.

Die Box2D Bibliothek hat zwar Fehler und Schwachstellen, wie zum Beispiel das Problem mit dem Hitboxübergang, diese lassen sich aber durch cleveres Nutzen der gegebenen Elemente weitestgehend umgehen bzw. minimieren.

Die ursprüngliche Spielidee konnte fast komplett umgesetzt werden. Lediglich einige wenige Funktionen wurden nicht umgesetzt, da diese den Rahmen dieser Arbeit übersteigen würden. Diese sind jedoch im Code als //TODO vermerkt.

7 Anhang

7.1 Code

Der Code umfasst mehrere tausend Zeilen und ist damit zu lange um ihn hier anzuhängen. Zudem passen zu wenige Zeichen in eine Reihe, was das ganze unübersichtlich machen würde, weshalb der Code auf Github hochgeladen wurde. LINK!!!!!!!!

7.2 Quellen

Für und in dem Spiel verwendete Tools

https://libgdx.badlogicgames.com/

https://www.mapeditor.org

https://github.com/libgdx/libgdx/wiki/Texture-packer

https://github.com/libgdx/libgdx/wiki/2D-Particle-Editor

In dem Spiel verwendete Texturen und Sheets

https://rottingpixels.itch.io/nature-platformer-tileset

https://rottingpixels.itch.io/enemy-characters-pack-free

https://opengameart.org/content/free-ui-asset-pack-1

https://flyclipart.com/heart-sprite-pixel-art-maker-sprite-logo-png-350324

https://arks.itch.io/dino-characters

https://thewisehedgehog.itch.io/thip

https://reff-sq.itch.io/slime-animations

Offizielle Dokumentationen

https://doc.mapeditor.org/en/stable/

https://github.com/libgdx/libgdx/wiki/Tile-maps

https://libgdx.badlogicgames.com/documentation/help/Documentation.html

Projekte, durch die ich gelernt habe

https://gamefromscratch.com/libgdx-tutorial-13-physics-with-box2d-part-2-force-impulses-and-torque/

https://github.com/Icohedron/LibGDX-Block-Bunny

https://github.com/oakes/libgdx-examples

http://moribitotechx.blogspot.com/p/tutorial-series-libgdx-mtx.html

https://github.com/libgdx/libgdx-demo-superjumper

https://github.com/wmora/martianrun

https://github.com/BrentAureli/SuperMario

https://github.com/rolangom/ruwimp

Spezielle Probleme, die auf Grundlage von Stack Overflow gelöst wurden

https://stackoverflow.com/questions/28000623/libgdx-flip-2d-sprite-animation

https://stackoverflow.com/questions/32626986/libgdx-texture-bleeding-issue

https://stackoverflow.com/questions/47164432/libgdx-ppm-world-conversion

https://stackoverflow.com/questions/34053532/jumping-slows-x-velocity-jump-and-run-with-libgdx-box2d

https://stackoverflow.com/questions/35857399/libgdx-how-to-control-touch-events-for-jumping-multiple-times/35857543

https://stackoverflow.com/questions/15733442/drawing-filled-polygon-with-libgdx

https://jvm-gaming.org/t/box2d-best-way-to-implement-rolling-friction-resistance/42177

https://stackoverflow.com/questions/53207557/black-stripes-in-libgdx