Entwicklung einer Labyrinth-App mit automatischer Konstruktion eines Spielfeldes auf Basis von fotografierten Handzeichnungen

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Tanja Blücher

2148548

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Departement Medientechnik

Erstprüfer: Prof. Torsten Edeler

Zweitprüfer: Prof. Andreas Plaß

Hamburg, 25.02.2016

Inhaltsverzeichnis

1. Motivation

Abb. 1 Holzlabyrinth der Firma BRIO

Die Motivation, eine App zu konzipieren, entwickelte ich in dem Unterrichtsfach "Mobile Systeme". Hier wurden die verschiedenen Sensoren behandelt, die sich in einem Smartphone befinden. Auf diesen Informationen aufbauend, habe ich eine erste kleine App entwickelt. Diese benutzt den Beschleunigungssensor um eine Höhe zu messen. Bei der Entwicklung ist mir aufgefallen, dass man Smartphones sehr gut in seine Umgebung einbinden kann und somit die reale Welt mit der digitalen Welt verbinden kann. Im gleichen Semester wurde in dem Fach „Image Processing“ die Bildverarbeitung mit OpenCV behandelt und ich habe einen ersten Einblick in die Kantenerkennung erhalten.

Ein Spiel aus meiner Kindheit ist das Labyrinth der Firma BRIO (siehe Abb. 1). Das Besondere an dem Labyrinth ist, dass es zwei Knöpfe an der Seite gibt. Wenn diese gedreht werden, kippt das Spielfeld in die Drehrichtung. Dadurch rollt die Stahlkugel durch das Labyrinth. Ziel dabei ist, die Kugel bis zum Ziel zu manövrieren, ohne dass sie in eines der Löcher fällt.

Außerdem ist es möglich, unterschiedlich schwierige Labyrinthe einzusetzen und zu bespielen.

Die Labyrinth-App soll das Spielgefühl des Holzlabyrinths vermitteln, aber gleichzeitig auch die Möglichkeit bieten, seine eigenen Ideen umzusetzen und diese in Labyrinthe zu verwandeln. Dies geschieht, indem man mit einem Stift auf ein reales Blatt Papier ein Labyrinth aufzeichnet, es abfotografiert, auf sein Tablet überträgt und sofort mit dem Spiel beginnen kann.

1. Konzept

In diesem Kapitel wird das grundlegende Konzept der Labyrinth-App erklärt. Außerdem wird dargelegt, mit welcher Entwicklungsumgebung und mit welcher externen Bibliothek gearbeitet wurde. Zusätzlich werden die Voraussetzungen dargelegt, die benötigt werden, um die Labyrinth-App auf einem mobilen Android Gerät installieren und benutzen zu können. Am Ende des Kapitels wird dargestellt, mit welchem Gerät Debugging betrieben wird.

1.1 Das Konzept der App

Es wird eine Android-App entwickelt, die ein von Hand gezeichnetes Bild eines Labyrinths abfotografiert. Anhand von Kantenerkennung werden die Umrisse des Labyrinths erkannt und ein Spielfeld konstruiert. Durch setzen eines Start- und Zielpunktes wird das Labyrinth auf dem Tablet spielbar. Der Startpunkt dient als bewegbare Kugel. Diese kann durch Kippen des Tablets in alle Richtungen bewegt werden. Dadurch wird die Kugel durch das Spielfeld bis zum Zielpunkt manövriert. Wenn man an eine Kante des Labyrinths trifft, prallt die Kugel ab.

Die App soll leicht zu bedienen sein und sich auf das Wesentliche beschränken.

Die App ist für die Nutzung auf einem Tablet konzipiert.

1.2 Entwicklungsumgebung und Kantenerkennung

Als Entwicklungsumgebung dient Android Studio, das auf IntelliJ IDEA basiert. IntelliJ IDEA ist eine Integrierte Entwicklungsumgebung für die Programmiersprache Java (Vgl. Android Open Source Project: „Android Studio Overview“). Zur Kantenerkennung wird das Software Developement Kit OpenCv4Android als Bibliothek in das Projekt integriert. Die App wurde mit der Android Version 4.2.2 entwickelt.

1.3 Voraussetzungen

Um die App optimal nutzen zu können, wird ein Endgerät mit mindestens sieben Zoll benötigt. Das Endgerät muss mindestens unter der Android API 15 laufen.

1.4 Debugging

Die App ist für das Asus ME173X optimiert und dieses wurde als Debugging Gerät verwendet. Es läuft unter der Android Version 4.2.2 mit der API 17.

2. Bedienungshandbuch

Im nächsten Abschnitt wird die Funktionsweise für den Endbenutzer dargestellt. Es werden alle Funktionsweisen der Labyrinth-App erklärt und aufgezeigt wie die App benutzt wird.

Es wird vorausgesetzt, dass man die Labyrinth-App auf dem Mobilgerät installiert hat.

Beschriftungen und Hinweise sind in Deutsch verfasst.

2.1 Das Starten der App

Abb. 2 Logo der Labyrinth-App

Um die Labyrinth-App zu starten, drückt der Benutzer auf das Icon der App (siehe Abb. 2) auf seinem Mobilgerät. Die App wird dann gestartet und der Startbildschirm erscheint (siehe Abb. 3).

Auf dem Startbildschirm gibt es zwei Knöpfe, die man drücken kann. Der Knopf *„SPIELEN“* leitet den Benutzer zur Spielfelderzeugung weiter. Der Knopf *„ANLEITUNG“* leitet den Benutzer zu einer ausführlichen Anleitung weiter.

Außerdem befindet sich noch der Name der App in der oberen Mitte des Bildschirms. Darunter steht worum es in der App geht: *„Manövriere mit einer Kugel durch dein selbst entworfenes Labyrinth (siehe* Abb. 3*).“*

Abb. 3 Screenshot: Startbildschirm

2.2 Ein Spielfeld erstellen

Um das eigene Spielfeld zu erstellen, muss der Benutzer im Startbildschirm auf den Knopf *„SPIELEN“* drücken (siehe Abb. 4). Dieser leitet ihn dann zu einem Auswahlbereich weiter (siehe Abb. 5).

Es gibt zwei unterschiedliche Wege ein Spielfeld zu generieren. Es gibt die Möglichkeit ein Foto von einem Labyrinth zu machen, das man zuvor auf ein Blatt Papier gezeichnet hat.

Oder man wählt ein Bild aus, das sich auf dem Mobilgerät befindet.

Jedes Foto, das mit der App aufgenommen wird, wird auch auf dem Mobilgerät gespeichert. Somit kann der Benutzer ein schon einmal aufgenommenes Foto von seinem gezeichneten Labyrinth wiederverwenden.

Abb. 4 Screenshot: Auswahl des SPIELEN Knopfs

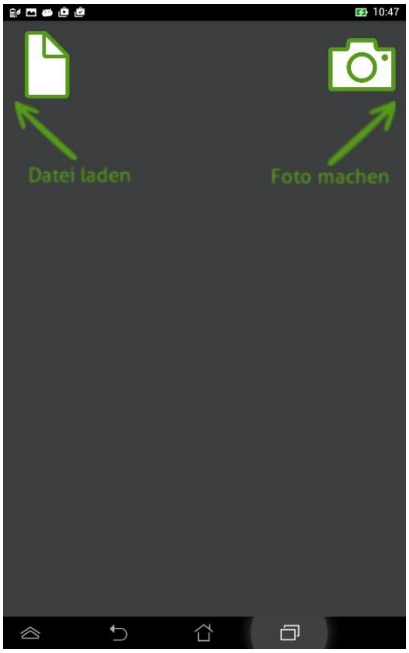

Abb. 5 Screenshot: Auswahlbildschirm

2.2.1 Ein Foto aufnehmen

Abb. 6 Kamerasymbol

Um ein Foto aufzunehmen, drückt der Benutzer im Auswahlbildschirm auf das Kamerasymbol (siehe Abb. 6). Es öffnet sich die Kamera und der Benutzer kann ein Foto von dem gezeichneten Labyrinth machen (siehe Abb. 7). In dem Foto werden die Kanten erkannt und diese auf das Foto gezeichnet. Das mit den Linien versehene Bild wird im Auswahlbildschirm dargestellt (siehe Abb. 8). Im oberen Teil des Auswahlbildschirms erscheint ein Haken und darunter der Schriftzug: *„Wenn du mit dem Bild zufrieden bist, klicke auf den grünen Haken (*siehe Abb. 8).“

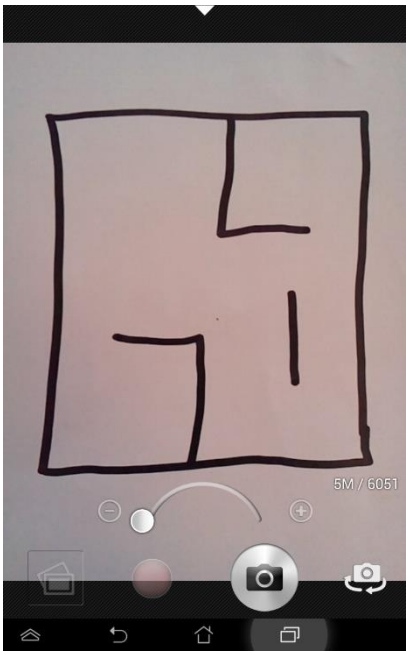

Abb. 7 Screenshot: Aufnahme eines Bildes

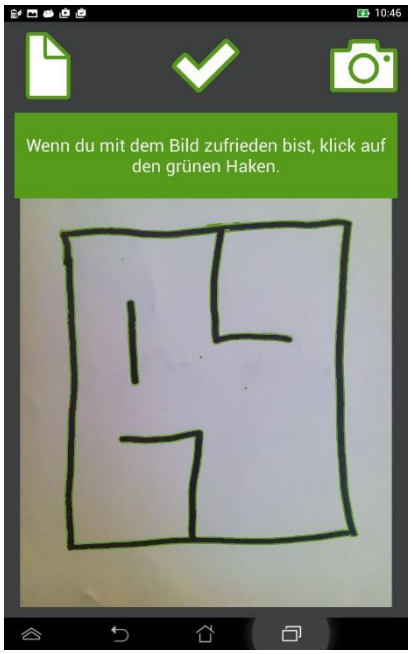

Abb. 8 Screenshot: Anzeige des aufgenommenen Bildes

2.2.2 Ein schon vorhandenes Foto auswählen

Abb. 9 Dateisymbol

Um ein schon vorhandenes Bild auszuwählen, klickt der Benutzer auf das linke obere Dateisymbol (siehe Abb. 9) im Auswahlbildschirm. Es öffnet sich das Fenster des Dateimanagers, in dem der Benutzer ein Bild auswählen kann (siehe Abb. 10).

Das ausgewählte Bild wird mit den gefundenen Linien versehen und in dem Auswahlbildschirm angezeigt (siehe Abb. 11).

Sobald das Bild angezeigt wird, erscheint ein Bestätigungshaken am oberen Rand des Bildschirms. Außerdem erscheint ein grüner Block mit dem Text: *"Wenn du mit dem Bild zufrieden bist, klicke auf den Haken (siehe* Abb. 11*)."*

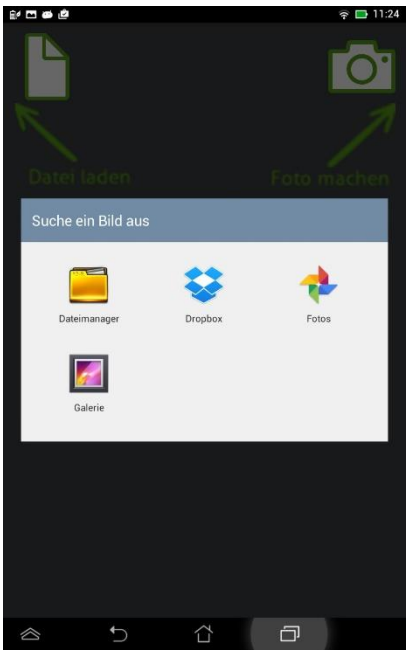

Abb. 10 Screenshot: Öffnen der Bildauswahl

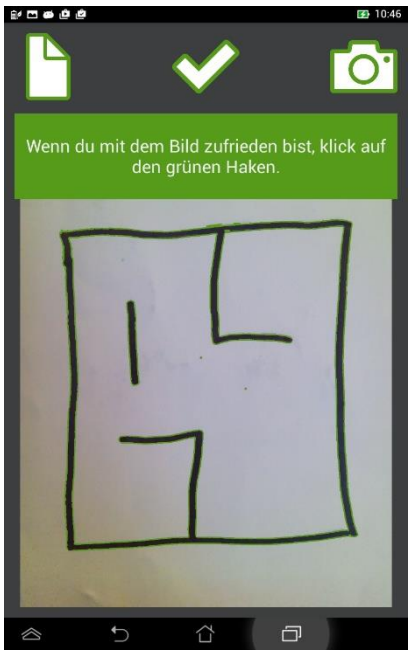

Abb. 11 Screenshot: Anzeige des ausgewählten Bildes

2.3 Das Labyrinth spielen

Abb. 12 Bestätigungshaken

Nach Drücken des Bestätigungshakens (siehe Abb. 12) wird das aus dem Foto konstruierte Spielfeld dargestellt. Es werden nur die Umrisse der Kanten in grün dargestellt (siehe Abb. 13).

Als nächstes legt der Benutzer einen Startpunkt und einen Zielpunkt fest. Dies geschieht, indem der Benutzer mit dem Finger auf die Bildschirmoberfläche des Mobilgeräts tippt. Beim ersten Tippen wird die Spielkugel platziert. Beim zweiten Tippen wird das Ziel festgelegt. Die Spielkugel wird in blau dargestellt. Das Ziel wird in grün dargestellt und ist größer als die Spielkugel (siehe Abb. 14).

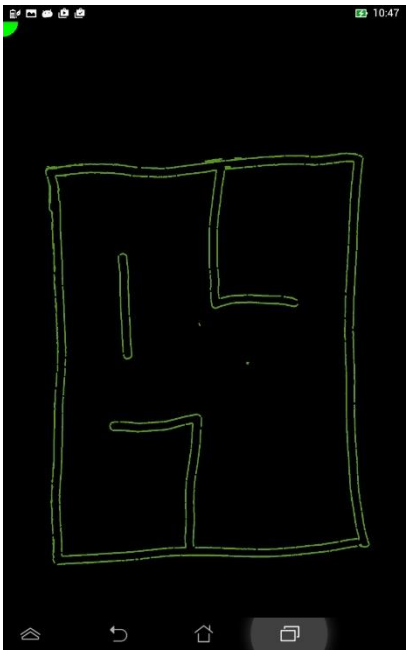

Abb. 13 Screenshot: Anzeige des konstruierten Spielfeldes

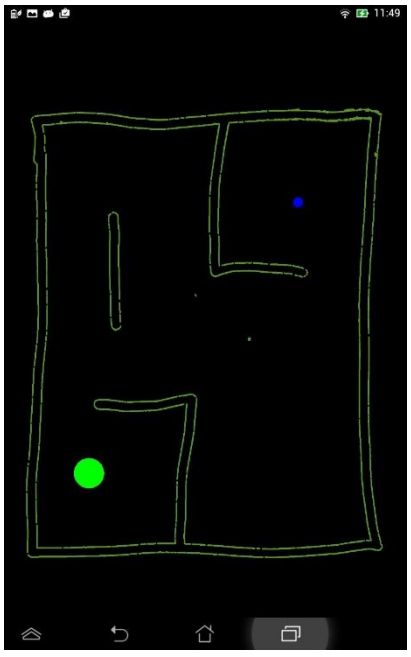

Abb. 14 Screenshot: Anzeige mit gesetzter Spielkugel und Ziel

Sobald das Ziel festgelegt ist, bewegt sich die Spielkugel. Sie lässt sich durch den Benutzer steuern, indem er das Mobilgerät in die gewünschte Richtung kippt. Je größer die Neigung des Geräts ist, desto schneller bewegt sich die Spielkugel. Durch ein kontrolliertes Bewegen der Spielkugel kann der Benutzer sie zum Ziel manövrieren. Sobald die Kugel eine Linie berührt, prallt sie von der Linie zurück. Je schneller die Kugel gegen die Line prallt, desto weiter prallt sie zurück. Sollte die Spielkugel durch eine Lücke in den Linien rollen, wird sie am Bildschirmrand aufgehalten.

Wenn der Benutzer mit der Spielkugel das Ziel erreicht, wird der Gewinnerbildschirm angezeigt (siehe Abb. 15). Nach einer kurzen Zeit gelangt der Benutzer wieder zurück auf den Auswahlbildschirm (siehe Abb. 11). Er kann das gleiche Labyrinth noch einmal spielen oder ein neues Labyrinth erstellen.

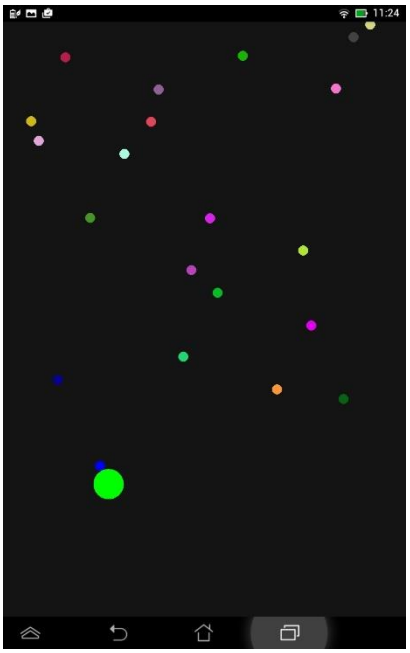

Abb. 15 Screenshot: Gewinnerbildschirm

2.4 Die Auswahl der Anleitung

Die Anleitung lässt sich im Startbildschirm aufrufen (siehe Abb. 3). Indem der Benutzer auf den Knopf *„ANLEITUNG“* drückt, wird er zur Spielanleitung weitergeleitet (siehe Abb. 16). Alle wichtigen Informationen zur Bedienung der Labyrinth-App sind dort aufgeführt. Der Benutzer kann außerdem nachlesen, wie er das bestmögliche Ergebnis für ein Labyrinth erreicht. Mit dem Knopf *„Zurück“* gelangt der Benutzer wieder zurück zum Startbildschirm.

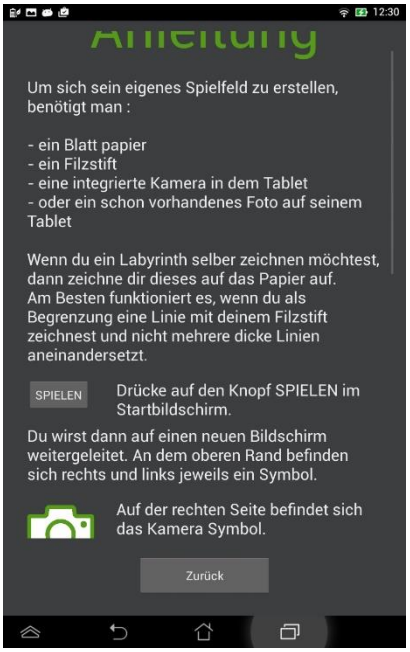

Abb. 16 Screenshot: Anzeige der Anleitung

3. Softwareentwicklung

In diesem Kapitel wird der Quelltext dargestellt. Dabei werden die Logik, sowie der Code erklärt und aufgezeigt. Der Fokus liegt dabei auf der Spielfeldgenerierung und der Spielballbewegung.

3.1 Projektstruktur

Das Projekt besteht aus unterschiedlichen Komponenten.

Eine Activity Klasse stellt einen Bildschirm bereit. Auf diesem kann ein User Interface dargestellt werden (Vgl. Android Open Source Project: „Activity“):

- class MainActivity extends Activity
- class CreateLevel extends Activity
- class PlayLevel extends Activity
- class Instructions extends Activity

Das Layout der Activity Klasse wird in einer .xml –Datei festgelegt:

- activity_main.xml
- activity_ create_Level.xml
- activity_play_level.xml
- activity_instuctions.xml

Ein SurfaceView bietet eine Zeichenfläche (Vgl. Android Open Source Project: „ public Class SurfaceView“):

- class DrawingView extends SufaceView

Eine Application Klasse ist eine Basis Klasse, die einen globalen Zustand besitzt. Sie kann Informationen zwischen Activities übergeben (Vgl. Android Open Source Project: „public Class Application“):

- class BaseApp extends Application

Die OpenCV Bibliothek ist in das Projekt integriert.

Außerdem sind dem Projekt Bildressourcen hinzugefügt.

3.2 UML Klassendiagramm der Java Klassen

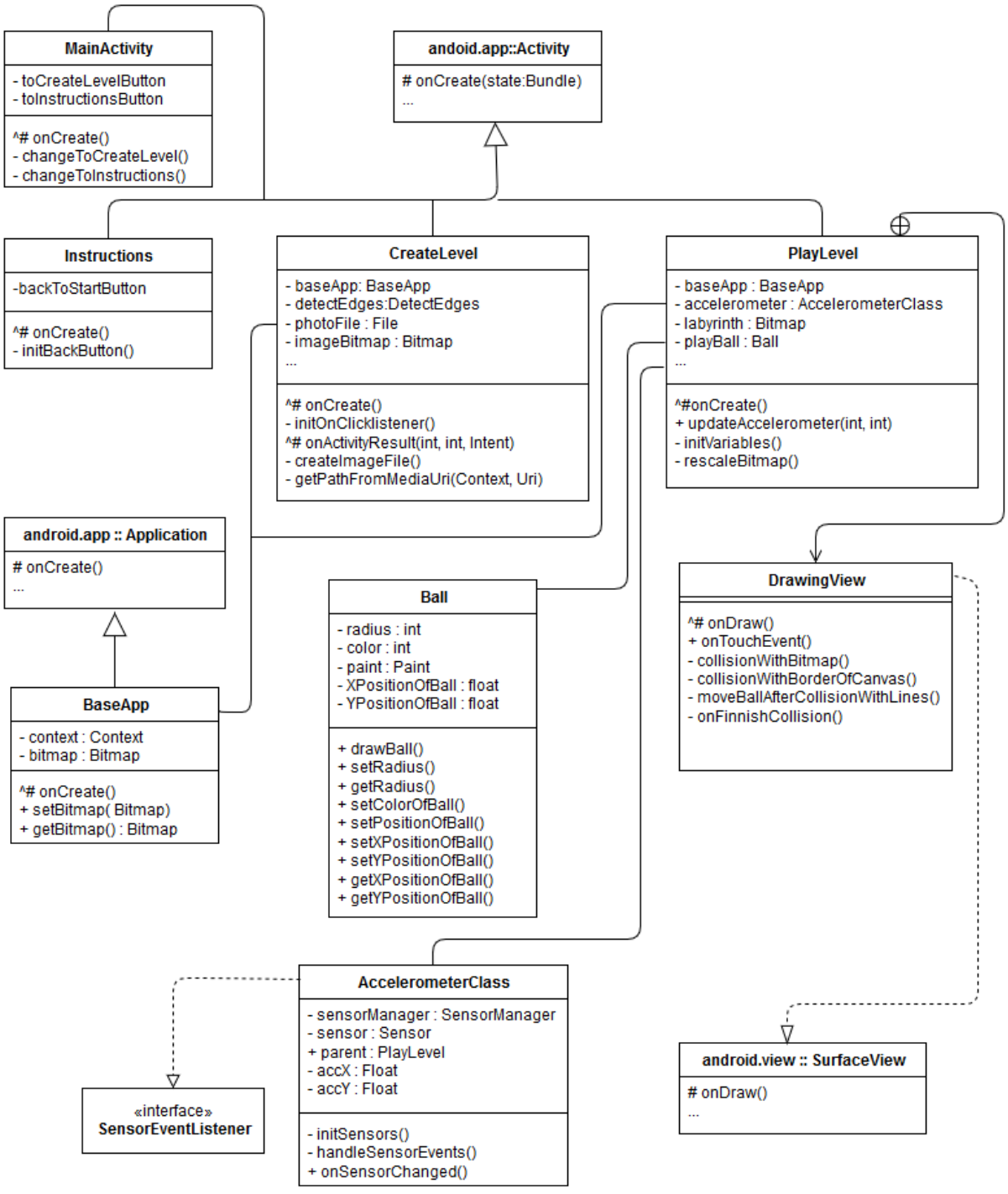

Abb. 17 UML Klassendiagramm

3.3 AndroidManifest.xml

```
1  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
2  <uses-feature android:name="android.hardware.camera" />
3
4  <application
5      android:name=".BaseApp"
6      android:allowBackup="true"
7      android:icon="@mipmap/icon_labyrinth"
8      android:label="@string/app_name"
9      android:theme="@style/AppTheme" >
10 </application>
```

Listing 1 AndoirdManifest.xml: Genehmigungen

Das Manifest beinhaltet wesentliche Informationen über die App. Diese werden an das Android System weitergegeben.

Um ein Foto zu erzeugen, wird die Camera API von Android benutzt. Im *AndroidManifext.xml* wird angegeben, dass zur Benutzung der App die Kamera benötigt wird (Zeile 2, Listing 1). Die Benutzung der Camera API wird später durch einen Intent geregelt. Daher ist es nicht nötig die Befugnis zur Benutzung der Kamera einzuholen, sondern nur die Hardwarevorraussetzungen anzugeben.

Außerdem sollen erzeugte Fotos im Speicher des Mobilgeräts gespeichert werden und wieder geladen werden. Dafür muss die Genehmigung zum Lesen und Schreiben des Speichers eingeholt werden (Zeile 1, Listing 1).

Die BaseApp.java Klasse speichert Informationen und gibt diese an andere Activities weiter. Damit das möglich ist, muss die Klasse *BaseApp.java* als Application gesetzt werden.

```
1 <activity
2     android:name=".PlayLevel"
3     android:screenOrientation="portrait" >
4 </activity>
```

Listing 2 AndroidManifest.xml: Bildschirmorientierung einer Activity festlegen

Die App ist im Portraitmodus spielbar. Damit sich das nicht ändert, ist in dem Manifest den Activities die Eigenschaft *screenOrientation* zugeteilt (Zeile 3, Listing 2).

In der *AndroidManifest.xml* Datei wird außerdem das Icon der App festgelegt. Es handelt sich dabei um eine .png Datei, die der Projektstruktur in den Ressourcen hinzugefügt wurde (Zeile 7, Listing 1).

3.4 Ein Foto aufnehmen und speichern

```
1  @Override
2  public void onClick(View v) {
3
4      Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
5
6      if (takePictureIntent.resolveActivity(getPackageManager()) != null) {
7
8          photoFile = null;
9           try {
10
11         photoFile = createImageFile();
12
13         } catch (IOException ex) {
14         // File konnte nicht erstellt werden
15
16         }
17
18         if (photoFile != null) {
19             takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT,
20                 Uri.fromFile(photoFile));
21             startActivityForResult(takePictureIntent, REQUEST_TAKE_PHOTO);
22         }
23     }
24
25 }
```

Listing 3 CreateLevel.java: Erzeugen und Ausführung des Kameraintents

Um ein Foto aufzunehmen, wird in der *CreateLevel.java* Klasse ein Intent Objekt erzeugt, das die Kamera Anwendung des Geräts startet (Zeile 4, Listing 3). Bevor der Intent gestartet wird, muss ein gültiger Dateipfad erzeugt werden.

Das Objekt *photoFile* ist ein File Objekt. In der Methode *createImageFile()* wird das File Objekt mit gültigem Namen und einem Typ überschrieben. Außerdem wird dem Objekt als Speicherverzeichnis das öffentlich zugängliche interne Speicherverzeichnis der Bilder zugewiesen (Zeile 11, Listing 3). Die Uri des File Objekts wird dem Intent übergeben, und das gemachte Foto wird an diese Stelle gespeichert (Zeile 19f, Listing 3).

Der Intent, der die Kamera Anwendung startet, wird mit der Methode *startActivityForResult()* gestartet. Der Methode werden der Intent und ein Erkennungsmerkmal übergeben (Zeile 21, Listing 3). Es wird dabei ein Resultat erwartet. Dieses wird als ein weiteres Intent Objekt geschickt. Er wird in der Callback-Methode *onActivityResult()* empfangen (siehe Kapitel 3.7).

3.5 Ein vorhandenes Bild benutzen

```
1  @Override
2 public void onClick(View v) {
3     Intent loadPictureIntent = new Intent();
4     loadPictureIntent.setType("image/*");
5     loadPictureIntent.setAction(Intent.ACTION_GET_CONTENT);
6     startActivityForResult(Intent.createChooser(loadPictureIntent, "Suche ein Bild
aus"),REQUEST_LOAD_PICTURE);
7 }
```

Listing 4 CreateLevel.java: Erzeugung des Bild laden Intents und Ausführung

Um ein Bild von dem Speicher des Mobilgeräts zu laden, wird ein neues Intent Objekt erzeugt (Zeile 2, Listing 4). Dem Intent Objekt wird die Aktion zugewiesen, dass nur bestimmte Daten ausgewählt und ausgegeben werden können (Zeile 3, Listing 4). Der Typ wird als Bild festgelegt (Zeile 4, Listing 4).

Der Methode *startActivityForResult()* wird ein Intent Objekt übergeben, das einen Chooser öffnet, dessen Auswahl an den vorherig erzeugten *loadPictureIntent* übergeben wird. Außerdem wird der Methode noch ein Erkennungsmerkmal übergeben, um den Intent bei der Resultatausgabe zu identifizieren (Zeile 6, Listing 4).

```
1 @Override
2 protected void onActivityResult(int requestCode, int resultCode, Intent data) {
3     super.onActivityResult(requestCode, resultCode, data);
4     if (resultCode == RESULT_OK) {
5
6         final Handler handler = new Handler();
7         detectEdges = new DetectEdges();
8
9         if (requestCode == REQUEST_TAKE_PHOTO) {
10
11
12         }
13         if (requestCode == REQUEST_LOAD_PICTURE) {
14                 Uri selectedPictureURI = data.getData();
15             String path = getPathFromMediaUri(this, selectedPictureURI);
16             photoFile = new File(path);
17         }
18
19
20     } else if (resultCode == RESULT_CANCELED) {
21
22     }
23
24 }
```

Listing 5 CreateLevel.java: onActivityResult()-Methode, ein Bild laden

Das geladene Bild befindet sich in der *onActivityResult()* Methode in dem Intent Objekt *data*. Der Methode onActivityResult() wurde ein Merkmal übergeben, dass das Resultat in Ordnung ist. Außerdem das vorherige Erkennungsmerkmal und die gewünschten Daten in einem Intent Objekt (Zeile 2, Listing 5).

Die Uri des Intent Objekts wird gespeichert (Zeile 14, Listing 5). Da das Bild zur Weiterbearbeitung als Objekt mit dem Datentyp File benötigt wird, wird die Uri in einen absoluten Pfad umgewandelt (Zeile 15, Listing 5) und ein neues File Objekt erzeugt.

3.6 Kantenerkennung und Linienerkennung

```
1  private Mat image;
2  private Mat lines;
3  private Mat imageWithLines;
4  private Mat imageOnlyLines;
5
6  public void drawLinesOnImage(File picture, int threshold, int minLineSize, int
lineGap) {
7      String filePath = picture.getAbsolutePath().toString();
8      lines = new Mat();
9
10
11
12     image = Imgcodecs.imread(filePath);
13
14     imageOnlyLines = new Mat(image.size(),image.type());
15     imageWithLines = image.clone();
16     Imgproc.cvtColor(image, image, Imgproc.COLOR_RGB2GRAY);
17
18     Size s = new Size(3, 3);
19     Imgproc.blur(image, image, s);
20     Imgproc.Canny(image,image, 40, 100);
21
22     Imgproc.HoughLinesP(image, lines, 1, Math.PI / 180, threshold, minLineSize,
lineGap);
23
24     drawLinesOnImage();
25
26     convertIntoBitmap();
27     convertIntoBitmapOnlyLines();
28 }
```

Listing 6 DetectEdges.java: Kanten und Linienerkennung

Zur Kantenerkennung wird die OpenCV Bibliothek benutzt. Um in einem Bild die Kanten zu erkennen, wird ein Objekt der Klasse *DetectEdges.java* erzeugt.

Der Methode *drawLinesOnImage()* wird ein File Objekt und drei Integer Werte übergeben (Zeile 6, Listing 6).

Das übergebene File Objekt wird in einem Mat Objekt gespeichert. Hierzu wird der absolute Pfad des Objekts ausgelesen (Zeile 7, Listing 6). Mit der Methode *imread()* wird das Bild eingelesen (Zeile 12, Listing 6).

Da die Canny-Kantenerkennung nur mit Graustufenbildern arbeiten kann, wird das Bild in ein Graustufenbild umgewandelt (Zeile 16, Listing 6). Um Störungen und Rauschen zu vermindern wird ein Blurfilter auf das Graustufenbild angewendet (Zeile 19, Listing 6).

Das Ergebnis wird in Abb. 18 dargestellt.

Danach wird mit der Canny-Kantenerkennung das Bild in ein Kantenbild umgewandelt.

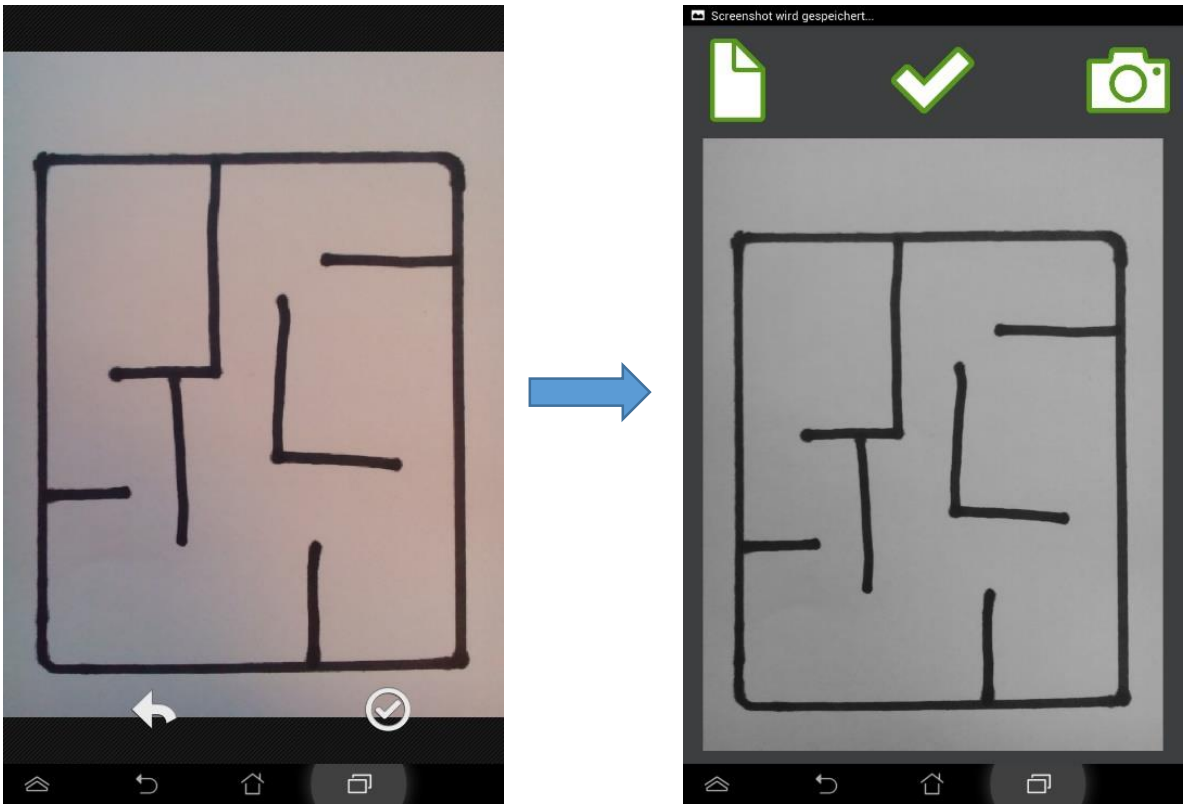

Abb. 18 Screenshot: a) Originalbild b)Graustufenbild mit angewendetem Blurfilter

Der Canny-Kantenerkennungs-Algorithmus besteht aus vier Schritten.

1. Das Eingangsbild wird mit dem Gaus-Filter geglättet.
2. Berechnung der Länge und Richtung des Gradienten. Der Gradient beschreibt den Intensitätsverlauf des Bildes. Die Länge des Gradienten beschreibt die Stärke der Kanten und er zeigt in die Richtung der stärksten Intensitätsänderung.
3. Anwendung der Non-Maximum Supression um Pixel, die nicht zu einer Kante gehören, zu entfernen und die Kante zu verdünnen.

4. Definierung eines „High“-Schwellenwerts und eines „Low“-Schwellenwerts. Wenn der Gradient an einer Pixelposition über dem „High“-Schwellenwert liegt, ist er ein Kantenpixel. Wenn ein Gradient an einer Pixelposition zwischen dem „High“-Schwellenwert und dem „Low“-Schwellenwert liegt, ist er nur ein Kantenpixel, wenn der Pixel mit einem anderen Kantenpixel verbunden ist. Wenn der Gradient an einer Pixelposition unter dem „Low“-Schwellenwert liegt, ist er kein Kantenpixel.

(Vgl. Rafael C. Gonzalez / Richard E. Woods, S.741 ff).

In der DetectEdges.java Klasse, wird der Imgproc.Canny()-Methode das Graustufenbild als Eingangsbild übergeben und das Objekt, in das das Kantenbild gespeichert werden soll. Außerdem werden der „Low“-und „High“-Schwellenwerte übergeben (Zeile 20, Listing 6).

Das Ergebnis der Ausführung der Canny-Kantenerkennung wird in Abb. 19 dargestellt.

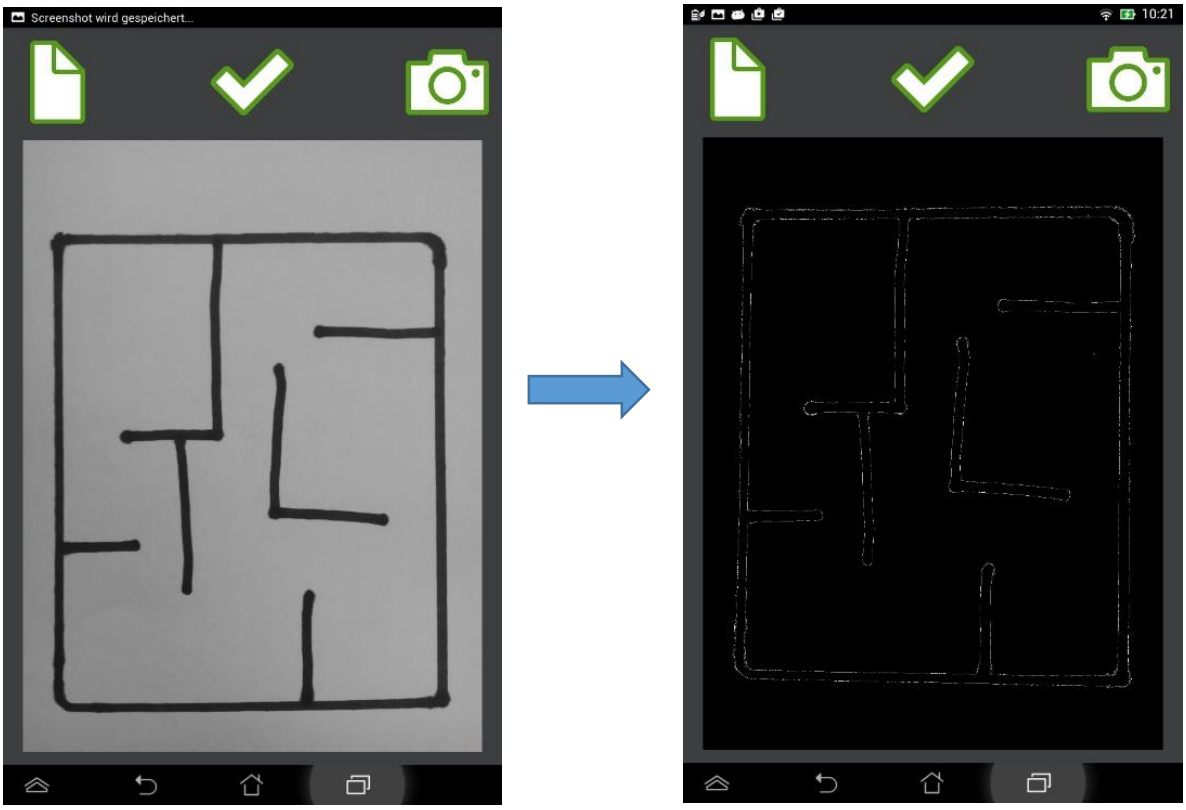

Abb. 19 Screenshot: a)Graustufenbild b) Kantenbild

Um die einzelnen Kantenpixel zu einer Linie zusammenzufassen, wird die Hough-Transformation angewendet.

Im kartesischen Koordinatensystem wird eine Gerade, die durch einen bestimmten Punkt geht mit der Geradengleichung $y_i = ax_i + b$ beschrieben. Dabei ist a die Steigung der Geraden und b der y-Achsenabschnitt. Unendlich viele Geraden gehen durch den Punkt (x_i, y_i). Sie erfüllen alle die Geradengleichung, jedoch besitzen sie alle unterschiedliche Werte a und b.

Für einen bestimmten Punkt (x_i, y_i) gilt: $b = (-x_i)a + y_i$. In einem ab-Raum gibt es für den Punkt (x_i, y_i) genau eine Gerade. Von allen Punkten, die sich im xy-Raum auf einer Geraden befinden, schneiden sich die Geraden im ab -Raum in einem Punkt. Der Schnittpunkt bestimmt die Parameter $a\ und\ b$ der gesuchten Geraden im xy-Raum.

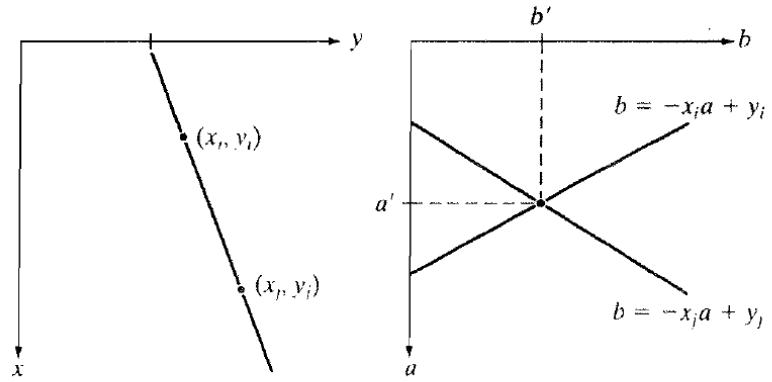

Abb. 20 a) Punkte auf einer Geraden im xy-Raum b) Geraden der Punkte im ab-Raum

(Quelle: Rafael C. Gonzalez / Richard E.Woods: „Digital Image Processing (Third Edition)“, S.755)

Da man im xy -Raum für vertikale Linien einen Wertebereich von $-\infty$ bis $+\infty$ bräuchte, wird die Geradengleichung durch $\rho = x\cos\theta + y\sin\theta$ dargestellt. ρ ist der Abstand zum Koordinatenursprung und θ ist der Winkel zur x-Achse (siehe Abb. 21).

Aus dem ab -Raum wird der $\rho\theta$-Raum, der Parameterraum. Jeder Punkt im xy–Raum wird durch eine sinusförmige Funktion im $\rho\theta$-Raum repräsentiert (siehe Abb. 21).

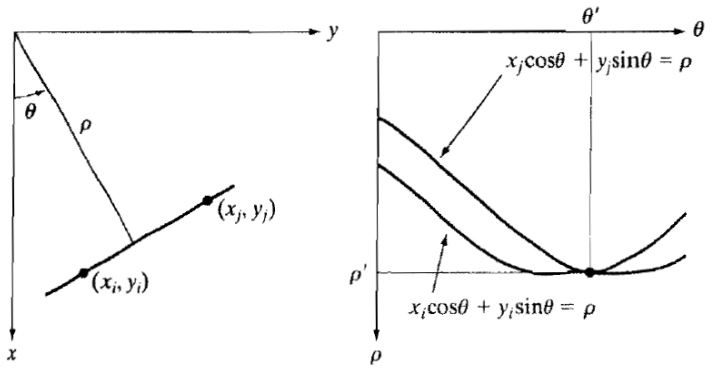

Abb. 21 a) Punkte auf einer Geraden im xy-Raum b) Darstellung der Punkte im Parameterraum

(Quelle: Rafael C. Gonzalez / Richard E.Woods: „Digital Image Processing (Third Edition)“, S.756)

Die Hough-Transformation unterteilt den $\rho\theta$ -Raum in Akkumulator-Zellen.

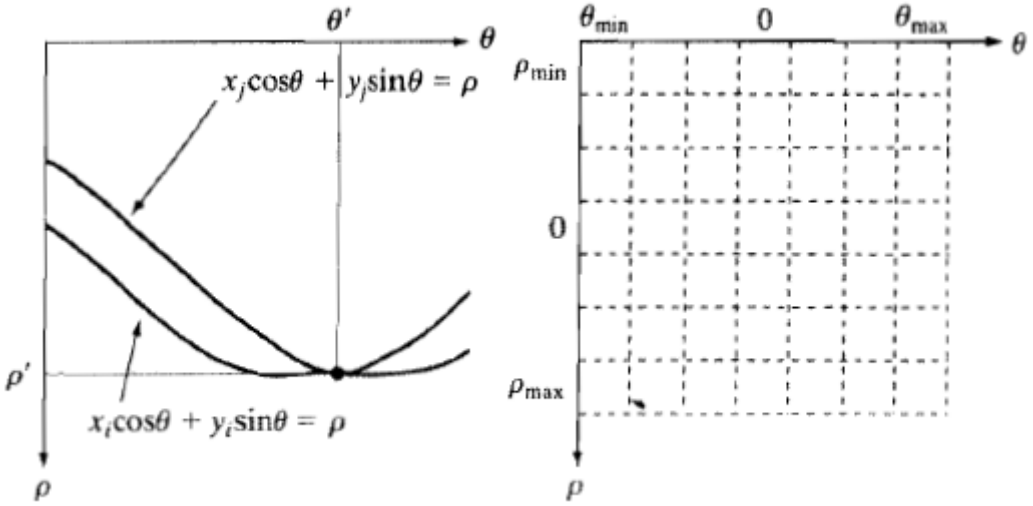

Abb. 22 a) Parameterraum b) Akkumulator

(Quelle: Rafael C. Gonzalez / Richard E.Woods: „Digital Image Processing (Third Edition)“, S.756)

Am Anfang sind wird in jede Zelle Null eingetragen. Wenn eine sinusförmige Funktion eine Zelle durchläuft, wird ihr Wert um 1 erhöht. An den Stellen, die den Parametern der Gerade im xy -Raum entsprechen, entstehen hohe Werte.

Durch die Angabe eines Minimum-Schwellenwerts, der die minimale Anzahl an überschneidenden Funktionen angibt, kann eine Linie definiert werden.

Die Hough-Transformation gibt einem die gesuchten $\rho - und\ \theta$ -Parameter für eine gefundene Line aus.

(Vgl. Rafael C. Gonzalez / Richard E. Woods , S. 755 ff)

Bei der probabilistischen Hough-Transformation werden nicht alle Punkte als mögliche Kandidaten betrachtet, sondern eine zufällige Teilmenge der Punkte. Dafür wird der Minimum- Schwellenwert von 250 auf 100 gesenkt. Außerdem kommen zwei neue Argumente dazu. Die minimale Länge einer Line und die maximale Lücke zwischen Linien, in der sie als eine Linie behandelt wird. Die Imgproc.HougLinesP()- Methode gibt nicht die $\rho - und\ \theta$ -Paramter aus, sondern x-und y-Koordinaten. Diese stellen den Start- und Endpunkte einer Line dar (Vgl. Opencv dev team: „Hough Line Transform“).

Der Imgproc.HoughLinesP()-Methode wird in der Klasse DetectEdges.java das Kantenbild übergeben. Außerdem ein Mat Objekt, das die x- und y-Koordinaten der Start- und Endpunkte speichert. Als nächstes wird das Argument übergeben, das festlegt, welche Auflösung der Parameter ρ in Pixeln hat. Das nächste Argument gibt die Auflösung der Parameters θ an. Als nächstes Argument wird der Parameter zur minimalen Überschneidungsanzahl übergeben. Die beiden letzten Argumente geben die Parameter der minimalen Länge einer Line und die maximale Lücke zwischen Linien, in der sie als eine Linie behandelt werden, an (Zeile 22, Listing 6). In Abb. 23 werden die gefundenen x-und y-Koordinaten zu Punkten zusammengefasst und dargestellt. Die Startpunkte der Linien sind in grün, die Endpunkte in rot markiert.

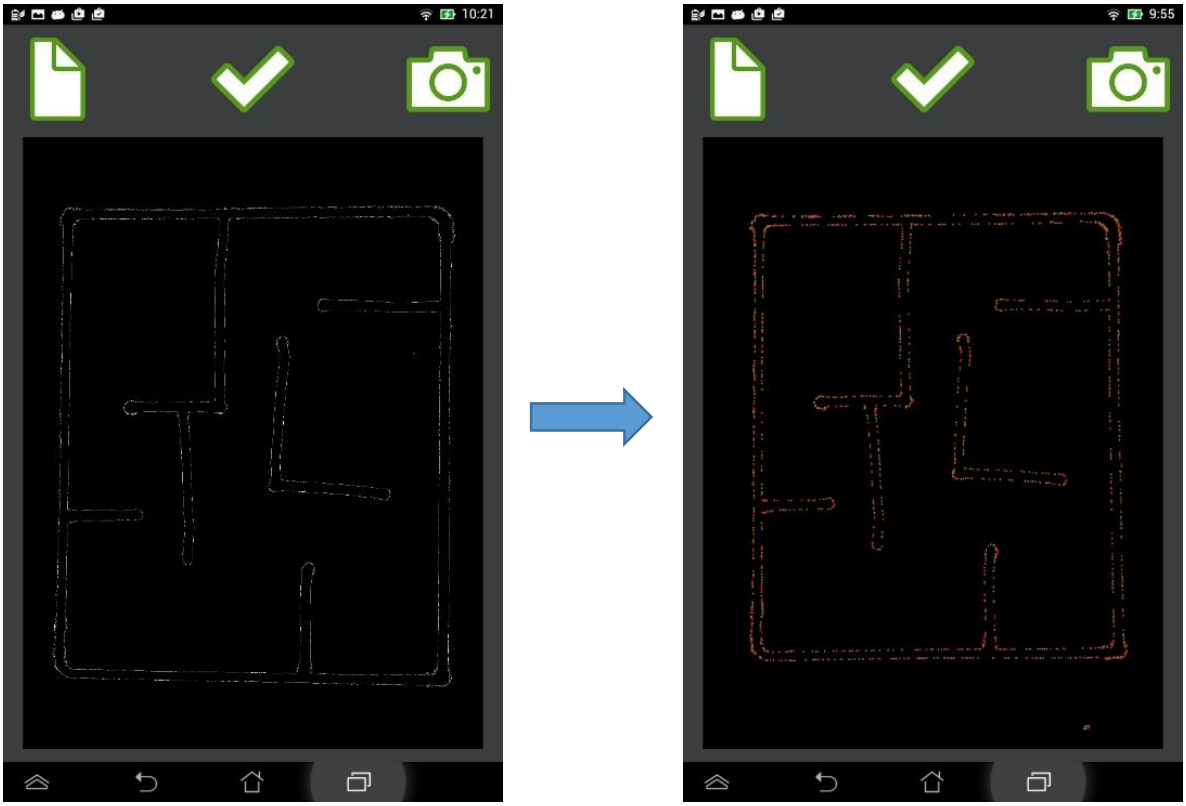

Abb. 23 Screenshot: a) Kantenbild b) gefundenen Start-und Endpunkten der Hough-Transformation

Um die gefundenen Linien darzustellen, werden die gefundenen Punkte der Hough-Transformation an die Imgproc.line()-Methode übergeben (Zeile 15, Listing 7). Insgesamt werden der Imgproc.line()-Methode das Mat Objekt übergeben, in das gezeichnet werden soll. Außerdem der Startpunkt und Endpunkt der Linie, sowie ein Farbwert, in dem die Linie gezeichnet werden soll. Der letzte übergebene Wert gibt die Stärke der gezeichneten Linien an. Das Ergebnis ist in Abb. 24 dargestellt. Die Methode zeichnet dann eine Linie vom Startpunkt bis zum Endpunkt.

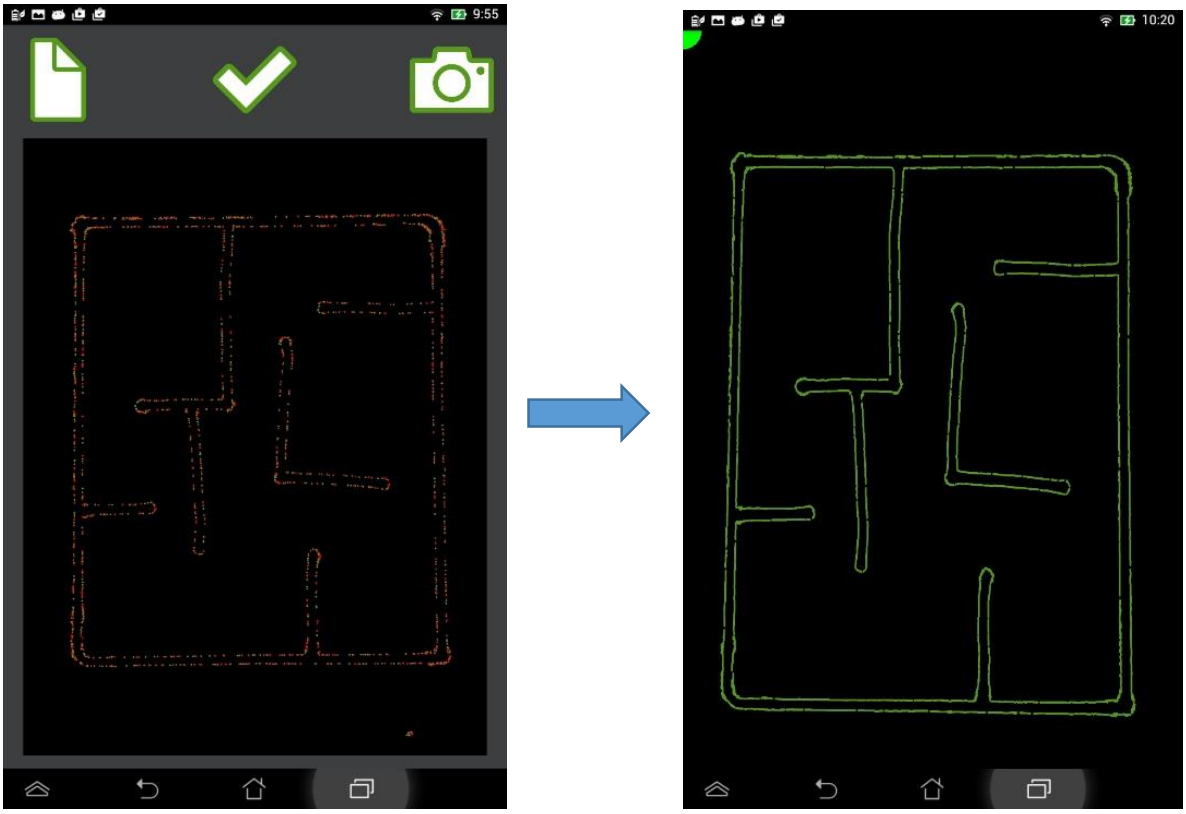

Abb. 24 Screenshot: a) Start- und Endpunkte b) Linienbild

```
1  private void drawLinesOnImage(){
2     for (int x = 0; x < lines.rows(); x++){
3
4         double[] vec = lines.get(x,0);
5         x1 = vec[0];
6         y1 = vec[1];
7         x2 = vec[2];
8         y2 = vec[3];
9
10        startOfLine = new Point(x1, y1);
11        endOfLine = new Point(x2, y2);
12
13
14
15        Imgproc.line(imageWithLines,startOfLine,endOfLine,new Scalar(87,155,25), 5);
16        Imgproc.line(imageOnlyLines,startOfLine,endOfLine,new Scalar(87,155,25), 5);
17    }
18 }
```

Listing 7 DetectEdges.java: Linien auf ein Mat Objekt zeichnen

Um die Linien auf das Originalbild zu zeichnen, wird ein Duplikat davon erstellt und in dem Mat Objekt *imageWithLines* abgelegt (Zeile 15, Listing 6).

Es werden die von der Hough-Transformation gefundenen Koordinaten ausgelesen und zu Punkten zusammengefasst (Zeile 2ff, Listing 6). Diese werden benutzt um die Linien auf das Mat Objekt *imageWithLines* zu zeichnen (Zeile 15, Listing 7).

Da das Labyrinth später aber nur aus den gefundenen Linien besteht, wird ein weiteres Mat Objekt erzeugt. Dieses hat die gleiche Größe wie das *image* Objekt und ist auch der gleiche Typ (Zeile 14, Listing 6).

In dem Mat Objekt *imageOnlyLines* werden die gefundenen Linien auch eingezeichnet (Zeile 16, Listing 7).

```
1 private void convertIntoBitmap(){
2         pBitmap = Bitmap.createBitmap(imageWithLines.cols(), imageWithLines.rows(),
Bitmap.Config.ARGB_8888);
3         Utils.matToBitmap(imageWithLines, pBitmap);
4 }
5 public Bitmap getImageBitmap(){ return pBitmap;  }
```

Listing 8 DetectEdges.java: Bitmap erzeugen

Beide Bilder mit Linien werden in der *convertIntoBitmap()-Methode* in eine Bitmap umgewandelt (Zeile 3, Listing 8). Über die Getter-Methoden kann man auf diese zugreifen (Zeile 5, Listing 8).

3.7 Bildverarbeitung und Anzeige

```
1     private ImageView tickImgView;
2     private Bitmap imageBitmap;
3
4     @Override
5      protected void onActivityResult(int requestCode, int resultCode, Intent data) {
6          super.onActivityResult(requestCode, resultCode, data);
7          if (resultCode == RESULT_OK) {
8
9              final Handler handler = new Handler();
10             detectEdges = new DetectEdges();
11
12             if (requestCode == REQUEST_TAKE_PHOTO) {
13
14                 new Thread(new Runnable() {
15                     @Override
16                     public void run() {
17                         detectEdges.drawLinesOnImage(photoFile, 10, 5, 5);
18                         imageBitmap = detectEdges.getImageBitmap();
19
20                         handler.post(new Runnable() {
21                             @Override
22                             public void run() {
23                                 showPicture.setImageBitmap(imageBitmap);
24
25                             }
26                         });
27                     }
28                 }).start();
29                 showTick = true;
30             }
31
32
33         } else if (resultCode == RESULT_CANCELED) {
34             showTick = false;
35         }
36
37     }
```

Listing 9 CreateLevel.java: Kantenerkennung anwenden

Die Kantenerkennung wird von der *CreateLevel.java* Klasse aus auf ein File Objekt angewendet. Das vorher mit der Kamera gemachte Foto wird weiterverarbeitet. Beim Starten des Kameraintents wurde ein Erkennungsattribut mitgegeben (siehe Kapitel 3.4). Dieses wird an die Callback-Methode *onActivityResult()* zurückgegeben (Zeile 5, Listing 9). Es wird überprüft, ob der Kameraintent ausgeführt wurde (Zeile 12, Listing 9) und ob dieses funktioniert hat (Zeile 7, Listing 9).

Es wird ein neues Objekt der Klasse *DetectEdges.java* erzeugt (Zeile 10, Listing 9). Auf dieses wird die Methode *drawLinesOnImage()* angewendet und das File Objekt übergeben (Zeile 17, Listing 9). Außerdem werden noch die drei Attribute für den

Überschneidungsschwellenwert, die minimale Linienlänge und die maximale Linienlücke übergeben.

Das bearbeitete Bild wird als Bitmap gespeichert. Hierzu wird die Bitmap Rückgabe - Methode des DetectEdges Objekt aufgerufen (Zeile 18, Listing 9). Um die Bitmap anzuzeigen, wird ein neues Handler Objekt erzeugt (Zeile 9, Listing 9). Dieses setzt die Bitmap in ein ImageView Objekt, das dieses auf dem Bildschirm anzeigt (Zeile 20ff, Listing 9).

```
1 BaseApp baseApp = (BaseApp) getApplicationContext();
2 baseApp.setBitmap(detectEdges.getOnlyLinesBitmap());
```

Listing 10 CreateLevel.java: BaseApp Objekt erzeugen und Parameter übergeben

Dieses angezeigte Bild dient als Vorschau. Das Originalfoto wird zusätzlich mit den generierten Linien angezeigt. Damit das Spielfeld im nächsten Schritt nur aus den gefundenen Linien besteht, wird ein BaseApp Objekt erzeugt (Zeile 1, Listing 10, siehe Kapitel 3.1) und an dieses das Bitmap Objekt mit den gefundenen Linien übergeben (Zeile 2, Listing 10).

3.8 Spielbares Labyrinth darstellen

```
1     @Override
2     protected void onCreate(Bundle savedInstanceState) {
3         super.onCreate(savedInstanceState);
4         DrawingView drawingView = new DrawingView(this);
5         setContentView(drawingView);
6           labyrinth = baseApp.getBitmap();
7
8     }
```

Listing 11 PlayLevel.java: Setzen des ContenViews

Das spielbare Labyrinth wird in *der PlayLevel.java* Klasse dargestellt. Hierzu wurde eine innere Klasse *DrawingView.java* erstellt, die von *der Klasse android.view :: SurfaceView* erbt. Sie besitzt eine *onDraw()* Methode, die überschrieben wird. Dadurch können Objekte gezeichnet werden (Vgl. Android Open Source Project: „public Class SurfaceView“). In der Klasse *PlayLevel.java* wird ein neues Objekt der Klasse DrawingView erzeugt (Zeile 4, Listing 11) und als Darstellungsebene gesetzt (Zeile 5, Listing 11). Da ein BaseApp Objekt nur Informationen zwischen Activities austauschen kann, wird die Bitmap mit den Linien in der Klasse *PlayLevel.java* in eine Bitmap gespeichert (Zeile 6, Listing 11). Da der DrawingView eine innere Klasse ist, kann sie später trotzdem auf die Bitmap zugreifen.

```
1 private void rescaleBitmap() {
2     DisplayMetrics metrics = new DisplayMetrics();
3     getWindowManager().getDefaultDisplay().getMetrics(metrics);
4     int screenHeight = metrics.heightPixels;
5     int screenWidth = metrics.widthPixels;
6
7     labyrinth = Bitmap.createScaledBitmap(labyrinth, screenWidth, screenHeight,
true);
8     }
```

Listing 12 PlayLevel.java: die Methode rescaleBitmap()

Bevor die Bitmap gezeichnet wird, muss sie noch in der Größe angepasst werden, sodass sie den ganzen Bildschirm ausfüllt. Ein neues DisplayMetrics Objekt wird erzeugt (Zeile 2, Listing 12). Die Abmessung des Displays wird diesem Objekt übergeben (Zeile 3, Listing 12) und die Höhe und Breite werden in Integer Objekten gespeichert (Zeile 4f, Listing 12). Diese Werte werden der Methode *createScaledBitmap()* der Klasse Bitmap übergeben. Und diese Methode auf die gewünschte Bitmap angewendet (Zeile 7, Listing 12).

```
1 @Override
2 protected void onDraw(Canvas canvas) {
3     canvas.drawBitmap(labyrinth, 0, 0, paint);
4 }
```

Listing 13 DrawingView: zeichnen der Bitmap

Die Bitmap wird in der *onDraw()* Methode auf ein Canvas Objekt gezeichnet (Zeile 3, Listing 13).

3.9 Spielballerzeugung und Zielerstellung

```
1 private Ball playBall = new Ball();
2 private Ball goalBall = new Ball();
3
4 goalBall.setColorOfBall(Color.GREEN);
5 goalBall.setRadiusOfBall(30);
```

Listing 14 PlayLevel.java: Erzeugen von zwei Ball Objekten und Attributzuteilung

Um den Spielball und das Ziel zu erstellen, werden zwei Objekte der Klasse *Ball.java* erstellt (Zeile 1f, Listing 14). Jedes Objekt der Klasse Ball besitzt eine Farbe, einen Radius und eine bestimmte Position. Auf diese Attribute kann mit den Getter - und Setter-Methoden zugegriffen werden. Durch die Methode *drawBall()* wird ein Ball gezeichnet (Zeile 2, Listing 15).

```
1 public void drawBall(Canvas canvas){
2         canvas.drawCircle(XPositionOfBall,YPositionOfBall,radius,paint);
3     }
```

Listing 15 Ball.java: drawBall()-Methode

```
1  @Override
2 public boolean onTouchEvent(MotionEvent event) {
3     if (event.getAction() == MotionEvent.ACTION_DOWN) {
4
5         if (createStartPoint) {
6             playBall.setPositionsOfBall(event.getX(), event.getY());
7             createStartPoint = false;
8              createFinishPoint = true;
9         } else if (createFinishPoint) {
10            goalBall.setPositionsOfBall(event.getX(), event.getY());
11            createFinishPoint = false;
12            canPlayBallMove = true;
13        }
14    }
15    return false;
16 }
```

Listing 16 DrawingView: onTouchEvent()-Methode

Wenn das Labyrinth gezeichnet wurde, setzt man den Startball, indem man einmal auf den Bildschirm tippt (Zeile 3, Listing 16). Dem ausgelösten Event werden die X-und Y-Koordinaten ausgelesen, an der Position an der das Event ausgelöst wurde. Die Koordinaten werden dem Ball Objekt über die Methode *setPositionOfBall()* als Attributwerte übergeben (Zeile 6, Listing 16). Die Position des Ziels wird auf die gleiche Weise gesetzt.

```
1 @Override
2 protected void onDraw(Canvas canvas) {
3     canvas.drawBitmap(labyrinth, 0, 0, paint);
4
5     playBall.drawBall(canvas);
6      goalBall.drawBall(canvas);
7 }
```

Listing 17 DrawingView: Startball und Ziel zeichnen

Beide Kreise werden in der *onDraw()*-Methode des DrawingViews gezeichnet (Zeile 5f, Listing 17).

3.10 Spielballbewegung

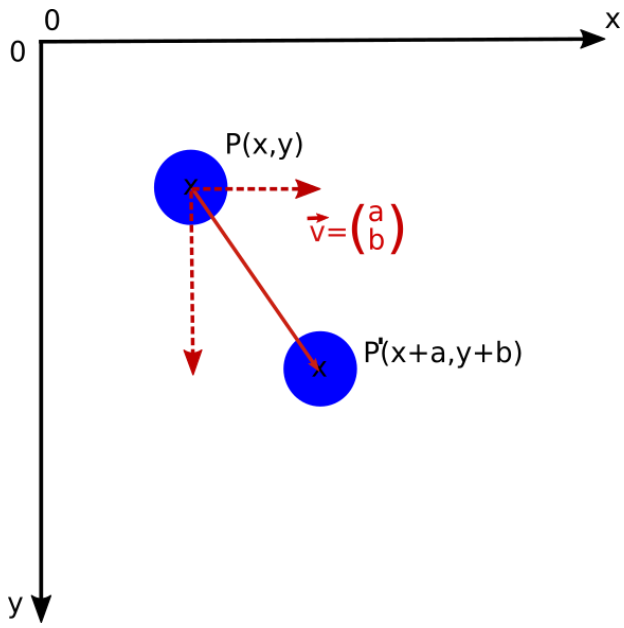

Abb. 25 Skizze: Ballbewegung

Der Ball wird von seiner derzeitigen Position um einen bestimmten Richtungsvektor verschoben (siehe Abb. 25).

Zur Bestimmung eines Richtungsvektors wird der eingebaute Beschleunigungssensor benutzt. Dieser misst die Beschleunigung des Geräts in die x-, y- und z-Richtung (Vgl. Android Open Source Project: „Sensor Overview“). Da auf das Gerät die Erdbeschleunigung wirkt, können geeignete Daten ausgelesen und als Richtungsvektor verwendet werden.

```
1  private PlayLevel parent = new PlayLevel();
2
3  public AccelerometerClass(Context context){
4      parent =(PlayLevel)context;
5      initSensors();
6  }
7   public void onSensorChanged(SensorEvent e) {
8
9      accX = e.values[0];
10     accY = e.values[1];
11
12     moveX = accX.intValue();
13     moveY = accY.intValue();
14
15     parent.updateAccelerometer(moveX,moveY);
16
17 }
```

Listing 18 AccelerometerClass.java: auslesen und Senden der Sensordaten

In der *AccelerometerClass.java* Klasse wird ein neues Objekt der Klasse *PlayerLevel.java* erstellt (Zeile 1, Listing 18). An diese sollen die ausgegebenen Sensordaten gesendet werden.

Der Bewegungssensor wird mit der Methode *initSensors()* initialisiert (Zeile 5, Listing 18) und gestartet. Die Methode *onSensorChanged()* wird von einem SensorEventListener Objekt aufgerufen, wenn sich die Werte des Sensors verändern (Vgl. Android Open Source Project: „Sensors Overview“).

Dem *onSensorChanged()* übergebenen SensorEvent Objekt werden die Beschleunigungswerte der x-Achse (Zeile 9, Listing 18) und die Beschleunigungswerte der y-Achse (Z.10, Listing 18) ausgelesen. Diese werden in Integer Werte umgewandelt und der PlayLevel Klasse mit der Methode *updateAccelerometer()* übergeben.

```
1 public void updateAccelerometer(int x, int y) {
2     moveX = x;
3     moveY = y;
4 }
```

Listing 19 PlayLevel.java: updateAccelerometer()-Methode

In der PlayLevel.java Klasse, aktualisiert die *updateAccelerometer()*-Methode die Werte für den Richtungsvektor des Balls (Zeile 1f, Listing 19).

```
1  @Override
2 protected void onDraw(Canvas canvas) {
3     playBall.drawBall(canvas);
4     playBall.setPositionsOfBall(playBall.getXPositionOfBall() + (speedX * moveX),
playBall.getYPositionOfBall() + (speedY * moveY));
5 }
```

Listing 20 DrawingView: errechne die neue Position des Balls

Jedes Mal, wenn die Methode *onDraw()* aufgerufen wird, wird die Position des Balls neu berechnet. Mit der Methode *setPositionOfBall()* wird das Ball Objekt auf eine neue Position gesetzt. Erst beim nächsten Aufruf der Methode wird der Ball dann auf die Position gezeichnet (Zeile 4, Listing 20).

Die x- Koordinate wird durch

$$x_{neu} = ((x_{alt} + (moveX * speedX))$$

mit

$$moveX \dots Beschleunigung\ in\ X - Richtung$$

und

$$speedX = -0{,}5$$

berechnet.

Das Koordinatensystem des Bewegungssensors und das Koordinatensystem zum Zeichnen stimmen nicht überein, daher wird der x-Wert mit der Konstante *speedX* negiert.

Die y-Koordinate wird durch

$$y_{neu} = ((y_{alt} + (moveY * speedY))$$

mit

$$moveY \dots Beschleunigung\ in\ Y - Richtung$$

und

$$speedY = 0{,}5$$

berechnet.

3.11 Kollisionskontrolle

3.11.1 Kollision mit einer Linie

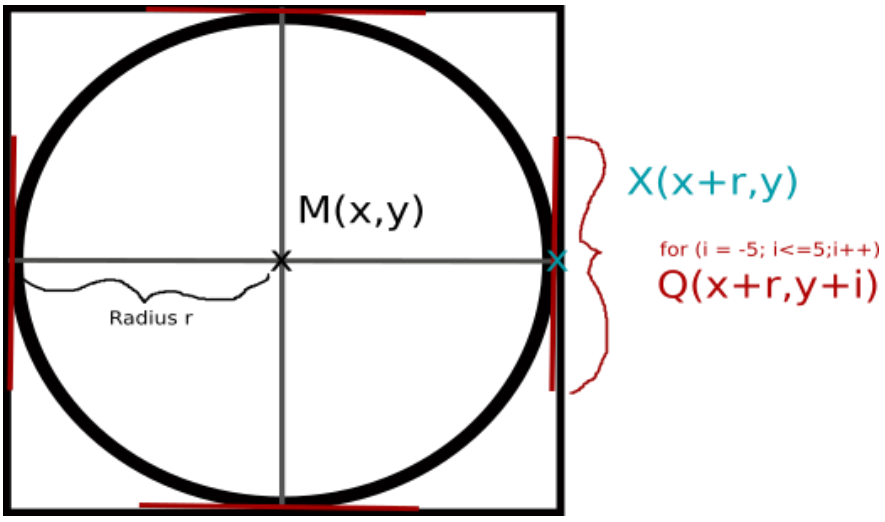

Abb. 26 Skizze: Kollision mit einer Linie

Um zu überprüfen, ob der Ball an eine Linie gestoßen ist, wird der Umkreis des Balls betrachtet. Der Pixelwert der gezeichneten Bitmap wird im Umkreis des Balls ausgelesen. Da bekannt ist, welchen RGB-Wert die Linien besitzen, wird überprüft, ob ein Randpixel des Kreises sich auf der Position eines Pixels der Bitmap befindet, die diesen Wert besitzt. Dabei werden nicht die gesamten Pixel des Umkreises benutzt, sondern Pixel auf Geraden, die die senkrechten und waagrechten Punkte des Kreises berühren und orthogonal zur x-Achse, bzw. y-Achse verlaufen.

Bei jedem Aufruf der onDraw()-Methode des DrawingViews, wird die Methode *collisionWithBitmap()* aufgerufen und dieser die aktuellen Koordinaten des Spielballs übergeben (Zeile 1, Listing 21).

```
1  private void collisionWithBitmap(int x, int y){
2      for (int i = -4; i <= 4; i++) {
3
4
5              if (labyrinth.getPixel(x - 5, y + i) == Color.rgb(87, 155, 25)) {
6
7                      collisionWithLine = true;
8                      pixelThatCollided = 4;
9                      canMoveChange = false;
10                     break;
11
12             } else if (labyrinth.getPixel(x + i, y - 5) == Color.rgb(87, 155, 25)) {
13                     collisionWithLine = true;
14                     pixelThatCollided = 3;
15                     canMoveChange = false;
16                     break;
17
18             } else if (labyrinth.getPixel(x + i, y + 5) == Color.rgb(87, 155, 25)) {
19                     collisionWithLine = true;
20                     pixelThatCollided = 1;
21                     break;
22
23                     canMoveChange = false;
24             } else if (labyrinth.getPixel(x + 5, y + i) == Color.rgb(87, 155, 25)) {
25                     collisionWithLine = true;
26                     pixelThatCollided = 2;
27                     canMoveChange = false;
28                     break;
29
30             else {
31                     if (canChangePixelThatCollided) {
32                             collisionWithLine = false;
33                             pixelThatCollided = 0;
34                     }
35
36             }
37     }
38 }
```

Listing 21 DrawingView: collisionWithBitmap()-Methode

Es werden die in Abb. 26 eingetragenen Tangentenabschnitte betrachtet. Es wird überprüft, ob die Positionen der Pixel, mit der Position einer Line der Bitmap übereinstimmen. Hierzu wird der RGB-Farbwert an der Position der Pixel der Geraden auf der Bitmap ausgelesen und mit dem RGB-Farbwert der Linien verglichen(Zeile 4, Listing 21). Dadurch, dass jede Seite des Kreises einzeln überprüft wird, kann genau

bestimmt werden an welcher Seite eine Kollision auftritt. Diese Information wird in der Variablen *pixelThatCollided* gespeichert (Zeile 8, Listing 21).

Nachdem eine Kollision festgestellt wird, bewegt sich der Ball mit der gleichen Geschwindigkeit weiter, nur dass der Richtungsvektor wechselt, sodass der Einfallswinkel dem Ausfallswinkel entspricht (siehe Abb. 27).

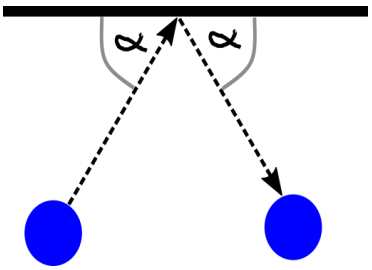

Abb. 27 Skizze: Abprall Bewegung

Danach wird die Methode *moveBallAfterCollisionWithLine()* aufgerufen.

Es wird die aktuelle Beschleunigung in den Variablen *moveXAfterCollision* und *moveYAfterCollision* abgelegt (Zeile 10f, Listing 22). Für jede Seite die kollidieren kann (Zeile 26, Listing 22), wird ein neuer Richtungsvektor berechnet (siehe Abb. 28) und das Ball Objekt an die neue Position gesetzt. Diese Methode wird in der *onDraw()*-Methode für vier *onDraw()* Aufrufe aufgerufen. Somit führt der Ball für diese Zeit eine konstante Abprall Bewegung durch. Danach wird der Richtungsvektor wieder durch die Beschleunigungssensorausgaben festgelegt.

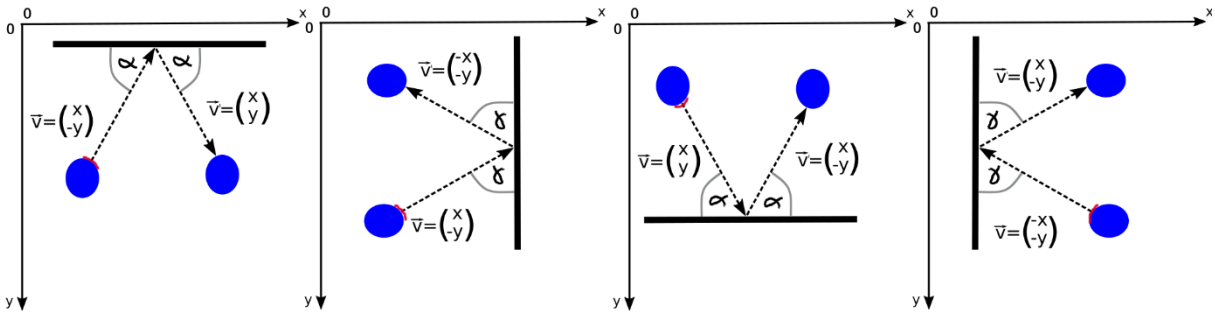

Abb. 28 Richtungsvektoren nach einer Kollision mit einer Linie

```
        private void moveBallAfterCollisionWithLine() {

            float x = playBall.getXPositionOfBall();
            float y = playBall.getYPositionOfBall();
            int speed = 1;



            if (!canMoveChange) {
                moveXAfterCollision = moveX * -1;
                moveYAfterCollision = moveY * -1;
                canMoveChange = true;
            }
            /*
                        1  1
                     *        *
                    4          2
                    4          2
                     *        *
                        3  3


                    */


            if (pixelThatCollided == 1) {

                playBall.setPositionsOfBall(x + moveXAfterCollision * speed, y +
moveYAfterCollision * speed);

            } else if (pixelThatCollided == 2) {

                playBall.setPositionsOfBall(x + moveXAfterCollision * -1 * speed, y
+ moveYAfterCollision * speed * -1);

            } else if (pixelThatCollided == 3) {

                playBall.setPositionsOfBall(x + moveXAfterCollision * speed, y +
moveYAfterCollision * speed);
            } else if (pixelThatCollided == 4) {

                playBall.setPositionsOfBall(x + moveXAfterCollision * -1 * speed, y
+ moveYAfterCollision * -1 * speed);
            }


        }
```

Listing 22 DrawingView: moveBallAfterCollisionWithLine()-Methode

3.11.2 Kollision mit dem Ziel

```
1       private boolean onFinishCollision() {
2             float x1 = playBall.getXPositionOfBall();
3             float y1 = playBall.getYPositionOfBall();
4             int r1 = playBall.getRadius();
5
6             float x2 = goalBall.getXPositionOfBall();
7             float y2 = goalBall.getYPositionOfBall();
8             int r2 = goalBall.getRadius();
9
10            if (Math.pow(x1 - x2, 2) + Math.pow(y1 - y2, 2) <= Math.pow(r1 + r2, 2))
{
11                return true;
12            }
13            return false;
14        }
```

Listing 23 DrawingView: onFinishCollision()-Methode

Ob das Ball Objekt das Ziel erreicht hat, wird in der Methode *onFinishCollision()* geprüft. Dabei wird überprüft, ob der Abstand zwischen dem Spielball und dem Zielpunkt kleiner oder gleich der Länge der Radien ist (Zeile 10, Listing 23). Es wird ein *true* oder *false* zurückgegeben (Zeile 11ff, Listing 23).

Wenn ein *true* zurückgegeben wird, wird in der onDraw()-Methode das Zeichnen des Gewinnerbildschirms eingeleitet.

3.11.3 Kollision mit dem Canvasrand

```
1       private void collisionWithBorderOfCanvas(Ball ball, Canvas canvas) {
2             int x = ((int) ball.getXPositionOfBall());
3             int y = ((int) ball.getYPositionOfBall());
4             int canvasWidth = canvas.getWidth();
5             int canvasHeight = canvas.getHeight();
6
7             if (canPlayBallMove) {
8                 if (x - 11 < 0) {
9                     ball.setXPositionOfBall(12);
10                    collisionWithBorder = true;
11
12
13                } else if (x + 11 > canvasWidth) {
14                    ball.setXPositionOfBall(canvasWidth - 12);
15                    collisionWithBorder = true;
16                } else if (y - 11 < 0) {
17                    ball.setYPositionOfBall(12);
18                    collisionWithBorder = true;
19                } else if (y + 11 > canvasHeight) {
20                    ball.setYPositionOfBall(canvasHeight - 12);
21                    collisionWithBorder = true;
22                } else {
23                    collisionWithBorder = false;
24                }
25            }
26
27
28        }
```

Listing 24 DrawingView: collisionWithBorderOfCanvas()-Methode

Zuletzt wird noch überprüft, ob sich der Ball am Canvasrand befindet, da er sich nicht aus dem Bildschirm bewegen darf. Hierzu wird der Methode *collisionWithBorderOfCanvas()* das Ball Objekt und das Canvas Objekt übergeben (Zeile 1, Listing 24). Die Position des Balls wird ausgelesen (Zeile 2f, Listing 24). Außerdem wird die Höhe und Breite des Canvas Objekts in die Variablen *canvasWidth* und *canvasHeight* gelegt. Es wird überprüft, ob die x-Koordinate des Balls minus des Radius kleiner als Null ist (Zeile 8, Listing 24). Falls dies der Fall ist, wird die x-Koordinate des Balls auf den Radius plus zwei gesetzt. Der Ball kann somit nicht mehr aus dem Bild rollen. Dies wird bei den anderen drei Seiten des Balls ebenfalls überprüft.

4. Fazit

Das Ziel dieser Bachelorarbeit war, eine funktionsfähige App zu entwickeln, die auf Papier gezeichnete Labyrinthe abfotografiert, erkennt und daraus Spielfelder generiert. Außerdem sollte eine funktionierende Kollisionskontrolle entworfen und in das Spiel integriert werden.

Ein weiteres Ziel war die Herstellung eines Quellcodes, der logisch und nachvollziehbar ist. Dieser Code sollte außerdem erweiterbar sein.

Zunächst musste verstanden werden, wie eine App aufgebaut ist und wie man ihre unterschiedlichen Komponenten miteinander verbindet. Hierfür wurde hauptsächlich mit der Android Dokumentation gearbeitet.

Die App wurde verschiedenen Personen zum Testen gegeben. Diese haben die App auf unterschiedlichen Geräten gespielt. Die Spielfeldgenerierung und die Kollisionskontrolle haben sehr gut funktioniert und die Benutzer hatten Spaß beim Spielen der App.

Die App wurde so programmiert, dass sie erweiterbar ist.

Derzeit ist die App so konstruiert, dass der Benutzer ein schnelles Spiel starten kann. Dabei sind die Parameter für die Linienerkennung aus Erfahrungswerten generiert worden. Diese Werte sind veränderbar. Daher wäre es in einem erweiterten Modus möglich, die Linienparameter so zu verändern, dass sie an neue Ansprüche angepasst werden können. Dies kann durch den Einsatz von Schieberegler realisiert werden.

Des Weiteren wurde der Spielball so programmiert, dass man seine Größe und Farbe verändern kann. Damit hat der Benutzer die Möglichkeit, sie nach seinen eigenen Wünschen zu gestalten.

Durch den Einsatz des Hotwire-Modus könnte ein weiterer Spielmodus eingefügt werden. Hierbei würde der Spielball, sobald er eine Linie berührt, auf seine Startposition zurückgesetzt werden. Dies hätte zur Folge, dass noch mehr Spielvielfalt und eine erhöhte Dynamik ins Spiel kämen.

Die Ausarbeitung der App stellte zunächst eine große Herausforderung dar. Die Schwierigkeit bestand darin, viele Ideen und komplizierte Zusammenhänge so zu strukturieren, dass sie nachvollziehbar dargestellt werden konnten.

Zusammenfassend ist zu sagen, dass die gesetzten Ziele erfüllt wurden.

5. Abbildungsverzeichnis

6. Literaturverzeichnis

Android Open Source Project: „Activities“, unter http://developer.android.com/guide/components/activities.html (letzter Aufruf: 17.02.2016).

Android Open Source Project: „Android Studio Overview“, unter http://developer.android.com/tools/studio/index.html (letzter Aufruf: 17.02.2016).

Android Open Source Project: „public Class Application“, unter http://developer.android.com/reference/android/view/SurfaceView.html (letzter Aufruf: 17.02.2016).

Android Open Source Project: „public Class SurfaceView“, unter http://developer.android.com/reference/android/view/SurfaceView.html (letzter Aufruf:17.02.2016).

Android Open Source Project: „Sensors Overview“, unter http://developer.android.com/guide/topics/sensors/sensors_overview.html (letzter Aufruf: 17.02.2016).

Opencv dev team: „Hough Line Transform“, unter http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_houghlines/py_houghlines.html (letzter Aufruf: 20.02.2016).

Rafael C. Gonzalez / Richard E.Woods: „Digital Image Processing (Third Edition)“, Pearson Education International, 2007

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelor-Thesis mit dem Titel:

„Entwicklung einer Labyrinth-App mit automatischer Konstruktion eines Spielfeldes auf Basis von fotografierten Handzeichnungen“

selbständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich wörtlich aus der Literatur oder aus anderen Quellen wie z.B. Internetseiten übernommen habe, habe ich deutlich als Zitat mit Angabe der Quelle kenntlich gemacht.

Hamburg, 25.02.2016

Ort, Datum Tanja Blücher