Fachhochschule Köln
Cologne University of Applied Sciences
Campus Gummersbach
Fakultät für Informatik und Ingenieurwissenschaften

Verbundstudiengang Wirtschaftsinformatik

Projektarbeit

Gestengesteuerte Funktionalität unter dem Android Betriebssystem

Prüfer:	Prof. Dr. Erich Ehses
vorgelegt am:	25. Oktober 2015
von cand.:	Stephan Wagner
aus:	Overath
Telefon-Nr.:	+49-176-80007570
Matrikel-Nr.:	1106011828
E-Mail-Adresse:	stephan.wagner.mi738@gmail.com

Zusammenfassung

In dieser Projektarbeit gilt es, den Problemstellungen in der Android Applikationsentwicklung bezüglich der Einstiegshürden und dem Problem des Code Copy entgegenzuwirken. Entsprechend wird geeigneter Ansatz zur Vereinfachung und Abstraktion von häufig verwendeten Methoden entwickelt und evaluiert. Um sich der Problemstellungen in Android, bezüglich der Einstiegshürden und dem Code Copy anzunähern, wird dessen konkrete Ausprägung am Beispiel „Steuerung von Funktionalität durch Gesten“ untersucht. Hierzu wird zunächst ein allgemeines Verständnis zu Gesten, sowie deren Erkennung aufgebaut, um dann an Hand von konkreten Bespielen, ein tieferes Verständnis der Steuerung von Funktionalität mittels Gesten zu erlangen. Die dabei gewonnenen Erkenntnisse werden in einem Abstraktionsprozess auf eine generische Implementierung übertragen, um weitergehend zu diskutieren, wie diese für andere App Projekte verfügbar gemacht werden kann.
Abschließend steht die Frage, in wie weit der Lösungsansatz valide ist gegenüber weiteren Ausprägungen der Eingangsproblematik bzw. ob sich daraus sogar ein allgemeines Lösungskonzept ableiten kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Applikationen für mobile Endgeräte sind in großer Vielfalt am Markt verfügbar und erfreuen sich quer durch alle Bevölkerungsschichten wachsender Beliebtheit. Dabei nehmen die Möglichkeiten, wie sogenannte Apps den Alltag bereichern und / oder erleichtern mit jeder neuen Hardwareschnittstelle zu. Die meist verbreiteten Betriebsysteme iOS und Android ermöglichen dabei mit dem jeweiligen Software Development Kit (kurz SDK) den Zugriff auf diese Schnittstellen. Applikationsentwickler haben damit die Möglichkeit dem Nutzer ein reichhaltiges Angebot von Applikationen zur Verfügung zu stellen.

1.1 Motivation

Die rasante Verbreitung der Technologien zu Mobilen Geräten erschließt ein großes Potential an Innovationsträgern für neue Applikationen. Da die Android Applikationsentwicklung auf der freien Entwicklung basiert (d.h. Jedem ist es erlaubt Apps zu schreiben), gehören neben dem Kreis professioneller Android Entwickler ebenso nicht professionelle und Einsteiger in der Android Entwicklung zu den Innovationsträgern. Jedoch ist die Einstiegshürde selbst für erfahrene Softwareentwickler unter Umständen so hoch, dass die Umsetzung von neuen Ideen bereits im Ansatz scheitern kann. Um die Komplexität der Entwicklung von Applikationen unter iOS zu reduzieren, stellt der Hersteller Apple zumindest eine Entwicklungsumgebung bereit, in der einfache Funktionalität auf einem Storyboard aus einer Auswahl an Funktionen zusammengestellt werden kann. Kenntnisse in der Programmierung sind somit nicht in allen Entwicklungsschritten zwingend notwendig. Dagegen bietet Android keine vergleichbare Unterstützung durch ein Storyboard, wodurch die Entwicklung von Android- Applikationen zahlreiche, hohe Einstiegshürden birgt. Bei einer Einstiegshürde handelt es sich um die technische Umsetzung von bestimmten Funktionalitäten, die für eine Applikation benötigt werden,

was dazu führen kann, dass ein und dieselbe Funktionalität in unterschiedlichen Applikationen von Entwicklern auf unterschiedliche Art und Weise implementiert wird. Für weniger erfahrene Entwickler oder Einsteiger besteht hier die Gefahr, Funktionalität fehlerhaft oder unperformant zu konzipieren.
Ein weiteres Problem ist das des Code-Copy. Dies beschreibt, wie gleiche Funktionalität durch Kopieren des jeweiligen Sourcecode in unterschiedliche Softwarekomponenten durch Dublizieren des Sourcecode integriert wird, anstatt auf eine zentrale Implementierung zu referenzieren.

1.2 Zielsetzung

Ziel dieser Projektarbeit ist es, einen Lösungsansatz zu entwickeln, um den Problemstellungen in der Android- App- Entwicklung bezüglich der Einstiegshürden und dem Problem des Code-Copy entgegenzuwirken. Hierzu sind mittels Hilfe von konkreten Implementierungsbeispielen die Einstiegshürden sowie die Gefahren des Code-Copy bei der Android-Entwicklung zu veranschaulichen und daraus Erkenntnisse für die Konzeption eines allgemeinen Lösungsansatzes abzuleiten. Dieser Ansatz ist abschließend dahingehend zu untersuchen, ob er für weitere technologische Problemstellungen in diesem Kontext der Android- Entwicklung anwendbar wäre. Die Arbeit, sowie die darin aufgeführten Codebeispiele, beziehen sich auf die Android- Version 4.x. Die Version 4.0 (API Level 14) stellt dabei die Mindestanforderung an das verwendete Betriebssystem dar.

1.3 Inhalt und Vorgehen

Um sich der Problemstellungen in Android bezüglich der Einstiegshürden und dem Code Copy anzunähern, wird dessen konkrete Ausprägung am Beispiel „Steuerung von Funktionalität durch Gesten“ untersucht. Hierzu wird zunächst ein allgemeines Verständnis zu Gesten, sowie deren Erkennung aufgebaut, um dann an Hand von konkreten Bespielen ein tieferes Verständnis der Steuerung von Funktionalität mittels Gesten zu erlangen. Die dabei gewonnenen Erkenntnisse werden in einem Abstraktionsprozess

auf eine generische Implementierung übertragen, um weitergehend zu diskutieren, wie diese für andere App Projekte verfügbar gemacht werden kann.
Abschließend steht die Frage, inwieweit der Lösungsansatz valide ist gegenüber weiteren Ausprägungen der Eingangsproblematik, bzw. ob sich daraus sogar ein allgemeines Lösungskonzept ableiten lässt.

2 Gesten im Allgemeinen

2.1 Begriffsdefinition

Gesten sind bewusste oder unbewusste Bewegungen bzw. Handlungen mit symbolischem Charakter eines Individuums, mit dem Ziel der nonverbalen Kommunikation oder der Unterstützung einer verbalen Kommunikation. Die Art und Weise einer Geste hängt dabei stark von den mentalen Modellen des Individuums ab, die es versucht durch die Geste zum Ausdruck zu bringen. Die Art und Weise der gestikulierenden Bewegung sowie die Semantik dahinter hängt u.a. vom kulturellen Hintergrund des Individuums, aber auch von körperlichen Eigenschaften wie Behinderungen ab.
Spätestens seitdem Dr. Sam Hurst 1974 den ersten Touchscreen entworfen hat, wird bei der Entwicklung technischer Systeme versucht, die Gesten als Interaktionsparadigma zu nutzen, um eine möglichst intuitive Mensch-Computer-Interaktion zu schaffen. Das durch die Gesten zu steuernde System registriert dabei vordefinierte Gesten mittels einer geeigneten Hardwareschnittstelle (z.B. Touchscreen) und führt pro Geste definierte Aktionen aus. In diesem Kontext wird unterschieden zwischen Single- und Multi-Touch-Gesten. Single-Touch-Gesten meinen einzelne Berührungen und Bewegungen (also z.B. mit einem Finger) auf einem Touchscreen. Der Android Style Guide nennt hierzu Gesten wie „Touch“, „Drag“ oder „Swipe and Drag“. Im Gegenzug bilden Multi-Touch-Gesten Bewegungen mit mehreren Fingern ab (siehe hierzu im Android Style Guide „Pinch-Open“ und „Pinch-Close“). Die Entwicklung der Sensoren zur Registrierung der einzelnen Berührungsformen ist dabei entscheidend.

2.2 Meilensteine der Sensortechnologie zur Registrierung von Gesten

Die Entwicklung der Sensortechnologie zur Registrierung von Gesten ist in den letzten Jahren stark vorangetrieben worden. Das IBM Simon (1992) gilt als das erste mobile Endgerät mit Touchscreen. Der Grundstein für die Weiterentwicklung des Touchscreen wurde jedoch bereits in den frühen 70er Jahren gelegt, indem Dr. Sam Hurst das erste transparente Touchdisplay entwickelte. Der darin verbaute Sensor konnte einfache Single-Touch-Gesten registrieren. In den folgenden Jahren wurden die unterschiedlichsten technischen Verfahren entwickelt. Daraus ist 1984 der erste Touchsensor hervorgegangen, mit dem Multi-Touch-Gesten registriert werden konnten. Der Zeitstrahl in Abbildung 1 gibt einen Einblick in eine Auswahl von wichtigen Meilensteinen bei der Entwicklung der Sensortechnologie:

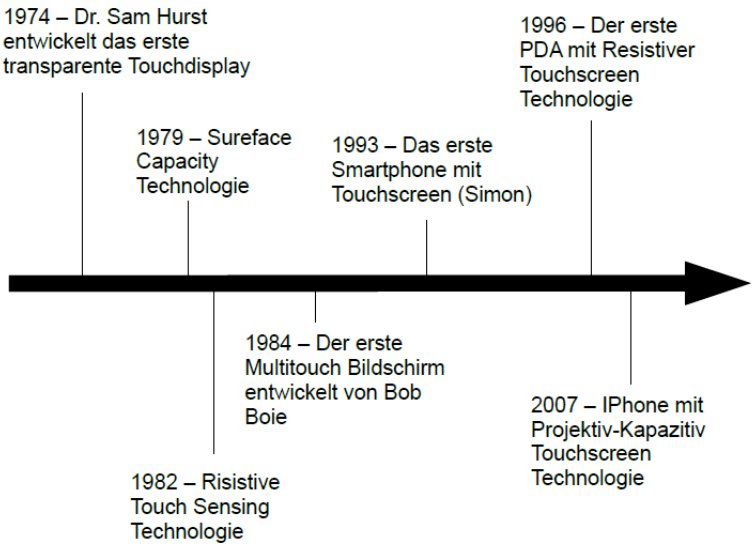

Abbildung 1: Meilensteine der Touchscreenentwicklung [**?**,]

Einen kurzen Einblick in die unterschiedlichen Sensortechnologien bietet das folgende Kapitel.

2.3 Sensortechnologieen zur Registrierung von Berührungen

Die Entwicklung der Gesten hängt unmittelbar mit der Entwicklung der Sensortechnologie zur Registrierung von Berührungen auf einem Bildschirm zusammen. Dabei sind die ersten Berührungsbildschirme, basierend auf optischen Bildschirmsensoren, beschränkt auf einfache Druck-Gesten. Diese Technologien sind aus der Entwicklung des Touchscreens hervorgegangen:

2.3.1 Biegewelle (Dispersive Signal) Technologie

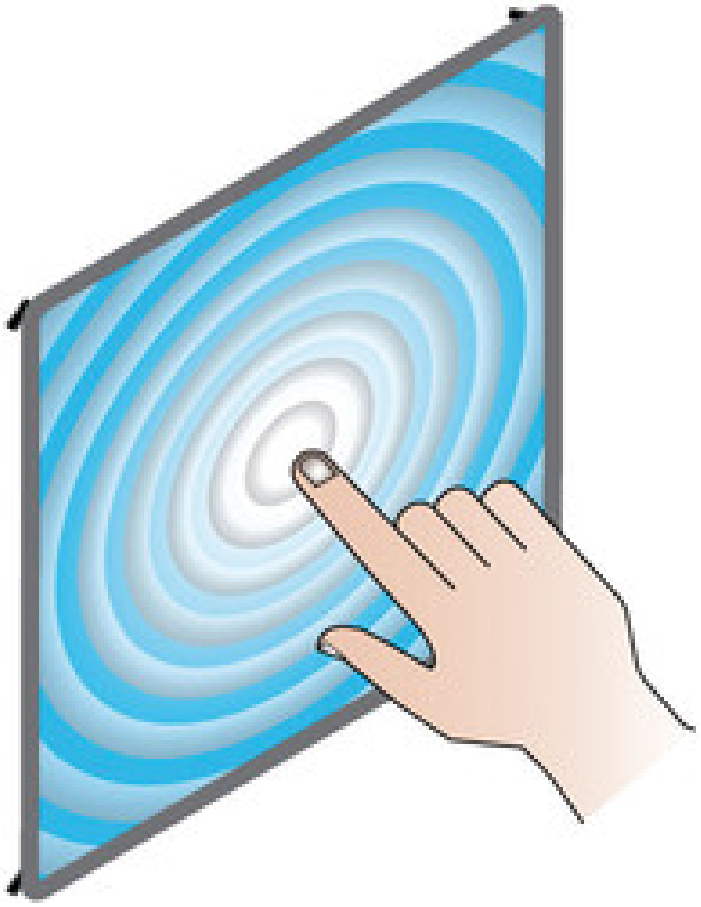

Abbildung 2: Biegewelle Technologie [**?**,]

Das Funktionsprinzip basiert auf der Ausbreitung von durch Berührung ausgelösten Schwingungen in einem bestimmten Medium, welches in den Bildschirm integriert ist. Diese Schwingungen werden an den Bildschirmekken durch Sensoren gemessen. Der relative Zeitunterschied in Bezug zu den Auftreffzeitpunkten der Welle auf die Sensoren wird für die Berechnung der Position der Berührung auf dem Bildschirm verwendet. Diese Technologie leistet nur die einfache Erkennung einer Berührung (z.B. nur ein Finger), da bei mehrfacher Berührung die Überlagerung von Wellen (Interferenz) zu ungenauen Berechnungen der Positionen führen würde.

2.3.2 Infrarot- Gitter Technologie

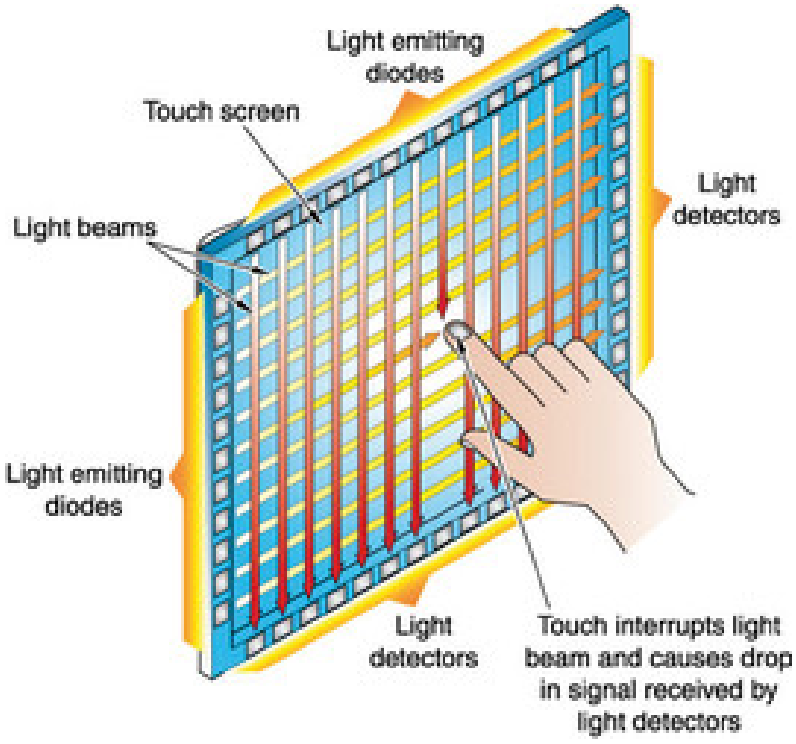

Abbildung 3: Infrarot Gitter Technologie [**?**,]

Die Infrarot-Gitter-Technologie teilt den Touchbildschirm mittels Infrarotstrahlen in ein Raster auf. Jede Berührungsgeste unterbricht einzelne Infrarotstrahlen, was durch entsprechende Sensoren wahrgenommen wird. Mehrfache Berührungen werden durch dieses Konzept prinzipiell unterstützt, wobei es zu technischen Problemen führen kann, wenn mehrere

dicht zusammenliegende Berührungspunkte zu unterscheiden sind, da hierfür die Maschen des Infrarotrasters eng genug konstruiert sein müssen.

2.3.3 Optische Rezeptor Technologie

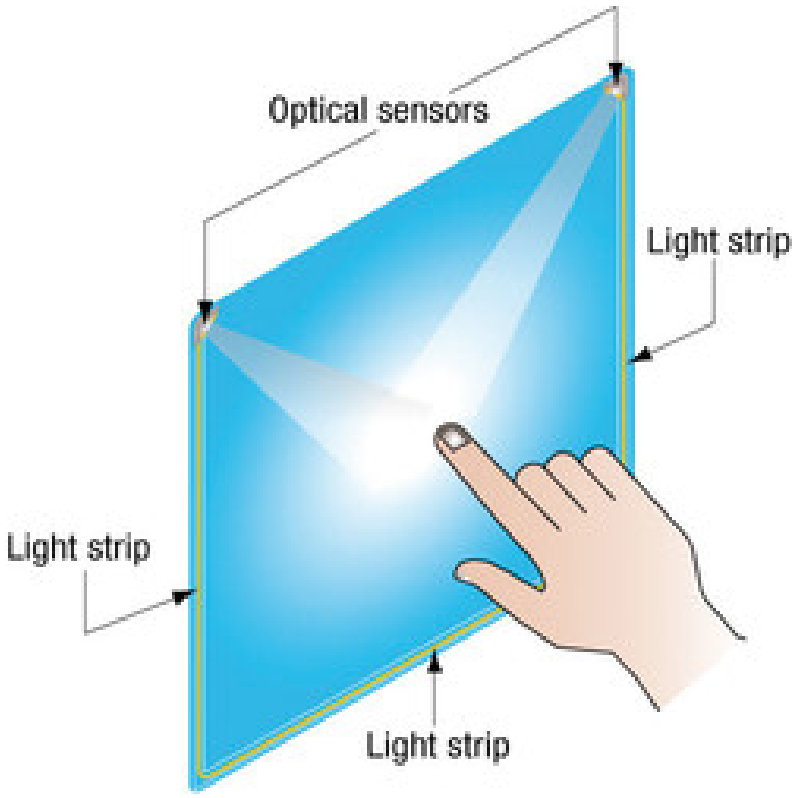

Abbildung 4: Optische Rezeptor Technologie [**?**,]

Die optische Rezeptorentechnologie zur Registrierung von Berührungen auf dem Touchbildschirm realisiert die Erkennung von Gesten mittels optischer Sensoren an den Bildschirmecken. Diese messen bei Berührung des Bildschirmes die Lichtunterbrechung des an den Kanten des Bildschirmes emittierten Lichtes. Ähnlich wie bei der Infrarottechnologie gibt es auch hier technologische Schwierigkeiten, dicht zusammenliegende Berührungspunkte auseinanderzuhalten, besonders, da lediglich zwei optische Sensoren verbaut wurden.

2.3.4 Resistive Touchbildschirm Technologie

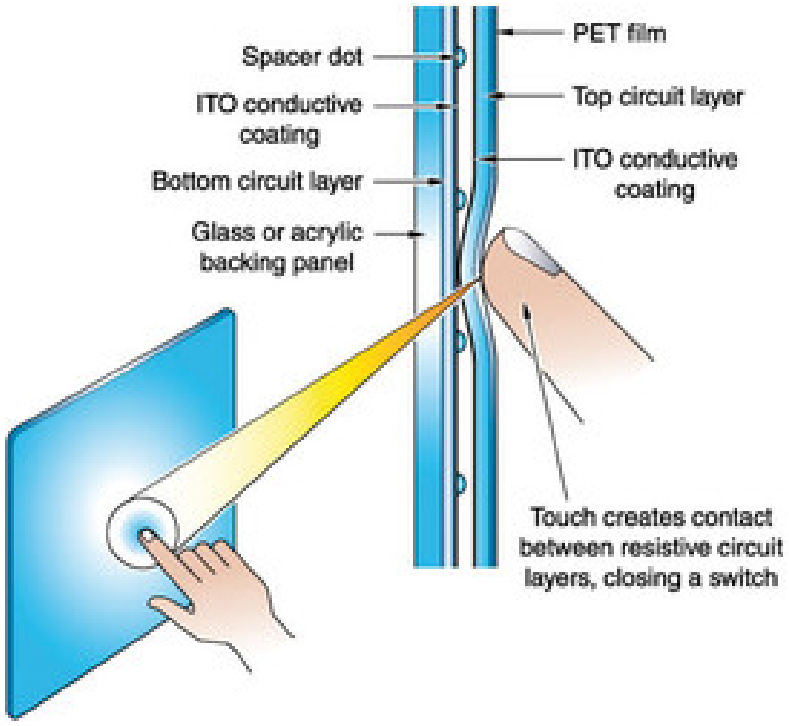

Abbildung 5: Resistive Technologie [?,]

In der resistiven Touchscreen-Technologie wird ein Touchbildschirm in Deckschicht und Grundschicht unterteilt. An den Innenkanten beider Schichten ist ein leitender Film angebracht, der je mit unterschiedlicher Spannung geladen ist. Wie in der Abbildung gezeigt, schließt eine Berührung den Stromkreis und mit Hilfe der an den Bildschirmkanten angebrachten Sensoren die genauen Koordinaten des Berührungspunktes ermittelt. Prinzipiell ist bei dieser Technologie die Erkennung von Mehrfachberührungen möglich, jedoch hängt dies von den verbauten Sensoren ab.

2.3.5 Surface Accustic Wave (SAW)

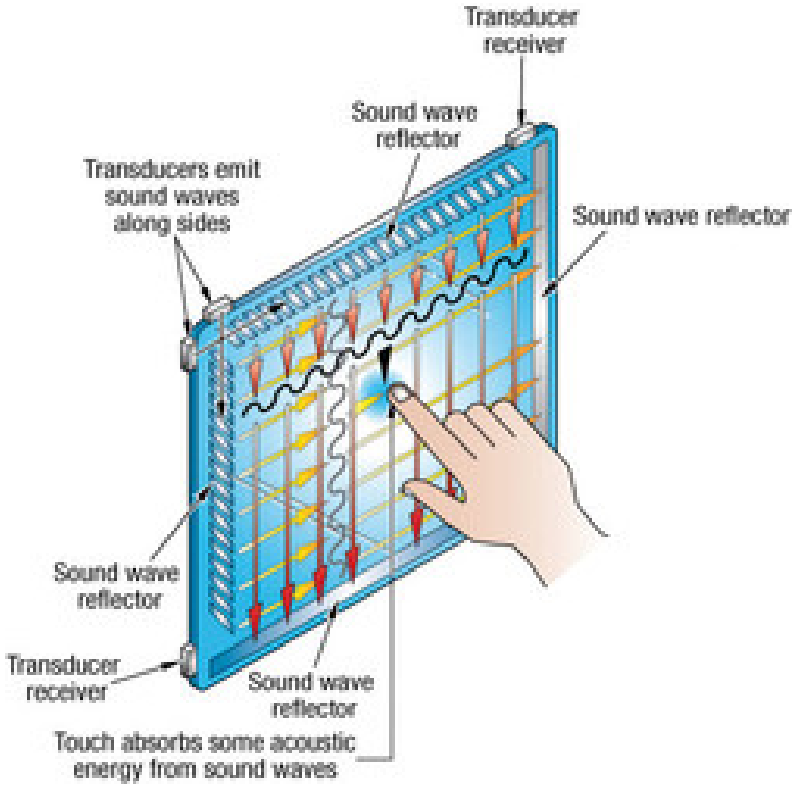

Abbildung 6: Surface Accustic Wave Technologie [**?**,]

Das technische Prinzip dieser Technologie ist vergleichbar mit dem des Infrarot-Gitters. Schallwellen werden an bestimmten Punkten des Touchbildschirms emittiert und an zwei Kanten (x;y) so reflektiert, dass ein Raster durch die Schallwellen im Berührungsbereich aufgespannt wird. An den zwei übrigen Kanten leiten Reflektoren die einzelnen Schallwellen zu einem Sensor. Die Berührung hat eine Dämpfung der Schallwellen (in x- und y-Richtung) zur Folge, die zum Zeitpunkt der Berührung den Berührungspunkt passieren. Da die Schallwellen zeitverzögert erzeugt werden, also nur immer eine Schallwelle in eine X-Richtung und nur eine in Y-Richtung, ist somit der Berührungspunkt genau zu ermitteln. Dies bedeutet jedoch auch, dass zum selben Zeitpunkt nur ein Berührungspunkt ermittelbar ist, womit Multi-Touch-Gesten durch diese Technologie nicht unterstützt werden.

2.3.6 Oberflächen-kapazitive Touchscreen Technologie

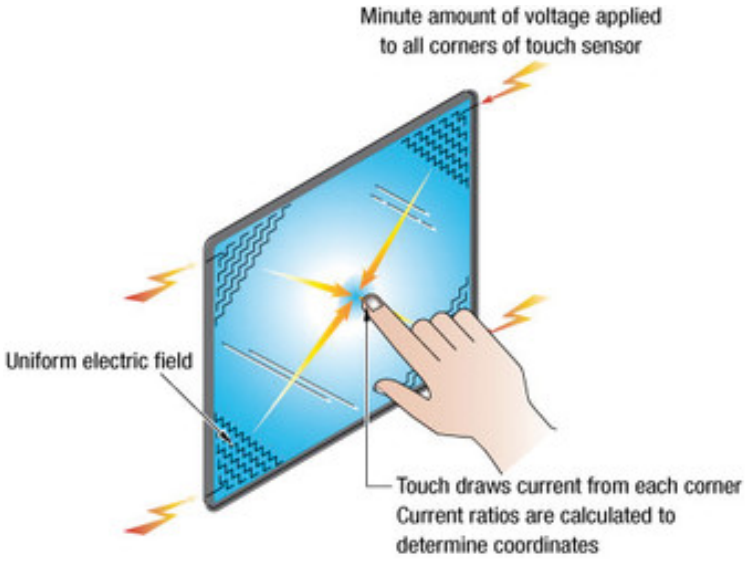

Abbildung 7: Oberflächen Kapazitive Technologie

Die kapazitive Touchscreen-Technologie registriert Berührungen durch die Ablenkung innerhalb eines elektronischen Feldes, welches durch Elektroden an den Bildschirmkanten erzeugt wird. Sensoren an den Bildschirmecken messen dabei das elektronische Feld und damit jegliche durch Berührung ausgelöste Veränderung des Feldes. Der Finger fungiert dabei als Erdung, über den Elektronen aus dem Feld abfließen und somit für einen Spannungsabfall während der Berührung sorgt. Die genaue X- und Y- Koordinate errechnet sich aus den Strömen, die an den Sensoren registriert werden. Auch hier beschränkt sich die Technologie auf die Registrierung von lediglich einer Geste.

2.3.7 Projiziert Kapazitiver Touchscreen Technologie

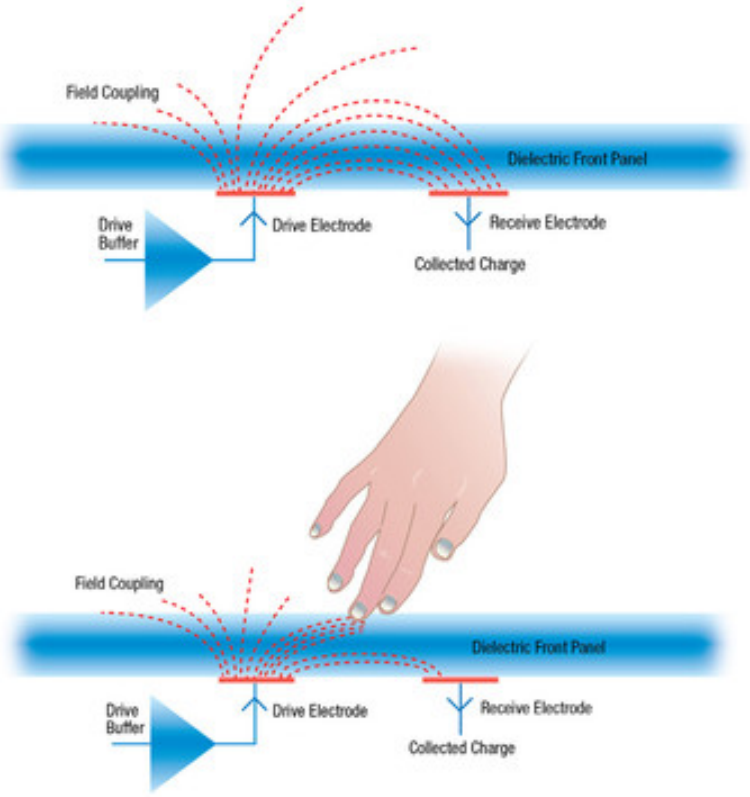

Abbildung 8: Projiziert Kapazitive Technologie [**?**,]

Die projiziert- kapazitive Touchscreen-Technologie erweitert die Oberflächen-Kapazitiv-Technologie um einzelne kleinere Felder auf dem Touchbildschirm und ist die aktuellste der genannten Technologien. Der wesentliche Vorteil gegenüber der Oberflächen-Kapazitiven-Technologie ist, dass hier schon die Näherung eines Fingers ausreicht, um eine Berührung zu registrieren. Dadurch ist es möglich selbst durch dünne Handschuhe eine Berührung zu registrieren, was einen klassischen Einsatzkontext für den medizinischen Bereich darstellt. Die Abbildung zeigt, wie allein durch die Annäherung eines Fingers Teilbereiche des Feldes durch natürliche Ladung in den Fingern abgelenkt werden.

3 Gestenerkennung und Steuerung von Funktionalität durch Gesten

3.1 Signalverarbeitung und erste Gesteninterpretation durch das Android OS

Die Auflistung der einzelnen Technologien zum Touchbildschirm hat gezeigt, wie unterschiedlich die Umsetzungen der Berührungserkennung und deren Leistungsfähigkeit in Bezug auf die Erkennung von einfachen und mehrfachen Berührungen zur selben Zeit sind. Das Android Betriebssystem ist darauf ausgelegt, die unterschiedlichen Formen von Touchbildschirmen zu unterstützen. Je nach verbauter Technologie ist dabei die Erkennung von mehreren Berührungspunkten freigeschaltet oder nicht. Für den Entwickler einer Android Applikation ist dies jedoch völlig transparent. Im Falle einer registrierten Berührung wirft die API ein sogenanntes „MotionEvent“. Die Android API leistet hier eine erste Interpretation der Gesten, denn je nach Art der Berührung wird ein bestimmtes „MotionEvent“ gefeuert. Diese Events können abgefangen werden und liefern Berührungsparameter (Koordinaten) für eine weitere, detailliertere Interpretation der Gesten. Der folgende Abschnitt beschreibt die prototypische Implementierung exemplarischer Gesten und geht neben den Besonderheiten und Schwierigkeiten der Gesteninterpretation auch auf die durch die Android API angebotenen MotionEvents ein.

3.2 Erkennung und Interpretation von Gesten sowie Steuerung jeweiliger Funktionalität durch Gesten

Für die Erkennung von Gesten bietet das Android SDK die Schnittstelle OnTouchListener an. Darin lässt sich die Methode

```
boolean onTouch(View zoomView, MotionEvent anEvent){...}
```

überschreiben. Über den Parameter anEvent vom Typ MotionEvent können unterschiedliche Berührungsgesten erkannt werden. Die Methode OnTouchListener wird in Abhängigkeit von bestimmten Ereignissen aufgerufen. Hierzu ist zu einer View-Instanz die konkrete Implementierung des OnTouchListener Interfaces zu adressieren:

```
viewInstance.setOnTouchListener(touchListenerImpl);
```

Wenn der Touchsensor eine Berührung registriert, wird für das jeweilige Layout geprüft, welche darin enthaltenen Views die Methoden des OnTouchListeners überschreiben und das Event an die jeweilige Implementierung weitergeleitet. Des Weiteren ist dem Objekt vom Typ MotionEvent zu entnehmen, um welche Art von Berührung es sich handelt. Je nach Art und Weise der Berührung liefert die Maskierung der zur Event Aktion

```
anEvent.getActionMasked()
```

ein Ergebnis, welches durch ein Element der Menge aus den folgenden Konstanten beschrieben wird:

Menge M

{

```
int ACTION_DOWN = 0;
int ACTION_UP = 1;
int ACTION_MOVE = 2;
int ACTION_CANCEL = 3;
int ACTION_OUTSIDE = 4;
int ACTION_POINTER_DOWN = 5;
int ACTION_POINTER_UP = 6;
int ACTION_HOVER_MOVE = 7;
int ACTION_SCROLL = 8;
```

```
int ACTION_HOVER_ENTER = 9;
int ACTION_HOVER_EXIT = 10;
}
```

Für die Interpretation der Gesten und weiter die Steuerung exemplarischer Funktionalitäten wie Zoom oder Drag gilt dann:

ACTION_DOWN: Der Touchsensor registriert eine Berührung in einem in sich geschlossenen Bereich.
Beispiel: Die Fingerkuppe eines Nutzers berührt den Touchscreen. Die Koordinaten erhält der Entwickler durch das Auslesen des Event-Parameters. Über die Methoden

`anEvent.getX()` und
`anEvent.getY()`

können die Koordinaten ausgelesen werden, auf denen sich ein Finger auf dem Touchbildschirm befindet. Diese Methoden geben im Fall von mehrerer Berührungen (z.B. mehrere Finger) immer die Koordinaten des ersten Berührungspunktes an.

ACTION_UP: Der Touchsensor registriert das Aufheben einer ursprünglich registrierten Berührung. Dies setzt zuvor die Registrierung einer ACTION_DOWN voraus.

ACTION_POINTER_DOWN: Der Touchsensor registriert eine zweite Berührung (zweiter Finger). Die Koordinaten der zweiten Berührung, sowie auch im Falle weiterer Berührungspunkte, lassen sich aus dem Event entsprechend der Reihenfolge der Berührung ermitteln:

```
anEvent.getX(0..n);
anEvent.getY(0..n);
```

Dies entspricht der Berührung mit dem ersten bis N-ten Finger. Für die Funktionalität Zoom ist es notwendig, die initiale Distanz zwischen den zwei Berührungspunkten zu ermitteln und temporär vorzuhalten.

ACTION_MOVE: Der Touchsensor registriert eine Bewegung weg vom ursprünglich registrierten Berührungspunkt, dessen Ermittlung diesem Event vorausgeht. Um die o.g. Funktionalitäten zu realisieren,

wird für die Ein-Finger-Geste "Drag"der Richtungsvektor bei einer Bewegung auf dem Touchbildschirm ermittelt, indem vom initialen Berührungsspunkt bis zur aktuellen Position des Fingers die Differenz in x wie in y Richtung errechnet wird.

3.3 Gestengesteuertes Vergrößern und Verkleinern sowie Veränderung des Fokus für Darstellungsobjekte vom Typ ImageView

Ian F. Darwin liefert mit seinem Kochbuch für die Android Entwicklung zu der Zoomfunktionalität folgenden Ansatz:

```
// Remember some things for zooming
PointF start = new PointF();

ImageView view = (ImageView) v;
// make the image scalable as a matrix
view.setScaleType(ImageView.ScaleType.MATRIX);
...
switch (event.getAction() & MotionEvent.ACTION_MASK) {
case MotionEvent.ACTION_DOWN: //first finger down only
start.set(event.getX(), event.getY());
break;
...
case MotionEvent.ACTION_MOVE:
if (mode == DRAG)
matrix.postTranslate(event.getX() - start.x, event.getY() -
start.y);

else if (mode == ZOOM)
matrix.postScale(scale, scale, mid.x, mid.y);

// Perform the transformation
view.setImageMatrix(matrix);
```

}
[?,]
Diese Auszüge aus Darwins Kochbuch zur Andorid Entwicklung zeigen schematisch die dynamische Skalierung einer ImageView in Abhängigkeit der erkannten Gesten mit Hilfe einer speziell für diesen Objekttyp anpassbaren Matrix. Die Matrix lässt sich direkt aus einer Objektinstanz vom Typ ImageView holen und mittels der Methoden

- `postTranslate()` und
- `postScale()`

verarbeiten. Dabei ist zu beachten, dass die Matrix nicht direkt verändert werden darf, sondern nur deren Kopie. Der Richtungsvektor für die neue Fokussierung und damit Transformation der Matix errechnet sich aus der Differenz der aktuellen Koordinaten zu den Anfangskoordinaten und wird zur Transformation der Matrix entsprechend übergeben:

```
matrix.postTranslate(event.getX() - start.x,
        event.getY() - start.y)
```

Im Fall von mehreren registrierten Berührungen, wie z.B. bei der Pinch-Open-Geste für die Ansteuerung einer Zoom-Funktionalität, muss zunächst die Anfangsdistanz zwischen den Berührungspunkten auf dem Touchbildschirm ermittelt werden. In einem Koordinatensystem aus zwei Dimensionen (x|y) wird die Länge einer Strecke von P1(xp1|yp1) und P2(xp2|yp2) wie folgt berechnet: $\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$

In Darwins Implementierung ist dies in der Methode `spacing()` realisiert. Weiter ist über das gesamte ACTION_MOVE Event hinweg, die neue Distanz der Berührungspunkte zu berechnen und mit der initialen Distanz in ein Verhältnis zu setzen. Der Quotient aus

$$ScaleFaktor = \frac{dist_{new}}{dist_{old}}$$

ergibt den Faktor für die Skalierung der ImageView. Die ImageView wird daraufhin entsprechend relativ zu der Positionsänderung der Berührungspunkte neu skaliert.

```
matrix.postScale(scale, scale, mid.x, mid.y);
```

Darwins Vorgehensweise stellt eine komfortable und einfache Möglichkeit dar, eine ImageView Instanz dynamisch zu skalieren oder den Fokus der Darstellung zu verändern. Jedoch bezieht sich die Lösung allein auf Darstellungselemente vom Typ ImageView, wohingegen die Superklasse View, die durch ImageView erweitert wird, es nicht vorsieht die Matrix zur Skalierung und Positionierung der View direkt zu verändern. Damit ist Darwins Lösung allein für Darstellungsobjekte vom Typ ImageView geeignet.

3.4 Abstraktion gestengesteuerter Funktionalitäten zum Skalieren und Fokussieren von Darstellungselementen

Dieser Abschnitt beschäftigt sich mit der Frage, inwieweit sich der Ansatz von Darwin generalisieren lässt und somit unabhängiger vom Typ des Darstellungsobjektes wird. Kann hierzu eine Lösung gefunden werden, so ist weitergehend zu diskutieren, inwieweit diese für andere Android App Projekte verfügbar gemacht werden kann.

3.4.1 Generalisierung der Funktionalitäten zum Skalieren und Fokussieren von Darstellungselementen

Die skizzierte Implementierung nach I.F. Darwin im letzten Abschnitt zeigt zwar, wie ein bestimmtes Darstellungselement in Abhängigkeit von bestimmten Gesten skaliert, jedoch gilt diese Lösung nicht für andere Darstellungselemente. Ein generischer Ansatz müsste die Skalierung und Fokussierung für alle Darstellungselemente leisten, mindestens jedoch für alle Elemente, die vom Typ android.view.View ableiten.
Der im Folgenden skizzierte Prototyp soll Darwins Ansatz um eine dynamische Anzahl von Darstellungselementen mit der Kompatibilität zu unterschiedlichen Darstellungstypen erweitern. Dieser Projektarbeit liegt

eine prototypische Implementierung bei (Klasse `Touch.java`), in der das Skalieren und Fokussieren von Darstellungselementen für allgemein alle Objekte ermöglicht wird, die vom Typ `android.view.View` ableiten (z.B. `TextView extends View`). Ein entsprechender Algorithmus könnte damit die Funktionalität zur Skalierung und Fokussierung applikationsübergreifend generalisieren. Für die Skalierung eines Darstellungsobjektes vom Typ View bietet die Android API folgende Methoden an:

- `setScaleX()`
- `setScaleY()`

und für die Veränderung des Fokus auf die View:

- `setTranslationX()`
- `setTranslationY()`

Ein dabei auftretendes Problem sind unerwünschte Seiteneffekte bei der Verwendung des Listeners zur Registrierung von Berührungen und Bewegungen. Dieser Listener kann einer View zugeordnet werden:

```
viewInstance.setOnTouchListener(touchListenerInstance)
```

Wird jedoch dieselbe View-Instanz innerhalb der TouchListener-Implementierung neu skaliert, kommt es auf Grund der angestoßenen Zoom-Funktionalität zu einem flackernden Übergang von der ursprünglichen Skalierung in die neue Skalierung. Dies liegt daran, dass das Koordinatensystem der View durch die Skalierung unmittelbar selbst verändert wird. Stellt diese View jedoch auch das Koordinatensystem für die Gestenerkennung, (`viewInstance.setOnTouchListener(...)`), so wird die Berechnung des Zoomfaktors gestört.
Die View, welche durch den Zoom neu skaliert wird, darf also im Falle einer gestengetriebenen Zoom-Funktionalität nie die Bemessungsgrundlage (Koordinaten) für die Berechnung des Skalierungsfaktors sein. Alternativ kann die sogenannte Parent-View der zu skalierenden View-Instanz als Bemessungsgrundlage herangezogen werden, wie es die folgende Abbildung zeigt:

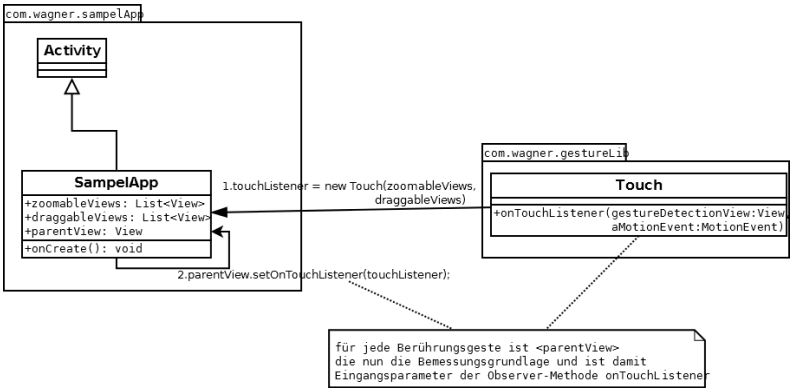

Abbildung 9: Konzeption eines generischen Touchlisteners

Die Parent-View ist in der Baumstruktur des Layouts einer View-Instanz übergeordnet.

3.4.2 Ausblick zur Integration der generischen Implementierung zu gestengesteuerten Funktionalitäten

Der letzte Abschnitt hat gezeigt, wie gestengesteuerte Funktionalitäten zum Skalieren und Fokussieren von Darstellungselementen, allgemein für unterschiedliche Darstellungstypen definiert werden können. Dieser Abschnitt gibt einen Ausblick, wie der entworfene Ansatz aus dem letzten Kapitel allgemein für Android Applikationsprojekte angeboten werden kann.
In Android Projekten lassen sich wie auch in anderen Java Projekten zusätzliche Bibliotheken in Form von .jar - Archiv Dateien einbinden. Grundsätzlich ist es möglich diese Bibliotheken manuell in die jeweilige Entwicklungsumgebung einzufügen. Sollten jedoch unterschiedliche Versionen zu den jeweiligen Android APIs unterschieden werden sind bestimmte Werkzeuge zur Versionverwaltung und dem übersichtlichen Management von Abhängigkeiten zu empfehlen. Auch wenn Tests zu Funktionalitäten der Bibliothek vor dem Erzeugen des .jar - Archives durchlaufen werden müssen, sollte dies automatisiert sein. Hierzu gibt es unterschiedliche Werkzeuge, die sich für das Auflösen von Abhängigkeiten sowie das finale Bauen der Applikation anbieten. Exemplarisch bietet sich hier das Build

Werkzeug „Maven“mit dem Android Plugin an. Der im letzten Kapitel entworfene Ansatz lässt sich damit in eine eigenständige Bibliothek auslagern, um diesen in unterschiedlichen Android-Applikationsprojekten zu nutzen.

3.5 Fazit

Eingangs stand die Frage im Raum, wie den Einstiegshürden in der Android-Entwicklung sowie dem Problem des Code-Copy entgegen gewirkt werden kann. Anhand der Entwicklung von gestengesteuerter Funktionalität (speziell hier Zoom und Drag) ist verdeutlicht worden, welche Schwierigkeiten auf Einsteiger in der Android Entwicklung warten. Gleichzeitig wird mit der Implementierung nach Darwin (Kapitel 3.3) deutlich, welche Mengen an komplexem Sourcecode zu gestengesteuerter Funktionalität (siehe Implementierung nach F.Darwin Kapitel 3.3) in Applikationsprojekte integriert werden müsste, um grundlegendes Verhalten zu definieren. Der in diesem Projekt geschaffene Ansatz zur Generalisierung von gestengesteuerter Funktionalität bietet die Möglichkeit den dahinterstehenden Programmcode zu zentralisieren, also Code Copy vorzubeugen und gleichzeitig die Android App-Entwicklung zu vereinfachen. Für weitere gestengesteuerte Funktionalitäten kann ähnlich verfahren werden, vorausgesetzt es kann ein geeigneter Abstraktionsgrad gefunden werden, wie er hier für die Skalierung und Fokussierung definiert ist.

Anhang

Listing 1: Die Klasse Touch.java

```
package com.wagner.android.gesturelib.gestureutils;

import android.graphics.PointF;
import android.util.FloatMath;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;

import java.util.List;

/**
 * This Class offers a generic Zoom and Drag
 * functionality that is controlled by using
 * Touch Gestures
 *
 * @author Stephan Wagner
 * @version 1.1.1.5
 *          Time: 22:06
 */
public class Touch implements View.OnTouchListener {

/**
 * The List of views that will be scaled
 * by using touch gestures.
 */
private List<View> transformableViews;

/**
 * The Tag for logging to this Listener.
 */
private final String TAG = getClass().getName();

/**
 * The initial coordinates of the first touch.
 */
private PointF start;
```

```
/**
 * The flag for defining the touch mode.
 */
private int mode;

/**
 *  Touch mode as functionality selector.
 */
private enum Mode{NONE, DRAG, ZOOM}

/**
 * The initial Distance for the
 * calculation of zoom.
 */
private float initDist;

/**
 * The initial Distance for the
 * calculation of zoom.
 */
private float newDist;

/**
 * The current scale factor of the zoom.
 */
private float scaleFactor;

/**
 * The maximal scale factor;
 */
private float maxScale;

/**
 * The minimal scale factor;
 */
private float minScale;

/**
 * The latency of the scale. Makes the zoom slower.
 */
private float latency;

/**
 * The Constructor of the touchListener
```

```
 * Implementation sets the initial not null parameters.
 *
 * @param aTransformableViewList The list of View
 *                               Instances that should
 *                               be transformed.
 * @param aMaxScaleFactor        the maximal scale factor.
 * @param aMinScaleFactor        the minimal scale factor.
 * @param aLatency               the latency for the
 *                               transformation, that is
 *                               used to make it slower.
 */
public Touch(final List<View> aTransformableViewList,
             final float aMaxScaleFactor,
             final float aMinScaleFactor,
             final float aLatency) {

    transformableViews = aTransformableViewList;
    start = new PointF();

    //Setting initial Values
    mode = Mode.NONE.ordinal();
    initDist = 1;
    newDist = 1;
    scaleFactor = 1;

    //setting scale
    maxScale = aMaxScaleFactor;
    minScale = aMinScaleFactor;

    //setting latency
    latency = aLatency;
}

/**
 * The implementation of TouchListener Interface,
 * that realizes drag and zoom functionality controlled
 * by multi-touch-gestures
 * @param touchView gives the base for the
 *                  registration of each touch.
 * @param anEvent holds the type and the
 *                coordinates of each touch.
 * @return returns true if listener has consumed the event.
 */
@Override
```

```
public boolean onTouch(final View touchView,
                       final MotionEvent anEvent) {

switch (anEvent.getActionMasked()) {
//first finger down only
case MotionEvent.ACTION_DOWN:
    //getting position
    start.set(anEvent.getX(), anEvent.getY());

    Log.d(TAG, "mode=DRAG");
    mode = Mode.DRAG.ordinal();
    break;

//first finger lifted
case MotionEvent.ACTION_UP:
    break;

//second finger lifted
case MotionEvent.ACTION_POINTER_UP:
    //new initial Distance is the last calculated distance
    //e.g. for more than one following pinch open gestures
    //and the connected zoom in function.
    initDist = newDist;

    mode = Mode.NONE.ordinal();
    Log.d(TAG, "mode=NONE");
    break;

//second finger down
case MotionEvent.ACTION_POINTER_DOWN:
    initDist = spacing(anEvent);
    mode = Mode.ZOOM.ordinal();
    Log.d(TAG, "mode=ZOOM");
    break;

case MotionEvent.ACTION_MOVE:
    if (mode == Mode.DRAG.ordinal()) {
    //movement of first finger

    float newX = anEvent.getX();
    float newY = anEvent.getY();

    float distanceX = (start.x - newX) / latency;
    float distanceY = (start.y - newY) / latency;
```

```
    for (final View draggableView : transformableViews) {
        draggableView.setTranslationX(
                draggableView.getTranslationX() - distanceX);
        draggableView.setTranslationY(
                draggableView.getTranslationY() - distanceY);
    }

    } else if (mode == Mode.ZOOM.ordinal()) { //pinch zooming
    float newDist = spacing(anEvent);

    if (newDist > initDist || newDist < initDist) {
        float factor = newDist / initDist;

        if (newDist > initDist && scaleFactor < maxScale)
        //pinch open --> zoom in
        {
            factor = factor / latency; //latency
            scaleFactor = scaleFactor + factor;
        }
        if (newDist < initDist &&
                scaleFactor > minScale &&
                factor < scaleFactor)
        //pinch close --> zoom out
        {
            factor = factor / latency; //latency
            scaleFactor = scaleFactor - factor;
        }

        for (final View scalableView : transformableViews) {
            scalableView.setScaleX(scaleFactor);
            scalableView.setScaleY(scaleFactor);
        }

        }
    }
    break;
}


return true;
}
```

```
/**
 * Calculates the distance between first
 * two touch points on touch screen.
 *
 * @param anEvent that contains the coordinates
 *                of the touch points.
 * @return the distance as float.
 */
private float spacing(final MotionEvent anEvent) {
    final float x = anEvent.getX(0) - anEvent.getX(1);
    final float y = anEvent.getY(0) - anEvent.getY(1);
    return FloatMath.sqrt(x * x + y * y);
}


}
```

A Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, den 25. Oktober 2015

(Unterschrift)