

F a c h h o c h s c h u l e W e d e l

Virtual Reality 2-Praktikum

Thema:

Invisiboga

INVISIBLE BOard GAme

Eingereicht von: Alexander Bertram

Erarbeitet im: 4. Fachsemester, 5. Verwaltungssemester

Abgabetermin: 14.11.2011

Betreuer: Martin Egge
Fachhochschule Wedel
Feldstrasse 143
22880 Wedel

Inhaltsverzeichnis

I. Allgemeine Problemstellung	2
1. Grundidee	2
2. Spiel	2
3. Spielfeld	2
4. Spielablauf	2
4.1. Vorbereitung	2
4.2. Ziehen	2
4.3. Sonderfelder	3
4.4. Ende	3
5. Augmented Reality	3
II. Benutzerhandbuch	4
1. Ablaufbedingungen	4
2. App-Installation	4
3. App-Start	4
4. Bedienungsanleitung	8
4.1. Initialisierung	8
4.2. Hinweise zur Bedienung	8
4.3. Spielfeld-Erzeugung, Neustart und nächste Spiel-Phase	9
4.4. Sonderfeld-Erzeugung	10
4.5. Startender Spieler	12
4.6. Würfeln	12
4.7. Besetztes Feld	12
4.8. Sonderfeld	12
4.9. Spiel-Ende	14
4.10. Verschiedene Perspektiven	14
5. Fehlermeldungen	16
6. Wiederanlaufbedingungen	16

III. Programmierhandbuch	17
1. Entwicklungskonfiguration	17
1.1. Computer	17
1.2. Smartphone	17
2. Problemanalyse und Realisation	17
2.1. Problemanalyse	17
2.1.1. Augmented Reality	17
2.1.2. Spielfeld	17
2.1.3. Grafik	18
2.1.4. Interaktion	18
2.1.5. Entwicklung für mobile Systeme	18
2.1.6. Logik	18
2.2. Realisationsanalyse	18
2.2.1. Augmented Reality	18
2.2.2. Spielfeld	19
2.2.3. Grafik	19
2.2.4. Interaktion	19
2.2.5. Entwicklung für mobile Systeme	20
2.2.6. Logik	20
2.3. Realisationsbeschreibung	21
2.3.1. Augmented Reality	21
2.3.2. Spielfeld	23
2.3.3. Grafik	24
2.3.4. Interaktion	27
2.3.5. Entwicklung für mobile Systeme	30
2.3.6. Logik	31
3. Beschreibung grundsätzlicher Datenstrukturen	33
3.1. Zustandsautomaten	33
3.1.1. Invisiboga (Abbildung 23)	33
3.1.2. Spiel (Abbildung 24)	34
3.1.3. Spieler (Abbildung 25)	34
3.1.4. Spielfigur (Abbildung 26)	34
3.2. Datenstrukturen	34
3.2.1. Light	34
3.2.2. Field	34
3.2.3. Pawn	35
3.2.4. Player	35
3.2.5. Space	36
3.2.6. TouchEvent	37
4. Architektur	37

5. Programmtests	38
6. Fazit	38
6.1. Einschränkungen und bekannte Probleme	38
6.1.1. Während der Entwicklung	38
6.1.2. Während der Ausführung	39
6.2. Erweiterbarkeit	39
6.2.1. Spieler	39
6.2.2. Zusätzliche Funktionalität	39
6.3. Abweichungen vom Konzept	39
6.4. Einschätzung der Projektergebnisse	40
7. Quellennachweise	40

Abbildungsverzeichnis

1. Marker	5
2. Installation von Nicht-Market-Anwendungen zulassen	6
3. Invisiboga.apk	6
4. Anwendung installieren	7
5. Anwendung installiert	7
6. Lade-Animation	8
7. Hinweis, dass die Kamera auf den Marker gerichtet sein muss	8
8. Hinweise, wie das Feld inklusive Sonderfelder erzeugt wird	9
9. Neustart	10
10. Beispiel-Feld	10
11. Weiter	11
12. Sonderfelder	11
13. Computer-Zug	12
14. Würfeln	13
15. Würfel-Ergebnis	13
16. Zielfeld ist belegt, Spieler bleibt auf dem vorherigen Feld stehen	13
17. Computer landet auf einem Sonderfeld	14
18. Computer gewinnt	14
19. Verschiedene Perspektiven	15
20. Marker, die von DroidAR verwendet werden	21
21. Feld-Texturen	24
22. Manipulation des User Interfaces	30
23. Invisiboga-Zustandsautomat	41
24. Spiel-Zustandsautomat	42
25. Spieler-Zustandsautomat	43
26. Spielfigur-Zustandsautomat	44
27. Architektur	45

Teil I.

Allgemeine Problemstellung

1. Grundidee

Im Rahmen des Virtual Reality 2-Praktikums soll eine Augmented Reality-Applikation, ein Spiel, für ein mobiles Gerät wie z.B. ein Smartphone oder ein Tablet entwickelt werden. In dem Spiel treten zwei Spieler, ein menschlicher und ein vom Computer gesteuerter, gegeneinander an.

2. Spiel

Das Spiel setzt sich aus einem Spielfeld, einem Würfel und je einer Spielfigur pro Spieler zusammen.

3. Spielfeld

Das Spielfeld besteht aus einzelnen Feldern, die den Spieldpfad darstellen. Entlang dieses Pfades werden die Spielfiguren bewegt. Der Anfang und das Ende werden durch je ein Start- und ein Zielfeld gekennzeichnet. Einige der Felder, die Sonderfelder, haben eine besondere Bedeutung. Bleibt eine Spielfigur auf einem dieser Felder stehen, so wird ein Ereignis ausgelöst, das eine bestimmte Aktion erfordert.

4. Spielablauf

4.1. Vorbereitung

Vor dem Spiel wird als Erstes aus den einzelnen Feldern ein Spielfeld erzeugt. Dann werden die Spielfiguren auf dem Startfeld platziert und der erste zu ziehende Spieler ausgelost. Das kann z.B. mit Hilfe des Würfels geschehen. Der Kontrahent mit der höheren Augenzahl eröffnet das Spiel.

4.2. Ziehen

Die beiden Spieler ziehen abwechselnd nacheinander. Um einen Zug durchzuführen, muss zunächst gewürfelt werden. Die Anzahl der aus dem Wurf resultierenden Augen auf dem Würfel entscheidet über die Schrittweite, um die die Spielfigur entlang des Spieldpfades Richtung Ziel bewegt werden darf.

4.3. Sonderfelder

Wenn eine der Figuren auf einem Sonderfeld stehen bleibt, darf die Spielfigur um ein paar Felder vor- oder muss um einige Felder zurückrücken. Die Richtung und die Anzahl der Felder wird per Zufall entschieden.

4.4. Ende

Erreicht eine Spielfigur das Zielfeld, so endet das Spiel. Sieger ist der Spieler, zu dem die Figur auf dem Zielfeld gehört.

5. Augmented Reality

Das komplette Spielgeschehen läuft in der virtuellen Realität ab. Alle Spielobjekte sind also auch virtuell. Das einzige reale Objekt, das für die Applikation nötig ist, ist ein bestimmtes Bild, der sogenannte Marker. Die Gerät-Kamera muss auf den Marker gerichtet sein, um das Spielgeschehen auf dem zugehörigen Bildschirm beobachten zu können.

Teil II.

Benutzerhandbuch

1. Ablaufbedingungen

Für die Ausführung der App wird ein mobiles Endgerät mit dem Betriebssystem Android (ab Version 2.2) und einer eingebauten Kamera benötigt. Zusätzlich muss der Marker aus Abbildung 1 ausgedruckt werden. Der Ausdruck sollte jedoch nicht zu klein sein und mindestens dem Format DIN A4 entsprechen.

2. App-Installation

Unter Android gibt es zwei Wege eine App zu installieren:

1. Android-Market
2. Installationsdatei

Da es diese App nicht im Android-Market gibt, wird im Folgenden die zweite Möglichkeit Schritt für Schritt beschrieben:

1. die Einstellung **Unbekannte Herkunft** unter **Einstellungen** → **Anwendungen** aktivieren und die nachfolgende Warnung bestätigen (Abbildung 2)
2. die Installationsdatei **Invisiboga.apk** in ein Verzeichnis auf dem Smartphone kopieren
3. mit einem Dateimanager in das Verzeichnis wechseln und die Installationsdatei öffnen (Abbildung 3)
4. durch Antippen des **Installieren**-Buttons die angeforderten Rechte bestätigen und warten, bis die App installiert ist (Abbildung 4)
Achtung: Die App verlangt das Recht, uneingeschränkt auf das Internet zuzugreifen zu dürfen. Dies wird benötigt, um Gerät-spezifische Einstellungen für die Kamera-Initialisierung herunter- und anonyme Nutzungsdaten hochzuladen. Falls dieses Verhalten unerwünscht ist, sollte die Installation jetzt durch das Antippen des **Abbrechen**-Buttons abgebrochen werden.
5. anschliessend die App durch Antippen des **Öffnen**-Buttons starten oder den Installationsdialog mit Hilfe des **Fertig**-Buttons schließen (Abbildung 5)

3. App-Start

Invisiboga kann entweder durch Antippen des **Öffnen**-Buttons am Ende des Installationsprozesses oder über das App-Menü gestartet werden.



Abbildung 1: Marker

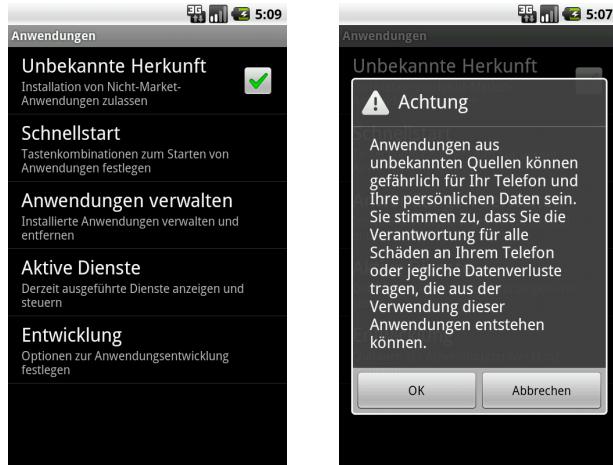


Abbildung 2: Installation von Nicht-Market-Anwendungen zulassen

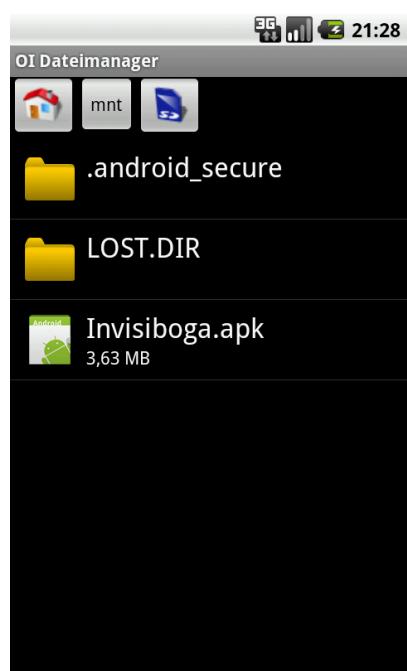


Abbildung 3: Invisiboga.apk



Abbildung 4: Anwendung installieren

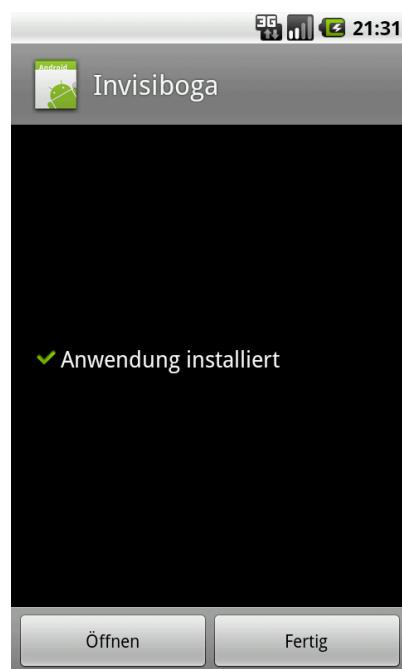


Abbildung 5: Anwendung installiert

4. Bedienungsanleitung

4.1. Initialisierung

Nach dem Start der App werden die für die Ausführung benötigten Daten geladen. Dies wird durch eine Lade-Animation signalisiert (Abbildung 6).



Abbildung 6: Lade-Animation

4.2. Hinweise zur Bedienung

Wenn alles Notwendige geladen ist, wird der Benutzer zunächst darauf hingewiesen, dass die Kamera auf den Marker gerichtet sein muss, um sowohl spielen als auch das Spielgeschehen beobachten zu können (Abbildung 7).

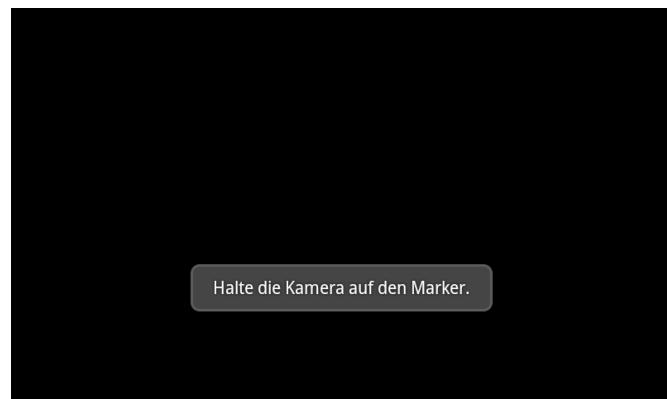


Abbildung 7: Hinweis, dass die Kamera auf den Marker gerichtet sein muss

Befolgt der Benutzer diesen Hinweis und hält die Kamera auf den Marker, erscheinen

nacheinander weitere Hinweise (Abbildung 8), die erklären, wie das Spielfeld und die Sonderfelder erzeugt werden können.

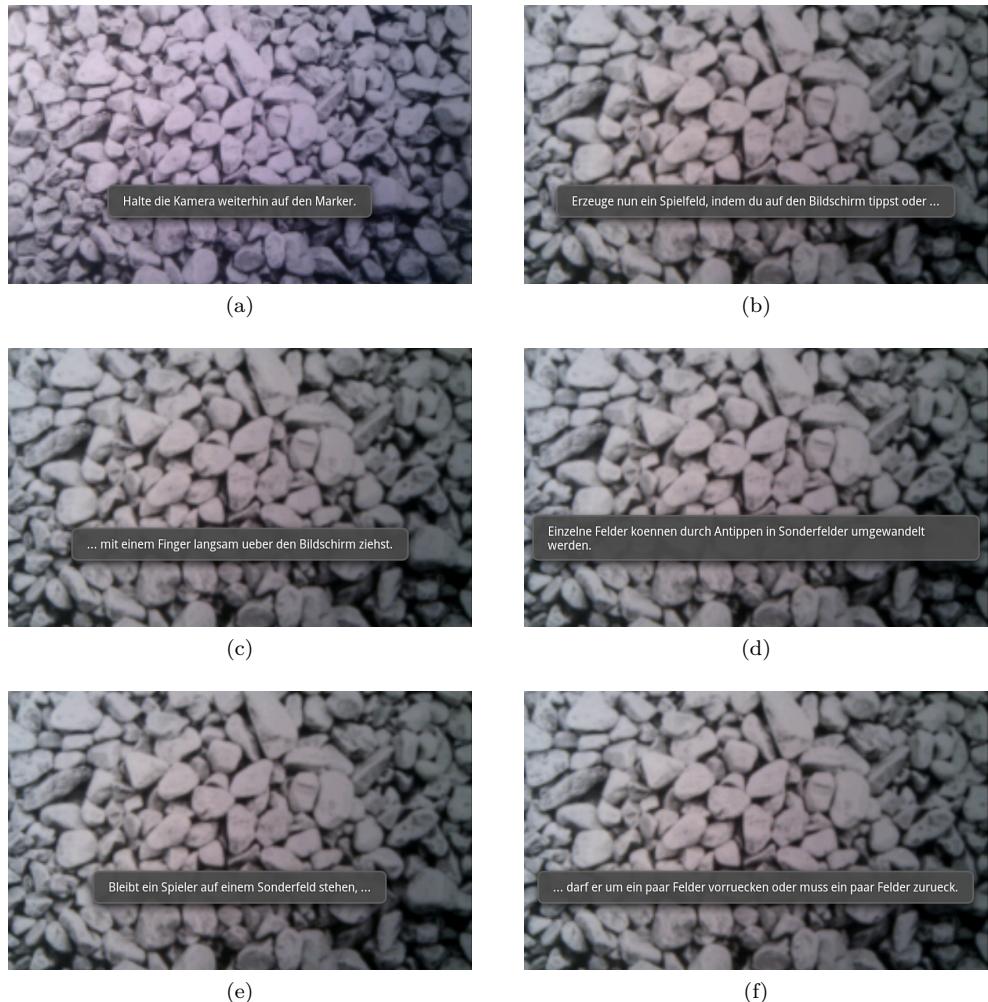


Abbildung 8: Hinweise, wie das Feld inklusive Sonderfelder erzeugt wird

4.3. Spielfeld-Erzeugung, Neustart und nächste Spiel-Phase

Wenn alle Hinweise angezeigt wurden, kann das Spielfeld erzeugt werden. Dafür muss entweder auf einzelne Stellen des Touchscreens getippt oder mit einem Finger über den Touchscreen gezogen werden. Das zuerst erzeugte Feld ist das Startfeld. Auf diesem beginnen die Spieler mit den ersten Zügen. Nach der Erzeugung des Startfeldes wird ein

Neustart-Button sichtbar, mit dem das Spiel nach einer Sicherheitsabfrage jederzeit neugestartet werden kann (Abbildung 9). Das zuletzt erzeugte Feld ist das Zielfeld. Die dazwischen liegenden Felder sind normale Felder (Abbildung 10). Nach der Erzeugung einer bestimmten Anzahl von Feldern wird ein **Weiter**-Button sichtbar, mit dem die nächste Spiel-Phase gestartet werden kann (Abbildung 11).

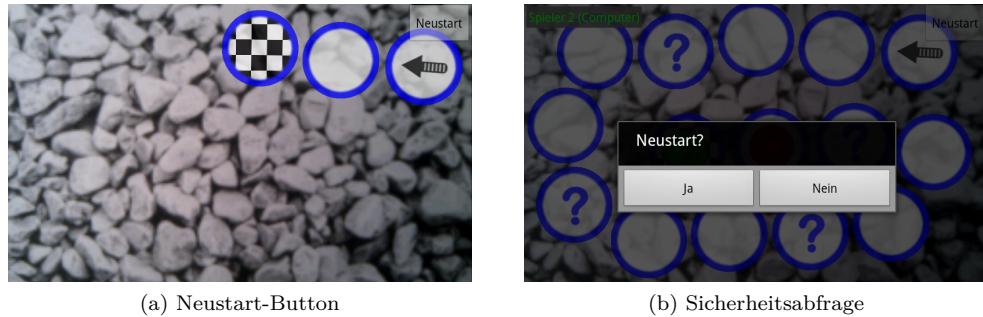


Abbildung 9: Neustart

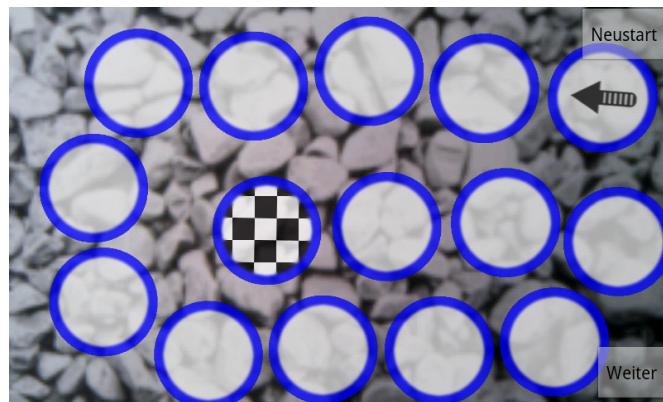


Abbildung 10: Beispiel-Feld

4.4. Sonderfeld-Erzeugung

Steht die Form des Spielfeldes fest, sollten einige normale Felder in Sonderfelder umgewandelt werden, um dem Spiel eine zusätzliche Spannung zu verschaffen. Es können nur die normalen Felder umgewandelt werden. Durch ein Antippen wird aus einem normalen Feld ein Sonderfeld. Durch ein Antippen eines Sonderfeldes kann dieses wieder in ein normales Feld umgewandelt werden (Abbildung 12).

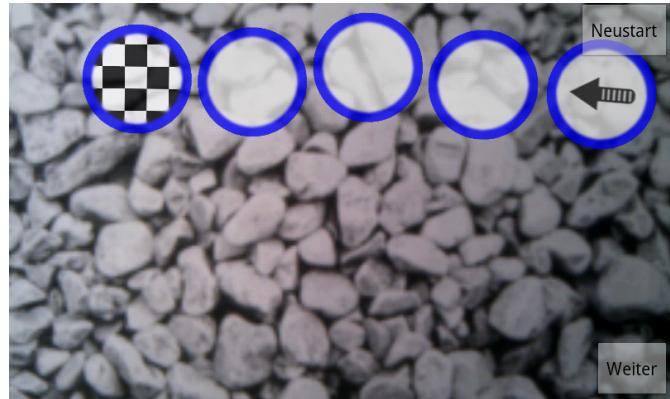


Abbildung 11: Weiter

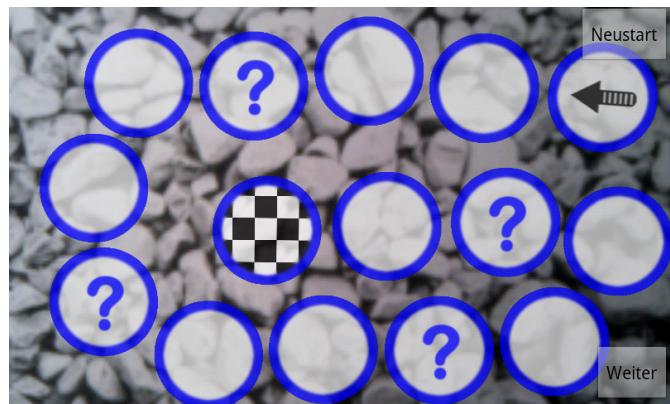


Abbildung 12: Sonderfelder

4.5. Startender Spieler

Wurde das Spielfeld um einige Sonderfelder erweitert, muss der Spieler den Weiter-Button antippen. Jetzt wird per Zufall der startende Spieler bestimmt. Die Kontrahenten dürfen abwechselnd nacheinander würfeln und ihre Spielfigur um die gewürfelte Augenzahl in Richtung des Zielfeldes bewegen. Links oben wird der Name und die Art des Spielers angezeigt, der gerade an der Reihe ist. Die Schriftfarbe entspricht der Spielfigur-Farbe (Abbildung 13).

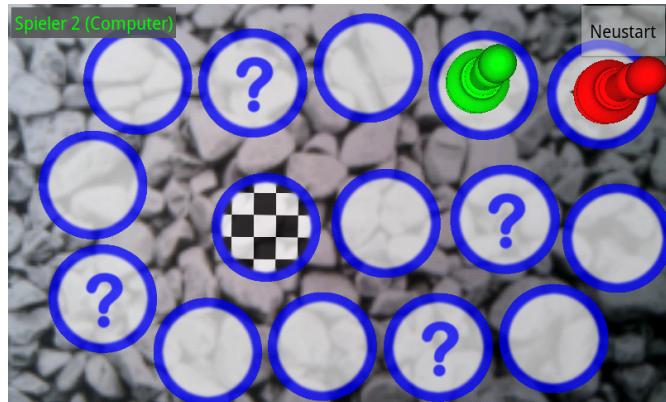


Abbildung 13: Computer-Zug

4.6. Würfeln

Das Würfeln wird für den Computerspieler von der App übernommen. Der menschliche Spieler muss selbst würfeln. Aus diesem Grund wird jedes Mal, wenn der menschliche Spieler an der Reihe ist, links unten ein Würfeln-Button eingeblendet (Abbildung 14). Nach jedem Würfel-Vorgang erscheint eine Meldung mit dem Würfel-Ergebnis (Abbildung 15).

4.7. Besetztes Feld

Wenn eine Spielfigur auf ein bereits besetztes Feld ziehen soll, erscheint eine Meldung und die Figur bleibt auf dem vorherigen Feld, vom Startfeld aus gesehen, stehen (Abbildung 16).

4.8. Sonderfeld

Landet ein Spieler auf einem Sonderfeld, so wird erneut gewürfelt und er darf seine Spielfigur um einige Felder vorrücken oder muss um ein paar Felder zurück (Abbildung 17).

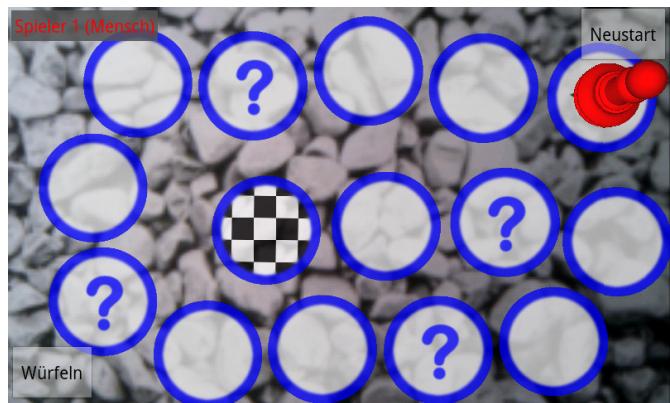


Abbildung 14: Würfeln

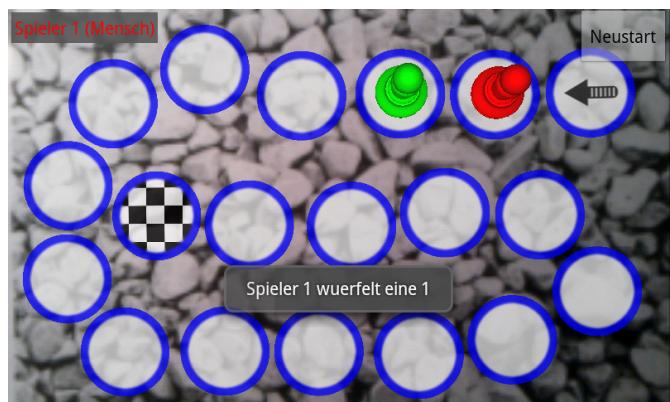
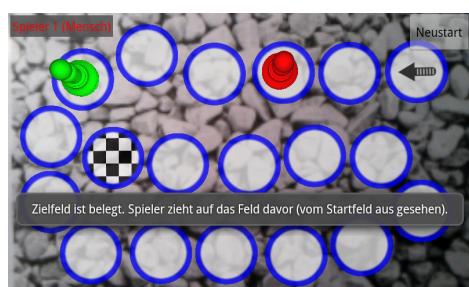
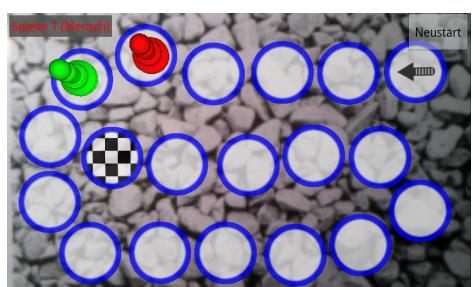


Abbildung 15: Würfel-Ergebnis



(a) Zielfeld belegt



(b) Spieler bleibt auf dem vorherigen Feld stehen

Abbildung 16: Zielfeld ist belegt, Spieler bleibt auf dem vorherigen Feld stehen

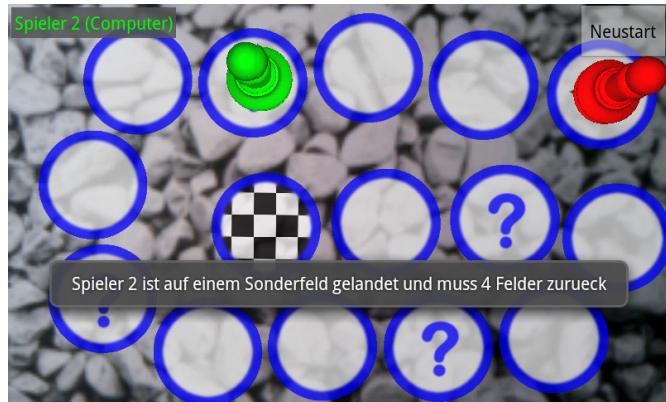


Abbildung 17: Computer landet auf einem Sonderfeld

4.9. Spiel-Ende

Der Spieler, der als Erster das Zielfeld erreicht, gewinnt das Spiel (Abbildung 18). In diesem Fall kann das Spiel durch das Antippen des Neustart-Buttons neugestartet werden.

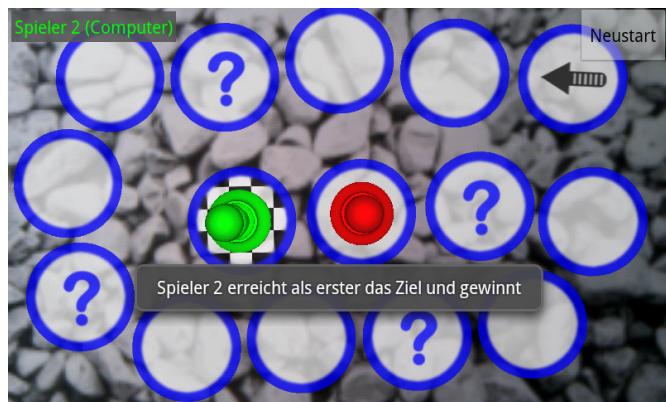


Abbildung 18: Computer gewinnt

4.10. Verschiedene Perspektiven

Das Spielgeschehen kann aus verschiedenen Perspektiven beobachtet werden, indem die Position der Smartphone-Kamera verändert wird (Abbildung 19).

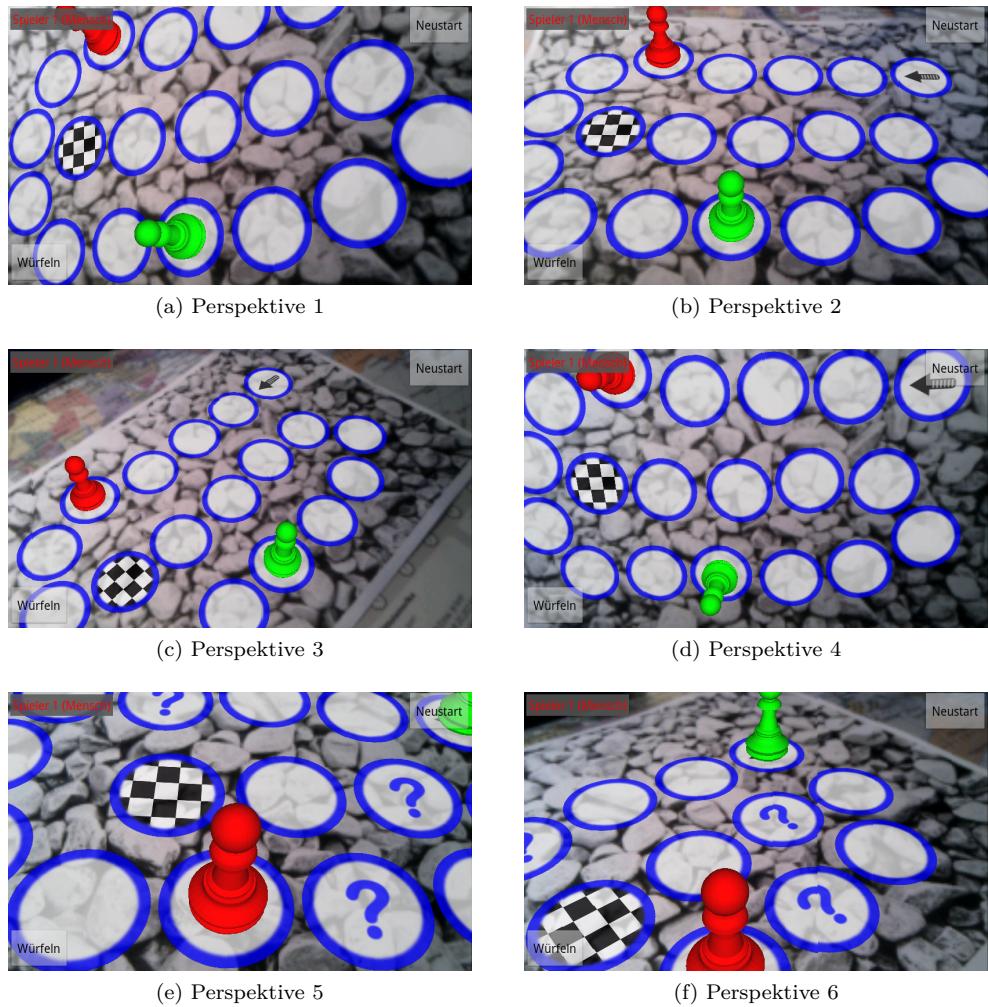


Abbildung 19: Verschiedene Perspektiven

5. Fehlermeldungen

Fehlermeldung	Fehlerursache	Behebungsmaßnahme
QCAR konnte nicht initialisiert werden, das Gerät wird nicht unterstützt.	Das verwendete Gerät wird nicht unterstützt.	Prüfen, ob auf dem verwendeten Gerät das Betriebssystem Android mindestens in der Version 2.2 installiert ist und die App neustartet. Wenn der Fehler weiterhin auftritt, dann ist das Gerät nicht für die App geeignet.
Keine Internet-Verbindung. Diese wird für die Initialisierung der Kamera-Einstellungen benötigt.	Für die Initialisierung wird eine Internet-Verbindung benötigt.	Sicherstellen, dass eine Internet-Verbindung besteht oder eine Verbindung herstellen und die App neustarten. Wenn der Fehler weiterhin auftritt, dann ist das Gerät nicht für die App geeignet.
QCAR konnte nicht initialisiert werden.	Ein unbekannter Fehler ist während der Initialisierung aufgetreten.	Das Gerät neustarten und die App erneut ausführen. Wenn der Fehler weiterhin auftritt, dann ist das Gerät nicht für die App geeignet.

6. Wiederanlaufbedingungen

Die App kann während der Ausführung z.B. durch einen eingehenden Anruf oder eine andere App unterbrochen werden. In diesem Fall wird die App intern pausiert und kann durch ein erneutes Starten weiter ausgeführt werden.

Teil III.

Programmierhandbuch

1. Entwicklungskonfiguration

1.1. Computer

- Intel Core i5, 2.50 GHz
- 8 GB RAM
- NVIDIA NVS 4200M
- Windows 7 Professional Service Pack 1, 64 Bit
- Eclipse Indigo Service Release 1
- Android SDK r14
- QCAR SDK 1.0.6
- Android NDK r6b

1.2. Smartphone

- HTC Desire
- Android 2.3

2. Problemanalyse und Realisation

2.1. Problemanalyse

2.1.1. Augmented Reality

Der Benutzer sollte beim Spielen das Gefühl haben, dass er nicht auf einen Bildschirm schaut, sondern durch ein Fenster eine (reale) Szene beobachtet. Um dies zu erreichen, müssen sich zumindest der Blickwinkel auf die Szene und deren Größe bei jeder Änderung des Kamera-Winkels und der Kamera-Entfernung zur Szene anpassen.

2.1.2. Spielfeld

Es muss eine bequeme Möglichkeit gefunden werden, wie das Spielfeld vom Benutzer erstellt oder eingegeben werden kann. Dieses muss daraufhin von der App erkannt und die Felder in richtiger Reihenfolge abgespeichert werden.

2.1.3. Grafik

Die einzelnen Spielobjekte wie das Spielfeld und die Spielfiguren müssen so dargestellt werden, dass der Benutzer diese auch als solche erkennt. Die Bewegung der Spielfiguren von Feld zu Feld darf nicht sprungartig geschehen. Diese sollte animiert werden, um dem Spieler ein besseres Spielgefühl zu vermitteln. Zusätzlich darf die Anpassung der Darstellung an die Position der Kamera keinen großen Extra-Aufwand bedeuten.

2.1.4. Interaktion

Der Benutzer muss den Spielablauf kontrollieren können. Es müssen Interaktionselemente existieren, um würfeln oder das Spiel neustarten zu können. Auch sollte der Spieler jederzeit wissen, in welchem Stadium sich das Spiel gerade befindet und welcher Spieler an der Reihe ist.

2.1.5. Entwicklung für mobile Systeme

Es darf nicht ausser Acht gelassen werden, dass die App für mobile Systeme entwickelt wird. Solchen Systemen stehen weniger Ressourcen zur Verfügung als Desktop-Systemen. Die App muss also in jeder Hinsicht **effizient** sein!

2.1.6. Logik

Das Spielkonzept verlangt keine besonders komplexe Logik und wird auch aus diesem Grund als letzter Punkt betrachtet. Jedes reale Spiel-Objekt

- Spielfeld
- Würfel
- Spieler
- Spielfigur

muss durch ein entsprechendes logisches Objekt in der App repräsentiert werden. Die beiden Spieler würfeln und ziehen nacheinander. Beim Erreichen eines Sonderfeldes darf der Spieler vor- oder muss zurückrücken. Für die Verwaltung der Zug-Reihenfolge wird ein virtueller Spielleiter benötigt. Dieser beendet auch das Spiel, wenn einer der Spieler das Zielfeld erreicht.

Es ist zu überlegen, ob gewisse Situationen als Konflikte angesehen und behandelt werden. Solch eine Situation entsteht z.B., wenn ein Spieler mit seiner Spielfigur seinen Gegner überholt oder wenn sich beide Spieler auf dem selben Feld befinden.

2.2. Realisationsanalyse

2.2.1. Augmented Reality

Einerseits kann dieses Problem durch die Entwicklung einer Komponente gelöst werden. Diese übernimmt das Tracking realer Objekte, deren Position im Raum relativ

zum Betrachter (in diesem Falle die Gerät-Kamera) ermittelt und die Ergebnisse in geeigneter Form zur Berechnung der Perspektive zur Verfügung stellt.

Andererseits gibt es genau für diesen Anwendungsbereich bereits Bibliotheken auf dem Markt. Diese gilt es herauszusuchen, zu evaluieren und die für das Problem am besten geeignete auszuwählen.

2.2.2. Spielfeld

Das Spielfeld bzw. dessen einzelne Felder können durch je einen Marker dargestellt werden. Es wird pro Feld-Art (Start-, normales -, Sonder- und Zielfeld) eine Marker-Art gebraucht, insgesamt also vier unterschiedliche Marker. Der Benutzer muss die benötigte Anzahl der Marker ausdrucken, evtl. ausschneiden und vor dem Spiel zu einem Spielfeld zusammenlegen. Die Aufgabe der App ist es im Anschluss einzelne Felder zu erkennen und daraus ein Spielfeld zu errechnen. Dabei gilt es das Problem zu lösen, eine vernünftige Reihenfolge der Felder untereinander zu erstellen. Die so erstellte Reihenfolge kann der Anwender im Nachhinein korrigieren oder komplett verändern. Dieser Ansatz erfordert praktische Experimente, um die optimale Marker-Größe zu ermitteln.

Eine weitere Herausforderung bei diesem Ansatz stellt die Größe der Marker dar. Diese dürfen nicht zu klein sein, weil die Gefahr besteht, dass sie von der App nicht mehr erkannt werden. Zu groß sollten sie aber auch nicht sein, weil der Abstand zwischen der Kamera und dem Spielfeld bereits bei einer mittleren Spielfeld-Größe (20-30 Felder) sehr groß müsste, um trotzdem alle Felder erfassen zu können.

Eine Inspiration zu einem komplett anderen Ansatz hat die Recherche nach existierenden Bibliotheken ergeben: <http://youtu.be/teKE5yy5iDk>. Es wird statt einzelner Marker pro Feld ein größerer Marker für das ganze Spielfeld verwendet. Die Felder werden mit Hilfe des Touchscreens erstellt. Dieser wird angetippt und an der angetippten Stelle erscheint ein Feld. Es ist auch möglich mit einem oder mehreren (vorausgesetzt, der Touchscreen ist multi-touch-fähig) Fingern über den Touchscreen zu ziehen und auf den von den Fingern gezogenen Pfaden in bestimmten Abständen Felder zu erstellen.

Beide Ansätze haben gemein, dass geeignete Marker erstellt werden müssen. Diese sollten möglichst viele wiedererkennbare Merkmale besitzen und sich so stark wie möglich von der Umgebung abheben. Somit ist eine optimale Software-seitige Erfassung sichergestellt.

2.2.3. Grafik

Die einfachste und bequemste Art, Spielobjekte grafisch in 3D darzustellen, ist OpenGL. Android unterstützt OpenGL ES in den Versionen 1.1 und 2.0.

2.2.4. Interaktion

Die Interaktion mit dem Benutzer kann durch die Verwendung von User Interface-Elementen, die vom Android-SDK zur Verfügung gestellt werden, realisiert werden.

Eine Alternative wäre es, die Objekte für die Interaktion in OpenGL zu erzeugen und mit der entsprechenden Funktionalität zu belegen.

Einen weiteren Ansatz bieten manche Bibliotheken an. So ist es möglich, einen Bereich auf einem realen Marker als einen virtuellen Knopf zu definieren. Eine reale Berührung dieses Bereiches kann dann mit Hilfe der Bibliothek registriert und eine benutzerdefinierte Aktion ausgelöst werden.

2.2.5. Entwicklung für mobile Systeme

Um eine möglichst hohe Effizienz zu erreichen, gilt es während der Implementierung folgende Punkte zu beachten:

- Wahl effizienter Algorithmen und Datenstrukturen
- unnötige Arbeit vermeiden
- keinen Speicher reservieren, wenn es vermeidbar ist

Diese Regeln gelten auch für die Entwicklung für Desktop-Systeme. Für mobile Systeme ist deren Einhaltung jedoch überlebensnotwendig. Bemerkt der Anwender, dass die App zu viele Ressourcen verbraucht, wird die App deinstalliert und evtl. nie wieder installiert.

2.2.6. Logik

Für die Entscheidung des Spielleiters, wann ein Spieler-Wechsel stattfinden soll und wann das Spiel zu Ende ist, wird ein aktueller Spiel-Zustand benötigt. Dazu gehört mindestens der Spieler, der am Zug ist und die aktuellen Positionen der Spieler auf dem Feld. Dies kann einerseits so realisiert werden, dass jeder Zug gespeichert wird, sodass der aktuelle Zustand jederzeit aus einem Start-Zustand und den bisherigen Zügen berechnet werden kann. Das hätte den Vorteil, dass auch Züge zurückgenommen werden können. Eine andere und für die Anforderungen besser passende Möglichkeit ist die Speicherung des aktuellen Zustandes. Für letztere Möglichkeit ist viel weniger Speicher und Berechnungsaufwand notwendig.

Die Speicherung der aktuellen Spieler-Positionen kann auf zwei Arten erfolgen. In der Datenstruktur für das Spielfeld wird pro Feld die Angabe gespeichert, welcher Spieler auf diesem Feld steht. Oder die aktuelle Position wird in der Spieler-Datenstruktur abgelegt. Das hat wieder den Vorteil, dass weniger Speicher und CPU verbraucht wird. Der eine Ansatz schließt den anderen aber nicht aus.

Wenn eine Spielfigur ein Feld passiert, das bereits von einer anderen Spielfigur besetzt ist, könnte die ziehende die stehende schlagen, sodass die geschlagene wieder zum Startfeld zurückkehren muss. Diese Möglichkeit würde dem Spiel eine zusätzliche Spannung verleihen.

Bleibt ein Spieler auf einem Feld stehen, das von einem anderen Spieler besetzt ist, könnte auch hier die Schlag-Regel zum Einsatz kommen. Ein anderer Ansatz wäre es, den zuletzt auf das besetzte Feld ziehenden Spieler auf dem Feld davor stehen zu lassen.

2.3. Realisationsbeschreibung

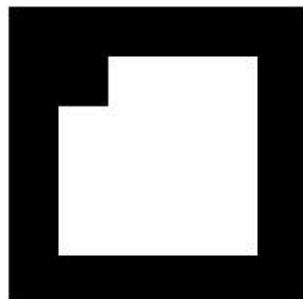
2.3.1. Augmented Reality

Eine selbst entwickelte Komponente würde zu viel Zeit in Anspruch nehmen und ist im Rahmen des Praktikums nicht realisierbar. Es soll also ein fremdes, für diesen Zweck entwickeltes, Framework zum Einsatz kommen.

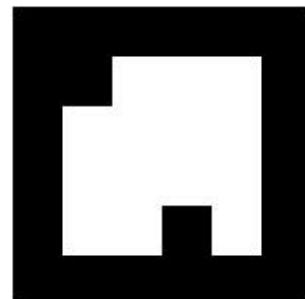
Nach einiger Recherche-Zeit und mehreren Experimenten hat sich ergeben, dass das **QCAR SDK** von der Firma Qualcomm für die Problemstellung am besten geeignet ist.

QCAR SDK Der wichtigste Grund, aus dem sich das QCAR SDK durchgesetzt hat, war die sehr stabile Marker-Erkennung. Zusätzlich haben folgende Faktoren eine Rolle gespielt:

- gute Dokumentation und viele Beispiele
- aktive Community und schnelle Hilfe
- keine einfache Marker, wie sie von den Frameworks NyARToolKit oder DroidAR verwendet werden (Abbildung 20), sondern komplexere Muster (Abbildung 1)



(a)



(b)

Abbildung 20: Marker, die von DroidAR verwendet werden

Das QCAR SDK ist in der Lage jedes von der Kamera aufgenommene Bild zu analysieren und nach einem oder mehreren Markern abzusuchen. Falls im Bild Marker gefunden werden, wird deren Position und Ausrichtung im Raum berechnet und der Umgebung zur Verfügung gestellt.

Android-Apps, die das QCAR SDK verwenden, müssen zu einem großen Teil in C bzw. C++ entwickelt werden. Dafür wird neben dem Android SDK das Android NDK (Native Development Kit) benötigt. Da C vom Android NDK sehr viel besser unterstützt wird als C++, wurde Invisiboga auch überwiegend in C entwickelt.

Android NDK Das Android NDK ermöglicht die Entwicklung Performance-kritischer Komponenten, zu denen auf jeden Fall die Bild-Verarbeitung zählt, auf nativer Ebene. Das NDK verwendet wiederum das JNI (Java Native Interface). Wie der Name schon sagt, stellt JNI eine Schnittstelle zwischen Java und nativem Code zur Verfügung. In Java findet nur noch die Initialisierung des Frameworks statt. Der native Code wird zu einer Bibliothek kompiliert, während der Erstellung der Installationsdatei in dieser verpackt und nach der Installation zur Laufzeit geladen.

```

1 static {
2     loadNativeLibrary(NATIVE_LIB_QCAR);
3     loadNativeLibrary(NATIVE_LIB_INVISIBOGA);
4 }
```

Damit Java native Funktionen aufrufen kann, muss sowohl auf der Java- als auch auf der nativen Ebene einiges beachtet werden. In Java muss zunächst eine Methode mit dem Zugriffsmodifizierer `native` deklariert werden:

```

1 private native void methodName();
```

Die entsprechende Funktion auf der nativen Ebene sieht wie folgt aus:

```

1 JNIEXPORT void JNICALL Java_ClassName_methodName(JNIEnv *env,
2                                                 jobject obj) {
3 }
```

`JNIEXPORT` und `JNICALL` sind vom JNI vordefinierte Makros, `ClassName` ist der vollqualifizierte Klassename, aus dem die native Funktion aufgerufen wird und `methodName` der in Java deklarierte Methodenname. `env` stellt eine Schnittstelle zur JVM (Java Virtual Machine) zur Verfügung und `obj` enthält alle nötigen Informationen über das Objekt, aus dem der Aufruf stattfand.

Der Aufruf einer Java-Funktion aus dem nativen Code ist dagegen ein wenig komplexer. So sieht die in Java aufzurufende Methode aus:

```

1 private void methodName() {
2     ...
3 }
```

Und das ist der entsprechende aufrufende Code auf der nativen Seite:

```

1 void methodName() {
2     JNIEnv *javaEnvironment;
3     javaVm->AttachCurrentThread(&javaEnvironment, NULL);
4     jmethodID method = javaEnvironment->GetMethodID(javaClass, "
5         methodName", "()V");
6     if (method != NULL) {
7         javaEnvironment->CallVoidMethod(javaObject, method);
```

7 }
8 }

Zunächst wird dafür gesorgt, dass die Methode in einem eigenen Thread aufgerufen wird, um Zugriffsverletzungen zu vermeiden. Anschließend wird die Methode anhand des Namens und der Parameter gesucht. Falls eine entsprechende Methode gefunden wird, wird diese aufgerufen.

Target Management System Die Marker, die für eine App benötigt werden, müssen in einem von Qualcomm online zur Verfügung gestellten “Target Management System” hochgeladen und dort verwaltet werden. Anschließend werden sie in eine Ressourcen-Datei überführt, die runtergeladen und in das Android-Projekt eingebunden werden muss.

2.3.2. Spielfeld

Das Spielfeld soll durch das Berühren des Touchscreens erzeugt werden. Für den Benutzer ist das mit Sicherheit am einfachsten und intuitivsten. Ein einfaches Antippen des Bildschirms erzeugt ein Feld. Das Ziehen mit dem Finger erzeugt mehrere Felder hintereinander.

Sobald der Spieler mit dem Touchscreen interagiert, wird auf der Java-Ebene ein Touch-Ereignis ausgelöst. Die Ereignis-Parameter

- Art des Ereignisses
 - Start der Berührung
 - Finger immer noch auf dem Touchscreen
 - Ende der Berührung
 - Abbruch der Berührung (z.B. durch eine andere Aktion)
- x-Koordinate
- y-Koordinate

werden an eine native Funktion weitergereicht und dort ausgewertet. So wird dort bestimmt, ob es sich bei dem aktuellen Ereignis um ein einfaches Tippen oder um ein Ziehen mit dem Finger über den Touchscreen handelt. Im Falle eines einfachen Tippens wird zunächst geprüft, ob sich an der angetippten Stelle bereits ein Feld befindet. Falls ja, wird dieses in ein Sonder-Feld umgewandelt. Ansonsten wird an der Stelle ein neues Feld erzeugt. Wenn ein Ziehen über den Touchscreen festgestellt wird, wird an der aktuellen Stelle ein neues Feld erzeugt.

Das grafische Spielgeschehen findet nicht in der Touchscreen- sondern in der vom Marker definierten Ebene statt. Bevor also ein neues Feld erzeugt wird, ist zunächst eine Koordinaten-Transformation notwendig. Das geschieht mit Hilfe der Projektionsmatrix, die vom QCAR SDK berechnet wird. Zusätzlich findet eine Prüfung statt, ob an der aktuellen Stelle ein Feld erzeugt werden kann, ohne dass sich Felder überlappen.

Das als erstes erzeugte Feld ist das Start-Feld. Dieses ist mit einem Pfeil versehen und wird, nachdem das zweite Feld erzeugt wurde, so gedreht, dass der Pfeil in die Richtung des zweiten Feldes zeigt. Nachdem dieses erzeugt wurde, erscheint ein **Neustart**-Button, mit dem das Spiel neugestartet werden kann. Das als letztes erzeugte Feld ist das Ziel-Feld. Wenn insgesamt fünf Felder erzeugt wurden, wird ein **Weiter**-Button sichtbar. Dieser versetzt das Spiel in die nächste Phase.

2.3.3. Grafik

Wie bereits in Abschnitt 2.2.3 erwähnt, wird der grafische Anteil in OpenGL ES umgesetzt. Es wird OpenGL ES 1.1 verwendet, weil noch viele Geräte im Umlauf sind, die keine vollständige Unterstützung für die Version 2.0 bieten.

Spielfeld Die Felder des Spielfeldes werden als transparente Quadrate gerendert, an die eine entsprechende Textur (Abbildung 21) gebunden ist.

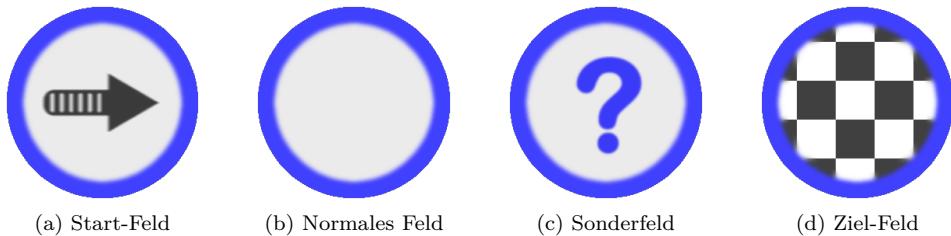


Abbildung 21: Feld-Texturen

Die Texturen werden im Java-Code in folgender Reihenfolge geladen:

1. Start-Feld
2. normales Feld
3. Sonderfeld
4. Ziel-Feld

```

1 private void loadTextures() {
2     mTextures.add(Texture.loadTextureFromApk("startSpace.png",
3         getAssets()));
4     mTextures.add(Texture.loadTextureFromApk("space.png",
5         getAssets()));
6     mTextures.add(Texture.loadTextureFromApk("specialSpace.png",
7         getAssets()));
8     mTextures.add(Texture.loadTextureFromApk("targetSpace.png",
9         getAssets()));
10 }
```

Während der Initialisierung werden die Texturen aus dem Java- in den nativen Code kopiert.

```

1 for (int i = 0; i < textureCount; ++i) {
2     ...
3     textures[i] = Texture::create(env, textureObject);
4 }
```

Auf der nativen Ebene existiert ein `enum`, das die verschiedenen Feld-Arten repräsentiert.

```

1 enum SpaceType {
2     START_SPACE,
3     SPACE,
4     SPECIAL_SPACE,
5     TARGET_SPACE
6 };
```

Bei der Erzeugung eines Feldes wird die Feld-Art (hier als Beispiel ein normales Feld) in der repräsentierenden Variable gespeichert.

```

1 space->type = SPACE;
```

Während des Renders wird die an das Quadrat zu bindende Textur anhand der gespeicherten Feld-Art bestimmt.

```

1 glBindTexture(GL_TEXTURE_2D, textures[space.type]->mTextureID);
```

Spielfiguren Für die Darstellung der Spielfiguren kommt eine Lichtquelle zum Einsatz. Bevor die Model-View-Matrix geladen wird, wird die Position der Lichtquelle definiert und dieser zugewiesen.

```

1 const GLfloat light0Position[] = { 0.0, 0.0, 0.0, 1.0 };
2 glLightfv(GL_LIGHT0, GL_POSITION, &light0Position[0]);
3
4 glMatrixMode(GL_MODELVIEW);
5 glLoadMatrixf(modelViewMatrix.data);
```

Durch diese Vorgehensweise wird die Lichtquelle an die Position des Beobachters (in diesem Fall die Geräte-Kamera) gebunden. Die Spielfiguren werden also immer aus der Richtung beleuchtet, aus der die Szene beobachtet wird (Abbildung 19).

Die Vertices und Normale wurden mit dem Skript `obj2opengl` aus einem frei im Internet verfügbaren 3D-Model im `obj`-Format erzeugt.

Die einzelnen Lichtkomponenten (ambient, diffus und spekular) je Spielfigur sind als Konstanten definiert.

```

1 static const Light PAWN_LIGHTS[PLAYER_COUNT] = {
2     {
3         // rot
4         { 1.f, 0.f, 0.f, 1.f },
5         { 1.f, 0.f, 0.f, 1.f },
6         { 1.f, 1.f, 1.f, 1.f } },
7     {
8         // gruen
9         { 0.f, 1.f, 0.f, 1.f },
10        { 0.f, 1.f, 0.f, 1.f },
11        { 1.f, 1.f, 1.f, 1.f } }
12 };

```

Diese Werte werden während der Initialisierung der jeweiligen Spielfigur

```

1 players[i].pawn.light = PAWN_LIGHTS[i];

```

und beim Rendern der Lichtquelle zugewiesen.

```

1 glLightfv(GL_LIGHT0, GL_AMBIENT, pawn->light.ambient);
2 glLightfv(GL_LIGHT0, GL_DIFFUSE, pawn->light.diffuse);
3 glLightfv(GL_LIGHT0, GL_SPECULAR, pawn->light.specular);

```

Animation Die Bewegung der Spielfiguren von Feld zu Feld ist animiert. Die Animation muss auf allen mobilen Geräten gleich schnell ablaufen. Sie darf also nicht von der CPU-Geschwindigkeit abhängen. Diese Anforderung wird wie folgt realisiert:

Bei jedem Aufruf der Render-Funktion wird das Zeitintervall seit letztem Rendern gespeichert.

```

1 float timeIntervalSinceLastFrame = (getCurrentTimeInMs() -
    lastFrameTime) / 1000.0f;

```

Die Geschwindigkeit wird für jedes Feld-Paar neu berechnet. Diese hängt einerseits von der Entfernung der beiden Felder ab und andererseits von einem konstanten Faktor.

```

1 QCAR::Vec2F velocity = vec2FSub(pawn->targetSpace->position, pawn
    ->currentSpace->position);
2 pawn->velocity = vec2FScale(velocity, TRANSLATIONS_PER_SECOND);

```

Aus der aktuellen Position der Spielfigur, ihrer aktuellen Geschwindigkeit und dem Zeitintervall seit letztem Rendern ergibt sich die neue Geschwindigkeit.

```

1 pawn->position = vec2FAdd(pawn->position, vec2FScale(pawn->
    velocity, timeIntervalSinceLastFrame));

```

2.3.4. Interaktion

Die Interaktion mit dem Benutzer wird zu einem Teil in OpenGL und zu einem anderen Teil durch die vom Android SDK zur Verfügung gestellten User Interface-Elemente realisiert.

Der OpenGL-Anteil betrifft die Spielfeld-Erzeugung und wurde bereits in Abschnitt 2.3.2 beschrieben.

Dank den Android UI-Elementen stehen dem Benutzer folgende Interaktionsobjekte zur Verfügung:

Neustart Dieser Knopf erscheint, nachdem das Start-Feld erzeugt wurde (Abbildung 9) und kann jederzeit für einen Neustart des Spiels verwendet werden.

Weiter Dieser Knopf wird sichtbar, wenn fünf Felder vorhanden sind (Abbildung 11). Er kann angetippt werden, wenn das Spielfeld erzeugt wurde. Anschließend wird gewürfelt und die Spielfiguren gezogen.

Würfeln Der menschliche Spieler tippt diesen Knopf an, um zu würfeln. Er erscheint immer, wenn der Computer-Spieler seinen Zug abgeschlossen hat.

Aktueller Spieler Dies ist ein Feld, in dem der aktuelle Spielername und die Spieler-Art angezeigt wird. Die Schriftfarbe entspricht der Farbe der Spielfigur (Abbildung 13).

Toast-Benachrichtigung So wird im Android SDK ein Hinweis genannt, der jederzeit dezent auf der Oberfläche erscheinen kann und den Benutzer über ein Ereignis informiert. Die Toasts werden z.B. genutzt, um dem Benutzer eine Hilfe-Stellung zu geben, über ein Würfel-Resultat oder den Gewinner des Spiels zu informieren (Abbildungen 7, 8, 15 und 18).

Das User-Interface einer Android-App besteht aus sogenannten `View`- und `ViewGroup`-Objekten. Diese sind hierarchisch untereinander angeordnet, wobei die Hierarchie beliebig komplex sein kann. Die `View`-Klasse dient als Basis für alle User Interface-Komponenten wie Eingabefelder oder Knöpfe.

Die Invisiboga-Oberfläche besteht aus zwei `Views`:

InvisibogaGLSurfaceView Stellt einen Bereich zur Verfügung, auf dem die per OpenGL erzeugten Inhalte dargestellt werden. Sie implementiert das Interface `Renderer` und die in bestimmten Abständen aufgerufene abstrakte Methode `onDrawFrame`, die wiederum die native Methode `renderFrame` aufruft (Listing 2.3.4).

OverlayView Eine transparente `View`, die als Container für UI-Elemente wie z.B. Knöpfe oder Textfelder dient.

```
1 @Override  
2 public void onDrawFrame(GL10 gl) {  
3     renderFrame();  
4 }
```

Beim Start einer Android-App wird vom Android SDK ein Hauptthread erzeugt. Dieser ist u.a. für die Kommunikation mit und unter den UI-Komponenten zuständig und wird aus diesem Grund auch UI-Thread genannt. Die Manipulation des UIs (Knöpfe ein-/ausblenden, Textfelder beschreiben usw.) ist nur aus diesem Thread möglich. Für das Rendern wird ein eigener Thread erzeugt, der GL-Thread. Die Methode `onDrawFrame` wird innerhalb dieses Threads aufgerufen. D.h., es ist nicht ohne weiteren Aufwand möglich, aus der nativen `renderFrame`-Funktion das UI zu verändern. Um diese Anforderung trotzdem zu realisieren, ist eine Nachrichten-basierte Thread-Kommunikation über einen `Handler` notwendig.

Zunächst ist ein `Handler` notwendig, der sowohl alle Nachrichten-Arten (hier als Beispiel die Nachricht zum Einblenden einer `View`) kennt, die verschickt werden, als auch die `View`, die die zu manipulierenden UI-Komponenten beinhaltet.

```

1 public class OverlayViewHandler extends Handler {
2     protected static final int SHOW_VIEW = 0;
3     ...
4
5     public OverlayViewHandler(OverlayView overlayView) {
6         mOverlayView = overlayView;
7     }
8 }
```

Bei der Instanziierung der `OverlayView` wird eine Instanz des `OverlayViewHandlers` erzeugt.

```

1 public OverlayView(Context context) {
2     ...
3     mHandler = new OverlayViewHandler(this);
4     ...
5 }
```

Eine Referenz auf die `OverlayView` ist in der `InvisibogaGlSurfaceView` gespeichert, um über diese Referenz Nachrichten an den `OverlayViewHandler` schicken zu können.

```

1 public InvisibogaGlSurfaceView(Context context, OverlayView
2     overlayView) {
3     ...
4     mOverlayView = overlayView;
5 }
```

Für das Einblenden einer `View` anhand des `View`-Namens vom nativen Code aus ist die Funktion `showView` zuständig.

```

1 void showView(const char *name) {
2     JNIEnv *javaEnvironment;
3     javaVm->AttachCurrentThread(&javaEnvironment, NULL);
```

```

4     jmethodID method = javaEnvironment->GetMethodID(javaClass, "showView", "(Ljava/lang/String;)V");
5     if (method != NULL) {
6         jstring javaString = javaEnvironment->NewStringUTF(name);
7         javaEnvironment->CallVoidMethod(javaObject, method,
8                                         javaString);
9     }
}

```

Diese native Funktion ruft die entsprechende Java-Methode auf.

```

1 private void showView(String name) {
2     Message.obtain(mOverlayView.mHandler, OverlayViewHandler.SHOW_VIEW, name).sendToTarget();
3 }

```

Es wird ein Nachricht-Objekt aus einem globalen Nachrichten-Pool entnommen, mit den nötigen Parametern

- Handler-Referenz
- Nachricht-Art
- View-Name

belegt und an den in den Parametern übergebenen Handler gesendet. Der `OverlayViewHandler` wertet beim Empfang der Nachricht die Nachricht-Art aus und führt die entsprechende(n) Aktion(en) durch.

```

1 @Override
2 public void handleMessage(Message message) {
3     switch (message.what) {
4         case SHOW_VIEW:
5             mOverlayView.showView((String) message.obj);
6             break;
7         ...
8     }
9 }

```

Die `showView`-Methode in der Klasse `OverlayView` sorgt letztendlich für das Einblenden der `View`.

```

1 protected void showView(String name) {
2     setViewVisibility(name, true);
3 }

```

Der ganze Ablauf ist in Abbildung 22 als Sequenzdiagramm grafisch abgebildet.

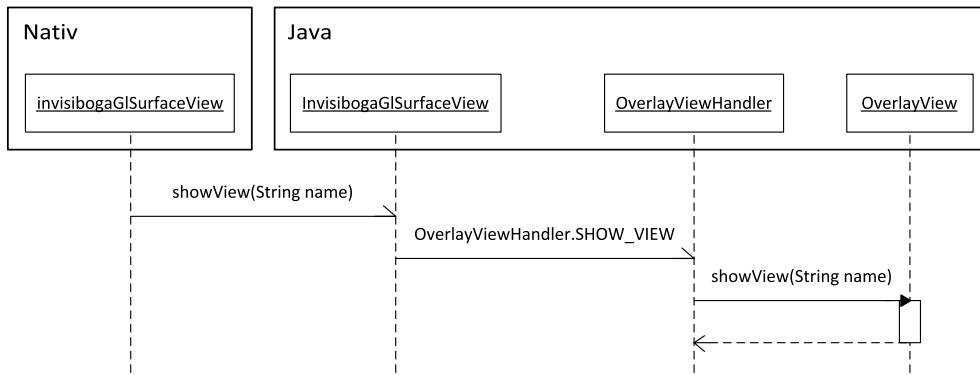


Abbildung 22: Manipulation des User Interfaces

2.3.5. Entwicklung für mobile Systeme

Hier unterstützen die Android-Entwickler die App-Entwickler und veröffentlichen auf [Android Developers](#) viele Tipps und Tricks sowohl auf Java als auch auf nativer Ebene, um Android-Apps möglichst effizient zu gestalten. U.a. werden dort folgende Aspekte angesprochen:

- Getter und Setter sollten im Gegensatz zur allgemeinen Vorgehensweise vermieden und lieber direkt auf die Variable zugegriffen werden.
- Konstante werden durch die Zugriffsmodifizierer `static final` deklariert.
- Für die Iteration über Arrays und alle Klassen, die das `Iterable`-Interface implementieren, wird die erweiterte `for`-Schleifen-Syntax `for (MyType a : mArray) { ... }` verwendet.
- Ganzzahlen-Operationen sind Gleitkommazahlen-Operationen vorzuziehen.

All diese und weitere Praktiken wurden während der Entwicklung beachtet und so gut wie möglich umgesetzt.

2.3.6. Logik

Der Spielleiter wird durch das Modul `game` repräsentiert. Hier wird auch der aktuelle Spiel-Zustand verwaltet.

Globale Variablen Dieser Zustand ist global, was Software-technisch nicht besonders gut, aber anders auch nicht realisierbar ist. Es gibt keine klassische `main`-Funktion, die als Programm-Einstiegspunkt dient und in der der Zustand lokal gespeichert werden kann. Die `renderFrame`-Funktion im Modul `InvisibogaGLSurfaceView` ist die einzige Funktion, die periodisch aufgerufen wird, wenn die App initialisiert wurde und bevor sie deinitialisiert wird. Die fehlende Indirektion und Kapselung hat aber auch Vorteile in Bezug auf Entwicklung für mobile Systeme wie in Abschnitt 2.3.5 erwähnt. Neben dem Spiel-Zustand existieren weitere globale Variablen wie z.B. das Spielfeld oder die vergangene Zeit seit der Berechnung des letzten Frames. Während der Entwicklung wurde jedoch darauf geachtet, dass die Anzahl der globalen Variablen so klein wie möglich bleibt.

Die am Spiel teilnehmenden Spieler sind in einem Array gespeichert.

```
1 Player players[PLAYER_COUNT];
```

Der aktuelle Spiel-Zustand ist durch die beiden Variablen `currentPlayerIndex` und `currentPlayer` repräsentiert. `currentPlayerIndex` ist dabei der Index des aktuellen Spielers im `players`-Array und `currentPlayer` ein Zeiger auf den Spieler selbst.

```
1 int currentPlayerIndex;
2 Player *currentPlayer;
```

Das aktuelle Feld, auf dem der Spieler steht, ist in der Spieler-Datenstruktur gespeichert. Neben dem aktuellen ist auch das Ziel-Feld, das für einen Zug notwendig ist, abgelegt und ebenso die Spielfigur.

```
1 typedef struct Player {
2     ...
3     int currentSpaceIndex;
4     int targetSpaceIndex;
5     Pawn pawn;
6     ...
7 } Player;
```

Ziehen

game Während der Initialisierung wird im `game`-Modul der beginnende Spieler per Zufall bestimmt.

```
1 setCurrentPlayer(rand() % PLAYER_COUNT);
```

Im späteren Spielverlauf wird in der Funktion `processMove` der nächste Spieler im Spieler-Array aktiviert.

```
1 set currentPlayer((currentPlayerIndex + 1) % PLAYER_COUNT);
```

Vor der Aktualisierung des Spielers wird jedoch geprüft, ob der aktuelle Zug und evtl. das Spiel damit zu Ende ist. Es wird auch kontrolliert, ob der Spieler auf einem Sonderfeld gelandet ist. In diesem Fall wird ein zufälliger Wert bestimmt, der sowohl negativ als auch positiv sein kann, dessen Absolutwert aber die Augenzahl auf dem Würfel nicht überschreiten darf.

```
1 int sign = rand() % 2;
2 int value = rand() % MAX_DIE_PIP_COUNT + 1;
3 value *= sign == 0 ? +1 : -1;
```

Am Ende übergibt die `processMove`-Funktion die Kontrolle an die `processPlayerMove`-Funktion im Modul `player`.

player Der Spieler bereitet seinen Zug vor, indem er würfelt und anhand des Ergebnisses das Zielfeld aktualisiert.

```
1 int diePips = rand() % MAX_DIE_PIP_COUNT + 1;
2 setPlayerTargetSpace(player, player->targetSpaceIndex + diePips);
3 prepareMove(player);
```

In der Funktion `prepareMove` wird zunächst das gesetzte Zielfeld analysiert. Falls es besetzt ist, wird ein neues Zielfeld, das Feld davor, gesetzt. Als nächster Schritt wird die Spielfigur des Spielers auf den Zug vorbereitet, aber nur, wenn der Spieler überhaupt ziehen kann. Das ist z.B. nicht der Fall, wenn ein Spieler eine Eins würfelt und das folgende Feld besetzt ist.

```
1 if (field.spaces[player->targetSpaceIndex].occupied) {
2     player->targetSpaceIndex = max(0, player->targetSpaceIndex -
3         1);
4 }
5 if (player->targetSpaceIndex != player->currentSpaceIndex) {
6     preparePawnMove(player);
7 }
8 ...
```

Der Ablauf beim Ziehen ist wie im wirklichen Leben realisiert. Der Spieler würfelt und hat somit ein bestimmtes Feld als ein globales Ziel. Die Spielfigur bekommt immer wieder die Anweisung von dem aktuellen Feld zum nächsten zu ziehen. Das wird solange fortgesetzt, bis die Spielfigur das Feld erreicht, das im Spieler als globales Ziel gesetzt ist. Dann ist der Zug des aktuellen Spielers zu Ende und die Kontrolle übernimmt

wieder das `game`-Modul. Beim Erreichen des jeweils nächsten Feldes wird das erreichte Feld zum aktuellen Feld, sowohl in der Spieler- als auch in der Spielfigur-Datenstruktur.

In der Funktion `preparePawnMove` wird das aktuelle Feld der Spielfigur mit dem aktuellen Feld des Spielers synchronisiert und abhängig von der aktuellen Ziehrichtung das nächste Zielfeld für die Spielfigur bestimmt.

```
1 setPawnCurrentSpace(&player->pawn, player->currentSpaceIndex);
2 int pawnTargetSpaceIndex = player->targetSpaceIndex >= player->
   currentSpaceIndex ? player->currentSpaceIndex + 1 : player->
   currentSpaceIndex - 1;
3 setPawnTargetSpace(&player->pawn, pawnTargetSpaceIndex);
```

Am Ende übergibt die `processPlayerMove`-Funktion die Kontrolle an die `processPawnMove`-Funktion im `pawn`-Modul.

pawn Analog zum `player`-Modul wird auch im `pawn`-Modul der Zug der Spielfigur zunächst vorbereitet, indem der Geschwindigkeitsvektor vom aktuellen Feld in Richtung des nächsten Feldes berechnet wird. Anschließend bewegt sich die Figur in die berechnete Richtung mit einer entsprechenden Geschwindigkeit. Wie das genau abläuft, wurde im Abschnitt [2.3.3](#) bereits beschrieben. Nach jedem Animationsschritt wird der Abstand vom aktuellen zum nächsten Feld und der Abstand vom aktuellen Feld zur Position der Spielfigur verglichen. Falls der zweite Abstand größer oder gleich dem ersten ist, dann hat die Spielfigur ihr Ziel erreicht und die Kontrolle wird wieder an das `player`-Modul abgegeben.

```
1 float sourceSpaceTargetSpaceDistance = vec2FDistance(pawn->
   currentSpace->position, pawn->targetSpace->position);
2 float sourceSpaceCurrentPositionDistance = vec2FDistance(pawn->
   currentSpace->position, pawn->position);
3 if (sourceSpaceCurrentPositionDistance >=
   sourceSpaceTargetSpaceDistance) {
4     ...
5 }
```

3. Beschreibung grundsätzlicher Datenstrukturen

3.1. Zustandsautomaten

In Invisiboga kommt an mehreren Stellen das Konzept der Zustandsautomaten zum Einsatz, um Abläufe zu verwalten und Probleme zu vermeiden, die aufgrund von asynchronen Aufrufen entstehen.

3.1.1. Invisiboga ([Abbildung 23](#))

Das ist der Zustandsautomat, der den App-Ablauf verwaltet. Hier werden sowohl die nativen Bibliotheken als auch die Texturen geladen und initialisiert. Die Kamera wird

ein- und ausgeschaltet, wenn es notwendig ist.

3.1.2. Spiel (Abbildung 24)

Dieser Automat wird verwendet, um z.B. festzustellen, wenn die Kamera auf den Marker gehalten wird und der entsprechende Hinweis angezeigt werden muss.

3.1.3. Spieler (Abbildung 25)

Für die Verwaltung der Zustände eines Spielers wird dieser Automat genutzt. Hier wird zwischen einem menschlichen und einem Computer-Spieler unterschieden. Der menschliche Spieler muss selbst würfeln. Es wird also ein Zustand benötigt, der auf eine Aktion wartet. Der Computer-Spieler kann diesen Zustand überspringen und gleich ziehen. Vor dem Ziehen muss der Zug aber vorbereitet werden. Dazu gehört z.B. das Würfeln und die Initialisierung der Spielfigur.

Es existiert auch ein Zustand, um den Spieler eine bestimmte Zeit in einem Zustand zu halten. Diese Möglichkeit wird z.B. genutzt, um Spieler-Wechsel zu verlangsamen und dem Spieler dadurch einen flüssigeren Spielablauf zu vermitteln.

3.1.4. Spielfigur (Abbildung 26)

Dieser Automat verwaltet die Zustände einer Spielfigur. Wie beim Spieler-Zustandsautomaten existieren Zustände für Situationen vor dem Zug, für den Zug selbst und nach dem Zug. Es ist auch eine Möglichkeit vorhanden, Aktionen zu verzögern. So wird z.B. eine kurze Pause eingelegt, wenn die Spielfigur ein Feld erreicht, der Zug aber noch nicht beendet ist.

3.2. Datenstrukturen

In diesem Abschnitt werden die für die App definierten Datenstrukturen beschrieben.

3.2.1. Light

```
1 typedef struct Light {  
2     GLfloat ambient [4];  
3     GLfloat diffuse [4];  
4     GLfloat specular [4];  
5 } Light;
```

Die Lichtquelle, von der eine Spielfigur beleuchtet wird, wird durch die ambivalenten, diffusen und spekularen RGBA-Anteile des Lichtes beschrieben.

3.2.2. Field

```

1  typedef struct Field {
2      int length;
3      Space spaces[MAX_SPACE_COUNT];
4 } Field;

```

Diese Datenstruktur repräsentiert das Spielfeld, das durch die Anzahl der Felder und die Felder selbst definiert ist. Der Aufbau eines Feldes wird in Abschnitt 3.2.5 beschrieben.

3.2.3. Pawn

```

1  typedef struct Pawn {
2      PawnState state;
3      PawnState nextState;
4      QCAR::Vec2F position;
5      QCAR::Vec2F velocity;
6      Space *currentSpace;
7      Space *targetSpace;
8      unsigned int continueTime;
9      Light light;
10 } Pawn;

```

Eine Spielfigur ist durch folgende Komponenten definiert:

state, nextState und continueTime Diese Komponenten sind für die Verwaltung der Zustände nötig. **state** ist der aktuelle Zustand. **nextState** ist der nächste Zustand und wird zum aktuellen, wenn die Zeit **continueTime** abgelaufen ist. Die einzelnen Zustände wurden bereits in Abschnitt 3.1.4 ausführlich beschrieben.

position aktueller Koordinatenvektor in der vom Marker definierten Ebene

velocity aktueller Geschwindigkeitsvektor

currentSpace und targetSpace aktuelles Feld, auf dem die Spielfigur steht und Zielfeld, auf das die Spielfigur ziehen soll

light Lichtkomponenten für die Leuchtquelle

3.2.4. Player

```

1  typedef struct Player {
2      char name[15];
3      PlayerType type;
4      PlayerState state;
5      PlayerState nextState;
6      int currentSpaceIndex;

```

```

7     int targetSpaceIndex;
8     Pawn pawn;
9     unsigned int continueTime;
10 } Player;

```

Folgende Felder beschreiben einen Spieler:

name Spieler-Name

type Spieler-Typ, entweder `PLAYER_TYPE_COMPUTER` (Computer-Spieler) oder `PLAYER_TYPE_HUMAN` (menschlicher Spieler)

state, nextState und continueTime der gleiche Verwendungszweck wie bei der Spielfigur-Datenstruktur

currentSpaceIndex und targetSpaceIndex Index des aktuellen Feldes, auf dem der Spieler bzw. seine Spielfigur steht und Index des Zielfeldes, auf das der Spieler ziehen darf bzw. muss

pawn Spielfigur des Spielers

3.2.5. Space

```

1 typedef struct Space {
2     int id;
3     SpaceType type;
4     QCAR::Vec2F position;
5     float angle;
6     bool occupied;
7 } Space;

```

Ein Feld ist definiert durch:

id Id und gleichzeitig die Position des Feldes im Feld-Array

type Typ des Feldes aus dem Wertebereich `START_SPACE` (Startfeld), `SPACE` (normales Feld), `SPECIAL_SPACE` (Sonderfeld) und `TARGET_SPACE` (Zielfeld)

position Koordinatenvektor in der vom Marker definierten Ebene

angle Winkel, um den das Feld evtl gedreht werden muss. Kommt momentan nur beim Startfeld zum Einsatz, um das Feld und somit auch die Textur mit dem Pfeil in Richtung des zweiten Feldes zu drehen.

occupied Flag, das anzeigt, ob das Feld von einer Spielfigur besetzt ist

3.2.6. TouchEvent

```
1 typedef struct TouchEvent {
2     ActionType actionType;
3     QCAR::Vec2F startPosition;
4     QCAR::Vec2F currentPosition;
5     QCAR::Vec2F lastPosition;
6     QCAR::Vec2F tapPosition;
7     bool isTap;
8     unsigned long startTime;
9     unsigned long lifeTime;
10    float startPositionLastPositionDistance;
11    bool isActive;
12 } TouchEvent;
```

Diese Datenstruktur ist für die Speicherung und Auswertung der Daten, die beim Berühren des Touchscreens entstehen, notwendig.

actionType Art der Aktion, die den Wert ACTION_DOWN (Start einer Geste), ACTION_MOVE (Fortsetzung einer Geste), ACTION_UP (Ende einer Geste) oder ACTION_CANCEL (Abbruch einer Geste) haben kann.

startPosition, lastPosition und currentPosition, startPositionLastPositionDistance
Start-, letzte und aktuelle Position der Berührung und die Distanz zwischen der Start- und letzten Position. Diese Angaben sind notwendig, um berechnen zu können, ob der Benutzer auf den Touchscreen getippt hat oder den Finger auf diesem bewegt.

tapPosition Tipp-Position, falls ein Tippen erkannt wurde

isTap Flag, ob ein Tippen erkannt wurde

startTime und lifeTime Startzeit und das Alter der Geste. Auch diese Angaben werden benötigt, um sicher ein Tippen auf den Touchscreen zu erkennen.

isActive Flag, das anzeigt, ob das Event aktiv ist

4. Architektur

In der Abbildung 27 sind zwei große Blöcke zu sehen, die beide kleinere Blöcke beinhalten. Der Block auf der linken Seite repräsentiert die Java- und der Block auf der rechten Seite die native Ebene.

Der Java-Block kann als ein Klassendiagramm gelesen werden. Die kleinen Blöcke stellen die einzelnen Klassen und die Pfeile die Beziehungen zwischen den Klassen dar. Die Klasse Log wird von jeder anderen Klasse verwendet. Die Pfeile wurden für eine bessere Übersicht weggelassen.

Im nativen Block steht jeder kleine Block für ein natives Modul. Die Pfeilrichtung beschreibt, welches Modul welche anderen Module einbindet. So bedeutet z.B. ein Pfeil vom Block x zu Block y, dass das Modul x das Modul y einbindet. Die separaten Blöcke auf der rechten Seite werden von so gut wie jedem anderen Modul eingebunden. Die Pfeile wurden für eine bessere Übersicht weggelassen.

Zwischen den großen Blöcken sind drei Pfeile zu sehen. Hier kann abgelesen werden, über welche Klassen und welche nativen Module die Kommunikation zwischen der Java- und der nativen Schicht abläuft. Der Pfeil von einer Java-Klasse zu einem Modul bedeutet, dass aus der Java-Klasse native Funktionen aufgerufen werden. Ein umgekehrter Pfeil heißtt, dass aus dem nativen Modul Java-Methoden aufgerufen werden.

5. Programmtests

Während der Entwicklung wurde die App immer wieder gestartet und zusätzlich zum normalen Ablauf gewisse Situationen provoziert und geprüft, wie die Anwendung reagiert. Insbesondere wurden folgende Konstellationen getestet:

- Beim ersten Halten der Geräte-Kamera auf den Marker wird ein Hinweis angezeigt.
- Das Antippen eines normalen Feldes wechselt den Status zum Sonderfeld. Ein erneutes Antippen wechselt den Status zurück zum normalen Feld.
- Die Spieler dürfen nicht über das Start- und das Zielfeld hinausziehen.
- Das Ziehen auf ein besetztes Feld ist nicht möglich, der ziehende Spieler bleibt auf dem vorherigen Feld stehen.
- Wenn ein Spieler eine Eins würfelt und das nächste Feld besetzt ist, darf er nicht ziehen.
- Wenn ein Spieler das Zielfeld erreicht, ist das Spiel zu Ende.

6. Fazit

6.1. Einschränkungen und bekannte Probleme

6.1.1. Während der Entwicklung

Das größte Problem während der Entwicklung war die Mischung zwischen Java- und nativem Code. Um genauer zu sein waren es die Aufrufe von Java-Methoden aus dem nativen Code. Die App stürzte oft mit kryptischen Fehlermeldungen ab. Die eingeschränkten Debug-Möglichkeiten auf der nativen Ebene mussten durch sehr ausführliche Log-Ausgaben ausgeglichen werden. Nach langer Recherche wurde auch der sogenannte CheckJNI-Modus entdeckt. Eine App, die native Bibliotheken verwendet,

kann in diesem Modus ausgeführt werden. Der Entwickler wird dann gewarnt, wenn fehlerhafte Stellen im Code gefunden werden oder wenn Aufrufe stattfinden, die Probleme verursachen könnten. Am Ende hat sich herausgestellt, dass die Aufrufe der Java-Methode aus dem nativen Code in einem falschen JVM-Thread gestartet wurden und die App durch fehlerhafte Speicherzugriffe abstürzt ist. Die Auslagerung der Aufrufe in einen separaten Thread durch `javaVm->AttachCurrentThread(&javaEnvironment, NULL)` hat das Problem behoben.

6.1.2. Während der Ausführung

Mit steigender Spielfeld-Größe ruckelt die Animation der Spielfiguren immer mehr. Dies resultiert vermutlich aus der geringen CPU-Geschwindigkeit und RAM-Größe. Da diese aber ständig wachsen, dürfte das in naher Zukunft keine große Rolle mehr spielen. Das Gerät, auf dem die App entwickelt wurde, ein HTC Desire, ist knapp zwei Jahre alt. Auf diesem ist das Ruckeln bereits bei einer Spielfeld-Größe von 20 Feldern zu sehen. Ein Test auf einem Samsung Nexus S, das wesentlich kürzer auf dem Markt ist, ergab, dass die Animation bei 30 Feldern immer noch flüssig war.

Ab und zu flackert der Bildschirm, während das Spiel läuft. Die Ursache konnte auch nach langer Suche nicht behoben werden. Das Flackern war auf dem Samsung Nexus S nicht auf Anhieb zu sehen. Es könnte also an den Treibern des HTC Desire liegen.

6.2. Erweiterbarkeit

Die Erweiterbarkeit ist grundsätzlich problemlos möglich. Der Aufwand ist davon abhängig, was erweitert werden soll.

6.2.1. Spieler

Um z.B. einen weiteren Spieler hinzuzufügen, müssen lediglich die Konstanten `PLAYER_COUNT`, `PAWN_LIGHTS` und `PLAYER_TYPES` angepasst werden.

Durch die Änderung der Konstanten `PLAYER_TYPES` kann auch sehr einfach der Modus Computer vs. Computer oder Mensch vs. Mensch realisiert werden.

6.2.2. Zusätzliche Funktionalität

Soll jedoch weitere Funktionalität, wie z.B. würfeln durch das Schütteln des Gerätes, implementiert werden, so ist schon ein bisschen mehr Arbeit notwendig. Das Ereignis muss auf der Java-Ebene registriert und an die native Ebene weitergeleitet werden. Auf der nativen Ebene muss dieses Ereignis ausgewertet und entsprechende Aktionen ausgeführt werden.

6.3. Abweichungen vom Konzept

Es gab eine einzige Abweichung vom Konzept. Diese war jedoch so gravierend, dass der vorgenommene Zeitplan nicht mehr eingehalten werden konnte. Der Konzept-Entwurf

hat vorgesehen, dass jedes Feld des Spielfeldes durch je einen physischen Marker repräsentiert werden soll. Diese Idee musste jedoch nach einigen Tests mit verfügbaren AR-Frameworks verworfen werden, da das Vorhaben sowohl Software- als auch Hardware-seitig nicht realisierbar war. Stattdessen wurde die Idee mit einem einzigen großen Marker und dem Zeichnen der Felder mit dem Finger auf dem Touchscreen übernommen.

6.4. Einschätzung der Projektergebnisse

Im Großen und Ganzen bin ich mit dem Projektverlauf und -ergebnis sehr zufrieden. Es hat sehr viel Spaß gemacht, sich in neue und unbekannte Technologien einzuarbeiten und mit diesen ein Spiel zu entwickeln. Auf der anderen Seite hat die Einarbeitung auch sehr viel Zeit und Mühe gekostet. Zusätzlich zu den oben genannten Abweichungen vom Konzept war das der Grund, warum der Zeitplan nicht eingehalten werden konnte.

Während der Entwicklung sind einige Ideen entstanden, wie das Spiel erweitert und somit interessanter und spannender gemacht werden kann. Diese werden evtl. später im privaten Rahmen realisiert.

7. Quellennachweise

<http://developer.android.com/index.html> Android SDK und NDK, ausführliche Dokumentation und viele Tipps und Tricks

<http://android-developers.blogspot.com/> viele Tipps und Trick zur Android-Entwicklung

<https://ar.qualcomm.at/qdevnet/sdk> QCAR-Framework inklusive Dokumentation, vieler Beispiele und Forum

<https://github.com/HBehrens/obj2opengl> Konvertierung eines obj-Models in OpenGL-kompatible Arrays

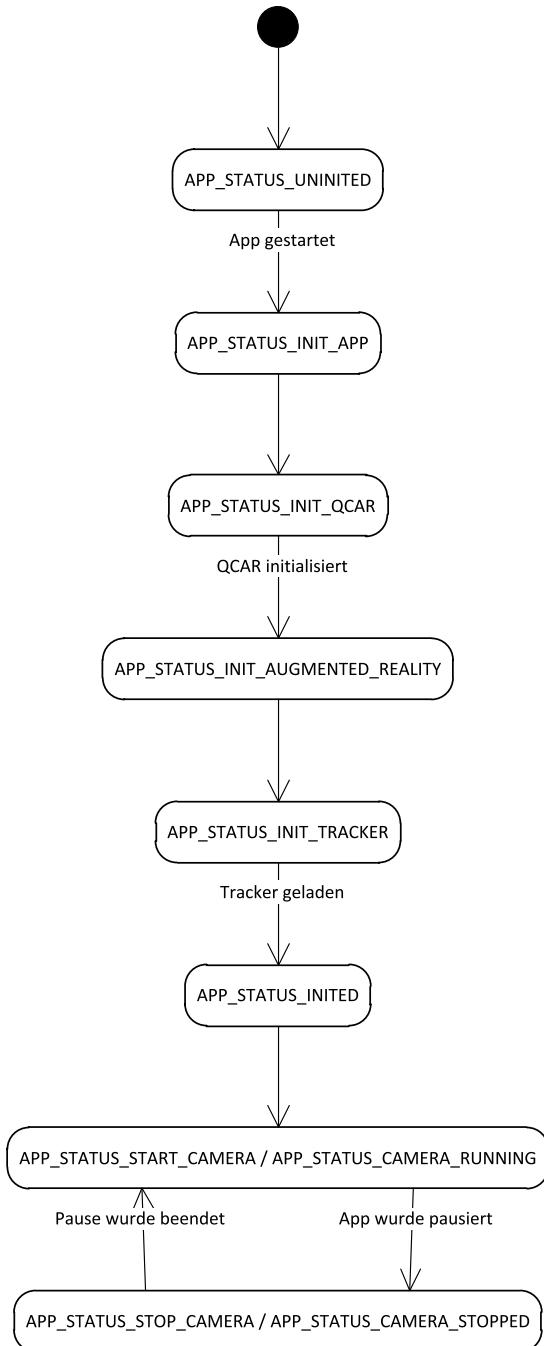


Abbildung 23: Invisiboga-Zustandsautomat



Abbildung 24: Spiel-Zustandsautomat



Abbildung 25: Spieler-Zustandsautomat

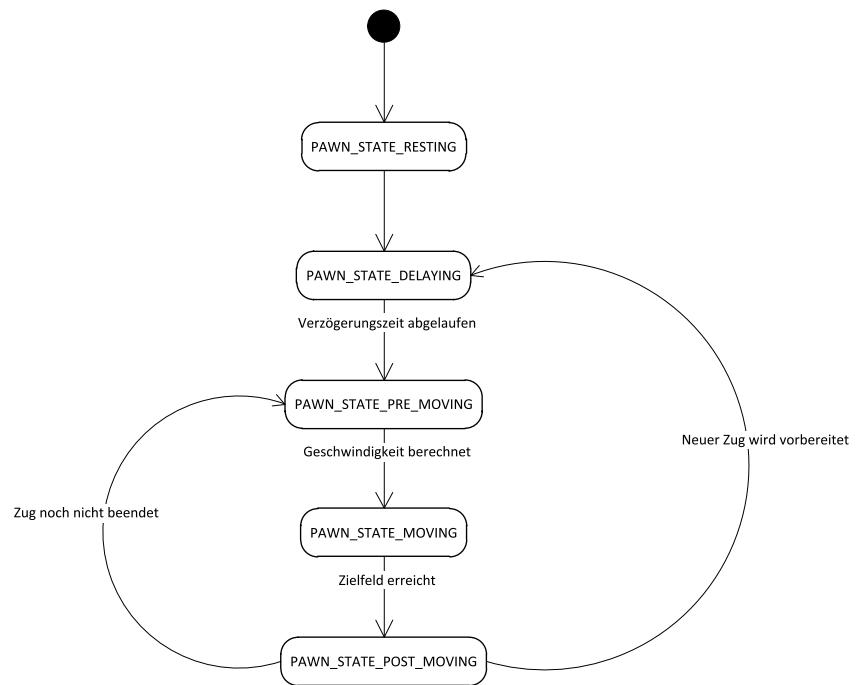


Abbildung 26: Spielfigur-Zustandsautomat

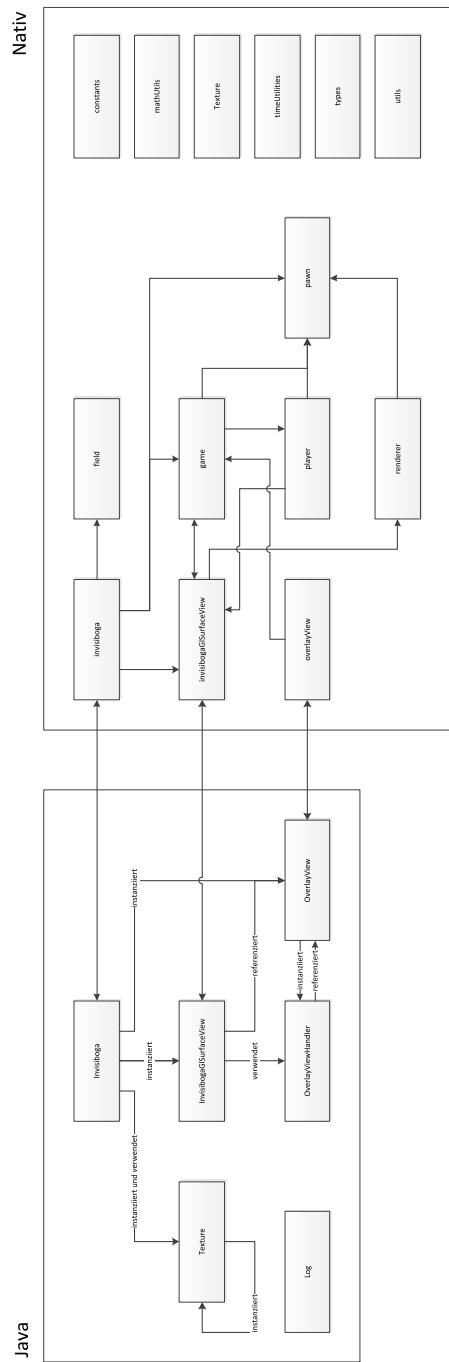


Abbildung 27: Architektur