UNIKASSEL
VERSITÄT

Fachbereich 16
Fachgebiet Software Engineering

Bachelorarbeit

im Bereich Informatik

über das Thema

Entwicklung von mobilen Applikationen unter Verwendung von Cross-Platform Game Engines

Autor: Tobias Gries
29201929
gritob@web.de

Prüfer: Prof. Dr. Albert Zündorf

Zweitprüfer: Prof. Dr. Lutz Wegner

Betreuer: M.Sc Tobias George

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich meine Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum: .....

(Unterschrift)

I Kurzfassung

Im Rahmen dieser Arbeit soll untersucht werden, inwiefern Cross-Platform Software Development Kits (folgend SDKs) im Bereich Game-Development für mobile Endgeräte mögliche Alternativen gegenüber der nativen Programmierung bieten. Dabei wird unter Verwendung von drei ausgewählten Development Kits jeweils die gleiche Applikation entwickelt, die möglichst identisch gehalten werden soll. Ziel von jedem SDK soll es sein, am Ende mit einer Code-Basis mehrere Plattformen ansprechen zu können. Die daraus entstandenen Applikationen sollen sich idealerweise weder in Performance noch im Look & Feel unterscheiden. Bei den ausgewählten SDKs müssen dabei mindestens die mobilen Plattformen Android und iOS unterstützt werden. Eine weitere Anforderung ist die Implementierung von grundlegenden Funktionen wie das Ansprechen von Sensorik, die Verwendung von Sound, Animationen und das Abspeichern von persistenten Daten. Am Ende werden daraufhin die verwendeten SDKs in einem Vergleich gegenübergestellt. Dazu werden die Unterschiede während der Entwicklungsphase näher erläutert, inwieweit plattformspezifischer Code eingebunden werden musste, die Handhabung der SDKs selbst, sowie der Vergleich auf den Endgeräten auf Basis von Android und iOS.

II Inhaltsverzeichnis

III Abbildungsverzeichnis

IV Listing-Verzeichnis

1 Einleitung

Mit der Einführung des ersten iPhone von Apple hat sich im Bereich der Spielindustrie viel geändert. Aus der Menge an verfügbaren Betriebssysteme für Handys und Tablets haben sich das Android Betriebssystem von Google und Apples iOS herauskristallisiert. Alle Applikationen, die heute in den verschiedenen Marktplätzen[1] zu finden sind, beinhalten in den meisten Fällen eine Version für Android, als auch eine Version für iOS. Aufgrund des großen Marktanteils beider Unternehmen im mobilen Sektor ist es für einen Entwickler besonders interessant, seine Applikationen für beide Plattformen anzubieten. Jedoch ist die native Programmierung für beide Plattformen in den meisten Fällen sehr aufwändig und erfordert viel Zeit oder ein großes Projektteam. Dies liegt an den technologischen Unterschieden zwischen den verschiedenen Systemen als auch an der großen Diversität von mobilen Geräten. Große Unternehmen haben gegenüber einer kleinen Entwicklergruppe meist die finanziellen Möglichkeiten eine Applikation nativ für beide Systeme zu entwickeln. Entwickler von Cross-Plattform SDKs werben mit hoher Effizienz, Geschwindigkeit und einfacher Handhabung ihrer Frameworks. Für selbstständige oder eine kleine Gruppe von Entwicklern wäre es daher besonders interessant Applikationen mit Hilfe solcher Frameworks zu entwickeln, da sie im Gegensatz zu großen Entwicklerstudios nicht die selben Ressourcen zur Verfügung haben, um jede Applikation mehrmals zu entwickeln. Andrew Burget gibt in seinem Vortrag über native und hybride Entwicklungsstrategien Aufschluss darüber, inwiefern die Implementierung mit Hilfe von Multiplattform-Frameworks vorteilhaft gegenüber der nativen Programmierung sein kann (siehe [1]). Darunter beschreibt er den Vorteil einer einzelnen Codebasis, die eine einfachere Abstraktion von der eigentlichen Entwicklerplattform definiert. Weiterhin sieht er Vorteile in der Verwendung von Cross-Plattform SDKs in kleinen Projekten, wo die Gruppe der Geräte für welche entwickelt wird, klar definiert ist. Er beschreibt zwar, dass die native Programmierung für Spiele und grafikintensive Applikationen besser geeignet ist, geht dabei jedoch von sehr großen Projekten aus, die im Bereich der mobilen Spielentwicklung nur selten zu finden sind. Bei den erfolgreichsten Spielen in diesem Bereich handelt es sich um Gelegenheitsspiele, welche für eine möglichst große Zielgruppe entwickelt wurden.

Inwiefern Cross-Platform SDKs mit dem Fokus auf Game-Development eine mögliche Option bieten, um eine konkurrenzfähige Applikation entwickeln zu können und wie der aktuelle Entwicklungsstand ausgesuchter Frameworks ist, soll in dieser Arbeit näher untersucht werden. Dies ist vor allem interessant, da der aktuelle Forschungsstand untersuchter Cross-Platform Frameworks in diesem Bereich eher gering ausfällt. Dadurch besteht durch diese Arbeit ein Informationsgewinn für kleinere Entwicklergruppen und einzel-

[1] elektronischer Marktplatz für Online-Handel von Software

ne Entwickler, welche ihre mobilen Applikationen unter Verwendung von Cross-Platform SDKs erstellen möchten. Das Thema ist weiterhin von großer Bedeutung, da es in naher Zukunft nicht absehbar ist, dass die Konkurrenz im Bereich des mobilen Marktes zwischen Android und Google mit einem entscheidenden Gewinner hervorgeht. Dadurch ist es in Zukunft möglich und heute auch schon zu großen Teilen praktiziert, dass geschriebene Applikationen für mobile Geräte für beide Betriebssysteme erwartet werden, um einen allgemeinen Standard zu erfüllen. Durch die Verwendung von Cross-Platform SDKs besteht die Möglichkeit, dass dieses Problem für keine Entwicklergruppen nahezu komplett verschwindet und je nach Entwicklungsstand dieser Frameworks, sie in Zukunft als Standard für die Entwicklung von Applikationen im mobilen Sektor angesehen werden können.

Zu Beginn der Ausführungen werden drei Cross-Platform Frameworks ausgewählt, welche sich im Bereich der mobilen Spielentwicklung bereits etabliert haben. Danach werden die Mindestanforderungen an eine geeignete Applikation definiert, die von den Frameworks realisiert werden können. Sie sollen die Grundfunktionen einer möglichen Anwendung abdecken, die ihren Fokus im Bereich der Spielentwicklung sieht. Im nächsten Schritt werden diese Anforderungen in einer Spielidee zusammengefasst, gefolgt von Designentscheidungen, um alle notwendigen Vorkehrungen für die Umsetzung in den Frameworks zu treffen. In der Entwicklungsphase wird die Implementierung unter Verwendung der ausgewählten SDKs durchgeführt und beschrieben. Daraufhin wird in einem weiteren Kapitel das Deployment auf die Testgeräte mit iOS und Android erläutert. Im letzten Schritt werden die Applikationen und die Frameworks einem Vergleich unterzogen, um in einem abschließendem Fazit die Fähigkeiten von Cross-Platform Game Engines zu bewerten.

2 Grundlagen

Bevor die verwendeten Frameworks erklärt werden und die Implementierung begonnen wird, müssen einige Begrifflichkeiten erläutert werden, da sie im Laufe der Ausführungen öfters Verwendung finden.

Motion Tweening

Motion Tweening beschreibt das Animieren eines Objektes zwischen zwei Punkten. Damit bei einer Animation nicht jedes Bild einzeln festgelegt werden muss, wird zu einem Startpunkt ein Endpunkt mit festgelegter Zeit definiert. Zwischen diesen beiden Punkten werden daraufhin alle Punkte interpoliert und ergeben eine Animation. Ein Start- und Endpunkt müssen dabei nicht Positionen in einem Koordinatensystem sein, sondern sind eher als Zustände eines Objektes zu sehen. Damit können auch Animationen, wie beispielsweise die Größe oder der Alphawert über Motion Tweening interpoliert werden.

Animierte Sprites

Sprites beschreiben ein zweidimensionales Bild, welches als einfache Grafik in eine größere Szene gesetzt wird. Animierte Sprites hingegen beschreiben eine Grafik mit mehreren Zuständen innerhalb eines einzelnen Bildes (sprite sheet), die richtig zusammengesetzt eine Animation ergeben. Dabei müssen die Koordination der einzelnen Frames in der Grafik bekannt sein. Mehrere verschiedene Animationen können innerhalb eines Bildes zusammengefasst sein und über die Auswahl der richtigen Koordinaten nacheinander abgespielt werden. Eine weitere Version animierte Sprites zu realisieren ist eine Menge von Einzelbildern, die nacheinander eingelesen die gewünschte Animation ergeben.

Easing

Im Bezug auf das beschriebene Motion Tweening beschreibt Easing die Funktion, welche die Interpolation zwischen zwei Punkten übernimmt. Ein lineares Easing beschreibt dabei einen stetig gleichbleibenden Verlauf von Start- zu Endpunkt. In den meisten Fällen folgt die Interpolation zwischen diesen Punkten einer mathematischen Funktion, da diese leicht zu implementieren sind.

Abbildung 1: Easing Bounce[1]

[1] Quelle: http://docs.coronalabs.com/api/library/easing

In Abbildung 1 sind weitere Möglichkeiten zur Interpolation als Beispiel dargestellt. Die x-Richtung stellt dabei die Zeit dar, wohingegen die y-Richtung von unten nach oben gesehen den Start- und Endpunkt darstellen.

Szenegraph

Der Szenegraph ist ein gängiger Begriff innerhalb von 3D Anwendungen. Er stellt das Wurzelelement einer Szene dar, an welchem alle weiteren Objekte als Kinderelemente gehangen werden. Dabei können diese Elemente weitere Elemente zu einer Gruppe zusammenfassen. Der Szenegraph folgt einer Baumstruktur, vergleichbar mit einer Ordnerstruktur in einem Dateisystem.

2.1 Überblick Cross-Platform SDK

Cross-Platform SDKs definieren Frameworks, unter welchen eine bestimmte Applikation für verschiedene Plattformen gleichzeitig programmiert werden kann. Plattform beschreibt dabei eine bestimmte Computerarchitektur, ein Betriebssystem oder einen Zusammenschluss aus beiden Faktoren. Ein Framework, welches mindestens zwei unterschiedliche Plattformen unterstützt, kann als Cross-Platform SDK definiert werden. Das Unterstützen mehrerer Plattformen lässt sich weiterhin in zwei Gruppen untergliedern. Zum einen kann ein Framework, welches für verschiedene Plattformen installiert werden kann und eine bestimmt Aufgabe erfüllt, als solch ein SDK definiert werden. Weiterhin kann ein Framework die Eigenschaft besitzen, dass die entwickelten Applikationen von mehr als einer Plattform unterstützt werden. Die Frameworks, welche innerhalb dieser Arbeit verwendet werden, sind Teil beider Gruppen von Cross-Platform SDKs. Sie können auf den Betriebssystemen Microsoft Windows und Mac OS X installiert werden und unterstützen mindestens die mobilen Plattformen Android und iOS.

2.2 Corona

Das Studio hinter dem Corona SDK wurde 2008 gegründet. In der aktuellen Version des Frameworks werden die Plattformen Android, iOS, Kindle Fire und NOOK unterstützt. Des Weiteren wirbt das Unternehmen hinter dem SDK mit einer zukünftigen Untersützung für das Betriebssystem Windows Phone 8. Das Framework setzt zudem auf Standards wie OpenGL, OpenAL oder der Physikengine Box2D und wirbt zudem mit einer Zahl von über 300.000 Entwicklern weltweit, die mit Hilfe des Frameworks verschiedenste Applikationen entwickeln. Das Corona SDK ist neben der Entwicklung für Spiele ebenfalls für Business-Applikationen und eBooks entwickelt worden. Neben einer Vielzahl von Lizenzen mit einzigartigen Features wird auch eine kostenlose Starterlizenz angeboten, welche keiner zeitlichen Begrenzung unterliegt. Die verwendete Programmiersprache des Frameworks ist Lua[1], welche in den meisten Fällen als eingebettete Skriptsprache für andere Programme verwendet wird. Das Corona SDK ist eines der ausgewählten Frameworks, da es durch seine große Anzahl an Entwicklern eines der meist verwendeten Frameworks im Bereich der Cross-Platform Development Kits ist.

2.3 Gideros

Das Gideros SDK unterstützt die Plattformen iOS und Android und steht für die Betriebssysteme Windows und Mac OS X zur Verfügung. Das Framework wirbt mit besonders hoher Effizienz, Geschwindigkeit, schnellen Testumgebungen und mit der Möglichkeit die entwickelten Applikationen auf mehreren Plattformen zu verwenden. Das Developement Kit steht kostenlos auf der Webseite des Unternehmens zur Verfügung und erlaubt mit einer Community-Lizenz das Entwickeln von Android und iOS Applikationen. Demgegenüber besteht die Möglichkeit, eine jährliche Lizenz zu erwerben unter welcher neue Features des Frameworks zur Verfügung stehen. Die Programmiersprache unter welcher mit dem SDK entwickelt werden kann ist Lua. Gideros wird oft in Konkurrenz zu dem Corona SDK gesehen, da beide Frameworks mit Verwendung der selben Programmiersprache das gleiche Ziel verfolgen. Das Framework hat besonders an Popularität gewonnen, als bei dem Corona SDK noch keine freie Starterlizenz zur Verfügung stand und Gideros somit eine kostenlose Alternative dargestellt hat. Aus diesen Gründen ist das Gideros SDK eines der Frameworks, unter welchem die Verwendung von Cross-Plattform Game Engines untersucht wird.

[1] `www.lua.org`

2.4 OpenFL

OpenFL steht für Open Flash Library und definiert ein Open Source Framework unter der MIT Lizenz (siehe [14]), welches auf der Flash API aufbaut, jedoch nicht auf das Flash Plugin angewiesen ist (siehe [13]). Das Framework unterstützt eine sehr große Anzahl an Plattformen, für welche die Applikationen erstellt werden können. Darunter zählen Windows, Mac, Linux, iOS, Android, Blackberry, Tizen, Flash und zu einem Teil HTML5. OpenFL war zuvor unter NME bekannt und wirbt mit schneller Entwicklungszeit und einfacher Handhabung. Das Framework steht unter Linux, Windows und Mac zur Verfügung und verwendet als Programmiersprache Haxe[1]. OpenFL wird als eines der ausgewählten Frameworks verwendet, weil es nahezu alle großen Plattformen unterstützt, mit der Open Source Philosophie im Kontrast zu den teilweise kommerziell vertriebenen Frameworks steht, sich seit der Änderung von NME zu OpenFL in ständiger Entwicklung befindet und bereits eine große Anzahl von Applikationen mit den Frameworks realisiert wurden.

[1] `www.haxe.org`

3 Design

In der Designphase werden die grundlegenden Mindestanforderungen an die Applikationen und den Frameworks erarbeitet. Diese Anforderungen fließen daraufhin in einem nächsten Schritt in einer Spielidee zusammen. In diesem Zusammenhang wird ebenfalls das Design des Spiels offengelegt, damit für die spätere Entwicklungsphase alle Vorkehrungen getroffen sind, um die Implementierung beginnen zu können.

3.1 Anforderungen

Um bei Fertigstellung der Applikation einen guten Vergleich unter den SDKs und den mobilen Geräten ziehen zu können, müssen einige Anforderungen von den SDKs und der daraus erstellten Applikationen erfüllt werden. Diese Anforderungen beziehen sich zum größten Teil direkt auf die Fähigkeiten der Frameworks. Da diese Frameworks ihren Fokus auf die Spielentwicklung legen, handelt es sich hier um grundlegende Funktionen, die zusammen kombiniert ein konkurrenzfähiges Produkt ergeben können, welches im Playstore[1] von Android oder im Appstore[2] von Apple bestehen kann. Bei diesen Funktionen kann es sich beispielsweise um mögliche Eingabeoptionen wie die Eingabe über das Display per Touch, Swipe[3] oder über Sensorik handeln. Weiterhin sind Sound und Musik sowie Möglichkeiten für die Erzielung von Animationen unbedingt erforderlich.

Auflösung

Gerade bei Handys und Tablets mit Android als Betriebssystem gibt es eine sehr große Auswahl an verschiedenen Displaygrößen und daraus resultierend verschiedene Auflösungen, die unterstützt werden müssen. Hinzu kommen die mobilen Geräte mit iOS als Betriebssystem, von denen im Vergleich zu den Android Endgeräten deutlich weniger existieren. Dennoch bringen die Handys und Tablets von Apple weitere verschiedene Displaygrößen mit sich und verlangen durch das teilweise verbaute Retina-Display auch möglichst hochauflösende Grafiken. Damit am Ende auch ein einheitliches Look & Feel erreicht werden kann, wird an die verwendeten Cross-Platform Frameworks die Anforderung gestellt, mögliche Optionen zu bieten, um mit sehr vielen verschiedenen Auflösungen umgehen zu können und dahingehend die verwendeten Grafiken passend zu skalieren.

[1] Online-Marktplatz von Google für Applikationen, Film und Musik
[2] Online-Marktplatz von Apple für Applikationen, Film und Musik
[3] Eingabe mittels Wischbewegung über das Display

Sensorik

An der Aufgabe ein Handy oder Tablet ohne Sensorik zu finden wird man mit hoher Wahrscheinlichkeit scheitern. Die heute wichtigsten verbauten Sensoren sind zum einen der Sensor zur Erkennung der Benutzereingabe über den Touchscreen und zum anderen der Beschleunigungssensor. Letzterer wird in der Spielentwicklung besonders oft verwendet um Objekte zu bewegen, die kontinuierlich von Benutzer gesteuert werden müssen. Das Verwenden dieses Sensors ist daher als grundlegende Funktion zu sehen, die von den Development Kits verlangt werden muss. Die spätere Applikation soll in der Entwicklungsphase offenlegen, inwiefern solche Sensoren angesprochen und wie einfach die Daten ausgelesen werden können. Das Spiel soll dazu die Sensorwerte auslesen und geeignet darauf reagieren. Dabei ist es besonders interessant zu sehen, ob verschiedene Geräte auch unterschiedliche Werte liefern. Gerade zwischen Android und iOS könnte es zu Unterschieden kommen, da es sich hier um zwei unterschiedliche Systeme auf unterschiedlichen Geräten handelt. Es darf nicht vergessen werden, dass die Entwicklung pro SDK auf einer Codebasis geschehen soll und das Framework bei unterschiedlichen Werten auch Optionen darbieten muss, um darauf entsprechend reagieren zu können. Weiterhin soll beobachtet werden, ob die Frameworks untereinander auf die selbe Art und Weise die geforderten Werte zurückliefern oder ob es marginale Unterschiede gibt. Die Eingabe über Touch ist dabei als Standardfunktion vorauszusetzen, da die heutigen Geräte in den seltensten Fällen mit einer echten Tastatur konstruiert werden und keine andere Möglichkeit zur Eingabe besitzen.

Menu & Buttons

Unabhängig davon, ob es sich um ein Spiel oder eine Business-Applikation handelt, wäre es aus designtechnischen Gesichtspunkten schlecht, wenn beim Starten des Programms direkt der Kern der eigentlichen Applikation startet. Um dies zu umgehen, wird ein Startmenü benötigt, mit dem vorher entsprechende Einstellungen vorgenommen werden können und das Spiel im Anschluss gestartet werden kann. Um auf Android und iOS auch das gleiche Look & Feel zu erhalten, dürfen im Menü und auch an keiner Stelle im Programm die spezifischen Buttons des Betriebssystems verwendet werden, weil diese sich im Design stark unterscheiden, wie in Abbildung 2 zu sehen ist.

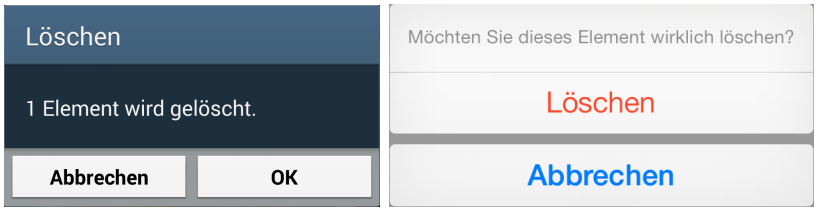

Abbildung 2: Vergleich Buttons Android 4.3 & iOS 7.0.4

Folglich müssen spezifische Grafiken verwendet werden, welche als Buttons fungieren. Im Normalfall ist dies auch die bessere Vorgehensweise, weil die verwendeten Bedienelemente einer Applikation immer an das generelle Design angepasst werden sollten, um einen einheitlichen Look zu gewährleisten. Da es sich hier um ein Spiel mit eigenständigen Design handeln soll, ist diese Vorgehensweise entsprechend zu bevorzugen.

Persistenz

Die Persistenz von Daten ist eine weitere Funktion, die von den Frameworks gefordert wird. Daten die beispielsweise dauerhaft gespeichert werden können sind in den meisten Fällen gesonderte Einstellungen eines Benutzers (Profil) oder der aktuelle Spielstand in einem Spiel. Wie viele oder auch welche Daten gespeichert werden ist für die geplante Applikation nicht relevant, sofern generell Daten gespeichert werden und die Funktion somit zum Tragen kommt. Auch hier wird es interessant zu sehen, inwiefern die Frameworks diese Funktion auf verschiedenen Systemen mit einem Code realisieren. Besonders bei Apple ist sehr stark vorgeschrieben, in welche Ordner auf dem System Daten geschrieben werden dürfen und in welche nicht. Werden Daten an unerwünschten Stellen gespeichert, so kann es dazu führen, dass die geschriebene Applikation von Apple abgelehnt wird und somit nicht in den Appstore gestellt werden darf. Möchte man im Gegensatz dazu bei Android Daten speichern, so müssen während der Installation gesonderte Rechte angefordert werden. Im Verlauf der Entwicklungsphase soll dann näher untersucht werden, wie diese Funktion von den einzelnen Frameworks umgesetzt wird.

Sound

Sound inklusive Musik sollte für ein Spiel selbstverständlich sein, weswegen dies als eine Mindestanforderung festgelegt wird. In der späteren Entwicklungsphase muss darauf geachtet werden, dass entweder Soundformate verwendet werden, die von beiden Systemen verstanden werden können, oder dass die Frameworks Möglichkeiten bieten, um zwischen verschiedenen Formaten je nach Android oder iOS zu unterscheiden. An welchen Stellen im Programm Sounds und Musik verwendet werden hängt von der Idee und Design des Spiels ab.

Animation

Für einen soliden Prototypen kommt ein Spiel auch komplett ohne Animationen aus. Animationen sind jedoch für konkurrenzfähige Applikationen nicht wegzudenken und stellen somit eine wichtige Funktion dar. Animation wird im Normalfall durch das Verwenden einer Physikbibliothek, durch animierte Sprites oder durch Motion Tweening realisiert.

Welche Art der Animation verwendet wird hängt hier auch wie beim Sound vom Design und der Idee des Spiels ab. Motion Tweening wird dennoch in jedem Fall verwendet werden um kleinere Effekte zu erzeugen oder Objekte gezielt zwischen zwei Punkten zu bewegen.

Schriftart

Eine eigens ausgesuchte Schriftart ist ebenfalls ein Aspekt, der unterstützt werden soll, um ein einheitliches Look & Feel zu gewährleisten. In den meisten Spiel werden in irgendeiner Art und Weise Texte verwendet, die Informationen vermitteln sollen. Sei dies eine Punktzahl, eine ablaufende Zeit einer Runde oder ein einfacher Text, um den Namen eines Levels zu beschreiben. Werden dafür keine echten Grafiken verwendet, so muss immer auf eingebettete Fonts[1] zurückgegriffen werden. Echte Grafiken bieten weiterhin den Vorteil, dass sie sehr leicht skalierbar sind, da es sich um Vektorgrafiken handelt. Wird keine eigene Schriftart in das Spiel integriert, so wird die aktuell eingestellte Font des Betriebssystems verwendet. Dadurch kann die Schrift auf jedem Gerät und je nach Nutzer anders sein und das Spiel ist nicht mehr einheitlich. Weiterhin kann es dabei zu Problemen in der Darstellung kommen, da nicht jede verwendete Schriftart alle gewünschten Zeichen enthält. Unterschiede gibt es auch in den Schriftgrößen, da diese ebenfalls nicht einheitlich gehalten sind. Die Frameworks müssen daher in der Lage sein, neue Schriftarten einbetten zu können, Optionen zur Bearbeitung anzubieten und diese auf den verschiedenen Betriebssystemen zur Verfügung zu stellen.

Skalierbarkeit

Skalierbarkeit beschreibt hier nicht den Inhalt im bereits angesprochenen Punkt *Auflösung*, sondern bezieht sich auf die Größe der Spielumgebung. Um am Ende der Entwicklung einen geeigneten Performancevergleich durchführen zu können, muss das Spiel selbst skalierbar sein. Beispielhaft könnte dies ein Schachspiel sein, welches in seiner Standardausführung ein Spielfeld mit 8x8 Feldern darstellt. Durch geeignete Skalierbarkeit kann das Feld auf größere Proportionen erweitert werden, wodurch die Framerate des Geräts sehr leicht manipuliert werden kann und leichter Vergleiche angestellt werden können. Skalierbarkeit ist keine Anforderung an die SDKs, da es sich bei diesem Unterpunkt um eine Designentscheidung handelt und somit als Anforderung an die eigentliche Applikation zu sehen ist.

[1] Schriftart (engl.)

3.2 Spielidee und Design

Bei der ausgewählten Spielidee wird sich an dem Prinzip von *Match 3 Games* orientiert. Bei diesem Spielprinzip existiert ein Spielfeld in ungefährer Größe eines Schachbretts, welches mit Spielsteinen verschiedenster Farben gefüllt ist. Der Spieler besitzt nun die Möglichkeit einen Spielstein mit einem benachbarten Stein in horizontaler oder vertikaler Richtung zu tauschen. Dieser Tausch wird vom Spiel dennoch nur akzeptiert, wenn durch das neu entstandene Spielfeld mindestens drei Spielsteine einer Farbe in einer Reihe oder einer Spalte zustande kommen. Die kombinierten Spielsteine verschwinden daraufhin und die überliegenden Steine füllen den leeren Platz aus, indem sie von oben nachgeschoben werden. Je mehr Spielsteine einer Farbe kombiniert werden, desto mehr Punkte erhält der Spieler. Die wohl bekanntesten Spiele dieser Kategorie sind Bejeweled (siehe Abbildung 3) und Candy Crush Saga (siehe Abbildung 4).

Abbildung 3: Bejeweled[1]

Abbildung 4: Candy Crush Saga[2]

Dieses Spielprinzip eignet sich perfekt für eine Umsetzung auf Handys oder Tablets. Um alle Mindestanforderungen erfüllen zu können und dem Spiel etwas Individualität zu geben, wird das Spielprinzip dennoch etwas verändert. Das Tauschen von zwei Spielsteinen wird durch eine erweiterte Variante ersetzt. Bei dieser Modifikation werden nicht zwei Steine getauscht, sondern vier, die sich gleichzeitig im Uhrzeigersinn verschieben. Je nachdem, welcher Spielstein ausgewählt wird und in welche Richtung er bewegt werden soll, befindet sich dieser an einer anderen Position der vier Steine, welche gedreht werden. Damit der Schwierigkeitsgrad durch diese Änderung nicht zu stark ansteigt, ist es dem Spieler erlaubt vier Steine miteinander zu tauschen, auch wenn nach Ende des Spielzuges keine Kombination aus mindestens drei Steinen zusammenkommt. Dadurch wird das Spiel strategischer, weil nun die Möglichkeit besteht durch geschicktes Drehen besonders vie-

[1] Quelle: http://blog.popcap.com/wp-content/blogs.dir/3/2012/05/BHDAppStore5.jpg
[2] Quelle: http://images.eurogamer.net/2014/usgamer/Candy-Crush-Saga-Screenshot-01.jpg

le oder große Kombinationen auf einmal zu generieren, wenn der Spieler gezielt darauf hinarbeitet. Abbildung 5 zeigt einen möglichen Spielzug, um das Prinzip zu verdeutlichen.

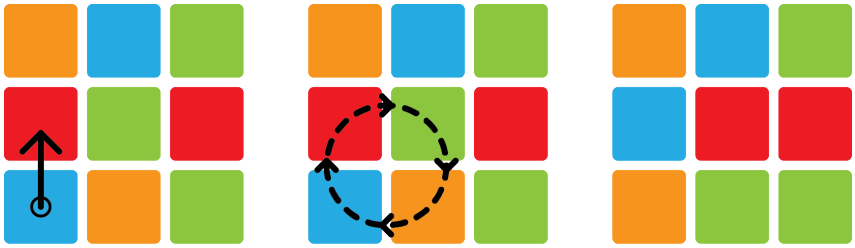

Abbildung 5: Möglicher Spielzug

Im linken Abschnitt der Abbildung wird exemplarisch in einem 3x3 großen Spielfeld der blaue Stein vom Spieler ausgewählt und nach oben bewegt. Dadurch nimmt der ausgewählte Stein die Position an der linken unteren Ecke des Vierecks ein, welches daraufhin im mittleren Teil der Abbildung gedreht wird. Das Resultat der vom Spieler gewünschten Drehung ist im letzten Abschnitt zu sehen. Die Möglichkeit, nur im Uhrzeigersinn zu drehen führt allerdings dazu, dass bestimmte Spielzüge nicht erlaubt sein dürfen. Diese unerlaubten Züge sind an den Rändern des Spielfeldes zu finden und beispielhaft in Abbildung 6 dargestellt.

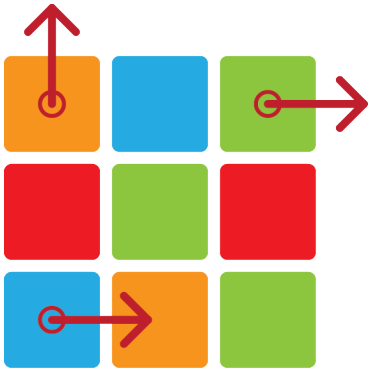

Abbildung 6: Unerlaubte Spielzüge

Da außerhalb des Feldes keine benachbarten Spielsteine zu finden sind, welche ein drehbares Viereck ergeben können, soll das Spiel diese Züge für ungültig erklären und auf die Eingaben nicht reagieren. Die in der Abbildung dargestellten Pfeile stellen dabei nicht alle unerlaubten Spielzüge dieses 3x3 Spielfeldes dar. Um in das Spiel den Beschleunigungssensor zu integrieren, wird weiterhin eine Änderung am Grundprinzip des Spiels vorgenommen. In *Match 3 Games* erhält der Spieler im Normalfall einen Bonus, sofern es ihm gelingt mehr als drei Spielsteine auf einmal zu kombinieren. Bei einer Kombination von fünf oder mehr Spielsteinen erscheint ein neuer einzigartiger Spielstein an der Stelle, an

welcher die Kombination durchgeführt wurde. Dieser Spielstein besitzt nun die Fähigkeit, dass er alle Spielsteine des Typs zerstört, mit dem er getauscht wurde. Weiterhin erhält der Spieler dadurch einen Punktebonus. Um bei der eigentlichen Idee zu bleiben und den Beschleunigungssensor in das Spiel zu integrieren, erhält der Spieler bei der Kombination aus mindestens fünf Spielsteinen eine besondere Aufladung mit welcher die Spielsteine eines Typs teilweise zerstört werden können. Die Aufladungen können ausgelöst werden, indem das Gerät in einem bestimmten Winkel nach links oder nach rechts geneigt wird. Aus designtechnischen Gründen sollte der Neigungswinkel etwa 45° erreichen, damit der Spieler bei der Aktivierung weiterhin das Spiel im Blick hat und die Aufladung nicht aus versehen ausgelöst wird, wenn das Gerät beim Spielen nur leicht geneigt wird. Die Neigungen in andere Richtungen werden nicht berücksichtigt, weil die Mindestanforderung an die Sensorik damit komplett abgedeckt wird. Die Neigung nach links oder rechts hat direkt Einfluss auf die Spielsteine die zerstört werden sollen. Dazu wird bei der Neigung nach links nur die linke Seite des Spielfelds betrachtet und vergleichbar dazu nur die rechte Seite, wenn das Gerät nach rechts geneigt wird. Der Spieler benötigt dabei noch eine Auskunft darüber, welche Steine jeweils links oder rechts des Spielfeldes zerstört werden. Dafür existiert außerhalb des Spielfeldes ein weiterer Spielstein, welcher nicht Teil des eigentlichen Spiels ist. Dieser Steinstein verändert seinen Typ alle paar Sekunden und ist Indikator dafür, welche Steine zerstört werden können. Der nachfolgende Typ des Spielsteins wird dabei komplett zufällig vom Spiel gewählt. Das genaue Intervall des Indikators muss am Ende der Implementierung im Balancing herausgearbeitet werden. In Abbildung 7 wird die Funktion des Beschleunigungssensors innerhalb des Spiels nochmals beispielhaft veranschaulicht.

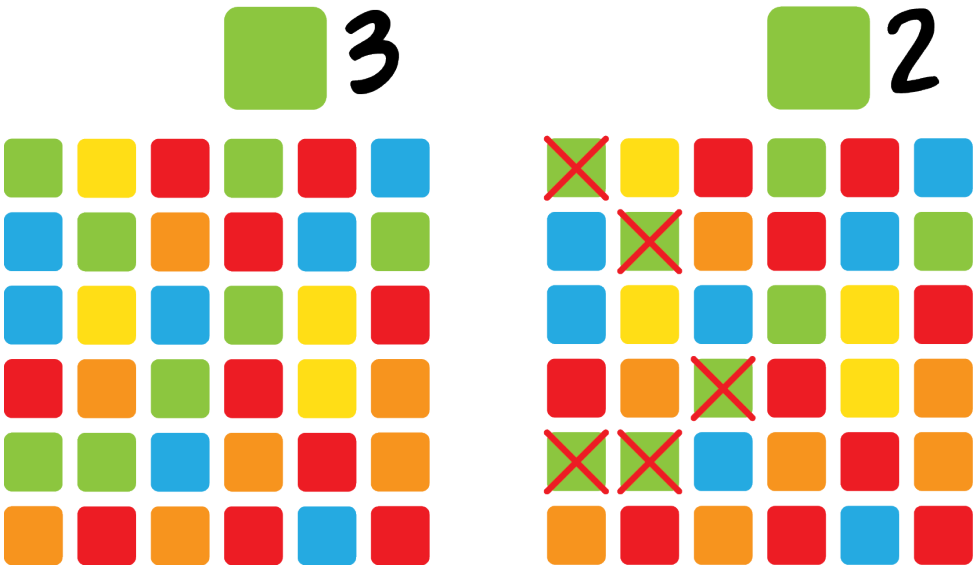

Abbildung 7: Verbrauch einer Aufladung mit Neigung links

Der Anfangszustand ist im linken Teil der Abbildung zu sehen. Der Spieler besitzt drei Aufladungen um die linke oder rechte Seite des Spielfeldes zu beeinflussen. Neigt der Spieler nun sein Gerät nach links, so entsteht die Situation, wie sie auf der rechten Seite der Abbildung aufgezeigt ist. Da momentan der grüne Stein vom Spiel ausgewählt wurde, werden bei der Neigung alle grünen Steine auf der linken Seite des Feldes zerstört. Daraufhin fallen die Steine von oben nach und die neu entstandenen leeren Plätze werden mit neuen Spielsteinen aufgefüllt. Weiterhin wird dem Spieler eine seiner Aufladungen abgezogen und seine Punktzahl erhöht. Die Punktzahl soll sich im oberen Bereich des Spielfeldes befinden. Am Ende einer Runde wird die Punktzahl mit der aktuell höchsten erreichten Punktzahl verglichen. Die Höchstpunktzahl wird auf dem Gerät gespeichert und erfüllt damit auch die Mindestanforderung an persistente Daten. Der Spieler soll die Möglichkeit besitzen, am unteren Ende des Displays den aktuellen Highscore zu sehen. Eine Spielrunde ist mit einem Zeitlimit von 60 Sekunden angesetzt. Ist dieses Limit erreicht, so wird der aktuelle Punktestand zurückgesetzt und der Spieler kann in einer neuen Runde versuchen, den aktuellen Highscore zu schlagen. In Abbildung 8 und Abbildung 9 sind Mockups abgebildet, welche das spätere Grunddesign des Spiels darstellen sollen. In Abbildung 8 ist der Startbildschirm abgebildet. Neben dem Logo und dem Startknopf befinden sich neben der dargestellten Zahl noch zwei weitere Buttons, welche der Skalierbarkeit des Spiels dienen und somit die Spielfeldgröße beschreiben. In der momentanen Einstellung ist das Spielfeld in 8x8 viele Steine aufgeteilt. Dabei kann das Spielfeld durch den rechten Knopf vergrößert und durch den linken Knopf verkleinert werden. Demzufolge können für den späteren Performancevergleich besonders einfach sehr viele Objekte erzeugt werden.

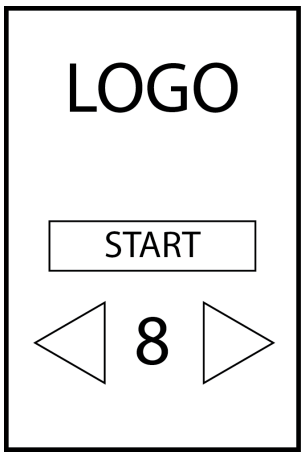

Abbildung 8: Mockup Startbildschirm

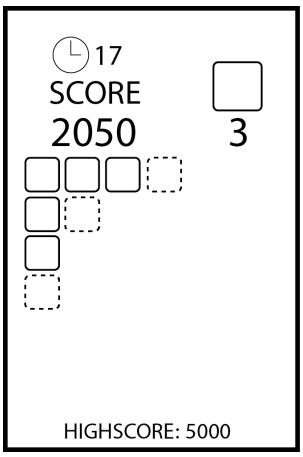

Abbildung 9: Mockup Spielbildschirm

Durch das Betätigen des Startknopfes gelangt der Spieler in das eigentliche Spiel, was exemplarisch in Abbildung 9 zu sehen ist. Informationen wie der Punktestand, die restliche

Rundenzeit und aktuell der ausgewählte Spielstein wird im oberen Bereich des Displays angezeigt. Mittig ist das Spielfeld zu sehen, welches den Hauptteil des Bildschirms einnehmen wird. Am unteren Ende des Displays befindet sich der aktuelle Highscore. Damit das Design des Spiels nicht einem einfachen Prototypen gleicht, müssen zudem entsprechende Grafiken verwendet werden. In Abbildung 10 sind die verwendeten Spielsteine aufgezeigt. Sie besitzen zwar alle dieselbe Form, sind demgegenüber aber farblich gut voneinander zu unterscheiden.

Abbildung 10: Verwendete Spielsteine[1]

Das Spiel ist auf ein 8x8 großes Spielfeld standardisiert und bietet mit sechs verschiedenen Typen von Spielsteinen eine gute Balance zwischen Schwierigkeit und Spaß. Bejeweled verwendet ebenfalls ein 8x8 großes Spielfeld allerdings mit sieben verschiedenen Spielsteinen, womit die möglichen Kombinationen etwas geringer ausfallen. Da das Drehen von vier Spielsteinen einen größeren Schwierigkeitsgrad mit sich bringt, als das einfache Vertauschen von zwei Steinen, wird das Spiel mit einem Spielstein weniger konzeptioniert. Die in Abbildung 10 ausgewählten Grafiken zeigen zudem, dass sich beim Design des Spiels für den Comicstil entschieden wurde. Infolgedessen handelt es sich bei der verwendeten Schriftart, wie in Abbildung 11 dargestellt, um eine Font im Comic-Look.

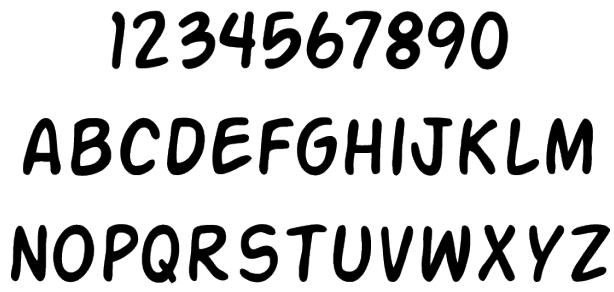

Abbildung 11: Verwendete Schriftart

Passend zu den Spielsteinen und der verwendeten Schriftart müssen auch die restlich ausgewählten Grafiken gehalten werden. Da die Spielsteine überwiegend Brauntöne verwenden, werden die weiteren Grafiken für Navigation und Text ebenfalls in diesen Farbtönen gehalten.

[1] `http://www.kenney.nl/`

Die Grafiken sind in Abbildung 12 dargestellt. In der linken unteren Ecke ist das Logo des Spiels zu sehen, welches den Namen *Four Twist* tragen soll. Daneben sind die Buttons zur Navigation zu finden, sowie die Uhr für den ablaufenden Timer. Der Stern dient als kleiner Effekt, welcher auftreten soll, wenn Spielsteine in der linken oder rechten Hälfte des Spielfeldes zerstört werden.

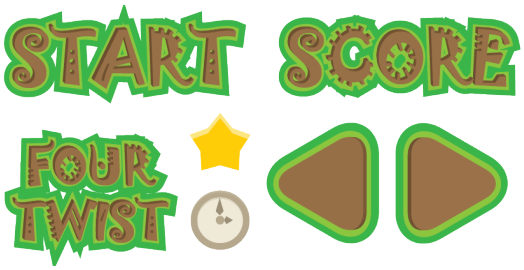

Abbildung 12: Restliche Grafiken

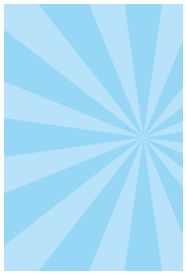

Abbildung 13: Hintergrundgrafik

Abbildung 13 zeigt den Hintergrund, der auf dem Startbildschirm und dem Spielbildschirm zu sehen sein wird. Durch die Verwendung eines blauen Hintergrundes und den braun-grünen Grafiken entsteht die Atmosphäre einer freien Landschaft. Der Hintergrund spiegelt dabei den Himmel wider, wohingegen die restlichen Grafiken Erde und Gras reflektieren.

4 Entwicklung

In der Entwicklungsphase wird die Implementierung der Applikation in den ausgewählten SDKs offengelegt. Dabei wird in einem ersten Teil der Grundlegende Aufbau der Applikation erklärt, welcher als Grundbaustein für alle Frameworks gelten soll. Die implementierten Applikationen sollen den selben Grundaufbau besitzen, damit später ein leichterer Vergleich durchgeführt werden kann. Es wird dabei dennoch berücksichtigt, dass Optimierungsmöglichkeiten den Vorrang erhalten, da diese die Performance der Applikation positiv beeinflussen können. Diese Vorgehensweise entspricht der Entwicklung einer realen Applikation. Vor der eigentlichen Implementierung werden die verwendeten Entwicklungsumgebungen beschrieben, um ebenso einen Einblick in den Umgang mit den SDKs unabhängig von der eigentlichen Programmierung zu erhalten. Dabei werden nach Möglichkeit die von den Entwicklern angebotenen oder empfohlenen Entwicklungsumgebungen verwendet. Mit Hilfe dieser werden daraufhin die verschiedenen Implementierungen in Corona, Gideros und OpenFL durchgeführt und näher erläutert.

4.1 Grundlegender Aufbau der Applikation

Der logische Ablauf der Applikation ist recht simpel gehalten. Beim Starten des Programms gelangt der Spieler in den Startbildschirm, welchen er mit dem Startbutton verlassen kann, um in das eigentliche Spiel zu gelangen. Dabei sind die beiden Bildschirme nicht direkt in zwei unterschiedliche Szenen unterteilt. Dies ergibt sich aus der Einfachheit des Startbildschirms, da dieser nur aus fünf Objekten besteht und somit ein sehr leichter Übergang in das Spiel geschaffen werden kann, indem die dargestellten Objekte einfach ausgeblendet werden. Der Spieler erhält dabei dennoch das Gefühl, als würde es sich um zwei unterschiedliche Szenen handeln. Bei der Sichtweise auf das Spiel gibt es insgesamt zwei Möglichkeiten. Dabei wird zwischen der Portraitansicht[1] und dem Landscapemodus[2] unterschieden. Die im Kapitel zuvor verwendeten Mockups haben bereits darauf hingedeutet, dass es sich um den Portraitmodus handeln wird. Dabei muss den Frameworks explizit gesagt werden, dass das Gerät das dargestellte Bild bei seitlicher Drehung nicht rotieren soll. Im Prinzip wäre es dennoch denkbar, das Spiel in beiden Versionen darzustellen und zu spielen. Die Implementierung des Beschleunigungssensors in das Spiel lässt diese Möglichkeit allerdings nicht zu, da die Neigung des Handys oder Tablets einen entscheidenden Mechanismus im Spiel darstellt und das Gerät somit bei jeder Aktivierung einer Aufladung das Bild rotieren lassen würde, was der Spieler als störend empfinden würde. Die in der Portraitansicht gewählte Auflösung muss hierbei ebenfalls gut überlegt werden, da die Anzahl der möglichen Geräte sehr hoch ist und sich die Auflösungen stark unterscheiden. Die Auflösung wird mit einem Seitenverhältnis von 3:2 angesetzt. Sie bietet gegenüber einer 16:10 oder 16:9 entscheidende Vorteile, wenn die Applikation neben dem Handy auch für Tablets entwickelt wird. Handys haben im Normalfall ein Bildverhältnis, welches dem Breitbild nahe kommt, wohingegen Tablets zudem auch in Auflösungen mit einem 4:3 Verhältnis produziert werden. Außerdem gibt es noch weitere einzigartige Verhältnisse wie 71:40, welches bei dem aktuellen iPhone 5[3] von Apple zum Einsatz kommt. Das 3:2 Verhältnis ermöglicht es, dass das Bild auf jedem Display nahezu den kompletten Bildschirm ausfüllen kann und einen guten Kompromiss zwischen Tablet und Handy bietet. Dies ist weiterhin auch der verwendeten Letterbox-Skalierung zu verdanken. Mit dieser Skalierungsart wird das 3:2 Verhältnis auf den größtmöglichen Raum gestreckt, bei dem das Grundseitenverhältnis bestehen bleibt und Grafiken auch außerhalb des Bereiches platziert werden können. Auf Handys mit Breitbildformat werden am oberen und unteren Bildschirmrand schwarze Ausschnitte zu sehen sein, wohingegen diese auf Tablets mit 4:3 Format jeweils auf der linken und rechten Seite zu erwarten

[1] Handy oder Tablet wird normal gehalten
[2] Gerät wird seitlich gehalten
[3] auch die Versionen 5C und 5S

sind. Würde die Auflösung beispielsweise an 4:3 Displays optimiert werden, so wären die Ausschnitte bei 16:9 und 16:10 Geräten deutlich größer. Die schwarzen Flächen werden in der finalen Applikation jedoch das Spiel nicht stören, da an dieser Stelle ein kleiner Trick angewendet werden kann, wo das 3:2 Verhältnis weiterhin eingehalten wird und die Ausschnitte nicht zu sehen sein werden. Ein hoch-aufgelöstes Hintergrundbild kann so weit gestreckt werden, dass es die Grenzen der definierten 3:2 Auflösung und darüber hinaus des Displays in alle Richtungen überschreitet. Dadurch ist es möglich, die schwarzen Ausschnitte durch dieses Bild zu überdecken und dem Benutzer die Illusion zu vermitteln, dass das Spiel direkt an das Display angepasst wurde. Der wirkliche Bereich in dem die Interaktion mit dem Spiel stattfindet, befindet sich weiterhin in dem 3:2 skaliertem Bereich. Um das Verhältnis der Auflösungen besser zu veranschaulichen, wurden die häufig auftretenden Displayverhältnisse in Abbildung 14 dargestellt.

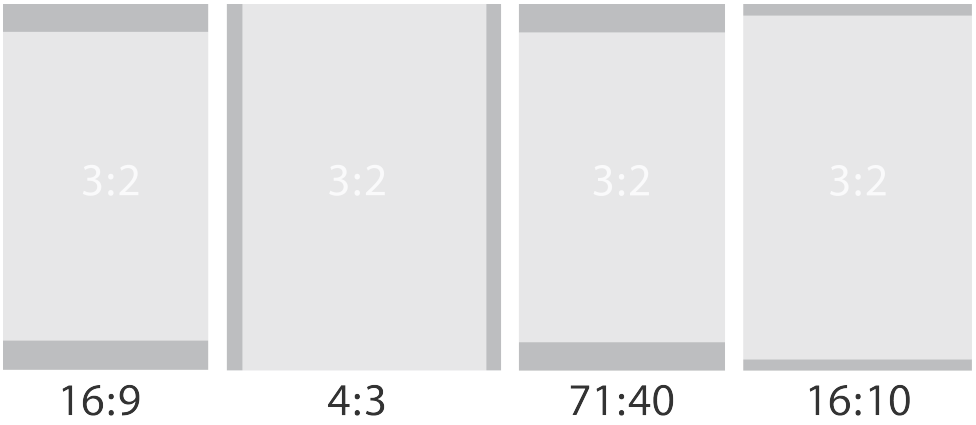

Abbildung 14: 3:2 Skalierung im Vergleich

Eine weitere Möglichkeit wäre, ein größeres Bild zu verwenden, welches auf allen Auflösungen den Bildschirm ausfüllen kann. Die Mindestauflösung eines Bildes, welches dies abdeckt ist mit `380 x 570` Bildpunkten oder ein Vielfaches dieser Auflösung gegeben.

Abbildung 15: Erweiterte Skalierung bei 16:9 Auflösung

Das Prinzip des weit skalierten Hintergrundbildes wird in Abbildung 15 nochmals veranschaulicht. Die grauen Bereiche spiegeln dabei das 3:2 skalierte Bild auf einem 16:9 Display wider, wie es auch in Abbildung 14 zu sehen ist. Das Hintergrundbild wird mit der linken oberen Ecke außerhalb des dargestellten Bereichs positioniert und deckt somit den kompletten Bildschirm ab. Neben der Auflösung und der daraus resultierenden Skalierung sind im Bereich des Spielablaufs besondere Timings zu beachten. Dabei handelt es sich gezielt darum, wann Eingaben vom Spiel akzeptiert werden und zu welchen Zeitpunkten Kombinationen unter den Spielsteinen erkannt werden dürfen. Daraus resultierend auch, zu welchen Zeitpunkten es erlaubt ist Spielsteine zu löschen und von oben nachzuschieben. Die Eingabe über den Touchsensor sind neben dem Beschleunigungssensor die einzigen Eingabemöglichkeiten in das Spiel. Befindet sich das Spielfeld gerade in Bewegung, so dürfen Eingaben vom Spiel nicht akzeptiert werden. Eine Bewegung kommt zustande, sofern der Spieler einen Stein auswählt, in eine Richtung bewegt und somit eine Drehung anstößt oder wenn daran schließend eine Kombination von mindestens drei Steinen ausgelöst wird. Befindet sich das Spiel nun in einem Zustand, bei dem Spielsteine animiert werden, könnte die Eingabe eines Spielers zu Fehlern im Spielfeld führen. Zum Beispiel wird zu einem Zeitpunkt vom Spiel eine Kombination erkannt und die Spielsteine gelöscht. Das Löschen der Spielsteine wird eine Animation starten, welche das Entfernen der Steine kurzzeitig verzögert. Zum selben Zeitpunkt wählt der Spieler einen Stein aus, welcher im Zusammenhang mit der Kombination steht. Dadurch dreht der Spieler vier Steine, wovon ein Stein nach Ablauf der Animation nicht mehr existieren wird. Das Resultat wäre ein freies Feld und somit ein Bug[1] im Spiel, welcher im schlimmsten Fall zum Absturz der Applikation führen kann. Weiterhin kann eine erkannte Kombination vom Spiel und daraus resultierender Animation und nachrücken neuer Spielsteine dazu führen, dass durch die Neuanordnung des Spielfeldes eine weitere Kombination entsteht. Auf jede erfolgreiche Kombination besteht immer die Möglichkeit, dass eine weitere folgt. Dies bedeutet entsprechend, dass Eingaben des Spielers während einer Animation vom Spiel ignoriert werden müssen, um mögliche Fehler zu umgehen. Dafür muss das Spiel wissen, zu welchen Zeitpunkten das Feld still steht und wann Bewegungen stattfinden. Weiterhin ist für die Eingabe über den Touchsensor zum Drehen der vier Steine nur der ausgewählte Stein wichtig. Dies bedeutet, dass bei der Auswahl des Spielsteins und folgender Wischgeste in eine Richtung nicht auf Kollision mit den benachbarten Steinen geprüft wird. Intuitiv könnte der Spieler davon ausgehen, dass er einen Spielstein anwählt und diesen auf den benachbarten Stein schieben muss, damit die Drehung vom Spiel akzeptiert wird. Intern wird dagegen nur die Richtung der Geste auf einen Mindestabstand geprüft. Der Abstand berechnet sich dabei aus der Differenz von Endkoordinate und Anfangsko-

[1] unerwünschter Fehler in der Software

ordinate des Touchevents[1]. Je nach Größe der Differenz kann das Spiel die gewünschte Drehrichtung des Spielers bestimmen. In Abbildung 16 ist diese Vorgehensweise beispielhaft veranschaulicht.

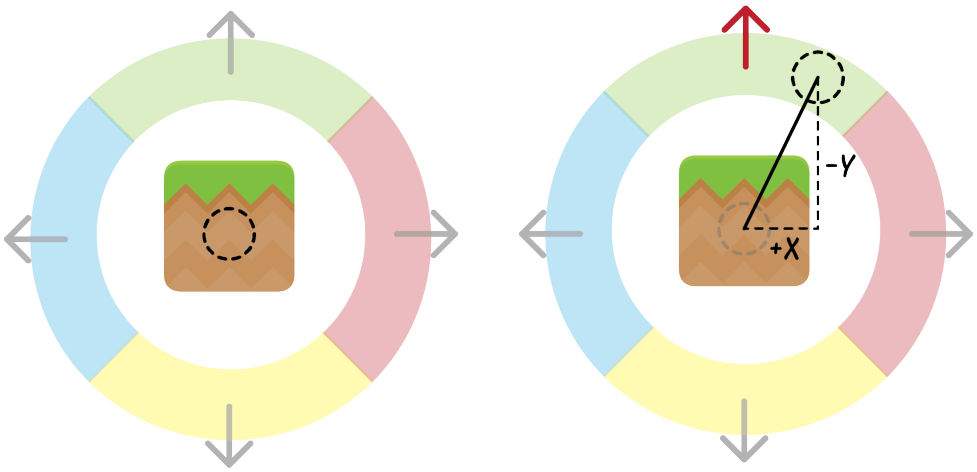

Abbildung 16: Interne Verarbeitung Spielzug

In der Abbildung repräsentiert der linke Ausschnitt den Startzustand eines Spielzuges. Der Spieler wählt den gewünschten Stein aus womit das Spiel den Start des Touchevents mittig des Ausschnitts registriert. Die farbig dargestellten Flächen geben den Mindestabstand an, den der Spieler mit dem Finger zurücklegen muss, damit der Spielzug als solcher registriert wird. Weiterhin geben die unterschiedlichen Farben die Zugrichtung an, wenn der Spieler in diesem Abschnitt (oder auch Winkel) seinen Zug beendet. Durch den Mindestabstand kann der Spieler nicht aus versehen einen Stein ziehen, wenn er das Display nur leicht an einer falschen Stelle berührt. Im Umkehrschluss bedeutet dies auch, dass er dadurch gezwungen wird eine längere Wischbewegung durchzuführen, welche im Prinzip als normale Swipegeste angesehen werden kann. Im rechten Teil der Abbildung beendet der Spieler seinen Zug im markierten Bereich des grünen Feldes. In welchem Bereich der Zug beendet wurde ist für das Spiel leicht zu erkennen, da lediglich die Länge der x und der y-Richtung bestimmt werden muss. In diesem Beispiel ist der Betrag der Länge in Richtung y größer als in x-Richtung weswegen es sich um einen Zug in vertikaler Richtung handeln muss. Da hier vom Startpunkt aus in negativer y-Richtung gezogen wurde, wird vom Spiel ein Zug nach oben registriert. Der y-Wert ist hier nach oben hin negativ, da in den Frameworks die verwendeten Koordinatensysteme ihren Ursprung an die obere linke Ecke des Displays legen und den positiven Bereich beider Achsen im 4. Quadranten eines kartesischen Koordinatensystems definieren. Um letztendlich einen Gesamtüberblick über den internen Aufbau der Applikation zu erhalten, ist in Abbildung 17 ein Klas-

[1]Ereignis bei Berühung des Displays mit einem kapazitiven Gegenstand (z.B. Finger)

sendiagramm zum besseren Verständnis abgebildet. Das Diagramm spiegelt dabei nicht die komplette Struktur inklusive Attribute und Methoden wider, zeigt aber die für das Verständnis wichtigsten Grundbausteine der Applikation.

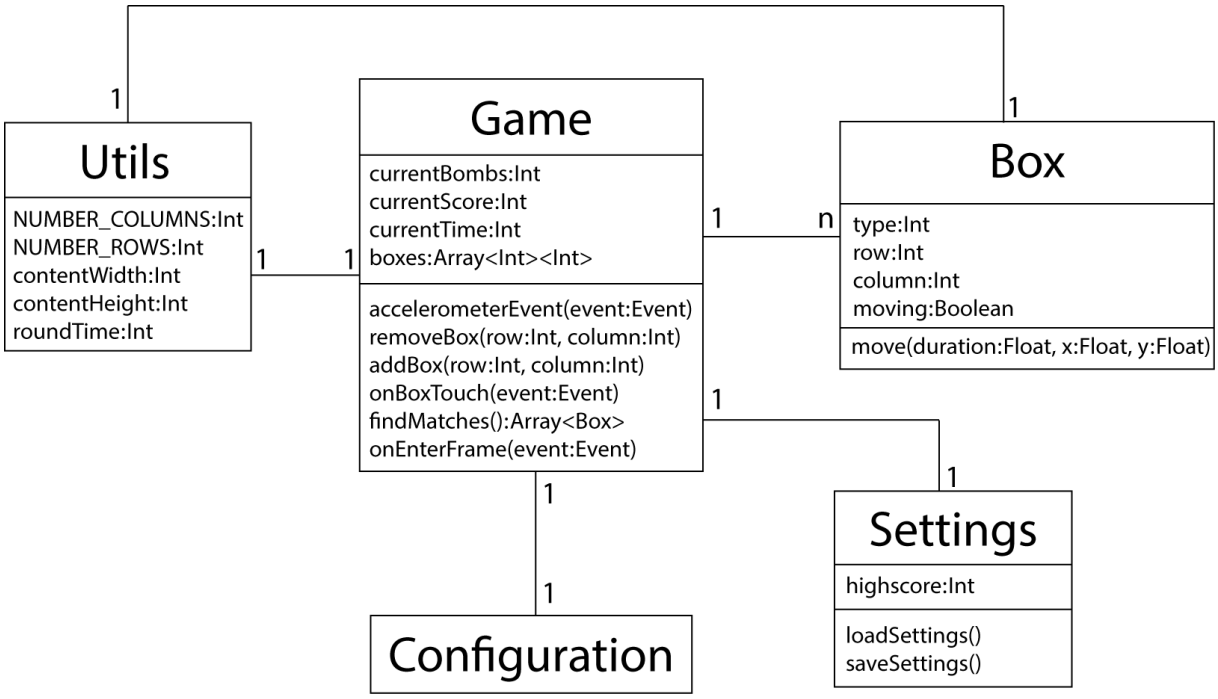

Abbildung 17: Klassendiagramm

Obwohl es sich bei der verwendeten Programmiersprache Lua von Corona und Gideros nicht um eine objektorientierte Sprache handelt, ist das Diagramm dort dennoch gültig. Anstatt Klassen können die verschiedenen Bausteine als Module betrachtet werden, die entsprechend eingebunden werden. In der Mitte des Diagramms ist die Klasse **Game** zu finden. Sie bildet das Herzstück der Applikation und beinhaltet die generelle Spiellogik, sowie wichtige Informationen über den aktuellen Zustand des Spiels. Von besonderer Bedeutung ist die *onEnterFrame*-Methode. Sie verhält sich wie eine *update*-Methode, wie sie in auch in anderen Game-Engines zu finden ist. Diese Methode wird pro Frame einmalig durchlaufen. Dadurch kann kontinuierlich auf bestimmte Events reagiert werden, sofern diese nicht über Listener abgefangen werden. Weiterhin steht die Klasse **Game** in Verbindung zu den verwendeten Spielsteinen, die über die Klasse **Box** repräsentiert werden. Sie beinhaltet neben der aktuellen Position eines Spielsteins, dem Typen und des Zustands im Bezug auf die Bewegung weiterhin Methoden, um die Bewegung eines Steins anzustoßen. Zudem kann mit Hilfe dieser Klasse der Bewegungszustand des aktuellen Spielfeldes festgestellt werden. Um dies zu erreichen wird jedem Spielstein ein Attribut gegeben, welches Auskunft darüber gibt, ob der Stein in Bewegung ist oder nicht. Dieses

Attribut wird gesetzt, sobald eine Art von Motion Tweening auf den Stein angewendet wird und wiederum entfernt, sobald die Animation abgeschlossen wurde. Das Spiel kann so leicht alle Steine überprüfen und feststellen, ob das Spielfeld eingefroren ist oder nicht und daraufhin die Eingabe akzeptieren oder ignorieren. Die Klasse **Utils** enthält Attribute, die über die Länge des Spiels immer konstant sind. Von besonderer Wichtigkeit sind hier die Attribute *contentWidth* und *contentHeight*. Sie beschreiben die aktuelle Größe des Displays in Höhe und Breite. Die Positionen aller Objekte im Spiel stehen immer in direkter Abhängigkeit zu diesen beiden Werten. Die 3:2 skalierte Auflösung ist in den Konfigurationen über `320 x 480` Bildpunkte abgesteckt. Dabei ist die am Ende verwendete Auflösung nicht diese, sondern eine an das Display skalierte Auflösung. Wird im Programmcode ein Objekt mittig des abgesteckten Rahmens an Koordinate `(160,240)` positioniert, dann befindet sich das Objekt auch in höheren Auflösung an dieser Stelle (mittig des Bildschirms). Weiterhin müssen dadurch die Grafiken ebenfalls in hoher Auflösung vorliegen, da diese ansonsten aufgrund des Qualitätsverlusts stark verschwimmen. Die Abhängigkeit aller Objekte zu diesen beiden Werten macht die Positionierung im Spiel wesentlich einfacher. Die eben beschriebene Koordinate `(160,240)` für die Position mittig des Displays ist für den Programmierer schwieriger zu verstehen als dieselbe Koordinate mit den Werten `(contentWidth/2,contentHeight/2)`. Zudem bleibt das Spiel dadurch in seiner Grundpositionierung erhalten, wenn die Skalierung zu einem späteren Zeitpunkt geändert werden sollte. Zusätzlich zu diesen Klassen gibt es die Klasse **Settings**. Sie beinhaltet alle Funktionen, die für die Mindestanforderung an die persistenten Daten zuständig sind und enthält als Attribut den aktuellen Höchstpunktestand des Spielers. Die Klasse **Configuration** steht nicht, wie es im Diagramm abgebildet ist, in allen Frameworks direkt als Modul zur Verfügung, soll aber hier die Projekteinstellungen darstellen. Dort werden Informationen wie beispielsweise Paketname, Auflösung, Pfade zu den Assets, mögliche Skalierung an das Display oder gewünschte FPS[1] hinterlegt.

[1] Frames per second - deutsch: Bilder pro Sekunde

4.2 Entwicklungsumgebungen

Unabhängig von den verwendeten Entwicklungsumgebungen der Frameworks werden die Applikationen auf einem Windows 7 64-Bit Betriebssystem entwickelt. Alle drei verwendeten SDKs können unter Microsoft Windows sowie Apples Mac OS X verwendet werden. Eine Version für Mac OS X ist dabei selbstverständlich, weil alle Frameworks iOS als mobiles Betriebssystem unterstützen und die Applikationen für iOS nur unter einem Apple Betriebsystem kompiliert werden können. Näheres dazu wird im Kapitel 4.6 erläutert. Bei den SDKs wird darauf geachtet, dass eine möglichst aktuelle aber auch stabile Version verwendet wird. Bei der Auswahl der Entwicklungsumgebungen für die Frameworks werden jeweils die empfohlenen oder bereits integrierten IDEs[1] verwendet.

Corona

Das Corona SDK wird in der Version 2014.1262 verwendet. Dabei handelt es sich um den letzten Build vor der Neuerung zu Coronas Graphics 2.0, wo viele Änderungen und Neuerungen im Bezug zur graphischen Darstellung implementiert wurden. Da zum Zeitpunkt der Entwicklung keine dieser neuen Funktionen verwendet werden, wird die letzte stabile Version vor dieser großen Änderung verwendet. Es ist zudem auch die erste Version, die explizit Apples iOS 7 Betriebssystem unterstützt. Im Bezug zur IDE empfiehlt das Studio hinter dem Corona SDK seit Oktober 2013 (siehe [2]) ihr eigens entwickeltes Plugin für den Sublime Text Editor[2]. Sublime Text wird in Version 2221 verwendet und ist ein häufig verwendeter Editor auf Mac OS X, welcher zudem sogar für Linux Distributionen zur Verfügung steht. Das dafür entwickelte Corona Plugin ermöglicht es, den Corona Simulator direkt aus dem Editor heraus zu verwenden und integriert zudem ein Modul zum Debuggen. Weiterhin wird eine Autovervollständigung und entsprechendes Syntax-Highlighting für Corona zum Editor hinzugefügt. Zudem können vorgefertigte Codemakros für bestimmte Funktionen eingefügt werden. Sie sind besonders hilfreich, um Zeit zu sparen und zu Beginn einen Überblick über die möglichen Funktionen zu erhalten. Dabei handelt es sich generell um Features, die für die eigentliche Programmierung irrelevant sind, dennoch kann neben der eingesparten Zeit mit dem SDK deutlich angenehmer gearbeitet werden. Der angesprochene Corona Simulator ist die erste Anlaufstelle um die Applikation zu testen. Es handelt sich dabei jedoch nicht um eine reale Virtualisierung der Geräte die vom Simulator abgebildet werden. Er ist dennoch sehr gut geeignet, um kleine Änderungen direkt zu überprüfen oder Debugging durchzuführen, da das Aufspielen der Applikation auf das echte Gerät zeitaufwendig ist. Toucheingaben werden im Simulator durch die Eingabe über die Maus abgebildet. Weiterhin bietet er eine eigenständige Kon-

[1] IDE : Integrated Development Environment - engl. für Entwicklungsumgebung
[2] `www.sublimetext.com`

sole auf denen Fehlermeldungen oder eigene Ausgaben dargestellt werden können. Diese deckt sich jedoch komplett mit der Konsole, die in Sublime Text integriert ist. Durch die Tatsache, dass der Simulator nahezu alle gängigen Geräte zur Anzeige anbietet, können verschiedene Auflösungen leicht getestet werden. Er besitzt im Kontrast dazu dennoch nicht die Möglichkeit Sensorik zu virtualisieren, so dass diese Funktion nicht über den Simulator getestet werden kann. Das Corona SDK mit dem integrierten Simulator ist von sich aus unabhängig von Sublime Text und kann auch mit anderen IDEs (z.B. Eclipse) verwendet werden oder es wird komplett auf eine intelligente Entwicklungsumgebung verzichtet.

Gideros

Gideros integriert in seiner Installation einen eigenen Editor mit dem Namen Gideros Studio. Die verwendete Version ist v2013.09.1 und ist zum Zeitpunkt der Entwicklung der aktuellste Build. Der Editor kommt mit den Standardfunktionen wie Autovervollständigung und einer Konsole für mögliche Ausgaben zum Debuggen. Weiterhin besitzt er ein extra Fenster mit Bildvorschau der importieren Bilder des Projekts und Möglichkeiten direkt zur offiziellen Dokumentation zu gelangen. Ein ähnlicher Simulator wie bei Corona wird auch bei Gideros verwendet, mit dem ein Entwickler die Funktionen seiner Applikationen schnell testen kann. Dafür verwendet Gideros den Gideros Player, welcher die Netzwerkschnittstelle verwendet, um die Applikation an den Player zu senden. Aus Gideros Studio heraus besitzt der Entwickler die Möglichkeit, seine Applikation an einen beliebigen Player im Netzwerk zu senden und dort abzuspielen. Dafür muss nur die Netzwerkadresse des Geräts bekannt sein, wo der Player gestartet wurde. Er bietet dabei verschiedene Auflösungen an, um das Format der Applikation zu testen und simuliert dabei notwendige Toucheingaben über die Eingaben der Maus. Wird der Player auf einem PC gestartet, so kann die Sensorik nicht getestet werden. Dennoch muss dabei nicht auf diese Funktion verzichtet werden, weil der Player auch für Android und iOS zur Verfügung steht. Er steht als Download für die beiden angesprochenen Betriebssysteme auf der Gideros Homepage zur Verfügung und erfordert lediglich, dass sich das Gerät im Netzwerk befindet und den Player gestartet hat. Wird die Applikation auf den mobilen Player gesendet, so kann dort auf die Sensorik des Geräts zugegriffen werden. Inwiefern der Gideros Player eine reale Repräsentation des Geräts widerspiegelt ist dabei ungeklärt. Das Gideros SDK steht hier in direkter Abhängigkeit zu Gideros Studio, da die entwickelten Applikationen nur aus dieser IDE heraus exportiert werden können.

OpenFL

OpenFL basiert auf Lime, welches Haxe als Programmiersprache verwendet. Auf der Homepage steht Haxe in Version 3.0.0 zur Verfügung. Die Installation von Haxe zieht gleichzeitig die Installation von Lime und OpenFL mit sich, welche über die Kommandozeile angestoßen wird. Bei der Entwicklung unter Windows wird FlashDevelop als IDE verwendet, welche letztendlich in Version 4.5.2.5 zur Verfügung steht. Mit FlashDevelop lassen sich direkt Vorlagen für OpenFL Projekte erstellen. Des Weiteren bietet die IDE eine sehr gute Übersicht über die verwendeten Bibliotheken, die eingebunden werden können. Ein extra Simulator von OpenFL für Android oder iOS existiert nicht, dafür kann das Projekt direkt auf Android Geräte aus der Entwicklungsumgebung heraus deployed und getestet werden. Da OpenFL mehr als nur mobile Plattformen unterstützt, kann die Applikation auch als Windows Programm oder als HTML5-Browserversion kompiliert werden. Dort kann die Sensorik zwar nicht angesprochen werden, dennoch können damit die Grundfunktionen und kleine Änderungen an der Applikation problemlos getestet werden. Wie für eine gute IDE üblich, bietet FlashDevelop auch eine Autovervollständigung für die möglichen Funktionen an, die sich explizit auf OpenFL und Haxe beziehen. OpenFL und FlashDevelop stehen nicht in direkter Abhängigkeit zueinander. Für OpenFL wäre es daher auch möglich, keine erweiterte IDE zu verwenden und die Applikation auf Kommandozeilenebene zu kompilieren.

4.3 Implementierung in Corona

Die einzige Voraussetzung für ein lauffähiges Programm ohne Inhalt ist eine komplett leere *main.lua*, eine *config.lua* und zuletzt eine *build.settings* Datei. Die Inhalte der letzten beiden angesprochenen Dateien werden zu einem späteren Zeitpunkt erklärt. Begonnen wird die Implementierung mit der einfachsten Funktion, dem Anzeigen von einfachen Bildern auf dem Display. Listing 1 zeigt hier das Einbinden des verwendeten Hintergrundbildes in Corona.

```
1 local backGround = display.newImageRect("images/background.png",320,480)
2 backGround.x = utils.contentWidth / 2
3 backGround.y = utils.contentHeight / 2
4 backGround.xScale = 1.5
5 backGround.yScale = 1.5
```

Listing 1: Einbinden Hintergrundbild Corona SDK

Für das einfache Anzeigen genügt theoretisch der Code in Zeile 1 des Listings. Wie schnell und einfach mit dem SDK umgegangen werden kann, wird an dieser Stelle zum ersten Mal deutlich, da alleine dieser Code komplett ausreicht, um ein Bild auf dem Display zu sehen. Es müssen keine weiteren Module oder anderer Code eingebunden werden, um diese Funktion zu erhalten. Alle Bilder befinden sich standardmäßig an Position `(0,0)` des Koordinatensystems, sofern keine genauen Werte angegeben werden. In Corona besitzen alle darstellbaren Objekte ihren Referenzpunkt im Mittelpunkt des Objekts. Grafiken werden in den meisten Fällen symmetrisch zu bestimmten Punkten platziert. Das hier eingebundene Hintergrundbild bietet dabei ein besonders gutes Beispiel, da es direkt mittig des Bildschirmes platziert wird und dadurch die Koordinaten `(utils.contentWidth/2,utils.contentHeight/2)` erhält. Zudem werden die Bilder immer um ihren Referenzpunkt hin skaliert und befinden sich nach Skalierung weiterhin immer an der richtigen Position. Alle verwendeten Grafiken werden auf diese Art in das Spiel eingebunden. Jede Grafik kann dabei gleichzeitig mehrere Funktionen einnehmen. Die Buttons, welche im Startbildschirm des Spiels zu sehen sind, werden ebenfalls als normale Grafiken eingebunden, allerdings mit dem Unterschied, dass an diesen jeweils ein Listener registriert wird. Dadurch kann die Mindestanforderung an Menu & Buttons aus dem Vorkapitel 3.1 realisiert werden. Explizit handelt es sich bei dem Listener um einen Touchlistener, welcher eingehende Eingaben über das Display registriert, wenn die entsprechende Grafik berührt wird. In Corona ist es möglich, mehrere Listener am selben Objekt zu registrieren, was jedoch an keiner Stelle des Programms verwendet wird.

Inwiefern das ausgelöste Ereignis vom registrierten Listener verarbeitet wird und wie der Listener an eine Grafik angemeldet werden, ist in Listing 2 dargestellt.

```
1  --init startButton
2  startButton:addEventListener( "touch", startUp )
3  [...]
4  function startUp(event)
5          if event.phase == "began" then
6          elseif event.phase == "moved" then
7          elseif (event.phase == "ended" or event.phase =="cancelled") then
8                  display.remove(startButton)
9                  --remove assets and switchToGame
10         end
11 end
```

Listing 2: Touchevent Corona

Events werden in Corona in einer Methode verarbeitet, die alle Phasen des Ereignisses abfängt. Der Startknopf soll dabei nur auf das Ende des Events hören, was auch der normalen Funktionsweise eines Buttons entspricht. Vorteilhaft an dieser Verarbeitung von Events ist, dass der komplette Code für ein Ereignis in einer einzelnen Methode zusammengefasst wird, wodurch der Code wesentlich übersichtlicher gehalten wird. Demgegenüber muss ein Listener registriert werden, welcher auf die Änderungen des Beschleunigungssensors reagiert, sowie eine Möglichkeit die in Kapitel 4.1 beschriebene *onEnterFrame*-Methode zu registrieren, in welchem ebenfalls die Anforderung an die Sensorik zu finden ist. Da es sich bei beiden Punkten um dauerhafte Ereignisse handelt, werden dafür Listener an dem *Runtime*-Objekt von Corona registriert. Dieses Objekt erbt von der Klasse EventListener in Corona und beschreibt ein Objekt, welches auf ständige globale Events hören kann (siehe: [3]). Um den Beschleunigungssensor anzusprechen, müssen nur zwei Funktionen des Frameworks verwendet werden. In Zeile 2 in Listing 3 wird zuerst die Abtastrate des Sensors definiert, welche mit 60Hz genau einmal pro gezeichnetem Frame (60 FPS) abgetastet wird.

```
1  Runtime:addEventListener( "enterFrame", onEnterFrame )
2  system.setAccelerometerInterval(60) -- 60Hz -> 60x per second
3  Runtime:addEventListener( "accelerometer", onAccelerometer)
```

Listing 3: Listener am Runtime Objekt

In Zeile 3 wird daraufhin der Listener für den Sensor am *Runtime*-Objekt registriert. Der dort beschriebenen *onAccerometer*-Methode wird vom Framework ein Event übergeben, über welches die aktuellen Werte des Sensors abgefragt werden können. Der Sensor hält dabei drei Werte für alle drei möglichen Achsen in x, y und z-Richtung. Die von Corona zurückgelieferten Werte des Gerätes werden zuvor durch einen smoothing Filter durchlaufen, womit haptische Bewegungen des Gerätes keinen so großen Einfluss auf die Werte haben. Der Hintergrund dieses Filters ist eine besonders weiche Bewegung, wenn Objekte dauerhaft über den Beschleunigungssensor gesteuert werden. Dieses Verhalten ist jedoch für das hier entwickelte Spiel unerwünscht, weil der Spieler durch eine kurze und schnelle Neigung das Spielfeld beeinflussen soll. Um dennoch auf eine zügige Neigung reagieren zu können, wird die vorhin bereits beschriebene Abtastrate mit 60Hz gewählt. Corona erlaubt Werte zwischen 10Hz und 100Hz, wobei 10Hz dem Standardwert entsprechen, wenn kein eigener Wert definiert wird. Mit einer Rate von 60Hz kann genau das gewünschte Verhalten simuliert werden. Leider erfordert eine höhere Frequenz auch eine größere Beanspruchung der Batterie, was als Nachteil festzuhalten ist (siehe: [4]). Zeile 1 in Listing 3 stellt die durchgeführte Implementierung der *onEnterFrame*-Methode dar. In ihr wird dauerhaft auf neue Zustände des Spiels reagiert, wie beispielsweise das Aktualisieren des Punktestands oder das Finden von Kombinationen im Spielfeld. Die kombinierten Spielsteine müssen vom Spiel gelöscht und animiert werden. Das Löschen von Grafiken in Corona funktioniert immer über den selben Weg. In Zeile 8 von Listing 2 wird die von Corona zur Verfügung gestellte remove-Methode für Objekte verwendet, bei der gleichzeitig zum Löschen die auf dem Objekt registrierten Listener entfernt werden. Die Spielsteine sollen jedoch mit Verzögerung aus dem Spiel entfernt werden. Ferner soll bis zu diesem Zeitpunkt der Spielstein animiert werden. Die Mindestanforderung aus der Designphase an die Animationen sind in Corona sehr leicht umzusetzen, da das Framework dafür passende Methoden anbietet. Für die Animation einer Grafik wird Motion Tweening verwendet, wo der aktuelle Zustand des Objekts auf einen gewünschten Endzustand interpoliert wird. Dafür können nahezu alle möglichen Attribute eines Objektes verwendet werden. Die wichtigsten Attribute sind dabei die aktuelle Position, der Alphawert und die Skalierung. Weiterhin kann eine Dauer der Animation angegeben werden, sowie eine Funktion, welche am Ende ausgeführt werden kann. In Listing 4 sind die beiden meist verwendeten Animationen des Spiels dargestellt. Darunter zählt zum einen die Bewegung eines Spielsteines, wenn er vom oberen Teil des Feldes nachrücken muss und zum anderen das Entfernen des Steins bei einer Kombination oder bei der Neigung des Gerätes nach links oder rechts. In Zeile 4 und 10 des Listings wird die Tweening-Funktion verwendet, welche als Übergabeparameter das zu bewegende Objekt erhält, sowie anschließend die Argumente der Animationsdauer, neuen Attributwerten und anknüpfend möglichen

Optionen.

```
1  --box.lua
2  function myBox.move(duration, newX, newY)
3          myBox.moving = true
4          transition.to(myBox, { time = duration, x = newX, y = newY,
5                  onComplete = function() myBox.moving = false end,
6                  transition = easing.OutQuad })
7  end
8
9  function myBox.remove()
10         myBox.trans = transition.to(myBox, { time = 300,
11                 xScale = 2, yScale = 2, alpha = 0,
12                 onComplete = function() display.remove(myBox) end})
13 end
```

Listing 4: Tweening in Corona

Das Entfernen des Steins nach Ablauf der Animation in der *remove*-Methode ist hier aufgrund der *onComplete*-Funktion sehr einfach (Zeile 12). In dem Beispiel des Listings 4 wird beim Löschen des Spielsteins die Interpolation auf die Skalierung und den Alphawert mit einer Dauer von 300 Millisekunden angewandt. Die Interpolation geschieht linear zwischen Start- und Endzeitpunkt. Es war jedoch möglich auch andere Versionen des Easings zu verwenden. Ein Beispiel ist in Zeile 6 zu sehen, wo *OutQuad* als Übergangsfunktion verwendet wird. Dadurch werden die Werte ausgehend quadratisch interpoliert, was dem linken Teil einer negativ quadratischen Funktion entspricht. Dies führt im Endeffekt dazu, dass der Spielstein gegen Anfang der Animation schneller und gegen Ende hin langsamer bewegt wird, was die Animation für den Spieler interessanter wirken lässt. Das Corona SDK ist zum Zeitpunkt der Implementierung in der Lage zwischen 42 verschiedenen Funktionen zu unterscheiden. Ein letztes Kapitel der dargestellten Grafiken bilden die Texte, welche Informationen über den aktuellen Spielstatus offenlegen sollen. Die in Kapitel 3.2 dargestellte Schriftart liegt als TTF[1]-Datei vor und kann von Corona direkt importiert werden. Dafür muss die Datei in der obersten Ordnerstruktur des Projektpfades abgelegt werden. Dies ist die erste und einzige Stelle der Implementierung (unter Verwendung dieses Frameworks), bei der zwischen verschiedenen Betriebssystemen unterschieden werden muss und Codeweichen[2] angewendet werden. Der Grund dafür ist jedoch weniger bei

[1] TTF - True Type Font
[2] Compileranweisung um zwischen Quellcode zu unterscheiden

Corona zu suchen, sondern eher bei der Art, wie die Betriebssysteme ihre Schriftarten ansprechen. Unter Android ist es wichtig, dass die Schriftart unter dem Namen angesprochen wurde, wie er im Dateisystem zu finden ist (ohne Dateiendung). Hingegen muss unter iOS der Name verwendet werden, wie er in der Schriftart selbst hinterlegt ist. In Listing 5 ist die Implementierung der Schriftart dargestellt. Zeile 13 und 14 stellen die Umsetzung eines einfachen Textfeldes mit dem Text, den Positionen, der Schriftart und der Schriftgröße als Übergabeparameter dar. Wichtig ist hier die übergebene *utils.myFont*-Methode, welche die Codeweiche für die Betriebssysteme enthält.

```
1  --utils.lua
2  utils.myFont = function()
3
4          if "Win" == system.getInfo( "platformName" ) then
5                  return "Action Man"
6          elseif "Android" == system.getInfo( "platformName" ) then
7                  return "Action_Man"
8          else -- Mac OS X and iOS
9                  return "Action Man"
10         end
11
12 --main.lua
13 local numberQuadText = display.newText(numberQuad, 0, 0,
14                                         utils.myFont(), 80)
15 end
```

Listing 5: Einbinden Font Corona

In der Methode sind weiterhin Einträge für das Windows und Mac OS X Betriebssystem zu finden. Diese Einträge sind notwendig, wenn die Schriftart auch im Corona Simulator sichtbar sein soll. Dafür muss die Font jedoch direkt im Betriebssystem installiert sein, wo auch der Simulator ausgeführt wird. Für die Android Geräte ist damit die Darstellung der Schriftart und daraus folgend die Realisierung der Anforderung an die Fonts komplett umgesetzt. Für die Darstellung unter iOS muss allerdings eine weitere Option berücksichtigt werden. Die Schriftart muss dafür in der *build.settings* Datei definiert werden. In dieser Datei werden Android oder iOS spezifische Optionen festgehalten. In Corona werden auch hier die notwendigen Rechte definiert, welche die Applikation vom Benutzer während der Installation erfordert. Die einzige Erlaubnis, die hier definiert wird, ist das Schreiben von eigenen Daten auf den internen Speicher. Dieses Recht ist für die

später erwähnten persistenten Daten besonders wichtig. Neben den Optionen für Android und iOS kann hier auch die grundlegende Ausrichtung des Gerätes festgehalten werden. Weitere Optionen werden in der *build.settings* Datei nicht definiert, da es keine Notwendigkeit dazu gibt. In Kombination zu diesen Einstellungen steht die *config.lua* Datei. Hier werden für die Applikation generelle Einstellungen festgehalten. Darunter zählen die Auflösung, die Letterbox-Skalierung und die gewünschten FPS. Das Framework bietet an dieser Stelle auch die Möglichkeit ein Suffix für Bilder zu definieren. Steht eine bestimmte Grafik in hoher und niedriger Auflösung zu Verfügung, so kann an den Dateinamen der hochauflösenden Grafik das selbst definiert Suffix angehangen werden. Das Framework sucht sich dabei für Geräte mit verschiedenen Auflösungen die passende Grafik heraus. Da die Applikation ausschließlich auf Geräten mit sehr hoher Auflösung getestet wird, wird auf diese Möglichkeit verzichtet. Bei der Entwicklung einer realen Applikation soll dies dennoch berücksichtigt werden, da an dieser Stelle der Speicher auf schwächeren Geräten mit kleineren Bildern gefüllt wird, was sich vorteilhaft auf die Performance auswirken kann. Durch die Implementierung der Schriftart können im Spiel Texte jeglicher Größe dargestellt werden. Neben dem Highscore und dem aktuellen Punktestand ist die verbleibende Rundenzeit eine wichtige Information für den Spieler. Eine eigene Funktion für eine ablaufende Rundenzeit wird nicht vom Framework angeboten. Jedoch kann hier ein Timer verwendet werden. Ein Timer in Corona führt eine bestimmte Funktion mit gewünschter Verzögerungsdauer aus. Die Verzögerung wird auch bei der Verwendung des Beschleunigungssensors verwendet. Wenn der Spieler das Gerät kurzzeitig nach links oder rechts neigt, so soll die Beeinflussung des Spielfeldes erst eine halbe Sekunde später geschehen, damit der Spieler den Bildschirm während der Aktion in richtiger Position vor sich sieht. Um den Timer näher zu beschreiben ist in Listing 6 eine kürzere Implementierung anhand der Rundenzeit abgebildet.

```
1 function startTimer()
2         if currentTime == 0 then
3                 --save Settings and reset
4         else
5                 --decrease time and update text
6                 roundTimer = timer.performWithDelay( 1000,startTimer,1)
7         end
8 end
```

Listing 6: Rundenzeit mittels Timer in Corona

Beim Setzen eines Timers in Zeile 6 wird dieser als Objekt zurückgegeben, mit dem die Möglichkeit besteht, ihn zu pausieren oder komplett zu stoppen. Als Übergabeparameter

wird die Zeit in Millisekunden übergeben, sowie nachfolgend die auszuführende Methode und anschließend die Anzahl der Wiederholungen des Timers. Die Rundenzeit wird so implementiert, dass die Rundenzeit pro Sekunde einmalig herunter gezählt wird, weswegen die Verzögerungszeit des Timers 1000 Millisekunden beträgt. Während der Implementierung werden verschiedene Verzögerungszeiten ausprobiert, um einen schöneren Effekt beim Herunterzählen des Timers zu erhalten. Aufgrund einer internen Implementierung im Corona SDK ist es nicht möglich, diese Zeit sehr klein zu wählen. Das Framework erstellt zur Laufzeit für jedes dargestellte Textfeld eine Bitmapgrafik, welche bei Veränderung des Textes komplett neu generiert wird. Dies bedeutet, dass bei einer getesteten Zeit von 5 Millisekunden nur durch das Updaten des einen Textfeldes insgesamt 200 Bilder pro Sekunde erzeugt werden. Dadurch wird die Performance des Spiels extrem stark beeinflusst, so dass das Spiel selbst auf dem leistungsstärksten Testgerät unspielbar ist, weswegen auf diesen Effekt letztendlich verzichtet wird. Den Abschluss der Implementierung in Corona bilden die im Vorkapitel 3.1 beschriebenen Mindestanforderungen an den Sound und die persistenten Daten. Sound und Musik sind sehr leicht in die Applikation einzupflegen. Wichtig ist hier unabhängig vom Framework, dass ein Format verwendet wird, welches sowohl von Android und iOS unterstützt wird. Für Musik wird daher das mp3-Format verwendet und bei normalen Sounds ebenfalls mp3-Dateien oder Dateien im wav-Format. Das nachgestellte Listing zeigt, inwiefern Sound und Musik in Corona implementiert werden.

```
1 --init
2 local switchBoxSound = audio.loadSound( "sound/switchBox.wav")
3 local song = audio.loadStream("sound/GasolinePiano.mp3")
4
5 audio.play(song, { loops=-1 })
6 audio.play(switchBoxSound)
```

Listing 7: Sound in Corona

Das Framework bietet die Möglichkeit, Musik als Stream zu laden. Dies bedeutet, dass das Lied immer nur stückchenweise geladen wird und somit keine langen Ladezeiten zustande kommen. Zum Abspielen in Zeile 5 und 6 können noch weitere Optionen an die Audio Library mitgegeben werden. Für die Musik ist an dieser Stelle nur wichtig, dass sie dauerhaft wiederholt werden. Beim Abspielen einer Datei wird automatisch einer aus 32 verfügbaren Kanäle des Gerätes verwendet. Die persistenten Daten sind im Vergleich zum Sound nicht so einfach zu implementieren, da das Framework von sich aus keine geeigneten Funktion mitliefert. Die Implementierung kann jedoch nahezu komplett alleine

durch Lua realisiert werden. Aufgrund der recht großen Community um das Corona SDK und der daraus resultierenden großen Nachfrage einer solchen Funktion wird von den Entwicklern des SDKs eine Lua-Datei zur Verfügung gestellt, mit der durch die Verwendung der JSON-Bibliothek[1] diese Funktion realisiert werden kann (siehe: [5]).

```
1  settings.highScore = currentScore --save highscore in table
2
3  function saveSettings()
4          loadsave.saveTable(settings, "projectcorona.json")
5  end
6
7  function loadSettings()
8          local highScoreTmp = loadsave.loadTable("projectcorona.json")
9          if highScoreTmp then
10                 settings.highScore = highScoreTmp.highScore
11         end
12 end
```

Listing 8: Peristente Daten in Corona

Das oben dargestellte Listing 8 zeigt die Verwendung des bereitgestellten Codes, mit dem die Daten gespeichert und geladen werden konnten. Beim Speichern in Zeile 4 muss lediglich ein Lua-Table mit den gespeicherten Werten übergeben werden (hier nur der Highscore), sowie ein Dateiname für die JSON-Datei, unter welchem sie daraufhin im Dateisystem zu finden ist. Die Datei wird in das JSON-Format kodiert und automatisch in den Documents-Pfad des Betriebssystems gespeichert, was aus Android und iOS Sicht der geeignete Ort für solche Dateien ist. Das Laden der Daten geschieht lediglich durch die Angabe des Dateinamens der JSON-Datei. Dabei wird die Datei zurück in einen Lua-Table dekodiert und kann somit beim Starten des Programms entsprechend ausgelesen werden.

[1] steht unter Lua direkt zur Verfügung

4.4 Implementierung in Gideros

Um im Gideros SDK ein erstes lauffähiges Programm zu erhalten, muss lediglich eine leere *main.lua* im Projekt angelegt werden. Weitere Dateien zur Konfiguration sind an dieser Stelle nicht notwendig, da die Implementierung mit dem mitgeliefertem Gideros Studio Editor geschieht und die Konfiguration somit über die Einstellungen des Editors durchgeführt werden können.

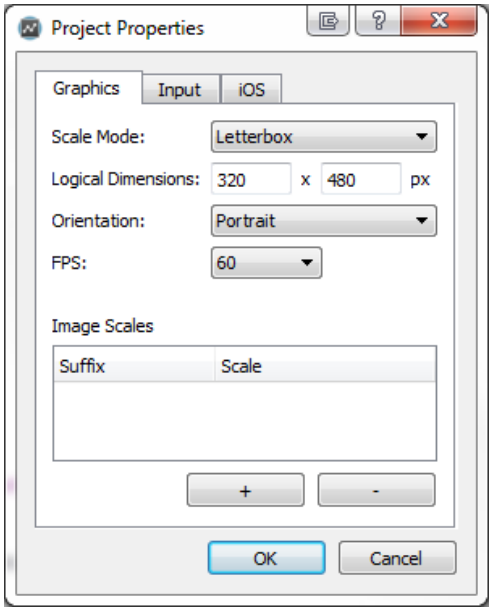

Abbildung 18: Projekteinstellungen Gideros

Diese Einstellungen sind in Abbildung 18 zu sehen und beinhalten neben der Auflösung auch das notwendige Letterbox-Scaling und die Ausrichtung im Portraitmodus. Weitere Konfigurationsmöglichkeiten stehen daraufhin während der Deploymentphase zur Verfügung. Das Einbinden eines einfachen Bildes in Gideros ist sehr ähnlich mit der Implementierung in Corona. Listing 9 zeigt den Code mit dem das Hintergrundbild in Gideros eingebunden wird.

```
1 local background = Bitmap.new(Texture.new("images/background.png"));
2 stage:addChild(background)
3 background:setAnchorPoint(0.5,0.5)
4 background:setPosition(utils.contentWidth/2, utils.contentHeight/2)
5 background:setScale(1.5,1.5)
```

Listing 9: Einbinden Hintergrundbild Gideros SDK

Zur besseren Handhabung, wird hier der Referenzpunkt eines jeden Bildes mittig gesetzt, was in dem Beispiel in Zeile 3 behandelt wird. Automatisch ist hier der Referenzpunkt einer Grafik immer an der linken oberen Ecke zu finden. Wichtig ist beim Einbinden einer Grafik, dass jedes Objekt dem Szenegraph explizit hinzugefügt wird, da das Bild ansonsten zwar geladen jedoch nicht dargestellt wird. Der Szenegraph steht bei Gideros unter dem globalen *stage*-Objekt zur Verfügung, welcher in Zeile 2 angesprochen wird. Das Prinzip der Listener an bestimmten Grafiken ist ebenfalls in Gideros wiederzufinden. Dies bedeutet, dass jede eingebettete Grafik ebenfalls als Button fungieren kann, womit die Mindestanforderung an benutzbare Knöpfe mit eigener Grafik aus dem Anforderungskapitel 3.1 realisiert werden kann. Inwiefern dies in Gideros realisiert wird ist in Listing 10 dargestellt.

```
1 startButton:addEventListener(Event.TOUCHES_END, startUp)
2 [...]
3 function startUp(event)
4         if startButton:hitTestPoint(event.touch.x, event.touch.y) then
5                 --remove graphics and switch scene
6         end
7 end
```

Listing 10: Listener und Touchevent Gideros

In Zeile 1 wird der Listener am Startknopf angemeldet. Dabei werden in diesem Framework nicht gleichzeitig mehrere Touchlistener angemeldet, sondern jeweils nur gezielt auf das Ende eines Touchevents. Dadurch wird bei der Implementierung von Buttons etwas an Code gespart, da nur auf die gewünschten Events gehört werden muss. Jedoch ist es dabei etwas aufwendiger, den Spielzug auf einem Spielstein zu implementieren. Grund dafür ist die Tatsache, dass für den Start des Events und für das Ende jeweils zwei verschiedene Methoden verwendet werden. Dadurch muss das ausgewählte Objekt des Touchevents zwischengespeichert werden und kann nicht direkt in einer Methode behandelt werden. Weiterhin hat Gideros die Eigenheit, dass angemeldete Listener für jedes Objekt registriert werden, sobald dieses Event generell ausgelöst wird. Dieses Problem kommt besonders bei den Spielsteinen zum Vorschein. Auf jedem dieser Steine wird ein Listener registriert, wie es auch in Listing 10 auf den Startknopf durchgeführt wurde. Der Unterschied ist hier, dass nicht auf das Ende eines Touchevents gehört wird, sondern auf dessen Anfang. Beim Berühren des Displays wird nun nicht nur die Methode des berührten Spielsteins aufgerufen, sondern alle Methoden der Spielsteine, die dieses Event als Listener mit dieser Methode registriert haben. Damit ist auch Zeile 4 des Listings zu erklären. Es muss in

jeder Methode eines Touchevents überprüft werden, welches Element sich an der Position der berührten x und y-Koordinate befindet. Damit auch das Ende eines Spielzuges korrekt durchgeführt wird, muss auch ein Listener für das Ende des Touchevents registriert werden. Da in Gideros jedes Event in einer eigenen Methode behandelt wird und der Start des Touchevents nur für das berührte Objekt wichtig ist, muss das Ende des Events global registriert werden. In Listing 11 ist in Zeile 3 diese Implementierung dargestellt.

```
1  require "accelerometer"
2  [...]
3  stage:addEventListener(Event.TOUCHES_END, onTouchEnd)
4  stage:addEventListener(Event.ENTER_FRAME, onEnterFrame)
5  accelerometer:start()
6
7  local accTimer = Timer.new(1000/40)
8  accTimer:addEventListener(Event.TIMER, checkAccelerometer)
9  accTimer:start()
10 [...]
11 function checkAccelerometer()
12         local x,y,z = accelerometer:getAcceleration()
13         [...]
14 end
```

Listing 11: Sensorik und globale Listener in Gideros

Über das global zur Verfügung gestellte *stage*-Objekt, können allgemeine Listener registriert werden. Darüber kann dabei auch anschließend die onEnterFrame-Methode realisiert werden. Weiterhin ist in dem Listing die Verwendung des Beschleunigungssensors zu erkennen, da dies eine notwendige Anforderung an die Applikation aus Kapitel 3.1 implementiert. Wichtig ist zu Beginn, dass das *accelerometer*-Modul von Gideros geladen wird, was in Zeile 1 zu sehen ist. Darüber kann auf alle Funktionen zugegriffen werden, welche die Verwendung des Sensors betreffen. Damit der Sensor auch Werte an das Framework zurückliefern kann, muss dieser erst gestartet werden (Zeile 5). Die Abfrage der Sensordaten geschieht in einem Intervall von 40Hz und muss über ein Timer-Objekt von Gideros angestoßen werden. An diesen Timer wird daraufhin der entsprechende Listener registriert, der die Methode zur Abfrage der Sensordaten ausführen soll. Anschließend muss der Timer noch gestartet werden, damit die Methode auch wirklich in diesem Intervall ausgeführt wird. In der Methode können daraufhin, wie in Zeile 12 erkenntlich, die Werte ohne großen Aufwand herausgelesen werden. An dieser Stelle sei jedoch erwähnt,

dass aus der Dokumentation von Gideros nicht klar hervorgeht, in welchen Zeitabständen der Hardwaresensor seine Daten tatsächlich erneuert. Der Timer wird zwar mit 40Hz definiert, dennoch handelt es sich dabei nur um die einfache Abfrage der Sensorwerte und nicht um das wirkliche Updateintervall des Sensors. Mit dem ausgewählten Intervall von 40Hz kann jedoch das gewünschte Verhalten realisiert werden. Für die notwendigen Animationen der verschiedenen Objekte kann Motion Tweening verwendet werden. Das notwendige Modul um die Interpolation zwischen zwei Punkten zu realisieren wird mit der Installation des SDKs nicht mitgeliefert. Die Entwickler hinter dem Framework stellen jedoch das Modul in ihrem offiziellen Git-Repository zur Verfügung. Das bereitgestellte Tweening-Modul basierte auf einer bestehenden Implementierung namens GTween, welches ehemals für ActionScript 3 entwickelt wurde (siehe: [7]). Durch das Einbinden des Moduls als Lua-Datei kann zwar einfaches Tweening realisiert werden, jedoch wird das gewünschte Easing noch nicht unterstützt. Dafür muss ein weiteres Module eingebunden werden, welches diesmal nicht direkt von Gideros angeboten wird, sondern von einem externen Entwickler zur Verfügung steht (siehe: [8]). Mit diesen beiden implementierten Dateien ist es daraufhin möglich, die gewünschten Animationen durchzuführen und somit den Teilbereich Animationen aus dem Anforderungskapitel 3.1 zu realisieren. Ein Auszug der Verwendung des Tweenings in Gideros ist in Listing 12 zu sehen.

```
1  function myBox.move(duration, newX, newY)
2          myBox.moving = true
3          local tween = GTween.new(myBox, duration,{ x = newX, y = newY },
4                  { dispatchEvents = true, ease = easing.outQuadratic })
5          tween:addEventListener("complete",
6                  function() myBox.moving = false end)
7  end
8
9  function myBox.remove()
10         local tween = GTween.new(myBox, 0.3, { scaleX = 1, scaleY = 1,
11                 alpha = 0} , { dispatchEvents = true })
12         tween:addEventListener("complete",
13                 function() stage:removeChild(myBox); myBox = nil end)
14 end
```

Listing 12: Tweening in Gideros

In der *move*-Methode des Listings wird ein Spielstein von einer Koordinate auf eine andere bewegt. Das GTween-Modul interpoliert die Position zwischen Start- und Endpunkt

und verschiebt das Objekt in der übergebenen Zeit. Die Zeit muss hier in vollen Sekunden angegeben werden, welches inkonsistent mit dem Timer in Gideros ist, bei dem Millisekunden übergeben wurden. Damit die Interpolation zwischen den Punkten nicht linear geschieht, wird hier als Option *easing.outQuadratic* übergeben. Weiterhin muss auf dem Tweening-Objekt ein Listener registriert werden, der bei Beendigung der Animation eine gewünschte Methode ausführt. Wichtig ist bei dieser Implementierung, dass der Animation die Option *dispatchEvents* mitgegeben wird. Dadurch ist es erlaubt an einem Tween-Objekt Events anzumelden, welche als *custom events* nach Gideros klassifiziert werden. Bei dem hier verwendeten *complete*-Event aus Zeile 5 und 12, handelt es sich um solch ein Event. Bei *build-in events* wäre die Option nicht notwendig gewesen (siehe: [9]). In der Animation der *remove*-Methode soll der Spielstein nach Beendigung dieser gelöscht werden. Um ein Objekt aus dem Szenegraph zu löschen, muss die entsprechende Methode auf dem *stage*-Objekt aufgerufen werden, welche in Zeile 13 zu sehen ist. Obwohl dies ausreichen sollte, um das Objekt komplett zu löschen und den Speicherbereich des Bildes freizugeben, wird die Referenz des Spielsteins zur Sicherheit nochmals auf *nil*[1] gesetzt. Weiterführend wird die Schriftart mit Gideros in die Applikation implementiert, um auch die Mindestanforderung an Fonts aus 3.1 zu erfüllen. Das Einbetten der Schriftart kann hier komplett ohne die eigentliche TTF-Datei bewerkstelligt werden. Diese muss nur einmalig vorliegen, damit aus der TTF-Datei eine Bitmap erstellt werden kann. Das Gideros SDK liefert mit der Installation ein Programm mit dem Namen *Gideros Font Creator*. Dieses beigelegte Programm erstellt aus einer TTF-Datei eine Bitmap, in welcher der komplette Zeichensatz dargestellt ist. Die Größe der Schriftart kann während des Exportierens eingestellt werden. Als Ausgabe liefert das Programm die erwähnte Bitmap, sowie eine TXT-Datei in welcher insbesondere die Koordinaten aller Zeichen, sowie weitere Informationen für das SDKs hinterlegt sind. Das Framework erwartet nun für jedes erstellte Textfeld ein Font-Objekt, welches vorab definiert werden muss. In Listing 13 ist dieses Objekt in Zeile 1 definiert.

```
1 local font = Font.new("font/Action_Man.txt", "font/Action_Man.png")
2 local scoreText = TextField.new(font, "0")
```

Listing 13: Font in Gideros

Als Übergabeparameter müssen die Ausgabedateien des Font Creators übergeben werden. Beim Export der Schriftart wird besonders darauf geachtet, dass die Schriftart sehr groß gewählt wird. Durch die Verwendung einer Bitmap mit kleinem Zeichensatz ist es denkbar, dass ein sehr großer Text eine starke Fragmentierung aufweist, da die Zeichen eine

[1] in anderen Programmiersprachen als *null* bekannt

feste Größe besitzen und nicht mehr in Vektorform vorliegen. In Zeile 2 des Listings wird das Font-Objekt im ersten Argument an das Textfeld übergeben. Obwohl ein Textfeld in Gideros nahezu identisch wie eine einfache Grafik behandelt werden kann, ist es nicht möglich, dass der Referenzpunkt des Feldes mittig gesetzt werden kann. Dennoch ist damit die Anforderung an eine eigene Schriftart vollständig implementiert worden. Während der Implementierung des Beschleunigungssensors wird bereits ein Timer-Objekt in die Applikation eingebunden, welches dauerhaft die Methode zum Abfragen der Sensorwerte ausführt. Neben dauerhaften Timern müssen weiterhin Timer verwendet werden, welche eine Funktion einmalig mit einer bestimmten Verzögerung ausführen können. Besonders bei der Neigung des Gerätes nach links oder rechts soll das Löschen der Steine mit einer kleinen Verzögerung durchgeführt werden. In Listing 14 ist der Timer während des Prozesses zum Löschen der Steine auf der linken Seite des Spielfeldes dargestellt.

```
1 function bombFieldLeft()
2         if bombTimer == nil then
3                 bombTimer = Timer.delayedCall(500, function() [...] end)
4         end
5         [...]
6 end
```

Listing 14: Timer in Gideros

Mit dem Aufruf von *delayedCall* an das globale Timer-Objekt kann die Verzögerungszeit sowie die Methode zum Ausführen nach der Verzögerung übergeben werden. Weiterhin ist es noch möglich an dieser Stelle weitere Optionen als dritten Parameter zu übergeben. Im Gegensatz zu dem Timer des Beschleunigungssensors, muss dieser Timer nicht extra gestartet werden und wird automatisch ausgeführt. Dabei wird ein Objekt zurückgegeben, über welches der Timer frühzeitig gestoppt oder pausiert werden kann, was an dieser Stelle jedoch nicht notwendig gewesen ist. Nachdem alle grafischen Implementierungen durchgeführt wurden, wird der Sound und die Musik in das Spiel eingebunden, um der Anforderung aus dem gleichnamigen Unterkapitel in 3.1 gerecht zu werden. Kurze Sounds und lange Musik werden in Gideros gleich behandelt. Das bedeutet, dass diese zur Laufzeit der Applikation komplett in den Speicher geladen werden. Listing 15 gibt einen Überblick, inwiefern einzelne Sounds in das Spiel implementiert werden können. Über das global zur Verfügung stehende Sound-Objekt können diese über die Angabe ihres Pfades geladen werden (Zeile 1,2). Durch das Laden eines Sounds gibt die Funktion ein Objekt zurück, über welches diese abgespielt werden können. Beim Abspielen kann zwischen verschiedenen Optionen unterschieden werden. Werden der *play*-Methode (Zeile 4) keine

Parameter übergeben, so wird der Sound einmalig abgespielt. Bei der Musikdatei ist es jedoch gewünscht, dass diese dauerhaft abgespielt wird.

```
local tapsound = Sound.new("sound/tapsound.wav")
local song = Sound.new("sound/GasolinePiano.mp3")
[...]
tapsound:play()
song:play(0,-1)
```

Listing 15: Sound in Gideros

Dafür muss in einem ersten Schritt die Startposition übergeben werden und anschließend, ob der eingeladene Sound wiederholt werden soll, was vom Framework durch die Übergabe von -1 registriert wird. Als letzte verbleibende Anforderung wird die Persistenz in Gideros implementiert. Eine eigene Implementierung zum Speichern der Daten wird vom SDK nicht angeboten. Jedoch gibt es hier ein extern entwickeltes Modul, welches unter Verwendung der JSON-Bibliothek die Daten entsprechend kodiert und auf dem Speicher des mobilen Gerätes ablegt (siehe: [10]). Dieses Modul unterstützt das Abspeichern von einzelnen Daten sowie einer Reihe von Daten, zusammengefasst in einem Lua-Table. Da in dieser Applikation nur der Höchstpunktstand gesichert werden muss, wird auch nur die Funktion zum Abspeichern einzelner Daten verwendet. Standardmäßig werden die Daten vom eingebetteten Modul in den Documents-Ordner des mobilen Betriebssystem gesichert und später wieder herausgeladen.

```
function saveSettings()
        dataSaver.saveValue("highScore", highScore)
end

function loadSettings()
        local highScoreTmp = dataSaver.loadValue("highScore")
        if highScoreTmp then
                highScore = highScoreTmp
        end
end
```

Listing 16: Persistenz in Gideros

Im oben aufgeführten Listing 16 ist die grundlegende Verwendung dieses Moduls dargestellt. Durch das Aufrufen der *saveValue*- oder *loadValue*-Methode können die Daten

entsprechend verarbeitet werden. Zum Speichern eines Wertes muss hier neben der Referenz auf den Wert auch ein eindeutiger Name vergeben werden, über welchen dieser Wert zu einem späteren Zeitpunkt wieder herausgelesen werden kann. Ein genauer Datentyp muss hier nicht angegeben werden, da Typsicherheit nicht Teil der verwendeten Programmiersprache ist.

4.5 Implementierung in OpenFL

Der Grundaufbau des Projekts kann von der Entwicklungsumgebung generiert werden. Für eine erste lauffähige Version der Applikation sind zwei Dateien von wichtiger Bedeutung. Zum einen eine *Main.hx* Datei, welche den Einstieg in das Programm ermöglicht, zum anderen eine *XML-Datei* im Wurzelverzeichnis des Projektes, welche alle Informationen über die Konfiguration der Applikation beinhaltet. In einem ersten Schritt wird das Hintergrundbild in die Applikation eingebunden. Das Einbinden einer Grafik in OpenFL geschieht über einen kleinen Umweg. Zuerst muss ein Objekt vom Typ *Sprite* angelegt werden, welches als Knoten das Bitmap-Objekt mit der eigentlichen Grafik zugewiesen bekommt. Das Eltern-Objekt wird danach dem allgemeinen Szenegraph hinzugefügt und das Bild ist daraufhin sichtbar. In Listing 17 ist die Implementierung des Hintergrundbildes dargestellt.

```
1  private var background:Sprite;
2  [...]
3  var image = new Bitmap(Assets.getBitmapData("img/background_new.png"));
4  image.x = 0 - image.width / 2;
5  image.y = 0 - image.height / 2;
6  background = new Sprite();
7  background.addChild(image);
8  background.x = Utils.contentWidth / 2;
9  background.y = Utils.contentHeight / 2;
10 background.scaleX = Utils.getInstance().getScaleFactor();
11 background.scaleY = Utils.getInstance().getScaleFactor();
12 addChild(background);
```

Listing 17: Einbinden einer Grafik in OpenFL

In Zeile 1 und 6 wird das übergeordnete Sprite-Objekt generiert, welches in Zeile 7 die Grafik als Kindknoten erhält. In Zeile 12 wird das zusammengesetzte Objekt in einem letzten Schritt dem Szenegraph hinzugefügt. Um den Referenzpunkt einer Grafik mittig zu erhalten, kann keine von OpenFL implementierte Funktion verwendet werden. Aus diesen Grund wird die Grafik beim Erstellen je um die Hälfte ihrer Größe nach links und nach oben versetzt. Da alle Konfigurationen an der Grafik über das Eltern-Objekt geschehen, befindet sich die Grafik immer mittig zu diesem Objekt und das gewünschte Verhalten kann so realisiert werden. Obwohl das Hintergrundbild, wie es bei der Letterbox-Skalierung üblich ist, an die Ränder des Displays skaliert werden, können durch die erhöhte Skalierung des Bildes die schwarzen Ränder außerhalb der 3:2 Auflösung nicht entfernt

werden. Dies liegt vor allem daran, dass außerhalb dieses Bereichs generell keine Bilder platziert werden können. Dazu kommt weiterhin das Problem, dass eine der Möglichkeiten um mit der Skalierung umzugehen zu dem Zeitpunkt der Implementierung bei Android Geräten mit einem Bug behaftet ist, welches bei Änderung des Modus für die Skalierung einen schwarzen Bildschirm zurückliefert (siehe: [11]). Jedoch bietet das Framework hier eine Alternative, welche den Umgang mit verschiedenen Auflösungen letztendlich vereinfacht hat und die Anforderung an mehrere Auflösungen aus 3.1 realisiert. Innerhalb der Konfigurationsdatei kann die Auflösung in Breite und Höhe mit jeweils 0 Pixel definiert werden. Dadurch wählt das SDK für jedes Gerät auf dem die Applikation ausgeführt wird die maximal mögliche Auflösung. Da es sich jetzt nicht mehr um einen realen 3:2 Bereich handelt, sondern um mehrere verschiedene Skalierungen, muss dafür ein Skalierungsfaktor berechnet werden, der Abhängig von der momentan verwendeten Auflösung ist. Weiterhin muss beachtet werden, dass der reale Bereich zwar nicht mehr 3:2 ist, jedoch der virtuelle Bereich immer noch mit einer 3:2 Skalierung behandelt werden soll. Dies bedeutet, dass je nach Verhältnis des echten Gerätes der Faktor gegen die Länge oder die Breite gerechnet werden muss. Das Hintergrundbild, welches in einem 3:2 Verhältnis vorliegt, muss hierfür als einzige Grafik von `320 x 480` Bildpunkten in `380 x 570` Bildpunkte geändert werden[1]. Aufgrund des wiederholten Musters auf der Grafik, bleibt das Look & Feel zum Vorgänger des Bildes nahezu unverändert und die Mindestanforderung für den Umgang verschiedener Auflösung kann somit vollständig realisiert werden. Listing 18 gibt nochmal einen kurzen Einblick darüber, inwiefern die Skalierung für jedes Objekt berücksichtigt werden muss.

```
1 private var factor:Float = Utils.getInstance().getScaleFactor();
2 [...]
3 logoImage.width = logoImage.width * factor * 0.9;
4 logoImage.height = logoImage.height * factor * 0.9;
```

Listing 18: Skalierung einzelner Bilder in OpenFL

Um das Bild um einen gewünschten Faktor zu skalieren (hier 0.9), muss zuvor die echte Höhe und Breite der Grafik mit dem berechneten Faktor multipliziert werden. In OpenFL besteht die Möglichkeit, dass jedes Bild einen oder mehrere Listener enthält, über welche sie beispielsweise zu Buttons weiter umfunktioniert werden. Da das Framework neben iOS und Android auch weitere Plattformen wie beispielsweise HTML5 unterstützt, werden Toucheingaben über Mouse-Events registriert. Listing 19 gibt ein Beispiel, inwiefern Listener mit dem Framework realisiert und verwendet werden.

[1] siehe dazu Kapitel 4.1

```
1 startButton.addEventListener(MouseEvent.MOUSE_UP, startUp);
2
3 public function startUp(event:MouseEvent) {
4         //init game
5 }
```

Listing 19: Listener und Events in OpenFL

In Zeile 1 wird der Listener der Grafik hinzugefügt, welcher auf das Ende eines Tastendrucks registriert wird. Als Übergabeparameter erhält die Funktion eine Methode, die beim Auslösen des Ereignisses ausgeführt werden soll. Der Start und das Ende eines Touchevents werden in OpenFL einzeln behandelt, was im Umkehrschluss bedeutet, dass bei den Spielsteinen das berührte Objekt zwischengespeichert werden muss, sofern der Spielzug als gültig registriert werden soll. Das Ende des Spielzuges muss als globaler Listener registriert werden. Globale Listener werden in OpenFL auf dem *stage*-Objekt aufgerufen. Wird die Methode auf keinem Objekt explizit aufgerufen, so wird automatisch das eben beschriebene Objekt verwendet. In Listing 20 ist dieses Verhalten nochmal dargestellt.

```
1 addEventListener (Event.ENTER_FRAME, onEnterFrame);
2 Lib.current.stage.addEventListener (MouseEvent.MOUSE_UP, onMouseUp);
```

Listing 20: Globale Listener in OpenFL

In Zeile 1 wird die *onEnterFrame*-Methode definiert, welche als update-Methode aus Kapitel 4.1 fungiert. Neben diesen definierten Listenern wird auch ein weiterer Listener für den Beschleunigungssensor implementiert, um der geforderten Mindestanforderungen an die Sensorik gerecht zu werden. Der generelle Zugriff auf den Sensor ist bereits im Framework integriert und es muss lediglich die passende Bibliothek geladen werden. OpenFL verwendet zum größten Teil Bibliotheken von Flash, unter welchen ebenfalls die Library für den Sensor zu finden ist. Aus der großen Auswahl an unterstützten Plattformen des Frameworks folgt jedoch, dass nicht unter jeder Plattform solch ein Sensor zu finden ist. Damit der Sensor explizit für die gewünschten Plattformen integriert wird, müssen an dieser Stelle auch Anweisungen an den Compiler gegeben werden, bei denen zwischen den mobilen Betriebssystemen Android und iOS und den verbleibenden Systemen unterschieden wird. Dies ist notwendig, da sich die Applikation sonst nur unter iOS und Android ausführen lässt. Da kein echter Simulator unter OpenFL existiert, wird zum Debuggen die

Windows-Version der Applikation verwendet. In dieser kann zwar kein Sensor angesprochen werden, jedoch können die Grundfunktionen getestet werden. Ohne die Codeweiche für die mobilen Systeme ist nicht Möglich gewesen, die Applikation unter Windows zu testen, da es zu Compilerfehlern kommt. In Listing 21 ist das Einbinden des Sensors dargestellt.

```
1 import flash.sensors.Accelerometer;
2 import flash.events.AccelerometerEvent;
3
4 #if (android || ios)
5         var accelerometer:Accelerometer = new Accelerometer();
6         accelerometer.setRequestedUpdateInterval(1000/40);
7         accelerometer.addEventListener(AccelerometerEvent.UPDATE,
8                 onAccelerometerChange);
9 #end
```

Listing 21: Zugriff Beschleunigungssensor in OpenFL

In den ersten Zeilen werden die notwendigen Bibliotheken importiert. Darauf folgend wird der Sensor angelegt und mit einer Pollingrate[1] von 40Hz konfiguriert (Zeile 6). An das erstellte Sensor-Objekt wird daraufhin ein Listener registriert, der als Übergabeparameter ein *AccelerometerEvent* erhält, sowie die auszuführende Methode, sobald das Event registriert wird. Diese Implementierung ist nur für iOS und Android gültig. Die auszuführende Methode muss nicht extra für die verschiedenen Plattformen definiert werden, da sie nur ausgeführt wird, sofern ein Sensor vorhanden ist. Innerhalb der Methode werden die Werte entsprechend abgefragt und verarbeitet. Aufgrund eines Bugs im Framework gibt es dabei jedoch eine Unstimmigkeit zwischen den Sensorwerten auf iOS und Android. Auf Geräten mit dem iOS Betriebssystem werden die Werte so herausgelesen, wie es zu erwarten ist. Die Neigung nach links liefert negative Werte, wohingegen die Neigung nach rechts positive Werte zurückgibt. Diese Zuordnung kommt von den Geräten selbst und hat nichts mit dem Framework zu tun. Bei Android Geräten werden die Werte jedoch in umgekehrter Weise herausgelesen. Dies führt dazu, dass in der Implementierung der Methode weitere Compileranweisungen gesetzt werden müssen, in denen bei den Sensorwerten zwischen Android und iOS verschieden reagiert wird. Nachdem die Anforderung des Kapitels 3.1 an die Sensorik implementiert wurde und die verschiedenen Grafiken im Spiel zu sehen waren, können für den nächsten Anforderungspunkt die notwendigen Animationen integriert werden. Für die Animationen muss eine extra Bibliothek mit dem Namen *Actuate*

[1] Zeitabstand in welchem der Sensor abgefragt wird

(siehe [12]) eingebunden werden. Wichtige Bibliotheken werden in OpenFL direkt über die Kommandozeile nachinstalliert[1]. Die neue Bibliothek wird dabei jedoch nicht direkt in das Projekt integriert und muss über die Konfigurationsdatei als Haxelibrary eingefügt werden. Dabei handelt es sich um die einzige Bibliothek, die über die Art nachinstalliert werden muss. Die Verwendung dieser Library erlaubt es daraufhin, die notwendigen Animationen auf die Grafiken im Spiel anzuwenden. In Listing 22 ist die beispielsweise Verwendung der Actuate-Bibliothek dargestellt.

```
1 //Box.hx
2 public function move(duration:Float, newX:Float, newY:Float) {
3         this.moving = true;
4         function onComplete() {
5                 this.moving = false;
6         }
7         Actuate.tween (this, duration, { x: newX, y: newY } )
8                 .ease (Quad.easeOut).onComplete (onComplete);
9 }
```

Listing 22: Tweening in OpenFL

In Zeile 7 findet die Verwendung der Actuate-Library statt. Die dargestellte Methode bewegt einen Spielstein zu einer gewählten Position in einer angegebenen Zeit. Als Übergabeparameter erhält die *tween*-Methode das zu bewegende Objekt, sowie eine Dauer der Animation und die neuen Werte zu denen die Bewegung interpoliert werden soll. Weiterhin gibt es die Möglichkeit, bei Beendigung der Animation eine bestimmte Methode aufzurufen. Das Easing zwischen den Start- und Endwerten wird mit *Quad.easeOut* definiert, womit ein besserer Effekt bei der Animation erzielt werden kann. Die notwendigen Bibliotheken des Easings werden mit der Actuate-Library mitgeliefert. In einem weiteren Schritt wird die Anforderung an die Schriftart angegangen. Damit die Schriftart von OpenFL verwendet werden kann, muss sie lediglich in einen beliebigen Ordner der Projektstruktur abgelegt werden. Danach kann sie als Asset in das Programm geladen werden. Dafür muss in OpenFL ein Textformat-Objekt erstellt werden, was die Schriftart, die Größe und die Farbe definiert. Dieses Objekt kann daraufhin einem Textfeld zugewiesen werden, welches im Folgenden die Eigenschaften des zugewiesenen Objekt angenommen hat. Inwiefern mit dem Format und den Textfelder umgegangen werden muss, ist in Listing 23 dargestellt.

[1] >haxelib install actuate

```
1  private var font:Font = Assets.getFont("font/Action_Man.ttf");
2  [...]
3  var textFormat = new TextFormat(font.fontName
4                  , 30 * Utils.getInstance().getScaleFactor(), 0xFFFFFF);
5  textFormat.align = TextFormatAlign.CENTER;
6
7  currentTimeText = new TextField();
8  currentTimeText.multiline = true;
9  currentTimeText.defaultTextFormat = textFormat;
10 [...]
11 currentTimeText.text = "\n" + Std.string(currentTime);
12 currentTimeText.selectable = false;
13 addChild(currentTimeText);
```

Listing 23: Font in OpenFL

In Zeile 3 wird das Textformat-Objekt generiert. In Zeile 9 bekommt das Textfeld das Format zugewiesen und übernimmt dadurch die Eigenschaften des Formats. Textfelder in OpenFL müssen mehrzeilig definiert werden. Dies liegt an dem Problem, dass die Schrift am oberen Ende des Textfeldes abgeschnitten wird. Inwiefern das mit der Schriftart selbst zu tun hat, wird nicht untersucht, weil sie durch die Planung der Applikation unter allen Umständen verwendet werden soll. Daher werden die Textfelder mit mehreren Zeilen definiert (Zeile 8) und jeder dargestellte Text besteht aus zwei Zeilen, bei der die oberste Zeile leer gelassen wird (Zeile 11). Weiterhin ist jedes in OpenFL implementierte Textfeld automatisch selektierbar. Dies bedeutet, dass es im Spiel möglich ist, den Text auszuwählen und den Inhalt zu ändern. Da dieses Verhalten unerwünscht ist, muss jedes Textfeld explizit als deselektierbar eingestellt werden (Zeile 12). Das dargestellte Textfeld in Listing 23 dient dem Anzeigen der aktuellen Rundenzeit. Diese muss über einen Timer sekündlich geändert werden. Timer in OpenFL können ebenfalls unter Zuhilfenahme der Actuate-Library verwendet werden.

```
1  private var roundTimer:IGenericActuator;
2  [...]
3  public function startTimer() {
4          [...]
5          roundTimer = Actuate.timer(1).onComplete(startTimer);
6  }
```

Listing 24: Timer in OpenFL

Das oben dargestellte Listing 24 stellt beispielsweise die Verwendung des Timers anhand der Rundenzeit dar. Der definierte Timer in Zeile 5 wird mit einer Zeit von einer Sekunde definiert, welcher bei Ablauf dieser Zeit die *startTimer*-Methode von neuem ausführt und den Timer neu startet. Die letzten beiden Anforderungen aus Kapitel 3.1, welche in der Entwicklungsphase von OpenFL implementiert werden, sind der Sound und die Persistenz der Daten. Sound kann in OpenFL durch wenige Eingriffe in der Applikation realisiert werden. Jeder definierte Sound muss als Objekt von Typ *Sound* in OpenFL eingeladen werden.

```
1 private var switchBoxSound:Sound;
2 private var song:Sound;
3 [...]
4 switchBoxSound = Assets.getSound("sound/switchBox.wav");
5 song = Assets.getSound("sound/GasolinePiano.mp3");
6 [...]
7 switchBoxSound.play();
8 song.play(0,-1);
```

Listing 25: Sound in OpenFL

In Zeile 1 und 2 des Listings 25 werden die Sounds aus dem Pfad in dem sie abgelegt wurden in das entsprechende Sound-Objekt geladen. Über dieses können sie daraufhin mit der *play*-Methode abgespielt werden. Wie in Zeile 8 erkennbar, können dieser Methode verschiedene Übergabeparameter zugeteilt werden, welche beispielsweise den abzuspielenden Sound in einer Schleife abspielen lassen (hier durch -1 realisiert). In einem letzten Schritt wird die Mindestanforderung an die Persistenz implementiert, um den Punktestand des Spielers abzuspeichern. Hier müssen keine extra Bibliotheken eingebunden werden, da bereits vorgefertigte Methoden vorhanden sind, die über die Flash-Bibliotheken implementiert wurden. Der wichtigste Teil der Funktionen, um Daten zu Speichern und zu Laden, sind in Listing 26 dargestellt. Das in Zeile 1 deklarierte Objekt vom Typ *SharedObject* ist zuständig für das Abspeichern und Laden der Daten. Dafür muss es mit einem Dateinamen initialisiert werden (Zeile 4), woraufhin anschließend die zu speichernden Daten an das *data*-Attribut des Objekts referenziert werden (Zeile 8). Unter Verwendung der *flush*-Methode auf dem Objekt, können die Daten daraufhin im Dateisystem des Betriebssystems gespeichert werden. Beim Herausladen der Daten muss das Objekt in der selben Weise wie beim Speichern initialisiert werden. Aus dem beim Speichern zugewiesenen Attribut kann daraufhin der Punktestand des Spielers in Zeile 19 in eine lokale Variable geladen werden und steht somit für weitere Zwecke zur Verfügung.

```
1  private var so:SharedObject;
2  [...]
3  public function saveSettings() {
4          so = SharedObject.getLocal( "saved_scores" );
5          if (so.data.scores == null) {
6                  so.data.scores = 0;
7          }
8          so.data.scores = currentScore;
9          [...]
10         flushStatus = so.flush() ; // Save the object
11         [...]
12 }
13
14 private function loadSettings() {
15         so = SharedObject.getLocal( "saved_scores" );
16         if (so.data.scores == null) {
17                 so.data.scores = 0;
18         }
19         highScore = so.data.scores;
20 }
```

Listing 26: Speichern und Laden eigener Daten in OpenFL

4.6 Deployment

Um die Applikationen in den Frameworks als lauffähige Version auf die mobilen Testgeräte zu bekommen, müssen einige Vorkehrungen getroffen werden. Für die Applikationen unter iOS müssen diese in jedem Fall unter einem Mac OS X Betriebssystem kompiliert werden, da zum Kompilieren der Programme XCode auf dem System installiert sein muss, welches nur für das Apple Betriebssystem zur Verfügung steht. Damit dies bewerkstelligt werden kann, muss weiterhin eine gültige Entwicklerlizenz von Apple für iOS vorliegen. Jede Applikation erhält dabei eine eindeutige *iOS App ID*, welche im Development Center von Apple festgelegt werden kann. Weiterhin muss diese ID mit einem entsprechenden *iOS Provisioning Profile* verknüpft werden. Dieses muss ebenfalls im Development Center erstellt und auf dem Entwicklersystem installiert werden. Dieses Profil kann weiterhin auch nur auf Systemen mit Mac OS X installiert werden. Mit diesen Vorkehrungen kann die Applikationen signiert und auf das mobile Testgerät aufgespielt werden. Das Kompilieren der Applikationen unter Android benötigt keine gültige Lizenz. Mit der Installation des SDKs wird automatisch ein Schlüssel zum Debuggen mitgeliefert, mit welchem die Applikationen ohne Probleme signiert werden können.

Deployment unter Corona

Bei der Verwendung des Corona Frameworks geschieht das Kompilieren der Programme ausschließlich über den mitgelieferten Corona Simulator, welcher im Hintergrund XCode verwendet. Die notwendigen Rechte, die zum installieren der erstellen Applikationen auf Android Geräten erforderlich sind, können in Corona in der *build.settings* Datei definiert werden. Bei Apple werden die erforderlichen Rechte innerhalb des generierten Profils im Development Center festgelegt. Der Simulator unter Windows bietet nur die Option unter Android zu bauen, wohingegen die Mac OS X Version das Kompilieren für beide Systeme unterstützt.

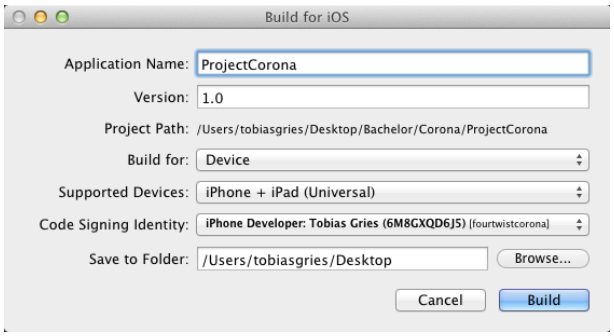

Abbildung 19: Kompilieren unter iOS Corona

In Abbildung 19 ist der dafür verwende Dialog unter Mac OS X abgebildet. Er deckt sich in den wichtigsten Teilen mit dem Kompilieren für Android Geräte. Unter dem Corona SDK ist es notwendig, dass zur Kompilierzeit immer eine Internetverbindung besteht, da es neben der freien Version des SDKs auch Premium-Versionen gibt, die letztendlich mit dem Account bei Corona verbunden sind. Am Ende des Vorgangs wird eine apk-Datei für Android oder eine app-Datei für iOS generiert. Die schnellste Möglichkeit die apk-Dateien unter Android zu installieren, ist das einfache Kopieren auf die Speicherkarte, woraufhin die Datei unter Zuhilfenahme eines geeigneten Dateimanagers installiert werden kann. Die app-Dateien für iOS können unter Mac OS X mittels iTunes in die Mediathek des Gerätes geladen und anschließend installiert werden. Dafür ist es weiterhin notwendig, dass das Testgerät als Entwicklergerät im Development Account von Apple registriert wird. Für Android kann jedes Gerät ohne Probleme verwendet werden.

Deployment unter Gideors

Die Verwendung von Gideros Studio für das Framework ist vorallem für den späteren Build-Prozess wichtig. Die in Gideros Studio entwickelte Applikation wird durch eine Export-Funktion in ein Eclipse oder ein XCode-Projekt exportiert. Dadurch wird das Projekt am Ende so kompiliert, wie es bei der nativen Programmierung ebenfalls der Fall gewesen wäre. Die notwendigen Rechte für die Installation unter Android können dann in der Manifest-Datei des Android Projekts festgelegt werden. Beim Erstellen der apk-Datei kann dabei ein entsprechender Debug-Schlüssel zum Signieren übergeben werden oder ein neuer Schlüssel erstellt werden. Weiterhin kann der Paketname übergeben werden, welcher eindeutig gewählt werden muss, sofern die Applikation später in den Playstore gestellt werden soll. Die erstellte apk-Datei kann danach über den Dateimanager im Android System installiert werden. Das für iOS exportierte Projekt kann ohne weiteres in XCode importiert werden. In XCode selbst sind die wichtigsten Einstellungen die aktuelle iOS SDK Version gegen welche die Applikation gebaut werden soll, sowie ein eindeutiger Paketname, welcher sich mit dem Paketnamen der App-ID im Development Center überschneiden muss. In einem letzten Schritt wird in XCode das korrekte Provisioning-Profil ausgewählt und die Applikation kann über die Build-Funktion von XCode in die entsprechende app-Datei kompiliert werden. Diese Datei wird daraufhin über iTunes in die Mediathek des Gerätes eingebunden und installiert.

Deployment unter OpenFL

Für OpenFL wird unter Windows die FlashDevelop IDE verwendet. Um die Applikation auf den Geräten mit dem Android Betriebssystem zu installieren, kann die Deploy-Funktion aus der Entwicklungsumgebung verwendet werden. Ist ein Android Gerät per USB an dem Entwicklersystem angeschlossen, so kann die Applikation mittels eines Knopfdruck gebaut und auf dem Gerät installiert werden. Dabei kann zwischen *Debug* und *Release*-Version unterschieden werden. In diesem Fall wird nur die Debug-Variante verwendet. Auf dem Android Gerät muss dazu weiterhin USB-Debugging erlaubt sein, welches über die Entwickleroptionen in den Einstellungen des Handys oder Tablets aktiviert werden kann. Nach Deployment der Applikation wird diese automatisch gestartet und kann entsprechend getestet werden. Der Build unter OS X geschieht komplett über die Kommandozeile. Mittels Aufruf von *openfl test ios* im Projektpfad wird der Build-Prozess gestartet und die app-Datei erstellt. Die Konfigurationsdatei von OpenFL hält dabei die Einstellungen mit dem eindeutigen Paketnamen, welcher sich mit der App-ID aus dem Development Center decken muss. Anhand dieses Namens wählt OpenFL während des Kompilierens automatisch das korrekte Provisioning-Profil, welches auf dem System installiert sein muss. Die app-Datei steht danach in einem angegebenen Ordner zur Verfügung und kann über iTunes auf dem Gerät installiert werden.

5 Allgemeiner Vergleich

In einem Allgemeinen Vergleich werden die Frameworks gegenübergestellt. Schon bei der verfügbaren Dokumentation der SDKs gibt es große Unterschiede. Die offizielle Dokumentation von OpenFL ist mit nur wenigen Seiten bestückt (siehe [15]). Dabei sind weiterhin nur die wichtigsten Funktionen dargestellt und mit Beispielen näher erklärt. Da das Meiste jedoch auf Flash bzw. der Haxe-Programmiersprache basiert, können viele Antworten auch von anderen Quellen bezogen werden. Daneben existiert ein offizielles Forum, bei dem Support gesucht werden kann. Dieses ist aktiv besucht, kann sich jedoch nicht mit den Besucherzahlen der anderen SDKs messen. Im Bereich des Einstiegs bietet OpenFL eine Reihe von Beispielprogrammen, die über die Kommandozeile nachinstalliert werden können. Anfangs bieten die Beispielprogramme einen guten Einstieg, sind allerdings weniger gut dokumentiert, so dass die Funktionsweise des Programmes selbst herausgearbeitet werden muss. Die Dokumentation des Gideros SDK (siehe [16]) kann direkt aus der eigens mitgelieferten Entwicklungsumgebung Gideros Studio aufgerufen werden. Die offizielle Dokumentation besteht im Prinzip aus zwei Teilen. Der erste Teil der Dokumentation bietet einen Überblick über die komplette API[1] des Frameworks. In dem meisten Fällen handelt es sich hier jedoch nur um eine Aufzählung aller Funktionen, die mit einem Satz in der Dokumentation beschrieben werden. Weiterhin ist die Syntax kurz erläutert, sowie die Übergabewerte dargestellt (sofern vorhanden). Neben der Gideros API sind hier jedoch auch die API der verwendeten Physikengine und der Programmiersprache Lua zu finden. Weiterhin gibt es in dieser Dokumentation Unterpunkte zu verschiedene Plugins, die in Gideros implementiert werden können. Der zweite Teil der Dokumentation fasst eine Erklärung der Funktionen im Sinne eines Tutorials für Einsteiger. Dies fasst eine umfangreiche Erklärung von der Erstellung eines Projekts, über das Einbinden von Grafiken und Verwenden von Events, bis hin zum Deployment der Applikation. Wie auch bei OpenFL werden Beispielprogramme mit der Installation des SDKs mitgeliefert. Diese sind nach Funktionsweise geordnet und erklären in sehr kleinen Programmen, wie bestimmte Funktionen umgesetzt werden können. Die Dokumentation des Corona SDKs ist allen Punkten den Dokumentationen der anderen getesteten SDKs überlegen (siehe [17]). Die API ist sehr umfangreich beschrieben und mit Codebeispielen bestückt. Weiterhin sind mehrere Videos mit Beispielimplementierungen auf der Videoplattform YouTube zu finden, die von offizieller Seite erstellt wurden. Daneben sind eine große Reihe von offiziellen Leitfäden und Tutorials zu finden, die alle Funktionen des SDKs abdecken. Desweiteren werden wie auch beim Gideros SDK eine große Anzahl von Beispielprogrammen mitgeliefert, welche alle notwendigen Funktionen in sehr kurzen Programmen erläutern.

[1] Application programming interface

5.1 Performance

Bei dem Performancetest nimmt die Mindestanforderung an die Skalierung eine wichtige Rolle ein. Durch das stetige Erhöhen der Menge an Spielsteinen, können die Applikationen verschiedener Frameworks in einem FPS-Test verglichen werden. Dabei bleibt das mobile Endgerät, auf dem der Test durchgeführt wird als Konstante im Test bestehen. Bei dem verwendeten Testgerät handelt es sich um ein Samsung Galaxy S4[1] mit dem Android Betriebssystem in Version 4.3. Die Menge an Spielsteinen wird variabel von 25 (5x5) bis zu einer Anzahl von 1400 (70x70) Spielsteinen gesetzt. Dabei werden die durchschnittlichen Bilder pro Sekunde im Ruhezustand des Spielfeldes gemessen. Das Resultat der durchgeführten Messung ist in Abbildung 20 dargestellt.

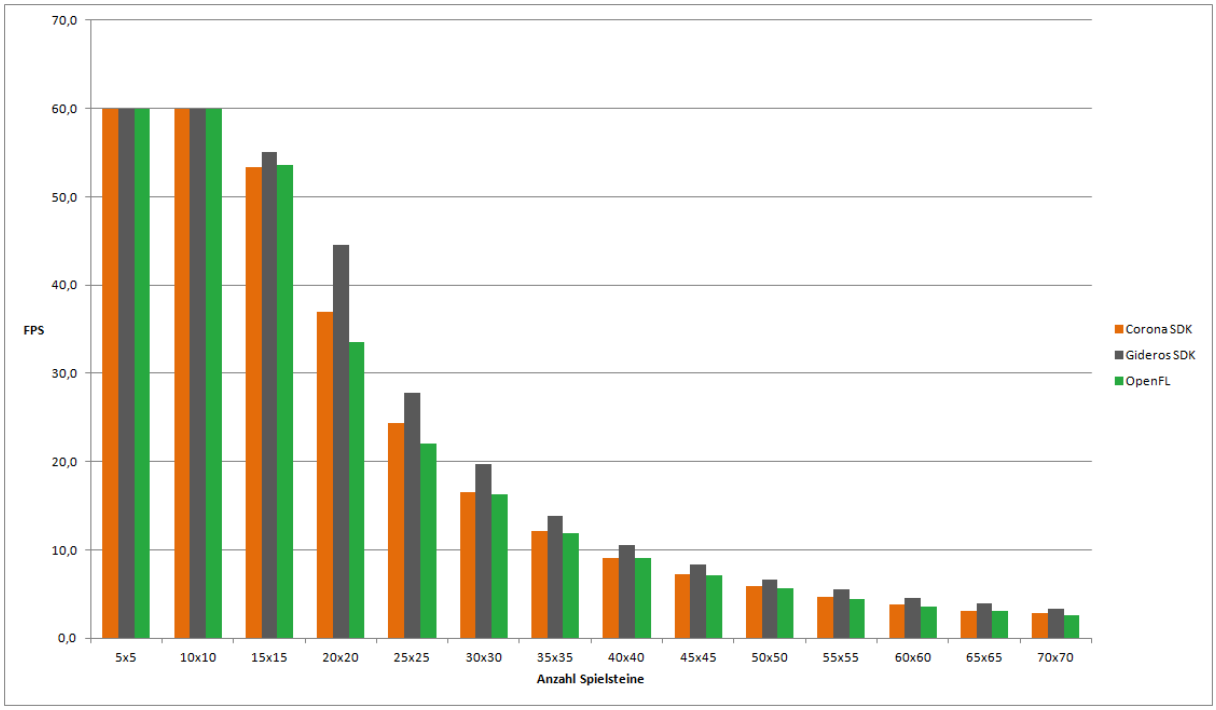

Abbildung 20: Erreichte FPS bei variabler Anzahl an Spielsteinen

Das Grunddesign des Spiels ist mit einem Spielfeld der Größe 8x8 angesetzt. Bei dieser Spielfeldgröße werden in jedem Framework die maximal möglichen FPS erreicht. Dieser Zustand fällt jedoch bei einer Menge von 15x15 Spielsteinen bereits ab. Obwohl das Spiel für den Benutzer immer noch als flüssig empfunden wird, sind hier bereits kleine Unterschiede zwischen den Frameworks zu sehen. Das Gideros SDK hat im Vergleich zu den anderen verwendeten SDK die beste Performance. Diese Differenz findet bei einem Messwert von 20x20 Spielsteinen ihren Höhepunkt. An dieser Stelle werden Unterschiede von bis zu 10 FPS erreicht. Über alle Messpunkte hinweg bietet das Gideros SDK die beste

[1] Modellnummer: GT-I9505

Performance. OpenFL und das Corona SDK erreichen nahezu die gleichen Werte, wobei die Applikation unter Corona allgemein ein wenig besser abschneidet. Ab einer Größe von 20x20 Spielsteinen wird die Performance des Spiels als unangenehm empfunden und stört den Spielfluss, da die Animationen der Spielsteine ab dieser Größe merkbar weniger flüssig durchgeführt werden. Das Starten der Applikationen auf dem Testgerät nimmt bei jedem Framework die gleiche Zeit ein. Diese Beobachtung gilt sowohl für die Android Version als auch für die Version und iOS. Dort kommt als Testgerät ein IPod Touch der 5. Generation mit iOS 7.0.4 zum Einsatz. Das Gerät ist von der Hardwareausstattung wesentlich schwächer als das Samsung Galaxy S4. Die Applikationen laufen jedoch in ihren vorgesehenen Spielfeldgrößen auch auf diesem Gerät auf der maximal möglichen FPS. Die Unterschiede zwischen den Frameworks im Bereich der Performance können sich generell mit den gemessenen Differenzen aus dem Test mit dem Android System decken.

5.2 Look & Feel zwischen Android und iOS

Ein Unterschied im Look & Feel zwischen Android und iOS ist bei jedem verwendeten Framework nicht vorhanden. Auch unter den Frameworks selbst sind alle Applikationen in ihrem Erscheinungsbild gleich und reagieren identisch auf die verschiedenen Eingaben. Die Eingabe über den Beschleunigungssensor konnte über ein geeignetes Anpassen der Werte auf ein identisches Verhalten zwischen den Frameworks abgestimmt werden. In Abbildung 21 ist als Beispiel ein Vergleich zwischen der Android und iOS Version unter dem Gideros SDK dargestellt.

Abbildung 21: Beispiel Gideros SDK Android und iOS

Die Android Version ist im linken Teil der Abbildung zu sehen, wobei die iOS Version im rechten Teil der Abbildung dargestellt ist. Ein Unterschied zwischen den beiden Versionen ist nicht zu erkennen. Alle Elemente befinden sich an der selben Position und reagieren identisch auf die gleichen Eingaben. Die verwendeten Grafiken werden in guter Qualität angezeigt und alle Animationen werden auf die selbe Art und Weise abgespielt. Jedoch gibt es einen Unterschied im Sound zwischen iOS und Android. Unter Android werden die Sounds mit einer kurzen Verzögerung abgespielt. Diese Verzögerung ist mit einer Dauer von annähernd 500ms spürbar, sofern dem Nutzer die Android- und die iOS-Versionen bekannt sind. Dieses Problem ist sowohl abhängig von Android als auch vom verwendeten Gerät. Es handelt sich dabei jedoch um einen Fehler, welcher unter allen Frameworks auftritt und in direkter Abhängigkeit mit Google steht (siehe: [6]). Ein weiterer Test auf einem Google Nexus 7 mit dem Android 4.3 Betriebssystem bestätigt dieses Problem weiterhin.

5.3 Unterschiede zwischen den SDKs

Zwischen den Frameworks können einige Unterschiede festgehalten werden. Beim Starten der Applikationen auf den verschiedenen Frameworks war unter der Applikation von Gideros SDK ein erzwungener Splashscreen zu sehen, welcher das Logo des Frameworks darstellt. Dieser Bildschirm kann bei Kauf einer Lizenz entfernt werden. Unter dem Corona SDK oder OpenFL ist solch ein Splashscreen nicht zu sehen. Mit dem Corona Framework ist es jedoch in der Standard-Version möglich, einen eigenen Splashscreen hinzuzufügen. Unter OpenFL ist dies ebenso möglich. Bei den notwendigen Lines of Code[1] (folgend LOC) erreicht OpenFL den höchsten Wert mit rund 1050 geschriebenen Zeilen. Gefolgt wird dieser Wert von Gideros mit 840 LOC, woraufhin das Corona SDK mit nur rund 800 LOC auskommt. Gideros und Corona erreichen nahezu den selben Wert, da sie unter anderem auch die selbe Programmiersprache zur Umsetzung verwenden und der Aufbau der Frameworks sich sehr ähnelt. Ein Großteil der Zeilen, die in OpenFL mehr benötigt wurden, sind durch die Textfelder zu erklären, die mit sehr vielen Codezeilen implementiert werden mussten. Weitere kleine Unterschiede sind in den verwendeten Standardbibliotheken der Programmiersprache Lua zu finden. Sowohl Corona als auch Gideros verwenden Lua als Sprache, jedoch gab es bei der Math-Bibliothek von Lua für mathematische Funktionen einen kleinen Unterschied, sodass die Implementierung in Gideros bestimmte Funktionen wie *round* nicht zur Verfügung standen und eigens definiert werden mussten, wohingegen diese Funktionen in Corona ohne weiteres verwendet werden konnten. Innerhalb der Programmierung können keine großen Unterschiede festgehalten werden. Alle verwendeten Frameworks sind in ihrer Funktionsweise recht identisch. Schlüsselfunktionen wie Ani-

[1] eigenständig geschriebener Code ohne externe Bibliotheken / Module

mationen, Timer, Sound, das Einbetten von Grafiken oder die Verwendung von Listenern geschehen in allen SDKs auf die gleiche Weise. Erst zum Zeitpunkt des Deployments treten Unterschiede auf, da sie zwar unterschiedliche Vorgehensweisen verwenden, im Endeffekt jedoch das selbe Ergebnis liefern.

6 Fazit und Ausblick

In einem ersten Punkt kann zusammengefasst werden, dass sowohl das Gideros SDK, OpenFL und das Corona SDK die geforderten Anforderungen erfüllen konnten. Sie geben weiterhin einen guten Aufschluss darüber, dass Cross-Platform SDKs eine sehr gute Alternative darstellen, um schnell und effizient konkurrenzfähige Applikationen im Bereich der Spielentwicklung zu erstellen. Begonnen beim Corona SDK ist der Einstieg besonders einfach, da von Seiten der Community als auch von offizieller Seite aus sehr viele Möglichkeiten geboten werden, um den Lernprozess mit dem SDK zu vereinfachen. Darunter zählen eine Reihe von Tutorials, welche den Aufbau einer einzelnen Funktionen beschreiben bis hin zu kompletten Spielen, die sehr aufwändig dokumentiert sind. Weiterhin stehen eine Vielzahl von Videotutorials und Beispielprogrammen zur Verfügung, die den Einstieg nochmals vereinfachen. Dazu kommt die geringe Komplexität und Effizienz der verwendeten Programmiersprache Lua. Selbes gilt daher ebenfalls für das Gideros SDK, da hier die selbe Sprache verwendet wird. Für das Gideros SDK stehen im Vergleich zu Corona nicht so viele Möglichkeiten für den Einstieg zur Verfügung, jedoch ist die Qualität der Dokumentation recht hoch, so dass ein Mehrwert bei Corona nicht gegeben ist. Da OpenFL das Neuste aus den drei ausgewählten SDKs ist, ist die Qualität als auch die Quantität an Tutorials für den Einstieg nicht sehr hoch. Die mitgelieferten Beispielprogramme sind jedoch ausreichend, um den generellen Umgang mit dem SDK zu erlernen. Die dafür notwendige Zeit ist zwar höher als bei Corona oder Gideros, jedoch hat der Entwickler das Gefühl mehr Kontrolle über sein Programm zu haben. Dies kommt vor allem durch Haxe als Programmiersprache, welche höherer Komplexität als Lua aufweist. Nachdem der Einstieg geschafft ist und die Entwicklung begonnen wird, können alle SDKs vergleichsweise gut verwendet werden. Es wird schnell ein generelles Schema deutlich, welches unter den Cross-Platform SDKs identisch ist. Die aus dem Entwicklungskapitel dargestellten Codeausschnitte sind dafür ein gutes Beispiel. Der Aufbau bestimmter Methoden, um spezielle Effekte erzielen zu können, wie beispielsweise Animationen, verzögerte Methodenaufrufe oder einfach das Abspielen von Sound sind sehr ähnlich zueinander. In den meisten Fällen reicht eine einzelne Funktion mit einer Anzahl von Übergabeparametern, um diese Funktionen zu realisieren. Dies wirkt sich vorteilhaft auf die Geschwindigkeit und Effizienz aus mit welcher entwickelt werden kann. Zudem bleibt der Code dadurch überschaubarer und aufgeräumter, da nur wenige Zeilen verwendet werden müssen, um gewünschtes Verhalten zu realisieren. Diese starke Abstraktion von der nativen Programmierung hat jedoch den Nachteil, dass der Entwickler generell weniger Kontrolle über sein Programm besitzt. In sehr großen Projekten könnte es daher passieren, dass die Fähigkeiten der Frameworks an ihre Grenzen stoßen und gegebenenfalls bestimmte Funktionen nicht implementiert werden können. Im Rahmen dieser Arbeit ist

dieses Problem jedoch nicht aufgetreten. Wird ein direkter Vergleich zwischen den Frameworks gezogen und dabei nur die Tatsache berücksichtigt, dass nur für die Plattformen iOS und Android entwickelt wird, so scheint das Corona SDK von der Handhabung her die meisten Vorteile zu bieten. Die Einstellungen über die Konfigurationsdateien sind wesentlicher einfacher zu handhaben als die Einstellungen bei Gideros, welche letztendlich über Eclipse oder XCode realisiert werden. OpenFL hingegen verwendet ein ähnliches Prinzip wie Corona, ist jedoch während der Entwicklungsphase etwas schwieriger zu verwenden. Obwohl Gideros hier am schlechtesten abschneidet, bietet es gegenüber den anderen Frameworks die beste Performance. Im Performancetest sind dabei unter Verwendung des gleichen Szenarios bis zu 10 FPS Unterschied zu den anderen SDKs erreicht worden. Unabhängig davon bietet OpenFL den Vorteil, dass extrem viele Plattformen unterstützt werden. Dabei handelt es sich weiterhin um die meist verwendeten Plattformen, darunter auch Windows oder die Möglichkeit seine Applikation direkt für den Browser zu entwickeln. Dabei sollte die Version für den Browser ebenso gut spielbar sein wie auf den mobilen Geräten, da sich Eingaben über den Touchsensor über einfach Mausklicks realisieren lassen. Hierbei ist es jedoch nicht möglich gerätespezifische Sensorik anzusprechen, so dass an dieser Stelle bestimmte Funktionen über andere Eingaben[1] realisiert werden müssen. OpenFL ist darüber hinaus Open Source und es besteht die Möglichkeit, dass bestimmte Bibliotheken nach eigener Präferenz ausgetauscht werden können. Beispielsweise gibt es bei der Physik-Bibliothek mehrere verschiedene Bibliotheken zwischen denen sich der Entwickler entscheiden und daraufhin implementieren kann. Im Ausblick kann positiv auf die verschiedenen Frameworks geschaut werden. Alle Cross-Platform SDKs befinden sich in einem stetigen Stadium der Entwicklung und es gibt keine Ansätze darüber, dass eines der SDKs in naher Zukunft nicht mehr weiterentwickelt wird. Auch während der Durchführung dieser Arbeit haben sich die Frameworks weiterentwickelt. So ist es in der aktuellen Version von OpenFL, verglichen mit der verwendeten Version, beispielsweise wesentlich einfacher Grafiken in die Szene zu setzen. Dabei ähnelt die Vorgehensweise der Art, wie es in Gideros oder Corona implementiert wurde. Das Corona SDK wird in naher Zukunft auch das Windows Phone 8 Betriebssystem unterstützen und kann somit nahezu den kompletten mobilen Markt bedienen. Darüber hinaus legt das SDK auch großen Fokus auf die Weiterentwicklung im Bereich der Business Applikationen, so dass es nicht nur als Framework für Spiele gesehen werden kann. Weiterhin besitzen die Frameworks wesentlich mehr Funktionen, als sie in dieser Arbeit behandelt wurden. Darunter zählen vor allem wichtige Social Media-Aspekte, die Verwendung der implementierten Physik Engine oder die Integration von Plugins für die Monetarisierungsmöglichkeiten mittels Werbung oder In-App-Einkäufe. Diese Möglichkeiten stehen unter allen Frame-

[1] beispielsweise Tastatur

works zur Verfügung und werden unter Gideros und Corona direkt mit der Installation des SDKs mitgeliefert, wohingegen bei OpenFL auf externe Quellen in Git-Repositories zurückgegriffen werden kann. Unter den Social Media-Aspekten können Facebook- oder Twitter-Plugins verstanden werden, welche es ermöglichen, direkt aus der Applikation heraus Beiträge zu verfassen. Dies ist natürlich besonders interessant für den Entwickler, weil er dadurch kostenlose Werbung für seine entwickelte Applikation erhalten kann. Auf der anderen Seite kann der Benutzer beispielsweise seinen Punktestand auf einer Social Media-Plattform kundtun und sich somit mit anderen Spielern messen. Weiterhin besteht bei OpenFL noch die Möglichkeit die Spiele anstatt in 2D auch in 3D zu entwickeln. Wie weit der Entwicklungsstand dieser Möglichkeit ist, wurde jedoch nicht untersucht, da die anderen Frameworks keine Anwendungen in 3D unterstützen und sie nicht mehr vergleichbar gewesen wären. Abschließend kann zusammengefasst werden, dass alle Cross-Platform SDKs alle Anforderungen erfüllen konnten. Weiterhin haben sie die nötige Effizienz, Geschwindigkeit und Einfachheit bewiesen mit der sie geworben haben. Unter den verwendeten Frameworks gibt es keines, welches sich als Bestes herauskristallisiert hat. Jedes Framework hat in gewissen Bereichen bestimmte Vorzüge. So ist das Gideros SDK von der Performance her den anderen Frameworks überlegen. Mit OpenFL können sehr viele zusätzliche Plattformen unterstützt oder Bibliotheken nach belieben ausgetauscht werden, wohingegen das Corona SDK am leichtesten zu verwenden ist und im Bereich der zusätzlichen Plugins den besten Support liefert.

7 Quellenverzeichnis

[1] Andrew Burgert, (März 2012), *CASMA Globant Mobile Presentation* [Online] Available: `http://de.slideshare.net/andrewburgert/globant-andrew-burgert-casma` [Zugriff: 13.03.2014]

[2] © Corona Labs, (Oktober 2013), *Corona Editor + Subline Text* [Online] Available: `https://coronalabs.com/blog/2013/10/14/corona-editor-sublime-text/` [Zugriff 13.02.2014]

[3] © Corona Labs, (Januar 2014), *Runtime API* [Online] Available: `https://docs.coronalabs.com/api/type/Runtime/` [Zugriff 17.02.2014]

[4] © Corona Labs, (Januar 2014), *Accelerometer API* [Online] Available: `https://docs.coronalabs.com/api/library/system/setAccelerometerInterval.html` [Zugriff 17.02.2014]

[5] Rob Miracle, (Februar 2012), *Save your game data in Corona SDK* [Online] Available: `http://omnigeek.robmiracle.com/2012/02/23/need-to-save-your-game-data-in-corona-sdk-check-out-this-little-bit-of-code/` [Zugriff 24.02.2014]

[6] Joshua Quick, (Januar 2013), *New Android Audio Lag* [Online] Available: `http://forums.coronalabs.com/topic/28091-new-android-audio-lag/` [Zugriff 09.03.2013]

[7] Grant Skinner, (Dezember 2009), *GTween Tweening and Animation Library for ActionScript 3* [Online] Available: `http://gskinner.com/libraries/gtween/` [Zugriff 26.02.2014]

[8] Robert Penner, *Robert Penner's Easing Functions* [Online] Available: `http://www.robertpenner.com/easing/` [Zugriff 26.02.2014]

[9] © Gideros Mobile, *Gideros Documentation Events* [Online] Available: `http://docs.giderosmobile.com/events.htm` [Zugriff 26.02.2014]

[10] Arturs Sosins, (Januar 2012), *Save and load data module for Gideros Mobile* [Online] Available: `http://appcodingeasy.com/Gideros-Mobile/Save-and-load-data-module-for-Gideros-Mobile` [Zugriff 28.02.2014]

[11] Erik Escoffier, (Dezember 2013), *StageScaleModes#156* [Online] Available: `https://github.com/openfl/openfl/issues/156` [Zugriff 01.03.2014]

[12] Joshua Granick, (Dezember 2013), *Actuate Library OpenFL* [Online]
Available: `https://github.com/jgranick/actuate` [Zugriff 21.03.2014]

[13] Joshua Granick, (Mai 2013), *Introducing OpenFL* [Online]
Available: `http://www.joshuagranick.com/blog/2013/05/30/introducing-openfl/`
[Zugriff 13.03.2014]

[14] Joshua Granick, (Januar 2014), *The MIT License (MIT)* [Online]
Available: `https://github.com/openfl/openfl/blob/master/LICENSE.md`
[Zugriff 13.03.2014]

[15] OpenFL, (2014), *OpenFL Documentation* [Online]
Available: `http://www.openfl.org/documentation/` [Zugriff 21.03.2014]

[16] © Gideros Mobile, *(*Gideros Documentation) [Online]
Available: `http://docs.giderosmobile.com/` [Zugriff 21.03.2014]

[17] © Corona Labs, (2014), *Corona Documentation* [Online]
Available: `https://docs.coronalabs.com/` [Zugriff 21.03.2014]