ulm university universität ulm

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Bachelorarbeit
im Studiengang Software Engineering

Einsatz von OpenGL in AREA

vorgelegt von

Micha Weilbach

Januar 2016

Gutachter	Prof. Dr. Manfred Reichert
Betreuer:	Dr. Rüdiger Pryss
Matrikelnummer	791761

Kurzfassung

Zur Darstellung grafischer Objekte auf einem Display gibt es Spezifikationen wie OpenGL, die speziell für diese Aufgabe konzipiert sind. Diese Arbeit hat sich das Ziel gesetzt, die bereits implementierte Applikation AREA zu überarbeiten und mithilfe von OpenGL umzusetzen. Im ersten Teil wird der Begriff des Augmented Reality erläutert sowie die Funktionalitäten von AREA aufgezeigt. Im zweiten Teil wird die Umsetzung von AREA durch das schrittweise Zusammensetzen der verschiedenen grafischen Komponenten, die auf dem Display angezeigt werden, geschildert. Wie mit OpenGL grafische Objekte erzeugt und auf dem Display dargestellt werden, wird in einem einleitendem Beispiel vorgestellt. Der Aufbau der verschiedenen grafischen Komponenten von AREA wird mithilfe von Skizzen veranschaulicht und die für das Generieren der Komponenten entwickelten Algorithmen erläutert. Durch komprimierte Listings mit Programmcode wird verdeutlicht, wie die verschiedenen grafischen Komponenten generiert und letztendlich auf dem Display dargestellt werden.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sinngemäße Übernahmen aus anderen Werken sind als solche kenntlich gemacht und mit genauer Quellenangabe (auch aus elektronischen Medien) versehen.

Ulm, den Name

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Zu einem guten Ende gehört auch ein guter Beginn.

Konfuzius, (551 - 479 v. Chr.)

1 Einleitung

Der heutige Lebensalltag wird immer häufiger von mobilen Applikationen begleitet. Laut einer Statistik der Statista GmbH wurden in Deutschland im Februar 2015 etwa 45,6 Mio. Smartphone-Nutzer registriert, 10% mehr als im Vorjahr [Gmb15]. Somit ist etwa jeder 2. Deutsche im Besitz eines Smartphones.
Durch einen andauernden Konkurrenzkampf unter den Smartmobilherstellern und einer weiter fortschreitenden Entwicklung der Hard- und Softwaretechnik, werden die angebotenen Smartmobilgeräte immer leistungsfähiger. Durch leistungsfähigere CPUs und GPUs können aufwendigere Applikationen mit höheren Systemanforderungen entwickelt werden. Die Möglichkeit, dass benutzerspezifisch kompilierte Programme auf der GPU ausgeführt werden können, bietet den Entwicklern wesentlich mehr Kontrolle und Flexibiltät über die grafische Funktionalität einer Applikation [BAH09].
Die Applikation AREA verwendet einige Sensoren, um die aktuelle Ausrichtung des verwendeten Mobilgerätes zu bestimmen und festzuhalten. Über GPS werden die Koordinaten des aktuellen Standorts ermittelt und anschließend über eine Internetverbindung Daten über sogenannte Point of Interests (POI) aus der Umgebung geladen. Wird ein Mobilgeräte ohne Touchscreen oder mit fehlender GPS- oder WLAN-Unterstützung verwendet, wäre die Verwendung von AREA gar nicht oder nur bedingt möglich.
In dieser Arbeit wird eine bereits implementierte Version von AREA für das Android Betriebssystem betrachtet. Als Ziel dieser Arbeit soll die Kameraansicht von AREA mithilfe von OpenGL ES 2.0 umgesetzt und optimiert werden. Die nachfolgende schriftliche Ausarbeitung ist so aufgebaut, dass die Kameraansicht Schritt für Schritt um eine mit OpenGL ES 2.0 umgesetzten Komponente erweitert wird. Den Anfang der Implementierung bildet das Zeichnen und Positionieren der POIs als Kreise auf der Kameraansicht. Im darauf folgenden Abschnitt wird der Aufbau und die Funktionalität des Radars beschrieben. Es werden die Problematiken des Textrendering in OpenGL ES 2.0 geschildert und die Idee der verwendeten Lösung erläutert. Zum Schluss werden die Interaktionsmöglichkeit über den Touchscreen und zwei Methoden zur Behandlung von Clusterbildung implementiert.
Zum besseren Verständnis werden in den ersten beiden Kapiteln die Grundlagen von Augmented Reality im Zusammenhang mit AREA geschildert sowie die Funktionsweise von OpenGL ES 2.0 anhand eines kleinen Tutorials erläutert.

Im Rahmen dieser Arbeit wurde ein Nexus 4 Smartphone ohne Sim Karte verwendet. Daten über POIs wurden ausschließlich über das Universitäts- und Heimnetzwerk empfangen. Der in dieser Arbeit angegebene Code wird hauptsächlich in Pseudocode angegeben und entspricht der in der Implementierung verwendeten Logik, kann aber vom tatsächlich implementierten Code abweichen.

2

Augmented Reality und AREA

AREA steht für Augmented Reality Engine Application und ist eine voll funktionstüchtige und anpassbare Funktionseinheit, welche eine erweiterte Realität auf Smartmobilegeräten unter Android oder Apple iOS ermöglicht [Ulm15]. Als Erweiterte Realität oder Augmented Reality wird eine computergestützte Wahrnehmung bezeichnet, bei der sich reale und virtuelle Welt vermischen [Gol15]. In der Augmented Reality werden visuelle Darstellungen mit weiteren Informationen wie z.B Text ergänzt. Als Beispiel könnten zwei, in der Realität nebeneinander stehende Bäume durch die Kamera eines Smarthphones betrachtet werden und eine virtuelle Verbindungslinie zwischen beiden Bäumen könnte zusätzliche Informationen über deren Abstand angeben (vgl. **Abbildung** 2.1). Dieses Beispiel würde auch als Augmented Reality bezeichnet werden, falls z.B nur die aktuelle Position in Form von Höhen- und Breitengraden dargestellt werden würde und beide Bäume oder weitere sichtbare Details unbeachtet bleiben würden.
Es gibt viele Formen, in welchen Augmented Reality auftritt und dargestellt werden kann. In AREA tritt Augmented Reality in der Form auf, dass, abhänig von der aktuellen Position, sogenannte Point of Interests graphisch dargestellt werden, welche sich in einem festgelegtem Radius um die aktuelle Position befinden [SPSR15] [GSP+14] [GPSR13]. Ein Point of Interest (kurz POI) ist ein Ort oder ein Gebäude und könnte für die Allgemeinheit von größerem Interesse sein. Gebäude zählen unter anderem als POI, falls sie namentlich benannt werden können, wie z.B eine Sehenswürdigkeit wie das Ulmer Münster, ein Firmen- oder ein Geschäftsgebäude. Anhand der GPS Koordinaten des Mobilgerätes werden umliegende POIs von der Google API ermittelt, gesammelt und an das Mobilgerät gesendet. Letztendlich entscheidet die Google API, was als POI bezeichnet werden darf.
In AREA gibt es zwei Ansichten, in welchen die naheliegenden POIs angezeigt werden. Beim Starten der Applikation wird die Kartenansicht geladen, in der der aktuelle Standort des Mobilgerätes durch einen Kreis und umliegende POIs durch ein Fähnchen markiert werden. Die in **Abbildung** 2.2 dargestellte Kartenansicht bietet dem Benutzer einen Überblick über alle POIs in der Umgebung. Der Benutzer des Mobilgerätes kann in dieser Ansicht über den Touchscreen durch die Kartenansicht navigieren, auf der Karte heran- oder herauszoomen und die Ansicht auf den aktuellen Standort zentrieren. Wird ein POI angeklickt, so wird der Name des POIs angezeigt und die Ansicht auf diesen zentriert.
Auf der Kartenansicht wird dem Benutzer die Möglichkeit geboten, auf die Kameraansicht zu wechseln. Die Kameraansicht stellt die Umgebung aus der Perspektive der Kamera des Mobilgerätes dar. In dieser Ansicht wird ein POI nur grafisch markiert, falls der POI sich im Blickfeld der Kamera befindet, unter der Annahme, dass der POI durch die Kamera erkennbar sein könnte, wenn alle anderen visuellen Objekte unsichtbar wären oder ignoriert werden würden. POIs werden in der Kameraansicht durch Kreise repräsentiert, die mit dem Namen

Abbildung 2.1: Augmented Reality - Abstand zwischen zwei realen Bäumen

des jeweiligen POI markiert sind und ihre Position auf dem Display mit der Bewegung des Mobilgerätes verändern. Wird die Kamera nach links gedreht, so bewegen sich die Kreise der POIs nach rechts, was analog für alle möglichen Bewegungen des Mobilgerätes gilt. Ziel der Kameraansicht von AREA ist es, einen 3-dimensionalen Raum zu erzeugen, in welchem das verwendete Mobilgeräte im Mittelpunkt steht und die geographische Position der POIs relativ genau wiedergegeben wird. Wird also mit der Kamera ein POI anvisiert, soll ein Kreis überlappend auf diesem angezeigt werden.
In der Kameraansicht werden alle umliegenden POIs in einem Radar angezeigt (vgl. **Abbildung** 2.3). Der Radar besteht aus einem kreisförmigen Körper, in welchem das Mobilgerät den Kreismittelpunkt darstellt. Alle umliegenden POIs werden mit relativ genauen geographischen Abständen zum Radarmittelpunkt auf dem Radar angezeigt. Zur Orientierung zeigt ein Pfeil, außerhalb des Radarkörpers, die Nordrichtung an. Die aktuelle Blickrichtung der Kamera des Mobilgerätes wird durch einen andersfarbigen, halbtransparenten Kreisausschnitt im Radar ersichtlich.
Ähnlich wie in der Kartenansicht, kann der Benutzer mit den angezeigten POIs interagieren, indem ein Kreis ausgewählt wird. Das Anwählen eines POIs in der Kameraansicht öffnet ein

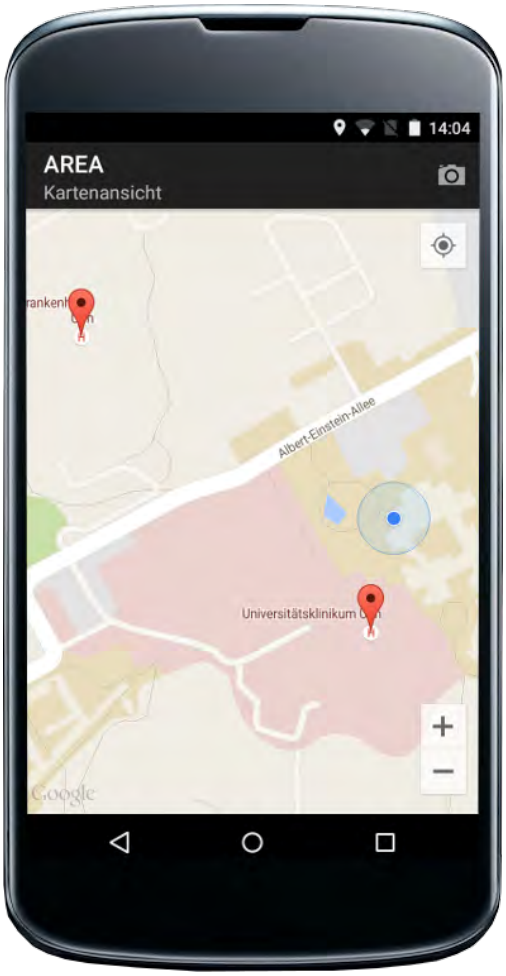

Abbildung 2.2: AREA-Kartenansicht

Informationsfenster, welches weitere Details über Namen, Höhenmeter, Distanz und Zusatzinformationen über den entsprechenden POI beinhaltet. Zusätzlich kann in der Kameraansicht der Umkreisradius festgelegt werden, in welchem POIs von der Google API geladen werden sollen. Dazu muss der Radar angewählt und in dem anschließend geöffnetem Distanzfenster der Schieberegler entsprechend verschoben werden.

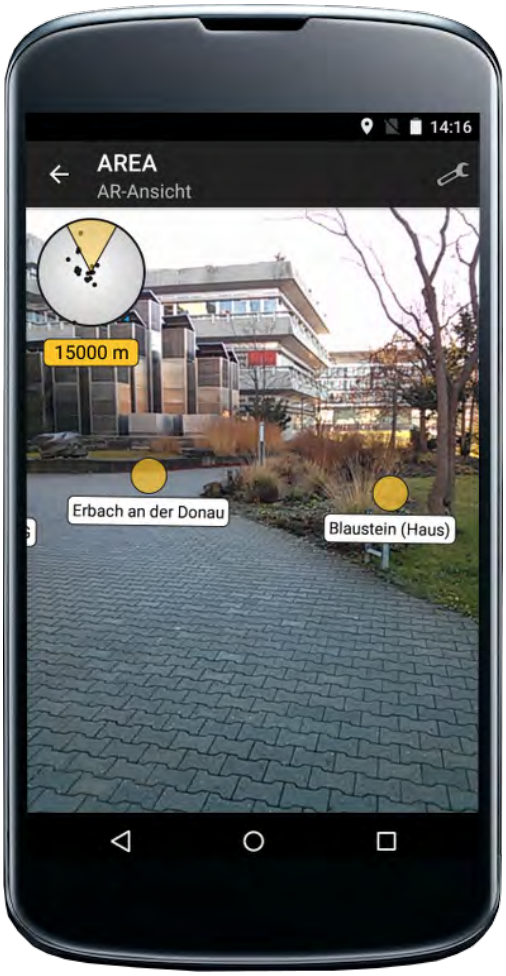

Abbildung 2.3: AREA-Kameraansicht

3
OpenGL ES 2.0

OpenGL steht für Open Graphics Library und ist eine low-level API, die dem Programmierer eine Schnittstelle zur Grafikhardware anbietet. Der wichtigste Vorteil von OpgenGL gegenüber anderen Graphik-APIs ist der, dass es auf vielen verschiedenen Plattformen wie Windows, Linux und Mac OSX funktioniert. OpenGL ES steht für Open Graphics Library for Embedded Systems und ermöglicht die Verwendung von OpenGL auf vielen Mobilgeräten, wie dem Iphone oder Android-Smartphones.
OpenGL wird für viele Arten von Anwendungen verwendet, von CAD Programmen bis hin zu Spielen wie Doom 3 und von wissenschaftlichen Simulationen bis hin zu 3D Modellierungsprogrammen [BAH09].
Dieses Kapitel soll als Einleitung in die Programmierung mit OpenGL ES 2.0 dienen und als Resultat ein farbig gezeichnetes Dreieck hervorbringen. Nachfolgende Einführung ist größtenteils von der Internetseite für Android Developer entnommen [Dev15b].

3.1 Form definieren

Um Formen mit OpenGL graphisch darstellen zu können, müssen diese Formen zuvor als eine Menge von Eckpunkten definiert werden. Dazu werden die Positionen der Eckpunkte, die eine Form bilden, als 3er-Tupel in einem Array gespeichert. Alle 3er-Tupel in einer Liste haben die gleiche Form *(x, y, z)* und geben die Position in einem 3-dimensionalen Koordinatensystem an. x gibt den x-Achsenwert, y den y-Achsenwert und z den z-Achsenwert an.
Das Koordinatensystem, welches in OpenGL ES verwendet wird, orientiert sich am Display des verwendeten Mobilgeräts, jedoch unabhängig vom Breiten:Höhen Verhältnis des Displays (vgl. **Abbildung** 3.1). Der Ursprung befindet sich in der Mitte des Displays, die Ecken haben die Wertigkeit von *1* bzw. *-1*, wodurch der sichtbare Wertebereich der x- und y-Achse des Koordinatensystems zwischen *-1* und *1* liegt [Dev15d]. Die z-Achse wird für eine 3D-Ansicht verwendet, da durch verschiedene z-Achsenwerte eine räumliche Darstellung leichter möglich ist.
Es bietet sich an, Formen innerhalb einer eigenen Klasse zu definieren. So können mehrere Instanzen aus diesen Klassen erzeugt werden. In einer solchen Klasse werden ein Bytebuffer, Farben und eine *draw()*-Methode definiert. Der Bytebuffer speichert alle Eckpunkte der Form in Byte ab. Je nachdem, ob die Form ein- oder mehrfarbig gezeichnet werden soll, muss ein weiterer Buffer für die Farbwerte definiert werden. Bei einfarbigen Formen reicht ein Array, welches Rot-, Grün-, Blau- und Alphawert dieser einen Farbe speichert. Bei mehrfarbigen Formen wird ein zusätzlicher Bytebuffer benötigt, welcher für jeden definierten Eckpunkt einen Farbwert

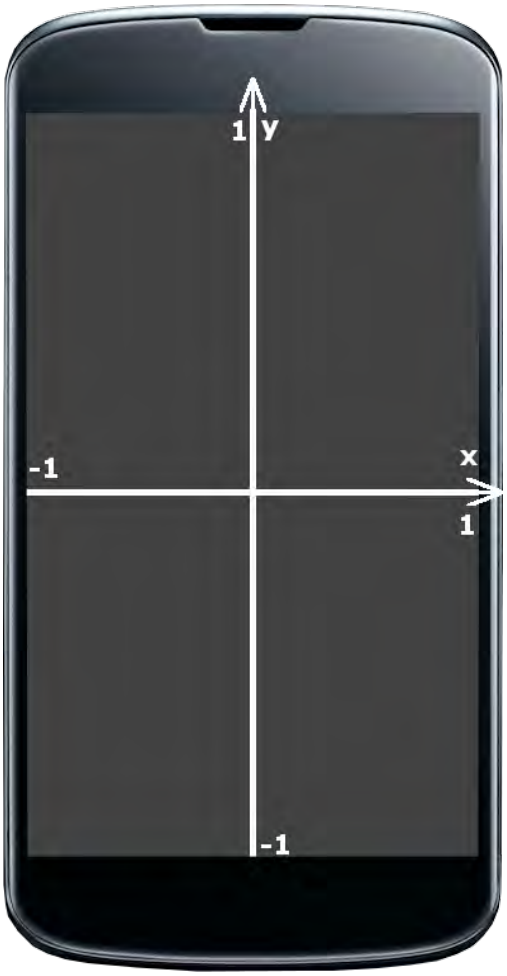

Abbildung 3.1: OpenGL ES Koordinatensystem

speichert. Die *draw()*-Methode wird von der Rendererklasse aufgerufen und bewirkt, dass die Form gezeichnet wird.

Listing 3.1: Klassendefinition eines Dreiecks mit festgelegter Position und Farbe

```
1  public class Triangle {
2      private FloatBuffer vertexBuffer;
3      static final int COORDS_PER_VERTEX = 3;
4      static float triangleCoords[] = {
5          0.0f, 0.622008459f, 0.0f,
6          -0.5f, -0.311004243f, 0.0f,
7          0.5f, -0.311004243f, 0.0f
8      };
9      float color[] = { 0.63671875f, 0.76953125f, 0.22265625f, 1.0f };
10
11     public Triangle(){
12         ByteBuffer bb = ByteBuffer.allocateDirect(
13             tringleCoords.length * 4);
```

```
14          bb.order(ByteOrder.nativeOrder());
15          vertexBuffer = bb.asFloatBuffer();
16          vertexBuffer.put(triangleCoords);
17          vertexBuffer.position(0);
18      }
19
20      public void draw(){
21          ...
22      }
23  }
```

Die Koordinaten des Dreiecks werden in *Zeile 4-7* gespeichert. Dass die Koordinaten jedes Eckpunktes als 3-Tupel gespeichert werden, wird durch die Variable *COORDS_PER_VERTEX* angegeben. In *Zeile 9* wird die Farbe festgelegt. Ein Floatwert von 1 würde dem RGB-Wert von 255 und ein Floatwert von 0 dem RGB-Wert von 0 entsprechen. Die in **Listing** 3.1 definierte Farbe entspricht umgerechnet etwa den RGB-Werten (162, 196, 57) - einem etwas hellerem Grün.
Im Konstruktor wird in *Zeile 12* der Bytebuffer erzeugt, der für jeden Eintrag im Array *trinangleCoords* 4 Byte reserviert. Die Reihenfolge, in der die Bytes gespeichert werden, wird in *Zeile 14* festgelegt [Ora15]. Aus dem erzeugten Bytebuffer wird ein Floatbuffer erzeugt, welcher mit den Koordinaten des Dreiecks gefüllt wird. in *Zeile 18* wird die Stelle im Floatbuffer festgelegt, an welcher der Lesevorgang beginnt.

3.2 Formen zeichnen

Um die im vorherigen Abschnitt definierte Form zeichnen zu können, wird in OpenGL ES eine Rendererklasse benötigt. Die Rendererklasse implementiert das Interface *GLSurfaceView.Renderer* und somit auch die Methoden *onDrawFrame()*, *onSurfaceChanged()* und *onSurfaceCreated()*. *onDrawFrame()* wird aufgerufen, um den aktuellen Frame zu zeichnen. *onSurfaceChanged()* wird aufgerufen, nachdem das Surface bzw. die Oberfläche, auf der mit OpenGL ES gezeichnet wird, erstellt wurde und wenn sich die Größe des Surface ändert. *onSurfaceCreated()* wird aufgerufen, wenn das Surface erstellt oder erneuert wurde [Dev15c].
Um das Dreieck zeichnen zu können, muss es vorher initialisiert werden.

Listing 3.2: Erzeugen einer Instanz der Dreiecksklasse

```
1  public class MyGLRenderer implements GLSurfaceView.Renderer {
2      ...
3      private Triangle mTriangle;
4
5      public void onSurfaceCreated(GL10 unused,
6          EGLConfig config) {
7          ...
8          mTriangle = new Triangle();
9      }
10     ...
11 }
```

Zum Zeichnen des Dreiecks müssen des Weiteren ein sogenannter *Vertex Shader*, ein *Fragment Shader* und ein *Program* definiert werden. Der Vertex Shader enthält OpenGL ES Shading Language Code(GLSL ES) und ist für die Transformation der Eckpunkte des Dreiecks verantwortlich. Der *Fragment Shader*, ebenfalls in GLSL ES geschrieben, übernimmt hingegen die Färbung und Texturierung der Fassade des Dreiecks.

Listing 3.3: Definition des Vertex Shaders und Fragment Shaders

```
1  public class Triangle{
2      private final String vertexShaderCode =
3          "attribute vec4 vPosition;" +
4          "void main() {" +
5          "  gl_Position = vPosition;" +
6          "}";
7
8      private final String fragmentShaderCode =
9          "precision mediump float;" +
10         "uniform vec4 vColor;" +
11         "void main() {" +
12         "  gl_FragColor = vColor;" +
13         "}";
14     ...
15 }
```

Die Positionen werden in *Zeile 3* als Attribute im *Vertex Shader* angegeben. *Zeile 9* gibt an, mit welcher Präzision Floatwerte kalkuliert werden, in diesem Beispiel also mit mittlerer Präzision. Durch *uniform* in *Zeile 10* wird *vColor* ein fester Wert zugewiesen. Eine *uniform* Variable kann vom Programm nur gelesen werden und hat für jeden verarbeiteten Eckpunkt den gleichen Wert [Clo15]. Da die Farbe des Dreiecks mit einer *uniform* Variablen festgelegt wird, werden alle Eckpunkte des Dreiecks mit dieser Farbe gezeichnet.
Um den in beiden Shaderstrings enthaltenen Code verwenden zu können, muss er kompiliert werden. Eine entsprechende Kompilermethode wird innerhalb der Rendererklasse definiert.

Listing 3.4: Kompilermethode zum kompilieren des Shadercodes

```
1 public static int loadShader(int type, String shaderCode){
2     int shader = GLES20.glCreateShader(type);
3     GLES20.glShaderSource(shader, shaderCode);
4     GLES20.glCompileShader(shader);
5
6     return shader;
7 }
```

Abhängig vom übergebenen Methodenparameter *type*, wird in *Zeile 2* ein Fragment oder Vertex Shader erzeugt. Der erzeugte Shader wird mit entsprechendem Shadercode kombiniert (*Zeile 3*), anschließend kompiliert (*Zeile 4*) und als Ergebnis zurückgeliefert. Um das Dreieck mit Shadern zeichnen zu können, muss der kompilierte Shadercode einem OpenGL ES Program Objekt hinzugefügt und das Program Objekt verlinkt werden. In dieser Einführung wird das Program Objekt im Konstruktor der Dreiecksklasse erzeugt.

Listing 3.5: Erzeugen und Verlinken eines OpenGL ES Program Objekts

```
1  public class Triangle(){
2      ...
3      private final int mProgram;
4
5      public Triangle(){
6          ...
7          int vertexShader = MyGLRenderer.loadShader(
8              GLES20.GL_VERTEX_SHADER, vertexShaderCode);
9          int fragmentShader = MyGLRenderer.loadShader(
10             GLES20.GL_FRAGMENT_SHADER, fragmentShaderCode);
11
12         mProgram = GLES20.glCreateProgram();
13         GLES20.glAttachShader(mProgram, vertexShader);
14         GLES20.glAttachShader(mProgram, fragmentShader);
15         GLES20.glLinkProgram(mProgram);
16     }
17 }
```

In *Zeile 7-10* werden die Shadercodes kompiliert und gespeichert. Ein leeres OpenGL ES Program Objekt wird initialisiert (*Zeile 12*) und die kompilierten Shader in das Program Objekt integriert (*Zeile 13,14*). Anschließend wird das Program Objekt verlinkt (*Zeile 15*).
Jetzt sind alle vorausgesetzten Komponenten vorhanden, um die *draw()* Methode zu definieren. In der *draw()* Methode werden die Position und Farbwerte des Dreiecks aus dem Vertex und dem Fragment Shader festgelegt und anschließend die Zeichnenfunktion ausgeführt.

Listing 3.6: Definition der draw-Methode zum Zeichnen eines Objekts

```
1  private int mPositionHandle;
2  private int mColorHandle;
3  private final int vertexCount = triangleCoords.length /
4     COORDS_PER_VERTEX;
5  private final int vertexStride = COORDS_PER_VERTEX * 4;
6
7  public void draw() {
8      GLES20.glUseProgram(mProgram);
9      mPositionHandle = GLES20.glGetAttribLocation(mProgram,
10         "vPosition");
11     GLES20.glEnableVertexAttribArray(mPositionHandle);
12     GLES20.glVertexAttribPointer(mPositionHandle,
13         COORDS_PER_VERTEX, GLES20.GL_FLOAT, false,
14         vertexStride, vertexBuffer);
15     mColorHandle = GLES20.glGetUniformLocation(mProgram,
16         "vColor");
17     GLES20.glUniform4fv(mColorHandle, 1, color, 0);
18     GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0,
19         vertexCount);
20     GLES20.glDisableVertexAttribArray(mPositionHandle);
```

```
21 }
```

Die Variable *vertexCount* in *Zeile 3* enthält die Anzahl der Eckpunkte, die gezeichnet werden sollen. Die Variable *vertexStride* in *Zeile 5* gibt an, wieviele Bytes für jeden Eckpunkt reserviert sind. In *Zeile 8* wird das zu verwendende OpenGL ES program Objekt ausgewählt. In *Zeile 9-14* werden die Positionsdaten und in *Zeile 15-17* die Farbe zum Zeichnen vorbereitet. Mit *Zeile 18+19* wird der Zeichenvorgang gestartet. Dabei gibt *GLES20.GL_TRIANGLES* den Zeichenmodus an, in diesem Beispiel werden immer nur Dreiecke, also drei Punkte mit Verbindungskanten gezeichnet und anschließend der Flächeninhalt mit entsprechender Farbe gefüllt. Der zweite Funktionsparameter gibt den Startindex des Arrays an, welches die zu zeichnenenden Eckpunkte enthält. Der letzte Parameter, hier *vertexCount*, gibt an, wieviele Eckpunkte insgesamt gezeichnet werden.
Damit das Dreieck auf der Benutzeroberfläche des Mobilgerätes dargestellt wird, muss die in **Listing** 3.6 definierte *draw()* Methode in der Rendererklasse aufgerufen werden. Dieser Aufruf muss in der *onDrawFrame()* Methode der Rendererklasse erfolgen.

Listing 3.7: Zeichnenbefehl an die Dreiecksklasse

```
1 public void onDrawFrame(GL10 unused){
2           ...
3           mTriangle.draw();
4 }
```

Als Ergebnis sollte ein grünes Dreieck auf der Benutzeroberfläche des Mobilgerätes angezeigt werden.
Diese Einführung in OpenGL ES 2.0 stellt eine Möglichkeit vor, wie ein Dreieck mit OpenGL ES 2.0 gezeichnet und dargestellt werden kann. Es gibt aber noch weitere Möglichkeiten, die ähnliche oder gleiche Resultate erzielen. Da die Umsetzung von AREA mit OpenGL ES 2.0 auf die Vorgehensweise, wie sie in diesem Abschnitt vorgestellt wurde, aufbaut, wird auf die Vorstellung von weiteren Möglichkeiten verzichtet.

4

Umsetzung von AREA mit OpenGL

In der Vorgängerversion von AREA, die noch kein OpenGL ES benutzt hat, werden die POIs und der Radar als Views generiert und dargestellt. Graphische Komponenten, wie Kreise, werden durch das Paint Interface gezeichnet und gefärbt. In diesem Kapitel soll die Vorgehensweise geschildert werden, die zu einer überarbeiteten Version von AREA führt, in welcher alle Komponenten in der Kameraansicht mit OpenGL ES 2.0 realisiert werden. Dabei wird der Großteil der funktionalen Operationen, wie die Bestimmung der sichtbaren POIs, aus der Vorgängerversion übernommen. Es werden neue Klassen, welche die Formen eines POI und des Radars definieren, erzeugt und eine Rendererklasse erstellt, welche die sichbaren POIs und den Radar bei jedem neuen Sensorereignis neu rendert.

4.1 Point of Interest

Ein Point of Interest, kurz POI, steht als Repräsentant für einen Ort oder ein Gebäude, der bzw. das für die Allgemeinheit von größerem Interesse sein kann. In AREA soll die relativ genaue Position eines oder mehrerer POIs angezeigt werden. In der Kameraansicht von AREA werden POIs als Kreise angezeigt, mit denen der Benutzer interagieren kann, um weitere Informationen über dieses POI zu erhalten. Damit ein POI richtig angezeigt werden kann, sind mehrer Faktoren wichtig: Die aktuellen GPS-Koordinaten des Mobilgerätes, die GPS-Koordinaten des POI und die aktuelle Haltung und Positionierung des Mobilgerätes. Zeigt die Kamera des Mobilgerätes von einem POI weg, so soll dieser POI auch nicht auf der Kameraansicht angezeigt werden. Gleiches gilt, falls beispielsweise nur der Himmel mit der Kamera betrachtet wird.

4.1.1 Darstellung als Kreis

Ein Kreis ist eine komplexere Form als ein Dreieck und mit OpenGL ES 2.0 nicht ohne weiteres darstellbar. Am einfachsten lässt sich ein Kreis als eine Ansammlung von identischen Dreiecken realisieren, die alle den Kreismittelpunkt als Eckpunkt gemeinsam haben. Diese Art, einen Kreis zu bilden, führt dazu, dass bei einer geringen Anzahl an Dreiecken der Kreis nicht wirklich rund, sondern eckig wird (vgl. **Abbildung** 4.1). Andererseits führen mehr Dreiecke, ab einer bestimmten Anzahl, zu keiner sichtbar glatteren Kreisrundung. Außerdem benötigen mehr Dreiecke auch mehr Speicherkapazität und sind ineffizienter zu rendern. Deshalb muss eine Anzahl Dreiecke festgelegt werden, mit der nicht unnötig Speicherkapazität verbraucht und gleichzeitig eine Kreisform erkennbar ist. Da in der Kameraansicht die Kreise relativ klein dargestellt werden, führt eine Anzahl von 50 Dreiecken pro Kreis zu einem annehmbaren Ergebnis.

Da das manuelle Berechnen der x-, y- und z-Koordinaten der Eckpunkte aller Dreiecke zu

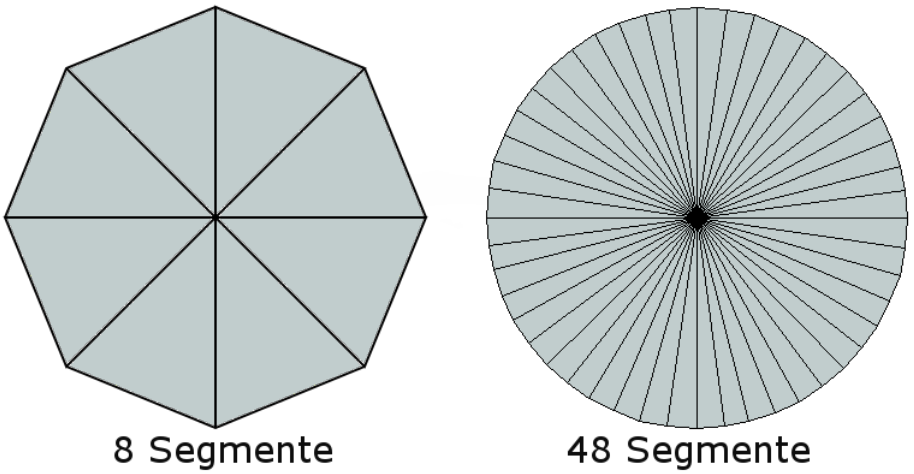

Abbildung 4.1: Kreise mit unterschiedlicher Anzahl Segmenten

aufwendig wäre, wird ein Algorithmus erstellt, der diesen Vorgang automatisiert. Hierzu gibt es wiederum mehrere Möglichkeiten, von denen nachfolgend zwei vorgestellt werden. In der ersten Variante werden die Koordinatenwerte mithilfe der Sinus- und Kosinusfunktionen berechnet. Die zweite Variante ist eine Optimierung der ersten Variante mit dem Ziel, die Anzahl der Sinus- und Kosinusoperationen und somit den benötigten Rechenaufwand zu minimieren [Pro15].

4.1.1.1 Standardalgorithmus zur Koordinatenberechnung

Der Standardalgorithmus berechnet die Koordinaten der Eckpunkte mithilfe der Sinus- und Kosinusfunktionen (vgl. **Abbildung** 4.2).

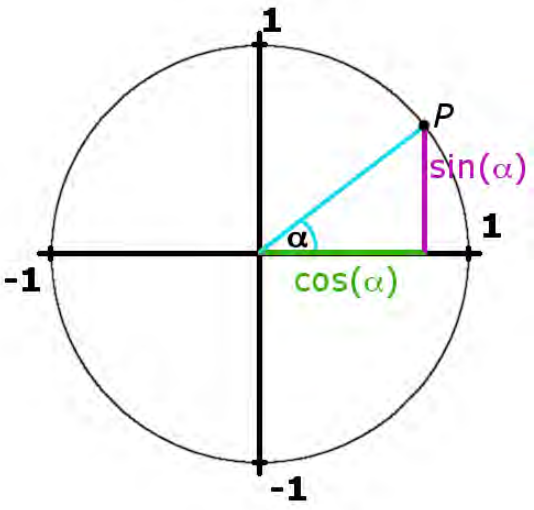

Abbildung 4.2: Berechnung der Kreiskoordinaten eines Einheitskreises

Alle Dreiecke, die den Kreis bilden, haben am Kreismittelpunkt den gleichen Winkel, welcher mit der Anzahl bereits durchgeführter Iterationen multipliziert wird und in die Sinus- und Kosinusfunktionen eingesetzt wird. Als Ergebnis erhält ein Eckpunkt einen x- und y-Achsenwert, die zusammen mit den Werten der vorherigen Iteration und dem Kreismittelpunkt ein Dreieck bilden. Für die erste Iteration müssen vor Schleifenbeginn die x- und y-Achsenwerte des Startpunktes angegeben werden.

Listing 4.1: Algorithmus zur Berechnung der Eckpunktkoordinaten als Pseudocode

```
1  public POI(float x, float y, float radius, int segments){
2          xOld = r;
3          yOld = 0.0f;
4          for(i = 1; i < segments; i++)
5                  α = 2π · i / segments;
6                  xNew = radius · cos(α);
7                  yNew = radius · sin(α);
8                  *Alte, Neue und Kreismittelpunktkoordinaten in
9                  *Array speichern
10                 xOld = xNew;
11                 yOld = yNew;
12         }
13         ...
14 }
```

Die Übergabeparameter x und y geben die Position des Kreismittelpunktes im OpenGL ES Koordinatensystem an. Da alle Kreise, sowohl die für POIs als auch die für den Radar, immer am Ursprung gezeichnet werden, sind die Übergabeparameter x und y unnötig, bieten aber die Möglichkeit für Erweiterungen und Optimierungen. Die Variablen *xOld* und *yOld* markieren in *Zeile 2+3* den Starteckpunkt. In *Zeile 5* wird der Winkel bzw. das Bogenmaß für die aktuelle Itaration berechnet. Der berechnete Wert für α wird für jede Iteration größer und beträgt in der letzten Iteration den Maximalwert 2π. Ausgehend von α werden die x- und y-Achsenwerte, welche die Koordinaten eines Eckpunkts festlegen, in *Zeile 6+7* berechnet. Der neu berechnete Eckpunkt bildet zusammen mit dem in der vorherigen Iteration berechneten Eckpunkt bzw. dem festgelegten Starteckpunkt und dem Koordinatenursprung ein Dreieck. Dieses Dreieck muss dem Array hinzugefügt werden, welches die x-, y- und z-Achsenwerte aller Eckpunkte beinhaltet und später zum Zeichnen des Kreises verwendet wird. Da die POIs nur 2-dimensional gezeichnet werden, wird der z-Achsenwert für alle Eckpunkte auf einen Wert von 0 festgelegt. Am Ende jeder Iteration müssen die neu berechneten Werte für die nächste Iteration vorbereitet werden, da diese einen Eckpunkt im nächsten Dreieck bilden (*Zeile 10+11*).

4.1.1.2 Optimierung des Standardalgorithmus

Im Standardalgorithmus werden pro Iterationsschritt die Sinus- und Kosinusfunktionen jeweils einmal aufgerufen. Aus Optimierungsgründen soll die Anzahl der Sinus- und Kosinus-Funktionsaufrufe minimiert werden. Für die Optimierung des Standardalgorithmus wird eine Idee aufgegriffen und in einer abgeänderten Form verwendet [Pro15].
Im Standardalgorithmus werden gleichschenklige Dreiecke betrachtet. Im optimierten Algorithmus werden rechtwinklige Dreiecke betrachtet, die weiterhin am Kreismittelpunkt den gleichen

Winkel innehaben. Zur Berechnung der Eckpunktkoordinaten, ragt die Hypothenuse aus der Kreislinie hinaus (vgl. **Abbildung** 4.3).

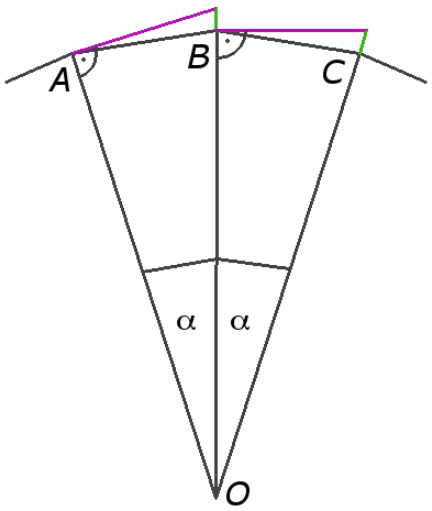

Abbildung 4.3: Eckpunktberechnung des optimierten Algorithmus

Die Idee hinter diesem Algorithmus ist es, die Eigenschaft, dass alle Dreiecke identisch und somit die Verbindungskante von Eckpunkt zu Eckpunkt immer gleichlang ist, auszunutzen. Dafür muss, wie im Standardalgorithmus, ein Starteckpunkt festgelegt werden. Die x- und y-Achsenwerte des Starteckpunkts bilden mit dem Kreismittelpunkt einen Vektor. Aus diesem Vektor wird ein Tangentenvektor gebildet, der mit einem vorher berechneten Tangentenfaktor gekürzt wird. Der berechnete Tangentenvektor wird mit dem Vektor vom Kreismittelpunkt zum Starteckpunkt addiert. Es entsteht ein neuer Vektor, der die Hypothenuse des ersten Dreiecks im Kreis bildet und den Eckpunkt, der berechnet werden soll, schneidet. Dieser Vektor, multipliziert mit einem vorher berechneten Radialfaktor, liefert die x- und y-Achsenwerte des nächsten Eckpunktes. Dieser Vorgang wird für alle weiteren Dreiecke wiederholt, bis wieder der Starteckpunkt erreicht ist.

Listing 4.2: Optimierungsalgorithmus zur Eckpunktkoordinatenberechnung als Pseudocode

```
1  public POI(float x, float y, float radius, int segments){
2          x = r;
3          y = 0.0f;
4          α = 2π / segments;
5          tangential_factor = tan(α);
6          radial_factor = cos(α);
7          for(i = 1; i < segments; i++)
8                  *x und y im Array speichern
9                  tx = -y;
10                 ty = x;
11
```

```
12                  x += tx * tangential_factor;
13                  y += ty * tangential_factor;
14
15                  x *= radial_factor;
16                  y *= radial_factor;
17                  *x, y und Kreismittelpunkt im Array speichern
18          }
19          ...
20  }
```

In *Zeile 2+3* werden die Koordinaten des Starteckpunkts festgelegt. α in *Zeile 4* ist der Winkel, welchen alle Dreiecke, die den Kreis bilden, am Kreismittelpunkt gemeinsam haben. Da für alle Dreiecke der Winkel α gilt, können die Tangential- und Radialfaktoren vorweg in *Zeile 5+6* festgelegt werden. Zu Beginn jeder Iteration werden die zuletzt berechneten x- und y-Werte in das Array gespeichert, welches später zum Zeichnen des Kreises verwendet wird. In *Zeile 9+10* wird der Tangentenvektor berechnet. Die Länge des berechneten Tangentenvektors entspricht dem Kreisradius und ist somit zu lang und muss gekürzt werden. Daher wird der Tangentenvektor mit dem Tangentialfaktor in *Zeile 12+13* multipliziert und anschließend mit dem ursprünglichen (x, y)-Vektor addiert. Es entsteht ein Vektor, der den Eckpunkt, der berechnet werden soll, schneidet. Die x- und y-Achsenwerte sind das Ergebnis aus der Multiplikation des Vektors mit dem Radialfaktor in *Zeile 15+16* und werden, zusammen mit den Kreismittelpunktkoordinaten, in das Array gespeichert. Pro Iterationsschritt werden die x-, y- und z-Koordinaten von zwei Eckpunkten und dem Kreismittelpunkt, die zusammen ein Dreieck innerhalb des Kreises bilden, in das Array gespeichert. Als z-Achsenwert wird, wie im Standardalgorithmus, für jeden Eckpunkt ein Wert von 0 festgelegt.
Durch die Optimierung konnten die Aufrufe der Sinus- und Kosinusfunktionen auf je einen Aufruf der Sinus- und Tangensfunktion minimiert werden. Da die Anzahl Iterationsschritte bei beiden Versionen gleich groß bleibt, ist eine Steigerung der Effizienz durch die optimierte Version zumindest nachvollziehbar.

4.1.2 Positionierung der POI

Bis jetzt werden die Kreise für die POIs nur in der Mitte des OpenGL ES Koordinatensystems gezeichnet und müssen noch verschoben werden, damit diese relativ genau auf die Position der POIs verweisen. Da AREA auf einem Mobilgerät ausgeführt wird, kann sich die Darstellung der Kameraansicht ständig ändern. Es müssen mehrere Faktoren zur Berechnung der Kreispositionen berücksichtigt werden: Die vertikale und horizontale Blickrichtung der Kamera des Mobilgerätes und die Haltung des Mobilgerätes selbst.
Für die Berechnung der horizontalen und vertikalen Positionen wird davon ausgegangen, dass das Mobilgeräte im Hochformat und parallel zum Boden gehalten wird. Um die Drehung des Mobilgerätes miteinzukalkulieren, wird der berechnete Kreis, entgegen der Drechrichtung des Mobilgerätes, um den Koordinatenursprung rotiert.

4.1.2.1 Horizontale Positionierung

Für die Berechnung der horizontalen Positionierung wird ein Kreis als Hilfsmittel verwendet. Der aktuelle Standort des Mobilgerätes bildet den Kreismittelpunkt. Ausgehend vom Kreismittelpunkt zeigt eine Achse die Nordrichtung an (vgl. **Abbildung** 4.4).

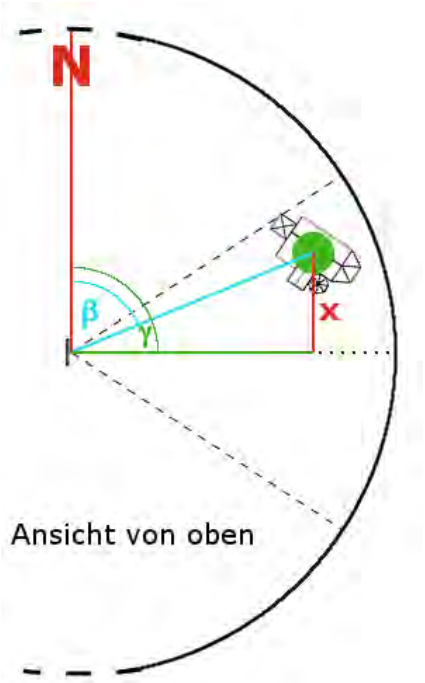

Abbildung 4.4: Berechnung der x-Achsenkoordinate mithilfe einer Ansicht von oben

Wichtig für die Berechnung sind die aktuelle Blickrichtung, ausgehend von der Mitte der Kameraansicht, die Richtung des POI, ausgehend vom Standort des Mobilgerätes, und das Blickfeldfenster der Kamera. Neigung und Drehung des Mobilgerätes werden hier noch nicht berücksichtigt. Die aktuelle Blickrichtung und die Position der POIs werden bereits in der Vorgängerversion von AREA durch entsprechende Funktionalität berechnet und können verwendet werden. Da sich die Breite des Blickfeldfensters je nach Drehung des Mobilgerätes ändert, kann kein fester Winkel für das Blickfeldfenster festgelegt werden. Stattdessen wird festgelegt, wieiviele Pixel 1° entsprechen. Ein Wert für die Anzahl Pixel pro 1° ist bereits in der Vorgängerversion festgelegt und berechnet sich aus der Displayhöhe des Mobilgerätes in Pixel dividiert durch 56°, was dem Winkel des Blickfeldfensters im Querformat entsprechen soll.
Die horizontale Positionierung im OpenGL ES Koordinatensystem lässt sich mit folgender Funktion berechnen.

Listing 4.3: Funktion zur Berechnung der x-Koordinate eines POI in Pseudocode

```
1 private float calcXCoord(POI){
2         poiDegree = POI.getHorizontalHeading();
3         xKoord = poiDegree - viewDegree;
4         xKoord /= pixelPerDegree;
5         return xKoord;
```

```
6 }
```

Für ein gegebenes POI wird dessen horizontale Ausrichtung angefordert (*Zeile 2*). Die Blickrichtung der Kamera ist mit der *public* Variablen *viewDegree* gegeben. Durch die Differenz in *Zeile 3* wird der Winkel von der Blickfeldmittelachse zum POI bestimmt. Das Ergebnis aus der Division des berechneten Winkels mit der Anzahl Pixel pro 1° in *Zeile 4* ist der x-Achsenwert des POIs im OpenGL ES Koordinatensystem. Ist der Wert größer 1 oder kleiner -1, so ist das POI zumindest nicht vollständig auf der Kameraansicht sichtbar.
Der in **Listing** 4.3 vorgestellte Algorithmus zur Berechnung des x-Achsenwertes ist in aktueller Form noch nicht korrekt. Beträgt die Blickrichtung der Kamera beispielsweise 1° und die horizontale Ausrichtung des POI 359°, so berechnet der Algorithmus einen Winkel von 358°. Folglich würde der POI fälschlicherweise nicht auf der Kameraansicht angezeigt werden. Als Lösung dieses Problems, wird der in **Listing** 4.3 *Zeile 3* berechnete Winkel mit 360° subtrahiert, falls dieser größer als 180° ist, oder mit 360° addiert, falls dieser kleiner als -180° ist. Der Wertebereich des Winkels liegt somit zwischen -180° und 180°.

Listing 4.4: Verbesserte Funktion zur Berechnung der x-Koordinate eines POI in Pseudocode

```
1  private float calcXCoord(POI){
2          poiDegree = POI.getHorizontalHeading();
3          xKoord = poiDegree - viewDegree;
4          if xKoord >= 180
5                  xKoord -= 360;
6          else if xKoord < -180
7                  xKoord += 360;
8          xKoord /= pixelPerDegree;
9          return xKoord;
10 }
```

4.1.2.2 Vertikale Positionierung

Für die Berechnung der vertikalen Positionierung, werden die Höhenmeter des POI, die Höhenmeter des aktuellen Mobilgerätstandorts, die Flugliniendistanz zwischen Standort und POI und die Neigung des Mobilgeräts verwendet. Aus der Differenz aus den Höhenmetern des POI und des Mobilgerätstandorts und der Distanz, lässt sich ein rechtwinkliges Dreieck skizzieren (vgl. **Abbildung** 4.5).
Da die Längen der An- und Gegenkathete bekannt sind, kann der Winkel α mithilfe der inversen Tangensfunktion berechnet werden. Der Winkel α wird dabei negativ, falls der POI höher als das Mobilgerät liegt.
Die Neigung des Mobilgeräts wird von einer bereits implementierten Funktion geliefert und wird auf den Winkel α addiert. Die Division von α mit der Anzahl Pixel pro 1° liefert als Ergebnis den gesuchten y-Achsenwert für das OpenGL ES Koordinatensystem.

Listing 4.5: Funktion zur Berechnung der y-Koordinate eines POI in Pseudocode

```
1  private float calcYCoord(POI){
2          poiAltitude = POI.getAltitude();
3          opposite = myAltitude - poiAltitude;
```

```
4            adjacent = POI.getDistance();
5            α = tan^-1(opposite/adjacent);
6            yKoord = α + viewYDegree;
7            yKoord /= pixelPerDegree;
8            return yKoord;
9    }
```

In *Zeile 2* werden die Höhenmeter des übergebenen POI angefordert. Die Variable *opposite* steht für die Strecke der Gegenkathete und bildet sich aus der Differenz aus den Höhenmetern des Mobilgerätes und des POI. Die Variable *adjacent* bestimmt die Länge der Ankathete, die der Entfernung zwischen Mobilgerät und POI entspricht. Der Winkel α lässt sich aus folgender Formel ableiten:

$$tan(\alpha) = \frac{Gegenkathete}{Ankathete} => \alpha = tan^{-1}(\frac{Gegenkathete}{Ankathete})$$

Der in *Zeile 5* berechnete Wert für α wäre korrekt, falls das Mobilgerät gerade und parallel zum Boden gehalten werden würde. In der Regel wird das Mobilgerät aber mit einer, zumindest minimalen, Neigung nach oben oder unten gehalten. In *Zeile 6* wird diese Neigung dem Winkel α hinzugerechnet. Das Ergebnis aus *Zeile 6*, dividiert mit der Anzahl Pixel pro 1° (*Zeile 7*), ergibt den gesuchten y-Achsenwert für das OpenGL ES Koordinatensystem.

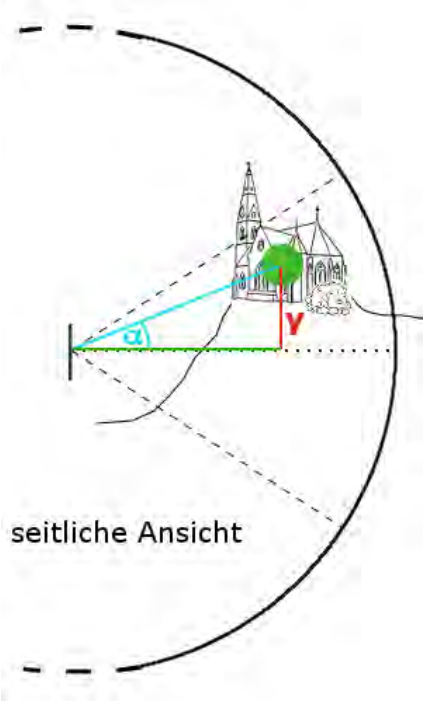

Abbildung 4.5: Berechnung der y-Achsenkoordinate mithilfe einer seitlichen Ansicht

4.1.2.3 Rotation des Mobilgerätes

Die x- und y-Achsenwerte werden zwar korrekt ausgerechnet, wird das Mobilgerät aber beispielsweise um 90° gedreht und somit vom Hoch- ins Querformat gewechselt, werden die Kreise trotzdem so angezeigt, als würde das Mobilgerät sich im Hochformat befinden. Folglich werde die POI nicht mehr richtig markiert.
Um die Drehung des Mobilgeräts zu berücksichtigen und die POI trotz Drehung richtig zu markieren, müssen die Kreise um den Ursprung des OpenGL ES Koordinatensystem rotieren. Der Winkel der Rotation entspricht dabei dem negierten Winkel, den das Mobilgerät von der Hochformathaltung abweicht. Der Winkel wird negiert, da die Kreise beispielsweise bei einer Linksdrehung des Mobilgerätes, diese Drehung mit einer Rechtsrotation ausgleichen müssen, um den POI weiterhin korrekt auf der Kameraansicht zu markieren (vgl. **Abbildung** 4.6).

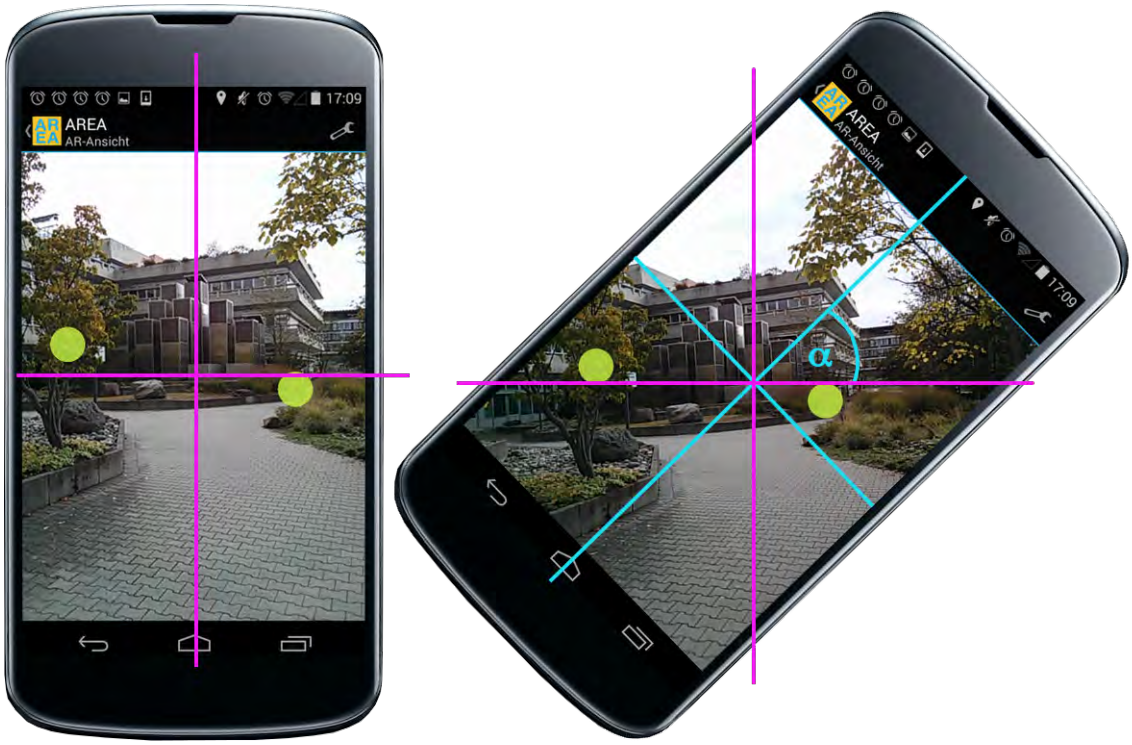

Abbildung 4.6: Der Rotationswinkel

Das verwendete Mobilgerät Nexus 4 von Android misst, bei einer Haltung des Mobilgeräts im Hochformat, einen Winkel von -90°. Somit beträgt der Winkel der Ausgangslage nicht, wie erwartet, 0°, sondern -90°. Die erhaltenen Sensordaten, bezüglich der aktuellen Drehung des Mobilgerätes, müssen entsprechend mit 90° addiert werden, um einen Winkel ausgehend von 0° zu erhalten. Der Wertebereich der von Sensoren registrierten Drehung liegt zwischen -180 und 180.

Listing 4.6: Funktion zur Berechnung der Mobilgerätrotation in Pseudocode

```
1 private float calcRotation(){
2         rotation = deviceRotation + 90;
3         if rotation > 180
```

```
4                   rotation -= 360;
5           return rotation
6   }
```

Mithilfe der berechneten Daten über die x- und y-Achsenwerte und die nötige Drehung, kann der erzeugte Kreis nun so gezeichnet werden, dass er zumindest die ungefähre Position eines POI markiert. Hierfür wird eine sogenannte Transformationsmatrix für den Kreis angelegt, die die Daten über die Positionierung des Kreises speichert. Der Kreis wird mit dem Koordinatenursprung als Mittelpunkt erzeugt, weshalb die Transformationsmatrix zu Beginn einer Nullmatrix entspricht. Wird die Transformationsmatrix auf eine Nullmatrix zurückgesetzt, bedeutet das, dass der Kreis seine ursprüngliche Position am Ursprung einnimmt. Damit der Kreis die gewünscht Position einnimmt, wird er zuerst um die berechnete Rotation des Mobilgerätes gedreht und anschließend um die berechneten x- und y-Werte verschoben.

Listing 4.7: Ausrichtung eines Kreises in Pseudocode

```
1  onDrawFrame(GL10 gl){
2           ...
3           POI p = new POI(0, 0, 0.1f, 50);
4           Matrix.setIdentityM(p.mTranslationMatrix, 0);
5           Matrix.setRotateM(p.mTranslationMatrix, 0,
6                   calcRotation(), 0, 0, 1);
7           Matrix.translateM(p.mTranslationMatrix, 0,
8                   calcXCoord(), calcYCoord(), 0);
9           Matrix.multiplyMM(scratch, 0, mMVPMatrix, 0,
10                  p.mTranslationMatrix, 0);
11          p.draw(scratch);
12          ...
13  }
```

In *Zeile 3* wird ein neuer Kreis *p* erzeugt und dessen Transformationsmatrix in *Zeile 4* auf eine Nullmatrix gesetzt. Die Transformationsmatrix wird in *Zeile 5+6* überarbeitet, indem der Kreis um einen bestimmten Winkel, der mit *calcRotation()* berechnet wird, rotiert wird. Dabei geben die letzten drei Parameter der Funktion *setRotateM()* an, um welche Achsen der Kreis rotiert werden soll, hier nur um die z-Achse, mit der Wertigkeit von *1* markiert.
In *Zeile 7+8* wird der Kreis um die mit den Funktionen *calcXCoord()* und *calcYCoord()* berechneten x- und y-Achsenwerte verschoben. Die überarbeitete Transformationsmatrix wird auf eine weitere Matrix *mMVPMatrix* multipliziert. Die Matrix *mMVPMatrix* enthält Informationen darüber, wie Objekte projeziert und aus einer virtuellen Sicht dargestellt werden müssen [Dev15a].
Anschließend wird der Kreis in *Zeile 11* gezeichnet und somit auf der Kameraansicht sichtbar. Kreise werden, einmal erzeugt, während der Laufzeit der Applikation gespeichert und nicht wie in **Listing** 4.7 bei jedem neuen Sensorenereignis neu erzeugt. Durch das Speichern der Kreise wird viel Rechenleistung eingespart, da so das aufwendige Berechnen der Eckpunkte jedes einzelnen Kreises nur einmal stattfindet und die bereits erzeugten Kreise wiederverwendet werden können. Es wäre denkbar, die Position von dargestellten Kreisen um einen bestimmten x- und y-Wert zu verschieben, wenn ein Sensorenereignis auftritt. Da Sensorenereignisse jedoch sehr häufig pro Sekunde auftreten und mit *float*-Werten, also Gleitkommazahlen mit begrenz-

ter Anzahl Nachkommastellen, gerechnet wird. Wirken sich bereits kleine Rundungsfehler sehr schnell und sichtbar aus, sodass Kreise schon nach kurzer Zeit nicht mehr die POI Positionen markieren.
Eine effektive Lösung, die das Rundungsfehlerproblem vermeiden soll, ist die, dass die Transformationsmatrix jedes Kreises bei einem auftretendem Sensorereignis auf eine Nullmatrix zurückgesetzt wird und der in **Listing** 4.7 dargestellte Vorgang wiederholt wird.

4.2 Der Radar

Auf der Kameraansicht angezeigte Kreise markieren zwar die ungefähre Position von POIs, die Entfernung, die die POIs vom aktuellen Standort entfernt sind, bleibt jedoch auf den ersten Blick unbekannt. Die Distanz zu ausgewählten POIs wird zwar, durch das Anwählen dieser, in einer Informationsbox angegeben, was aber mühsam sein kann, falls die Distanzen mehrerer POIs verglichen werden sollen.
Der Radar soll einen Überblick über alle POIs in der Umgebung schaffen und anzeigen, wieviele POIs in aktueller Blickrichtung angezeigt werden können. Die Anzahl der POIs, die tatsächlich angezeigt werden, hängt von der Neigung des Mobilgerätes ab. Der Radar analysiert die Umgebung nur aus der Vogelperspektive, also von oben. Als Hilfestellung, damit auch aus dem Radar die ungefähre Distanz eines POI ausgelesen werden kann, werden Grenzlinien in 2000m Abständen eingezeichnet.
Auch die Haltung des Mobilgerätes muss für die Anzeige des Radars miteinkalkuliert werden. Da das Blickfeld im Querformat breiter als im Hochformat und in der Diagonalhaltung am breitesten ist, muss das Blickfeld auf dem Radar, abhängig von der Neigung des Mobilgerätes, breiter oder schmaler dargestellt werden.
Ein Orientierungspunkt, der wie ein Kompass immer nach Norden zeigen soll, ist für die Funktionsweise des Radars unerlässlich und wird in Form eines Pfeiles an der Hülle des Radars angeheftet.

4.2.1 Der Nordpfeil

Da die Eckpunkte einer Form, die mit OpenGL ES gezeichnet werden soll, beim Arrayindex mit dem Wert *0* beginnend, gezeichnet werden, können früh gezeichnete Eckpunkte von später gezeichneten Eckpunkten übermalt werden. Damit sich der Nordpfeil mit dem Radarkörper verbindet, wird zuerst der Nordpfeil und danach der Radarkörper gezeichnet.
Für die Initialisieung werden die Eckpunkte des Nordpfeils so definiert, dass die Pfeilspitze die Koordinatenachsenwerte 0 für die x-Achse und die Summe aus Radarradius und Pfeilhöhe für die y-Achse einnimmt.
Die Eckpunkte des in **Abbildung** 4.7 gezeigten Nordpfeils werden in der Radarklasse definiert.

Listing 4.8: Eckpunkte des Nordpfeils in Pseudocode

```
1 public Radar(x, y, radius, segments, distanceRadius,
2         List<locations>){
3     ...
4     xCoord = 0f;
```

```
5       yCoord = radius + 0.03f;
6       zCoord = 0f;
7       * x,y,z Koordinaten in Array der Eckpunkte speichern
8       xCoord = 0.04f;
9       yCoord = r - 0.01f;
10      zCoord = 0f;
11      * x,y,z Koordinaten in Array der Eckpunkte speichern
12      xCoord = -xCoord;
13      * x,y,z Koordinaten in Array der Eckpunkte speichern
14      ...
15  }
```

Die Übergabeparameter x und y könnten die Positionierung des Radars festlegen, werden aber im weiteren Verlauf nicht verwendet. Weiter Übergabeparameter sind der Radius des zu zeichnenden Radars mit *radius*, die Anzahl der Dreiecke, welche die Kreisform des Radars bilden, mit *segments*, der Umgebungsradius, innerhalb welchem POIs von der Google API angefragt werden, mit *distanceRadius* und eine Liste von registrierten POIs, die auf dem Radar angezeigt werden sollen, mit *List<locations>*.
In *Zeile 4-13* werden die Koordinatenwerte für die drei Eckpunkte des Nordpfeils festgelegt und in das Array gespeichert, welches die Koordinaten aller Eckpunkte, die den Radar bilden, beinhaltet.

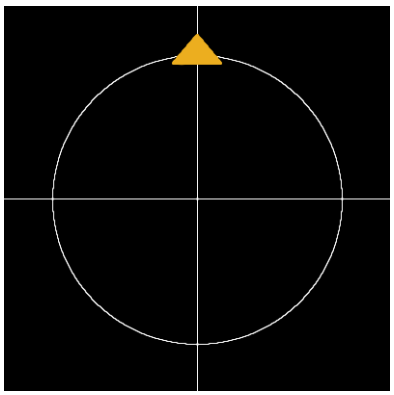

Abbildung 4.7: Positionierung des Radarnordpfeils

4.2.2 Der Radarkörper

Der Radarkörper stellt die Umgebung in einem bestimmten, aber änderbaren Erkennungsradius dar und besteht aus einer Umrandung, einer Nord-Süd- und Ost-West-Achse und Grenzlinien, die ausgehend vom Radarmittelpunkt 2000m Abstände markieren. Mittelpunkt des Radars bildet das Mobilgerät, auf welchem AREA ausgeführt wird.
Um eine Umrandung zu realisieren, wird ein Kreis gezeichnet, der von einem weiteren, minimal kleineren Kreis überzeichnet wird. Durch Verwendung von zwei verschiedenen Farben, wird eine Umrandung sichtbar, die dicker wird, je kleiner der zweite Kreis gezeichnet wird. Ein Kreis, der gleichzeitig für die Umrandung des Radars und die Nord-Süd- und Ost-West-Achsen verwendet

wird, bildet das Fundament des Radars. Auf den Fundamentkreis müssen weiter die Grenzlinien und die Bereiche zwischen den Grenzlinien gezeichnet werden, sodass die Nord-Süd- und Ost-West-Achsen nicht übermalt werden (vgl. **Abbildung** 4.8).

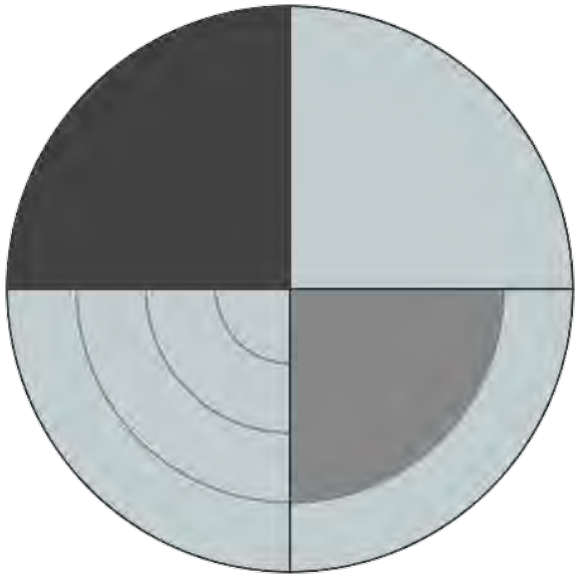

Abbildung 4.8: Aufbau des Radarkörpers (in Etappen)

Dafür wird ein Algorithmus verwendet, der die Eckpunktkoordinaten immer kleiner werdender Viertelkreise berechnet und speichert.

Listing 4.9: Algorithmus zum Berechnen der Radareckpunkte in Pseudocode

```
1  public Radar(x, y, radius, segments, distanceRadius,
2          List<locations>){
3      ...
4      numSegments = (int)(segments / 4);
5      numFragments = (int)(distanceRadius/2000);
6      for(i = 0; i < 4; i++){
7          for(j = 0; j < numFragments; j++){
8              if(j mod 2 == 0)
9                  * Bereich zwischen Grenzlinien
10                 * soll gezeichnet werden
11             else
12                 * Grenzlinie wird gezeichnet
13             for(k = 0; k < numSegments; k++){
14                 * Eckpunktkoordinaten berechnen
15                 * und im Array speichern
16             }
17         }
18     }
19     ...
```

```
20  }
```

Die Variable *numSegments* in *Zeile 4* entspricht der Anzahl Dreiecke, die einen Viertelkreis erzeugen. In *Zeile 5* wird die Anzahl der Grenzlinien berechnet. Durch drei ineinander verschachtelte Schleifen werden die Eckpunktkoordinaten berechnet und gespeichert. Die erste Schleife in *Zeile 6* durchläuft insgesamt vier Iterationen, die jeweils die Eckpunkte für einen Viertelkreis mit Grenzlinien berechnen und speichern. Die zweite Schleife in *Zeile 7* teilt den Viertelkreis in Fragmente, welche immer kleiner werdenden Viertelkreisen entsprechen und, mit alternierender Verwendung von Rand- und Füllfarbe, die Grenzlinien auf dem Radarkörper sichtbar machen.
In der dritten und letzten Schleife findet die tatsächliche Berechnung der Eckpunkte statt. Es werden die Dreieckskoordinaten für das aktuell ausgewählte Fragment berechnet und im Array abgespeichert (*Zeile 13-16*).
Anders als bei einem Kreis für ein POI, werden für den Radar unterschiedliche Farben verwendet. Die Verwendung der in **Listing** 3.3 *Zeile 10* als *uniform* definierten Farbe wäre für das Zeichnen des Radars ungünstig, da durch *uniform* eine einzige Farbe festgelegt wird, die für die Form verwendet wird.
Für mehrfarbige Formen kann im Fragment und Vertex Shader *varying* verwendet werden. Es muss, neben dem ByteBuffer für die Eckpunkte, ein weiterer Buffer für die Farben angelegt und in der *draw()*-Methode jedem Eckpunkt eine Farbe zugewiesen werden.
Die angepasste Version von **Listing** 3.3 sieht wie folgt aus.

Listing 4.10: Definition des Vertex Shaders und Fragment Shaders für mehrfarbige Formen

```
1  public class Radar{
2          private final String vertexShaderCode =
3                  "uniform mat4 uMVPMatrix;" +
4                  "attribute vec4 vPosition;" +
5                  "attribute vec4 vColor;" +
6                  "varying vec4 vecColor;"+
7                  "void main() {" +
8                  "  vecColor = vColor;" +
9                  "  gl_Position = uMVPMatrix * vPosition;" +
10                 "}";
11
12         private final String fragmentShaderCode =
13                 ...
14                 "varying vec4 vColor;" +
15                 ...;
16         ...
17 }
```

Der Vertex Shader Code enthält in *Zeile 3* eine Matrix *uMVPMatrix*, welche bereits in **Listing** 4.7 verwendet worden ist. Die *uMVPMatrix* enthält beispielsweise Transformationsdaten, mit welchen vermieden wird, dass Formen falsch bzw. anders dargestellt werden, falls sich die Höhe und Breite der Zeichenfläche unterscheiden. Da mithilfe der *uMVPMatrix* die eigentliche Darstellung verändert wird, muss sie im Vertex Shader Code definiert und mit dem *vPosition* Attribut verrechnet werden, um überhaupt erst diesen Effekt zu bewirken.

Für das mehrfarbige Zeichnen wird ein Attribut *vColor* in *Zeile 5* und ein variierbarer Vektor *vecColor* in *Zeile 6* definiert. Die einzige Änderung im Fragment Shader Code ist das wechseln von *uniform* in *varying* in *Zeile 14*.

Listing 4.11: Definition der draw-Methode zum Zeichnen eines Objekts

```
1  public void draw() {
2          ...
3          /* Alt
4          mColorHandle = GLES20.glGetUniformLocation(mProgram,
5                  "vColor");
6          GLES20.glUniform4fv(mColorHandle, 1, color, 0);
7          */
8          // Neu
9          mColorHandle = GLES20.glGetAttribLocation(mProgram,
10                 "vColor");
11         GLES20.glEnableVertexAttribArray(mColorHandle);
12         GLES20.glVertexAttribPointer(mColorHandle, 4,
13                 GLES20.GL_FLOAT, false, 0, colorBuffer);
14         ...
15 }
```

In *Zeile 3-7* wird die Handhabung einer festen Farbe, in auskommentierter Form, zum Vergleich aufgeführt. In *Zeile 9* wird das im Vertex Shader Code neu definierte Attribut *vColor* lokalisiert. In *Zeile 10* wird das Attributarray für den Renderingprozess freigegeben und in *Zeile 11* für den Renderingprozess vorbereitet, indem die Farbwerte aus dem FloatBuffer *colorBuffer* ausgelesen werden. Der FloatBuffer für die Farbwerte wird auf die gleiche Weise wie der FloatBuffer für die Koordinaten der Eckpunkte erzeugt.

4.2.3 Die POIs

Damit der Radar seinen Hauptzweck, nämlich POIs anzeigen, erfüllen kann, müssen diese korrekt auf dem Radar eingezeichnet werden. Ein POI wird auf dem Radar als Kreis dargestellt, dessen Positionierung sich aus der Entfernung und der nördlichen Ausrichtung errechnet (vgl. **Abbildung** 4.9).
Die Koordinaten des Mittelpunkts eines Kreises auf dem Radar lassen sich wie folgt berechnen.

Listing 4.12: Berechnen der Eckpunktkoordinaten eines POI auf dem Radar in Pseudocode

```
1  public Radar(x, y, radius, segments, distanceRadius,
2                  List<locations>){
3          ...
4          for all loc in locations{
5                  locationDistance = loc.getDistance();
6                  α = loc.getHoricontalHeading();
7                  distance = locationDistance / distanceRadius
8                          · radius;
```

```
 9                    centerX = cos(α) · distance;
10                    centerY = sin(α) · distance;
11                    for(i = 0; i < segments; i++){
12                          * Berechne Eckpunktkoordinaten
13                          * Speichere Eckpunktkoordinaten
14                    }
15              }
16              ...
17  }
```

In *Zeile 4* wird eine Schleife gestartet, welche alle im Umkreis bekannten POI durcharbeitet. Für die Berechnung Kreismittelpunktes werden die Distanz zwischen Mobilgerät und POI in *Zeile 5* und die Abweichung α von der Nordachse in *Zeile 6* aus vorhanden Informationen gelesen. Die Distanz wird in *Zeile 7* in den Maßstab des Radars umgerechnet, indem der Anteil der Distanz am Umgebungsradius mit dem Radarradius multipliziert wird. Die x- und y-Koordinatenwerte ergeben sich aus dem Sinus und Kosinus von α multipliziert mit der Distanz im Radarmaßstab (*Zeile 9+10*).
Ausgehend vom berechneten Kreismittelpunkt, werden die Eckpunkte des Kreises in einer weiteren Schleife in *Zeile 11-14* berechnet und dem Array angehängt, welches bereits die Eckpunktkoordinaten für den Nordpfeil und den Radarkörper beinhaltet.

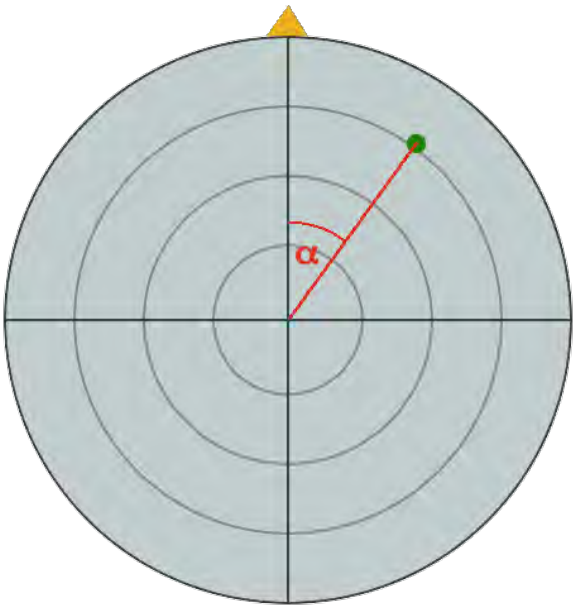

Abbildung 4.9: Radar mit eingezeichnetem POI

4.2.4 Das Sichtfeld

Das Sichtfeld soll als optisches Hilfsmittel dienen, welches die aktuelle Blickrichtung der Kamera farbig auf dem Radar markiert. Der Radarkörper dreht sich, wenn das Mobilgerät nach links

oder rechts geschwenkt wird oder die Haltung des Mobilgerätes rotiert. Das Sichtfeld dreht sich, im Gegensatz zum Radarkörper, nur, wenn das Mobilgerät rotiert und es ändert sich zusätzlich, abhänig von der Rotation, die Sichtfeldbreite. Der Radarkörper und das Sichtfeld verhalten sich somit unterschiedlich, wodurch das Sichtfeld durch ein extra Klasse und somit als eigene Form definiert wird.
Durch das Sichtfeld soll auf dem Radar die Fläche markiert werden, die von der Kamera erfasst werden kann. Deshalb wird das Sichtfeld so platziert, dass es, abhängig von der Rotation des Mobilgerätes, immer nach oben, also senkrecht vom Boden weg, zeigt (vgl. **Abbildung** 4.10).

Abbildung 4.10: Änderung des Sichtfelds für unterschiedliche Rotationen

Da das Sichtfeld der Kamera im Querformat breiter als im Hochformat und am breitesten in der Diagonalhaltung ist, soll dieser Breitenunterschied auf die Sichtfeldbreite übernommen werden. Damit das Sichtfeld den Radar und somit auch die auf dem Radarkörper dargestellten POI nicht übermalt, muss für das Sichtfeld eine transparente Farbe gewählt werden. Dafür wird

der Alphawert der verwendeten Farbe entsprechend angepasst. Mit einem Alphawert von 0 wird eine Farbe unsichtbar bzw. volltransparent. Ein Alphawert von 1 entspricht einer Farbe ohne Transparenz. Ein Alphawert von 0.5 erreicht den erwünschten Effekt und wird für den Alphawert der Sichtfeldfarbe verwendet.
Um den Winkel des Sichtfelds zu berechnen, muss die maximale Breite des Displays in aktueller Haltung des Mobilgerätes bestimmt werden (vgl. **Abbildung** 4.11).

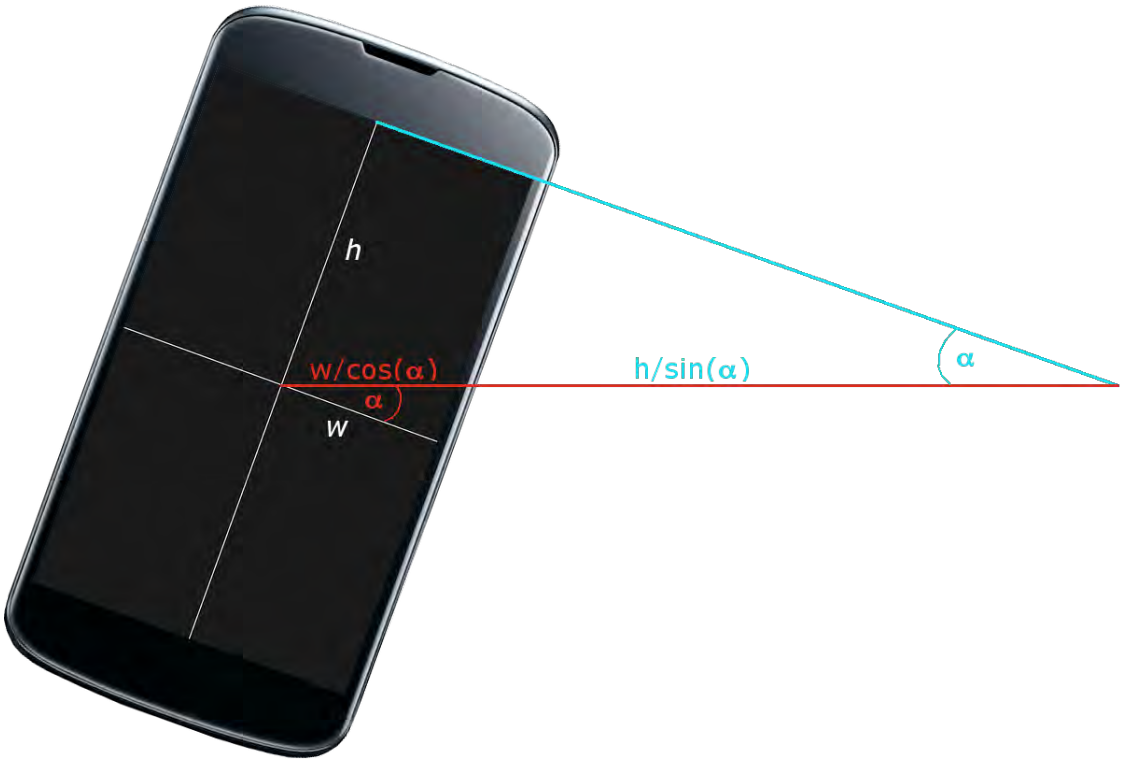

Abbildung 4.11: Berechnung der maximalen Breite abhängig von der Haltung

Der Winkel ergibt sich anschließend aus der Breite dividiert mit der Anzahl Pixel pro 1°.
Die aktuelle maximale Breite wird wie folgt berechnet.

Listing 4.13: Berechnen des Sichtfeldwinkels in Pseudocode

```
1  public FieldofView(x, y, radius, segments, rotation){
2          ...
3          rotation += 90;
4          if(rotation < 0){
5                  rotation += 360;
6          count = (int)(rotation mod 90);
7          rotation = rotation mod 90;
8
9          if(count mod 2 == 0){
10                 tmp1 = deviceWidth/cos(rotation);
11                 tmp2 = deviceHeight/sin(rotation);
12                 maxWidth = min{tmp1, tmp2};
13         }else{
```

```
14                      tmp1 = deviceWidth/sin(rotation);
15                      tmp2 = deviceHeight/cos(rotation);
16                      maxWidth = min{tmp1, tmp2};
17              }
18              α = maxWidth / pixelPerDegree;
19              ...
20  }
```

In *Zeile 3* wird der Rotationswinkel mit 90° addiert, um die in der Hochformathaltung des verwendeten Nexus 4 erhaltenen -90° auszugleichen. Damit mit positiven Winkeln zwischen 0° und 360° gerechnet werden kann, werden negative Winkel in *Zeile 4+5* entsprechend um 360° vergrößert. Das Koordinatensystem wird in seine vier Quadranten aufgeteilt (*Zeile 6+7*). Durch den Rotationswinkel lässt sich bestimmen, in welchem Quadranten sich die aktuell breiteste Strecke, von einem Displayende zum anderen, befindet. Abhängig vom ermittelten Quadranten werden zwei rechtwinklige Dreiecke mithilfe des Rotationswinkels und der Displayhöhe bzw. -breite gebildet. Je nach Quadrant bildet beispielsweise die Displayhöhe die Ankathete in einem rechtwinkligen Dreieck und die Displaybreite die Gegenkathete des anderen Dreiecks. Es wird jeweils die Länge der Hypothenuse berechnet, wobei die kürzere der beiden Hypothenusen als Ergebnis verwendet wird (*Zeile 10-12, 14-16*). Der Winkel des Sichtfelds ergibt sich aus der Division der berechneten Hypothenusenlänge durch die Anzahl Pixel pro 1° in *Zeile 18*.
Damit der Radar vollständig angezeigt wird, muss der Radarkörper vor dem Sichtfeld gezeichnet werden, da sonst das Sichtfeld vom Radarkörper überdeckt werden würde. Der Mittelpunkt des Radarkörpers und des Sichtfeldes, als Teilkreis betrachtet, ist identisch und hat auf dem Display eine feste Position. Wie die Kreise für die POIs, wird der Radarkörper bei jedem eingehendem Sensorereignis auf den OpenGL ES Koordinatensystemursprung zurückgesetzt, gedreht und neu ausgerichtet.
Da sich die Sichtfeldbreite durch die Rotation des Mobilgerätes ständig ändert, muss die Form des Sichtfelds neu berechnet werden. Das Sichtfeld wird also nicht wie der Radarkörper auf den Ursprung zurückgesetzt, sondern muss neu erzeugt werden.

4.3 Textrendering

Ein POI wird durch einen Kreis markiert, indem der Kreis auf der ungefähren Position des POI auf dem Display dargestellt wird. Da ein Kreis alleine wenig über den POI aussagt und dem Benutzer das ständige Anwählen der Kreise zur Informationsbeschaffung erspart werden soll, wird ein Kreis mit einem Namensschild erweitert, der den Namen des POI angibt. Der Name eines POI wird bereits von der Google API geliefert und kann zur Realisierung eines Namensschilds verwendet werden.

4.3.1 Problematik

Textrendering ist mit OpenGL ES 2.0 nicht ohne weiteres möglich, da anscheinend keine Funktionalität zum Zeichnen von Text gegeben ist. Jeder einzelne Buchstabe, jedes einzelne Sonderzeichen und jede einzelne Ziffer müsste als Form, bestehend aus Eckpunkten und Kanten, definiert werden. Diese Methode, Text zu zeichnen, wäre sehr mühsam und in der Umsetzung

vermutlich sehr ineffzient, wenn beispielsweise für ein **a** etwa 100 oder mehr Eckpunkte gezeichnet werden müssten. Weiterführend müssten für das Einbinden von verschiedenen Schriftarten neue Formen definiert werden.

4.3.2 Lösung

Um Text zeichnen zu können, wird eine bereits implementierte Lösung verwendet, die für OpenGL ES 1.0 erstellt worden ist, aber auch in einer überarbeiteten Version für OpenGL ES 2.0 verfügbar ist [fra15] [Kod15].
Die Idee hinter diesem Konzept ist, dass zur Laufzeit eine font Bitmap aus einer TrueType (.ttf) font Datei generiert wird. Schrittweise werden die einzelnen Zeichen des zu zeichnenden Texts in der generierten font Bitmap gesucht, als Texturblöcke mit festgelegtem Abstand nebeneinandergelegt und gezeichnet.
Nachfolgend wird der grobe Aufbau der beiden wesentlichen Methoden *load()* und *draw()* der Klasse *GLText* beschrieben. In der *load()*-Methode wird die font Bitmap aus der TrueType font Datei generiert. Durch den Aufruf der *draw()*-Methode wird ein übergebener Text gezeichnet.

Listing 4.14: Methodenkopf der load()-Methode

```
public boolean load(String file, int size, int padX, int padY)
```

Mit *file* wird der Dateiname der TrueType font Datei angegeben. Die Schriftgröße wird mit *size* festgelegt. Mit den beiden Parametern *padX* und *padY* wird der Abstand, der links, rechts, über und unter jedem einzelnen Zeichen freigehalten werden soll, festgelegt.

Listing 4.15: Verarbeiten der TrueType Datei in einr Paint Instanz

```
1 Typeface tf = Typeface.createFromAsset(assets, file);
2 Paint paint = new Paint();
3 paint.setAntiAlias( true );
4 paint.setTextSize( size );
5 paint.setColor( 0xffffffff );
6 paint.setTypeface( tf );
```

In *Zeile 1* wird aus der TrueType font Datei ein Typeface erzeugt. In *Zeile 2-5* wird eine Paint Instanz erzeugt und konfiguriert, die später verwendet wird, um die einzelnen Zeichen, die mit *GLText* gerendert werden können, als Textur in die Bitmap zu zeichnen. Für *paint* wird als Typeface das in *Zeile 1* erzeugte Typeface *tf* gesetzt (*Zeile 6*).
Mithilfe der Paint Instanz *paint* wird die exakte Breite jedes Zeichens ermittelt.

Listing 4.16: Bestimmen der Breite jedes Zeichens in der Bitmap

```
1 char[] s = new char[2];
2 charWidthMax = charHeight = 0;
3 float[] w = new float[2];
4 int cnt = 0;
5 for ( char c = CHAR_START; c <= CHAR_END; c++ ) {
6     s[0] = c;
7     paint.getTextWidths( s, 0, 1, w );
```

```
 8       charWidths[cnt] = w[0];
 9       if ( charWidths[cnt] > charWidthMax )
10           charWidthMax = charWidths[cnt];
11       cnt++;
12   }
13   s[0] = CHAR_NONE;
14   paint.getTextWidths( s, 0, 1, w );
15   charWidths[cnt] = w[0];
16   if ( charWidths[cnt] > charWidthMax )
17       charWidthMax = charWidths[cnt];
18   cnt++;
19
20   charHeight = fontHeight;
```

In *Zeile 5* werden alle Zeichen, die mit *GLText* gerendert werden können, in einer Schleife einzeln durcharbeitet. *CHAR_START* entspricht dem ASCII-Wert des ersten und *CHAR_END* dem des letzen Zeichens, der gerendert werden kann. Die Werte für *CHAR_START* und *CHAR_END* sind manuell auf die Werte 32 und 126 gesetzt worden. Die Breite jedes Zeichens wird in *Zeile 7* bestimmt und in *Zeile 8* in einem Array gespeichert. Die Breite des breitesten Zeichens wird ebenfalls gespeichert (*Zeile 9+10*).
Für *GLText* unbekannte Zeichen, deren ASCII-Wert nicht innerhalb von 32 und 126 liegt, werden durch *CHAR_NONE* mit dem Wert 32 ersetzt, was in der ASCII-Tabelle dem Leerzeichen entspricht. Die Breite des Leerzeichens ist bereits mit *CHAR_START* berechnet worden, das *CHAR_NONE*-Zeichen muss aber dennoch in *Zeile 13-17* extra berechnet werden, da dessen ASCII-Wert manuell geändert werden könnte.
Ist die Breite des breitesten Zeichens bestimmt, können die Zellenmaße festgelegt werden, die für jedes Zeichen verwendet werden.

Listing 4.17: Festlegen der Zellenmaße für alle renderbaren Zeichen

```
1  cellWidth = (int)charWidthMax + ( 2 * fontPadX );
2  cellHeight = (int)charHeight + ( 2 * fontPadY );
3  int maxSize = cellWidth > cellHeight ? cellWidth : cellHeight;
4  if ( maxSize < FONT_SIZE_MIN || maxSize > FONT_SIZE_MAX )
5      return false;
```

Die Zellenbreite ergibt sich aus der in **Listing** 4.16 bestimmten maximalen Breite addiert mit dem freien Abstand links und rechts jedes Zeichens in *Zeile 1*. Die Zellenhöhe ergibt sich aus der festgelegten Schriftgröße addiert mit dem freien Abstand über und unter jedem Zeichen in *Zeile 2*. Passen die berechneten Zellenmaße nicht in die festgelegte Max-/Minskala, wird dies als Fehler in *Zeile 4+5* zurückgesendet.
Mit dem Wert für *maxSize* kann nun die größe der Textur bzw. der Bitmap bestimmt werden.

Listing 4.18: Festlegen der Textur- bzw. Bitmapgröße

```
1  if ( maxSize <= 24 )
2      textureSize = 256;
3  else if ( maxSize <= 40 )
4      textureSize = 512;
```

```
5  else if ( maxSize <= 80 )
6      textureSize = 1024;
7  else
8      textureSize = 2048;
```

Je größer die Zellen mit *maxSize* bestimmt sind, desto größer wird die Bitmap.
Als nächster Schritt wird die Bitmap erzeugt.

Listing 4.19: Erzeugen einer Bitmap mit bestimmer Größe

```
1  Bitmap bitmap;
2  bitmap = Bitmap.createBitmap( textureSize, textureSize,
3      Bitmap.Config.ALPHA_8 );
4  Canvas canvas = new Canvas( bitmap );
5  bitmap.eraseColor( 0x00000000 );
6
7  colCnt = textureSize / cellWidth;
8  rowCnt = (int)Math.ceil( (float)CHAR_CNT / (float)colCnt );
```

Die Bitmap wird mit der in **Listing** 4.18 bestimmten Texturgröße erstellt (*Zeile 1-3*) und in *Zeile 5* mit transparenter Farbe, hier schwarz mit einem Alphawert von 0, gefüllt. In *Zeile 4* wird ein Canvas erzeugt, dem die Bitmap angehängt wird. In *Zeile 7+8* werden die Anzahl der Zeilen und Spalten der Bitmap berechnet.
Um zum Rendern von Text verwendet werden zu können, muss die noch leere Bitmap mit fonts, bzw. Schriftzeichen, gefüllt werden.

Listing 4.20: Generieren der font Bitmap

```
1  float x = fontPadX;
2  float y = ( cellHeight - 1 ) - fontDescent - fontPadY;
3  for ( char c = CHAR_START; c <= CHAR_END; c++ ) {
4      s[0] = c;
5      canvas.drawText( s, 0, 1, x, y, paint );
6      x += cellWidth;
7      if ( ( x + cellWidth - fontPadX ) > textureSize ) {
8          x = fontPadX;
9          y += cellHeight;
10     }
11 }
12 s[0] = CHAR_NONE;
13 canvas.drawText( s, 0, 1, x, y, paint );
```

Zeilenweise werden die definierten Zeichen in die erzeugte Canvas Instanz gerendert. Hierfür werden die Startwerte für die (x,y) Koordinaten in *Zeile 1+2* auf die linke unterste Zelle festgelegt. Die einzelnen Zeichen werden in *Zeile 5* gerendert. Anschließend werden die neuen (x,y) Koordinaten für das nächste Zeichen in *Zeile 8-10* bestimmt. Zum Schluss wird in *Zeile 13* das Zeichen gerendert, das für die undefinierten Zeichen eingesetzt wird.

Listing 4.21: Generierung einer OpenGL ES 2.0 Textur

```
1  final int[] textureHandle = new int[1];
2  GLES20.glGenTextures(1, textureHandle, 0);
3  if (textureHandle[0] != 0){
4      GLES20.glBindTexture(GLES20.GL_TEXTURE_2D, textureHandle[0]);
5      GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
6          GLES20.GL_TEXTURE_MIN_FILTER, GLES20.GL_LINEAR);
7      GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
8          GLES20.GL_TEXTURE_MAG_FILTER, GLES20.GL_LINEAR);
9      GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
10         GLES20.GL_TEXTURE_WRAP_S, GLES20.GL_CLAMP_TO_EDGE );
11     GLES20.glTexParameterf(GLES20.GL_TEXTURE_2D,
12         GLES20.GL_TEXTURE_WRAP_T, GLES20.GL_CLAMP_TO_EDGE );
13
14     GLUtils.texImage2D(GLES20.GL_TEXTURE_2D, 0, bitmap, 0);
15     bitmap.recycle();
16 }
17 if (textureHandle[0] == 0){
18     throw new RuntimeException("Error loading texture.");
19 }
```

Eine Textur wird in *Zeile 2* erzeugt und in *Zeile 4* in OpenGL gebunden. In *Zeile 5-8* werden die Filter- und in *Zeile 9-12* die Wrappingeigenschaften der Textur gesetzt. In *Zeile 14* wird die Bitmap in die gebundene Textur geladen und in *Zeile 15* recycelt. Tritt beim Laden der Textur ein Fehler auf, wird mit *Zeile 17-19* entsprechend eine Exception geworfen.
Am Ende der *load()*-Methode wird für jedes definierte Zeichen eine Texturregion eingerichtet.

Listing 4.22: Einrichtung einer Texturregion für jedes einzelne Zeichen

```
1  x = 0;
2  y = 0;
3  for ( int c = 0; c < CHAR_CNT; c++ ) {
4      charRgn[c] = new TextureRegion( textureSize, textureSize,
5                                      x, y, cellWidth-1, cellHeight-1 );
6      x += cellWidth;
7      if ( x + cellWidth > textureSize ) {
8          x = 0;
9          y += cellHeight;
10     }
11 }
```

Eine Schleife mit gleicher Anzahl Iterationen wie definierter Zeichen, füllt ein Array *charRgn* mit gleich großen, aber unterschiedlich positionierten Texturregionen in *Zeile 4*. Am Ende jeder Iteration werden die nächsten (x,y) Koordinaten auf der Textur für die nächste Texturregion in *Zeile 6-10* bestimmt.
Beim Rendern werden die Texturregionen verwendet, damit das Zeichen, das gerendert werden soll, schnell selektiert werden kann. Ist die *load()*-Methode erfolgreich durchgeführt, somit eine font Bitmap erstellt und alle nötigen Informationen, um die Bitmap zum Rendern von Text verwenden zu können, beschafft worden, kann der Renderprozess in der *draw()*-Methode aus-

geführt werden.
Der *draw()*-Methodenkopf sieht wie folgt aus.

Listing 4.23: Methodenkopf der draw()-Methode

```
public void draw(String text, float x, float y)
```

Mit dem *text*-Parameter wird der Text vorgegeben, der gerendert werden soll. Die Parameter x und y legen die Position auf den OpenGL ES Koordinatenachsen fest, an denen *text* gerendert werden soll.
Der übergebene Text *text* muss jetzt noch gerendert werden.

Listing 4.24: Funktionalität der draw()-Methode zum Textrendering

```
1  float chrHeight = cellHeight * scaleY;
2  float chrWidth = cellWidth * scaleX;
3  int len = text.length();
4  x += ( chrWidth / 2.0f ) - ( fontPadX * scaleX );
5  y += ( chrHeight / 2.0f ) - ( fontPadY * scaleY );
6  for ( int i = 0; i < len; i++ ) {
7      int c = (int)text.charAt( i ) - CHAR_START;
8      if ( c < 0 || c >= CHAR_CNT )
9          c = CHAR_UNKNOWN;
10     batch.drawSprite( x, y, chrWidth, chrHeight, charRgn[c] );
11     x += ( charWidths[c] + spaceX ) * scaleX;
12 }
```

In *Zeile 1+2* wird die Zellenbreite und -höhe und in *Zeile 3* die Textlänge kalkuliert. Eine Schleife in *Zeile 6* durchläuft einzeln jedes Zeichen des Texts. Die Position des jeweiligen Zeichens im Textregion-Array wird in *Zeile 7* ermittelt, indem der ASCII-Wert des Zeichens mit dem ASCII-Wert des ersten definierten Zeichens der Bitmap subtrahiert wird. Ist die ermittelte Position nicht innerhalb der Reichtweite des Arrays, wird das Zeichen als unbekannt markiert und durch ein festgelegtes, definiertes Zeichen in *Zeile 8+9* ersetzt. Mithilfe eines *SpriteBatch* wird in *Zeile 10* das Zeichen gezeichnet, indem die (x,y) Koordinaten, die zu rendernde Größe und die Position des Zeichens, auf der Textur, angegeben werden. Anschließend wird die x-Achsen Koordinate für den nächsten *char* angepasst.
Damit Text mit der *GLText*-Klasse gerendert werden kann, wird eine neue Instanz in der *onSurfaceCreated()*-Methode in der Rendererklasse erstellt.

Listing 4.25: Einrichtung einer Texturregion für jedes einzelne Zeichen

```
1  private GLText glText;
2
3  public void onSurfaceCreated(GL10 gl, EGLConfig config) {
4      glText = new GLText( gl, context.getAssets() );
5      glText.load( "Roboto-Regular.ttf", 14, 2, 2 );
6  }
```

In *Zeile 4* wird eine neue *GLText* Instanz erzeugt und in *Zeile 5* eine TrueType font Datei geladen, wobei die Schriftgröße auf 14 Pixel und das Padding in x- und y-Richtung jeweils auf 2 Pixel festgelegt wird.

Als Vorbereitung für den Renderprozess, werden die 2D Texturierung und Alpha Blending in der *onSurfaceCreated()*-Methode aktiviert.

Listing 4.26: Aktivieren von 2D Texturierung und Alpha Blending

```
1 public void onSurfaceCreated(GL10 gl, EGLConfig config) {
2     ...
3     GLES20.glEnable(GLES20.GL_BLEND);
4     GLES20.glBlendFunc(GLES20.GL_ONE, GLES20.GL_ONE_MINUS_SRC_ALPHA);
5 }
```

Die in **Listing** 4.23+4.24 definierte *draw()*-Methode ermöglicht das Rendern von Text in einer festgelegten (x,y) Position. Für eine Umsetzung in AREA muss Text aber auch in einem bestimmten Winkel verdreht dargestellt werden können. In der verwendeten Implementierung sind bereits weitere *draw()*-Methoden definiert worden, von denen eine verwendet wird, die Text verdreht rendern kann. Der Methodenkopf der verwendeten *draw()*-Methode ist wie folgt aufgebaut.

Listing 4.27: Methodenkopf der draw()-Methode für rotierbaren Text

```
public float drawC(String text, float x, float y, float angleDeg)
```

Die *drawC()*-Methode rendert den Text *text*, anders als die *draw()*-Methode, zentriert an den gegebenen (x,y) Koordinaten. Die ersten drei Parameter, *text*, x und y, entsprechen denen der Standardmethode. Mit dem vierten Parameter *angleDeg* kann ein Winkel festgelegt werden, um welchen der Text *text* an den gegebenen (x,y) Koordinaten rotiert wird.
Der Name des POI wird oberhalb des Kreises gerendert. Ein Kreis wird auf den OpenGL ES Koordinatenursprung gezeichnet, in x- und y-Richtung verschoben und anschließend um den Koordinatenursprung gedreht. Text wird mit der implementierten Version der *drawC()*-Methode an den (x,y)-Koordinaten rotiert. Damit Text um den Koordinatenursprung rotiert wird und sich somit Kreis und Text gleich verhalten, wird die *drawC()*-Methode entsprechend angepasst.

Listing 4.28: Änderung des Rotationsverhaltens der drawC()-Methode

```
1  drawC(...){
2      ...
3      // Alte Version
4      //Matrix.rotateM(modelMatrix, 0, angleDegZ, 0, 0, 1);
5      //Matrix.rotateM(modelMatrix, 0, angleDegX, 1, 0, 0);
6      //Matrix.rotateM(modelMatrix, 0, angleDegY, 0, 1, 0);
7      // Neue Version
8      Matrix.setIdentityM(modelMatrix, 0);
9      Matrix.setRotateM(modelMatrix, 0, angleDegZ, 0, 0, 1);
10     Matrix.translateM(modelMatrix, 0, x, y, z);
11     ...
12 }
```

Anstatt den Text um die einzelnen Achsen zu rotieren, wird zum Positionieren des Textes die gleiche Methode wie für die Kreise, die POIs markieren, verwendet. In *Zeile 8* wird die Transformationsmatrix *modelMatrix* auf eine Nullmatrix zurückgesetzt, in *Zeile 9* um die gewünschte

Rotationsmatrix erweitert und in *Zeile 10* mit Informationen über die gewünschten (x,y) Koordinaten gefüllt.
In der *onDrawFrame()*-Methode der Rendererklasse wird der Renderingprozess für einen POI-Namen mit den Positions- und Rotationsdaten des entsprechenden Kreises eingeleitet.

Listing 4.29: Rendern eines POI-Namen gebunden an einem Kreis in Pseudocode

```
1  onDrawFrame(){
2      ...
3      glText.begin(0f, 0f, 0f, 1f, mVPMatrix);
4      String text = POI.getName();
5      if(text.isEmpty() || text == null)
6          text = " ";
7      x = calcXCoord(POI) · min(width, height) / 2;
8      y = (calcYCoord(POI) + POI.radius + 0.04f)
9          · min(width, height) / 2;
10     y += (glText.cellHeight / 2);
11     α = calcRotation();
12     glText.drawC(text, x, y, α);
13                 glText.end();
```

In *Zeile 3* werden die RGB-Werte und der Alphawert der Schriftfarbe, hier schwarz, festgelegt. Der Name des POI wird ausgelesen und durch ein Leerzeichen ersetzt, falls für den POI kein Name vorhanden ist (*Zeile 4-6*). Die (x,y) Koordinaten und der Winkel α werden mithilfe der Methoden aus **Listing** 4.4, 4.5 und 4.6 berechnet (*Zeile 7-11*). Da der Text oberhalb des Kreises gezeichnet werden soll, wird der y-Wert entsprechend um den Kreisradius und einen zusätzlichen Abstand verschoben. In der verwendeten Implementierung zum Rendern von Text wird nicht der gleiche Koordinatensystemaßstab wie für das Rendern von Kreisen verwendet. Die (x,y) Koordinaten müssen im Pixelformat angegeben werden, weshalb die Ergebnisse von x und y mit der halben Displaybreite bzw. -höhe multipliziert werden (*Zeile 7, 9*). In *Zeile 12* wird der Renderingprozess ausgeführt.
Da der gerenderte Text wie in **Abbildung** 4.12 nicht immer gut zu erkennen ist, wird ein Hintergrundlabel, mit angepasster Größe, hinter den Text platziert. Da der Kreis und das zugehörige Label auf die Bewegungen des Mobilgerätes gleich reagieren, muss das Label nicht als extra Klasse defniert werden, sondern wird in die Klasse, die die Form des Kreises definiert, integriert.
Das Label wird aus zwei Rechtecken, die übereinander gezeichnet ein Kreuz ergeben, und vier Viertelkreisen als abgerundete Ecken gebildet (vgl. **Abbildung** 4.13). Diese sechs Komponenten werden einmal in schwarz und einmal, in einer etwas kleineren Variante, in weiß gezeichnet, damit eine Umrandung um das Label entsteht.
Die (x,y,z) Koordinaten der Eckpunkte des Labels werden dem Array, mit den Eckpunkten des Kreises, angehängt. Da das Label mit zwei unterschiedlichen Farben bemalt werden soll, für die Kreisform aber bis jetzt eine einzelne Farbe festgelegt worden ist, muss der Shadercode der Kreisform angepasst werden. Der angepasste Shadercode des Kreises entspricht dem des Radars, welcher bereits in **Listing** 4.10 definiert worden ist. Ein einfaches Array mit vier Elementen für die RGB- und Alphawerte ist ebenfalls nicht mehr ausreichend, sondern wird, wie bereits beim Radar, durch ein Array ersetzt, welches parallel zu jedem Eckpunkt eine Farbe speichert. Damit der Name des POI auf das Label gezeichnet wird, wird zuerst der Kreis mit Label und

Abbildung 4.12: Markierter POI mit POI-Namen

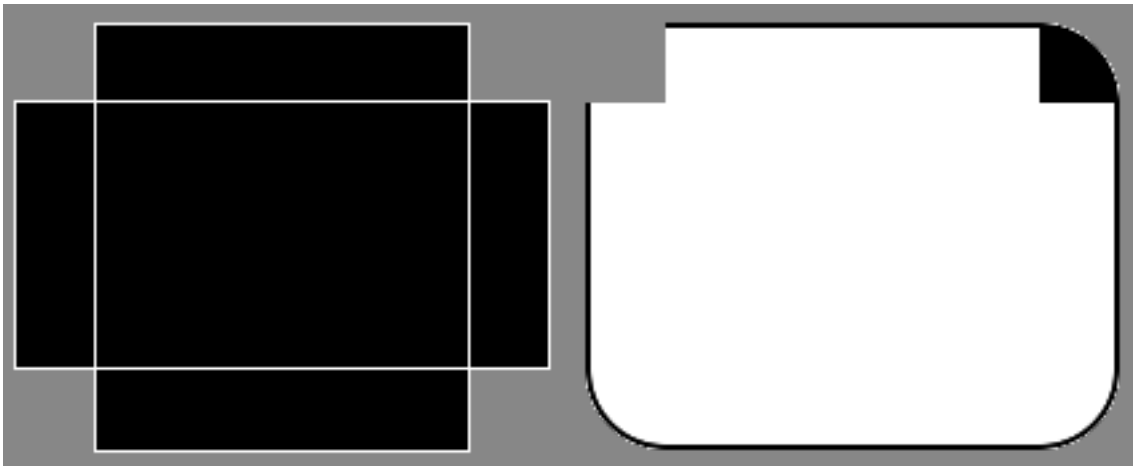

Abbildung 4.13: Aufbau des Textlabels

dann der Text gerendert. **Abbildung** 4.14 zeigt einen in AREA gezeichneten Kreis mit Text und Label.

4.4 Clusterbildung

Dem Benutzer von AREA soll die Möglichkeit geboten werden, weitere Informationen über ein POI erhalten zu können. Dazu kann ein POI über das Touchscreen des Mobilgerätes angewählt werden und ein Informationsfenster zum gewählten POI wird angezeigt. Damit das System erkennt, welches POI angewählt worden ist, werden die (x,y) Koordinaten des auf dem Touchscreen berührten Punkts ausgelesen und in das OpenGL ES Koordinatensystemformat

Abbildung 4.14: Textlabel eines POI in AREA

umgerechnet.
Das Auslesen der (x,y) Koordinaten findet in der *SurfaceView*-Klasse in der *onTouchEvent()*-Methode statt. Die Methode *onTouchEvent(MotionEvent e)* ist eine bereits implementierte Methode der Superklasse *View* und kann verwendet werden, um eingehende Touchscreenereignisse zu verarbeiten.

Listing 4.30: Auslesen der Koordinaten der berührten Touchscreenstelle

```
1 public boolean onTouchEvent(MotionEvent e){
2     if(e.getAction() == MotionEvent.ACTION_DOWN) {
3         onTouchStartX = e.getX();
4         onTouchStartY = e.getY();
5         mRenderer.registerPOIonTouchEvent(onTouchStartX,
6             onTouchStartY);
7     }
8     ...
9 }
```

Wird eine Berührung mit dem Touchscreen registriert (*Zeile 2*), werden die (x,y) Koordinaten an der berührten Stelle in *Zeile 3+4* ausgelesen und einer Funktion in der Rendererklasse *mRenderer* in *Zeile 5* übergeben.
Die Funktion *registerPOIonTouchEvent()* der Rendererklasse überprüft, ob an der berührten (x,y) Koordinate der Radar positioniert ist oder ein POI markiert wird. Werden sowohl der Radar als auch ein POI angewählt, hat der Radar eine höhere Priorität und es wird das Distanzfenster geöffnet. Wird ein POI an der berührten Stelle markiert, öffnet sich ein Informationsfenster zu diesem POI.

Listing 4.31: Ermitteln des angewählten Elements mit entsprechender Ausgabe in Pseudocode

```
1  registerPOIonTouchEvent(x, y){
2      coords = convertToGL(x,y);
3      if(coords inside radarrange)
4          showDistancePopup();
5      else{
6          coords = rotateCoords(coords);
7          for all p in POI_List{
8              if(coords inside p-range){
9                  showPointOfInterestPopup(p);
10                 break;
11             }
12     }
13 }
```

Die Parameter x und y werden als Pixelwerte übergeben und in *Zeile 2* für die weitere Verwendung in OpenGL ES Koordinatenwerte umgerechnet. In *Zeile 3* wird zuerst geprüft, ob sich die konvertierten Koordinatenwerte innerhalb des gezeichneten Radars befinden. Bei einem positiven Ergebnis wird in *Zeile 4* die Distanzview angefordert.
Ist der Radar nicht angewählt worden, wird überprüft, ob sich die Koordinaten der berührten Stelle auf einem markierten POI befinden. Die in *Zeile 2* konvertierten Koordinaten eignen sich aber noch nicht zum Abgleichen mit den in und **Listing** 4.4 und 4.5 berechneten Kreiskoordinaten. Die Kreiskoordinaten werden davon ausgehend, dass das Mobilgerät senkrecht gehalten wird, bestimmt. Daher werden in *Zeile 6* die konvertierten Koordinaten um den Koordinatenursprung rotiert, sodass ebenfalls Koordinatenwerte entstehen, die von einer senkrechten Haltung ausgehen.
Die konvertierten und rotierten Koordinatenwerte können nun verwendet werden, um sie mit den Kreiskoordinaten abzugleichen. Wird in einer Iteration festgestellt, dass der aktuell betrachtete POI angewählt worden ist, wird in *Zeile 9* eine POIView zum entsprechenden POI angefordert und die Schleife frühzeitig beendet.

4.4.1 Problematik

In bestimmten Situationen ist das im vorherigen Abschnitt geschilderte Verfahren zum Anwählen eines POI ungenau und führt zu einem falschen Resultat. Befinden sich vom Benutzer aus gesehen mehrere POIs auf einer oder fast gleicher Linie, werden diese auf der Kameraansicht von AREA an fast gleicher Stelle markiert (vgl. **Abbildung** 4.15). Kreise werden von anderen Kreisen überlappt oder sogar ganz verdeckt. Dieses Ereignis wird hier als Clusterbildung bezeichnet, da sich mehrere Kreise an einem Punkt ansammeln. Die Clusterbildung erschwert dem Benutzer das Anwählen eines bestimmten POI oder macht es sogar unmöglich.
Die in **Listing** 4.31 vorgestellte Methode überprüft zwar, ob ein POI auf der Kameraansicht angewählt worden ist, kann aber selbst nicht wissen, welcher POI vom Benutzer beabsichtigt angewählt worden ist.

Abbildung 4.15: Mehrere Markierungen bilden einen Cluster

4.4.2 1. Lösung: Auswahlliste

Als einfache Lösung für die Clusterbildungproblematik werden alle POIs, welche sich innerhalb des angewählten Clusters befinden, ermittelt und in einer Liste gespeichert. Der Benutzer bekommt diese Liste angezeigt und kann den gewünschte POI, dessen Informationsfenster angezeigt werden soll, in der Liste auswählen (vgl. **Abbildung** 4.16). Beinhaltet die ermittelte Liste nur ein einziges POI, so wird sofort das Informationsfenster zu diesem einen POI angezeigt.
Da die in **Listing** 4.31 definierte Methode bereits überprüft, ob ein POI angewählt worden ist, kann diese weiter verwendet werden. Damit die Methode nicht bereits nach dem ersten Treffer beendet wird, sondern alle sichtbaren POI überprüft und alle Treffer in einer Liste speichert, muss die Methode erweitert werden.

Listing 4.32: Erweiterte Methode durch Registrieren aller angewählten POIs in Pseudocode

```
1  registerPOIonTouchEvent(x, y){
2      ...
3      listOfSelectedPOI = new List();
4      for all p in POI_List{
5          if(coords inside p-range)
6              listOfSelectedPOI.add(p);
7      }
8      if(listOfSelectedPOI.size() > 0){
9          if(listOfSelectedPOI.size() == 1}{
10             showPointOfInterestPopup(listOfSelectedPOI.get(0));
11         else{
```

```
12                  showListOfPOIsInRange(listOfSelectedPOI);
13              }
14          }
15      }
```

In *Zeile 3* wird eine Liste erzeugt, in welcher alle POIs gespeichert werden, die von dem auf dem Touchscreen berührten Punkt getroffen werden. Ob ein POI getroffen wird, wird in *Zeile 5* überprüft. Bei einem positiven Ergebnis in *Zeile 5*, wird in *Zeile 6* der aktuell geprüfte POI *p* der Liste *listOfSelectedPOI* hinzugefügt. Sind alle POIs überprüft worden, wird in *Zeile 8* ermittelt, ob überhaupt ein POI getroffen worden ist. Beinhaltet die Liste *listOfSelectedPOI* genau einen POI, so wird in *Zeile 10* dessen Informationsfenster angefordert. Beinhaltet die Liste mehr als einen POI, so wird in *Zeile 12* eine Ansicht angefordert, die alle POI der Liste namentlich auflistet und jeden einzelnen POI anwählbar macht.

Abbildung 4.16: Auswahlliste über POIs in einem Cluster

4.4.3 2. Lösung: onTouch-Interaktion

In einem Cluster kann es passieren, dass das Namensschild eines Kreises von anderen Kreisen übermalt wird und der Name des markierten POI unleserlich wird. Der Benutzer kann zwar

durch die implementierte Lösung des vorherigen Abschnitts den Namen eines POI durch die Auswahlliste leicht herausfinden, muss sich aber Namen und Position des POI merken, da das Namensschild weiterhin verdeckt bleibt.
Es wird eine weitere Lösung zur Clusterbildungproblematik implementiert, die es dem Benutzer ermöglichen soll, die gezeichneten Kreise innerhalb eines festgelegten Radius zu verschieben. Damit ein verschobener Kreis weiterhin einen POI markiert, wird eine Verbindungslinie zwischen verschobenem Kreis und der ursprünglichen Position gezeichnet. Verschobene Kreise werden durch eine Farbänderung erkennbar gemacht.
Ein Kreis wird erst dann versetzt gezeichnet, wenn eine Wischbewegung auf dem Display registriert wird und der Kreis als Startpunkt der Wischbewegung angewählt worden ist. Ab welcher bewegten Distanz von einer Wischbewegung gesprochen werden darf, muss festgelegt werden, damit selbst ein kurzes Anwählen eines POI, mit der Absicht, dessen Informationsfenster angezeigt zu bekommen, nicht fälschlicherweise für eine Wischbewegung gehalten wird. Dazu wird die in **Listing** 4.30 verwendete *onTouchEvent()*-Methode erweitert.

Listing 4.33: onTouchEvent() Erweiterung zur Registrierung einer Wischbewegung

```
1  public boolean onTouchEvent(MotionEvent e){
2      ...
3      else if(e.getAction() == MotionEvent.ACTION_MOVE){
4          distanceMovedWhileTouched = movedDistanceWhileDown(
5              e.getX(), e.getY());
6          if(distanceMovedWhileTouched >= 100){
7              poiPositionHasToChange = true;
8              mRenderer.movePOIonTouchEvent(e.getX(), e.getY());
9          }
10     }
11     else if(e.getAction() == MotionEvent.ACTION_UP){
12         if(movedDistanceWhileDown(e.getX(), e.getY()) < 100){
13             mRenderer.displayClicked();
14         }
15         mRenderer.toMove = false;
16         mRenderer.POIsInRangeOfTouchEvent = null;
17         poiPositionHasToChange = false;
18     }
19     return true;
20 }
```

Wird in *Zeile 3* ein *ACTION_MOVE* Ereignis registriert, wird vermutet, dass der Benutzer einen Kreis verschieben möchte. Da aber bereits ein Zittern dieses *ACTION_MOVE* Ereignis auslösen kann, wird eine Distanz festgelegt, die die registrierte Bewegung vom Ereignisstartpunkt, der in **Listing** 4.30 bestimmt wird, mindestens zurückgelegt haben muss. Die zurückgelegte Distanz wird in *Zeile 4+5* mit folgender Formel bestimmt.

$$distance = \sqrt{\Delta x^2 + \Delta y^2}$$

Wird eine Bewegungsdistanz von mindestens 100 Pixeln festgestellt, wird von einer Wischbewegung ausgegangen (*Zeile 7*) und in *Zeile 8* ein Kreis gegebenfalls verschoben. Wird das

ACTION_UP Ereignis in *Zeile 11* registriert, bedeutet das, dass die Interaktion mit dem Touchscreen beendet worden ist. Beträgt die zurückgelegte Distanz vom Berührungsstartpunkt weniger als 100 Pixel, so wird von einem einfachen Anwählen eines POI ausgegangen und in *Zeile 13* die Methodik der implementierten Lösung aus dem vorherigem Abschnitt verwendet. Zum Schluss wird die Liste von POIs, die angewählt worden sind, geleert.
Soll ein Kreis verschoben werden, wird er auf die am Touchscreen aktuell berührte Position platziert. Da ein Kreis aber nur innerhalb eines festgelegten Radius verschoben werden kann, muss die neue Kreisposition extra berechnet werden, falls die Wischbewegung den festgelegten Radius überschreitet (vgl. **Abbildung** 4.17).

Abbildung 4.17: Wischbewegung zum Verschieben überschreitet den festgelegten Radius

Die (x,y) Koordinaten des *onTouchEvents* müssen in das OpenGL ES Koordinatensystem-format umgerechnet werden. Es wird ein Koordinatensystem mit dem Kreismittelpunkt als Ursprung betrachtet. Abhängig davon, in welchem Quadranten sich die aktuelle Position der Wischbewegung befindet, werden die neuen (x,y) Koordinaten des Kreises mit den Sinus- und Kosinusfunktionen berechnet.

Listing 4.34: Bestimmen der Koordinaten des verschobenen Kreises in Pseudocode

```
1 movePOIonTouchEvent(x, y) {
```

```
2       poiToMove = listOfSelectedPOI.get(0);
3       coords = convertToGL(x,y);
4       touchedX = coords[0];
5       touchedY = coords[1];
6       poiX = calcXCoord(poiToMove);
7       poiY = calcYCoord(poiToMove);
8
9       Δx = poiX - touchedX;
10      Δy = poiY - touchedY;
11      hypotenuse = √(Δx² + Δy²);
12      if(hypotenuse > 100){
13          α = sin⁻¹(Δx / hypotenuse);
14          hypotenuse = 100;
15          if(1. Quadrant){
16              touchedX = cos(α) · hypotenuse + poiX;
17              touchedY = sin(α) · hypotenuse + poiY;
18          else if(2. Quadrant){
19              ...
20          }
21      }
22      moveX = touchedX;
23      moveY = touchedY;
24      toMove = true;
25  }
```

In *Zeile 2* wird der POI ausgewählt, welcher verschoben werden soll. Die (x,y) Koordinaten der aktuell berührten Stelle auf dem Touchscreen und des ausgewählten POI werden in *Zeile 3-7* bestimmt. Die Koordinaten der berührten Stelle werden wiederum so umgerechnet, sodass diese die Stelle markieren, wenn das Mobilgerät senkrecht gehalten werden würde. In *Zeile 9+10* wird jeweils die Differenz der x- und y-Koordinaten gebildet. Mithilfe der Differenzen wird in *Zeile 11* die Distanz zwischen Kreismittelpunkt und berührter Stelle berechnet. Ist die Distanz größer als 100 Pixel und somit die berührte Stelle außerhalb des Radius, innerhalb welchem der Kreis verschoben werden kann, müssen die neuen (x,y) Koordinaten weiter berechnet werden. Mit α in *Zeile 13* wird der Winkel zwischen *hypotenuse* und y-Achse bestimmt. Anschließend wird *hypotenuse* auf den Maximalwert von 100 gesetzt. Abhängig vom Quadranten, in welchem sich die berührte Stelle befindet, werden die (x,y) Koordinaten des zu verschiebenden Kreises berechnet. In *Zeile 22-24* werden die berechneten Koordinaten in öffentlichen Variablen gespeichert. Die Variable *toMove* gibt an, ob ein POI bewegt werden muss und wird während des Renderingprozesses zur Überprüfung verwendet.
Damit ein verschobener Kreis an seiner neuen Stelle anwählbar bleibt, werden die Werte, um die der Kreis in x- und y-Richtung verschoben worden ist, im Kreisobjekt gespeichert. Dazu werden zwei Variablen in der POI-Klasse deklariert.

Listing 4.35: Deklaration der onTouchEvent-Ereignsivariablen in Pseudocode

```
1  POI(x, y, radius, segments){
2      ...
3      toMoveX = 0.0f;
```

```
4       toMoveY = 0.0f;
5       ...
6   }
```

Die Variablen *toMoveX* und *toMoveY* werden mit einer Wertigkeit von 0 implementiert, da ein Kreis zu Beginn an seiner ursprünglichen Position gezeichnet wird und erst dann verschoben werden kann. In der *onDrawFrame()*-Methode der Rendererklasse werden die *toMoveX* und *toMoveY* Variablen in die Berechnung der Kreisposition mit einbezogen.

Listing 4.36: Überarbeitete Version der Positionierung eines Kreises in Pseudocode

```
1  onDrawFrame(){
2      ...
3      for (all p in listOfPOI){
4          if(toMove){
5              if(p equals POIsInRangeOfTouchEvent.get(0)){
6                  p = new POI(p.x, p.y, p.radius, p.segments);
7                  p.toMoveX = moveX;
8                  p.toMoveY = moveY;
9              }
10         }
11         x = calcXCoord(p) + p.toMoveX;
12         y = calcYCoord(p) + p.toMoveY;
13         α = calcRotation();
14
15         * p rotieren, verschieben und dann zeichnen
16     }
17     ...
18 }
```

Für jeden POI wird in *Zeile 4* überprüft, ob momentan eine Wischbewegung durchgeführt wird. Entspricht in *Zeile 5* der aktuell iterierte Kreis dem, der durch die Wischbewegung verschoben werden soll, wird der Kreis in *Zeile 6* neu erzeugt, um eine andere Farbe zu erhalten. In *Zeile 7+8* werden die in **Listing** 4.35 deklarierten Variablen neu gesetzt. Die Koordinaten des Kreises und der Rotationswinkel werden ermittelt, wobei die Koordinaten mit den Kreisvariablen *toMoveX* und *toMoveY* verrechnet werden, damit der Kreis an seiner verschobenen Position korrekt gezeichnet wird. Der Kreis wird, wie bereits in **Listing** 4.7 beschrieben, um den Winkel α rotiert, an den Koordinatenachsen verschoben und anschließend durch einen *draw()*-Aufruf gezeichnet.
Jetzt muss noch der verschobene Kreis mit seiner ursprünglichen Position verbunden werden.

Listing 4.37: Zeichnen einer Verbindungslinie in Pseudocode

```
1  onDrawFrame(){
2      for(all p in listOfPOI){
3          ...
4          if (p.toMoveX != 0.0f OR p.toMoveY != 0.0f) {
5              regularPos[] = convertCoord(x, y);
6              movedPos[] = convertCoord((x - p.toMoveX),
```

```
7                   (y - p.toMoveY));
8               startPoint[] = {regularPos[0], regularPos[1], 0f};
9               endPoint = {movedPos[0], movedPos[1], 0f};
10              GLLine line = new GLLine(startPoint, endPoint);
11              line.draw(mMVPMatrix);
12          }
13          ...
14      }
15      ...
16  }
```

Zu einem bestimmten Kreis *p* muss eine Verbindungslinie gezeichnet werden, falls in *Zeile 4* festgestellt wird, dass der Kreis verschoben worden ist. In *Zeile 5-9* werden die Koordinaten des Start- und des Endpunkts der Verbindungslinie bestimmt und als (x,y,z) Koordinaten gespeichert. Mithilfe der berechneten Start- und Endpunktkoordinaten wird die Verbindungslinie durch die Form *GLLine* gezeichnet.
Die Kreise für ein POI und der Radar werden mit der Zeichenart *GLES20.GL_TRIANGLES*, also aus Dreiecken geformt, gezeichnet. Mit OpenGL ES können aber auch einfache Linien mit der Zeichenart *GLES20.GL_LINE* gezeichnet werden.

Listing 4.38: Definition einer Linie in Pseudocode

```
1  class GLLine{
2      ...
3      GLLine(start[], end[]){
4          lineCoords = {
5              start[0], start[1], start[2],
6              end[0], end[1], end[2]};
7          ...
8      }
9      draw(){
10         ...
11         GLES20.glDrawElements(GLES20.GL_LINES, lineCoords.length/3,
12             GLES20.GL_UNSIGNED_SHORT, drawListBuffer);
13     }
14 }
```

Die Verbindungslinie wird in *Zeile 3* mit dessen Start- und Endpunktkoordinaten initialisiert. In *Zeile 5+6* werden die Koordinaten nacheinander in ein Array gespeichert. Das Array wird, wie in **Listing** 3.1 beschrieben, in einen FloatBuffer *drawListBuffer* umgeschrieben. In der *draw()*-Methode wird die Verbindungslinie in *Zeile 11+12* durch die Zeichenart *GLES20.GL_LINE* gezeichnet.
Ein Kreis wird nur solange in seiner verschobenen Position gezeichnet, bis entweder AREA vollständig beendet oder erneut in die Kameraansicht gewechselt wird. Dass ein Kreis in seine ursprüngliche Position zurückgesetzt wird, wenn in die Kameraansicht gewechselt wird, hat folgenden Grund: Wenn die Kameraansicht angefordert wird, wird die Rendererklasse neu initialisiert. Da die Rendererklasse alle Informationen über die Kreispositionen festhält, werden diese bei einer Neuinitialisierung verworfen.

Damit einzelne, verschobene Kreise auch manuell auf ihre ursprüngliche Position zurückgesetzt werden können, wird auf dem Informationsfenster eines POI ein Zurücksetzen-Button angezeigt (vgl. **Abbildung** 4.18).

Abbildung 4.18: Der Zurücksetzen-Button für einen verschobenen POI

Mit dem Zurücksetzen-Button kann der Benutzer einen angewählten Kreis auf seine ursprüngliche Position zurücksetzen. Dieser Button erscheint nur dann, wenn der Kreis verschoben worden ist und bewirkt die Neuinitialisierung des angewählten Kreises.

Listing 4.39: Zurücksetzen eines verschobenen Kreises in Pseudocode

```
1 onDrawFrame(){
2     ...
3     for(all p in listOfPOI){
4         ...
5         if(resetButtonClicked){
6             if(p equals poiToReset){
7                 p = new POI(p.x, p.y, p.radius, p.segments);
8                 updatePOIInList(p);
```

```
 9                  resetButtonClicked = false;
10              }
11          }
12          ...
13          p.draw();
14          }
15          ...
16  }
```

Bevor ein Kreis gezeichnet wird, wird überprüft, ob kürzlich der Reset-Button für den POI, welcher vom Kreis markiert wird, angewählt worden ist und somit der Kreis auf seine ursprüngliche Position zurückgesetzt werden muss. Da in OpenGL ES ein gezeichnetes Objekt nur über Methoden der Rendererklasse angesprochen werden kann, kann erst innerhalb der *onDrawFrame()*-Methode auf das ClickEvent reagiert werden. Dazu muss in *Zeile 6* für jeden gezeichneten Kreis überprüft werden, ob es dem selektierten Kreis entspricht und somit zurückgesetzt werden muss. Der Kreis wird in *Zeile 7* neu erzeugt, wodurch alle Informationen über die Verschiebung in x- und y-Richtung verworfen werden, und ersetzt in *Zeile 8* den alten Kreis in der Liste der sichtbaren Kreise. Ist der zu einem ClickEvent gehörende Kreis gefunden, ist eine diesbezügliche Überprüfung für weitere Kreise überflüssig, bis das nächste ClickEvent eintrifft (*Zeile 9*). Mit dem nächsten *draw()* Aufruf in *Zeile 13* wird der Kreis wieder an seiner ursprünglichen Position gezeichnet.

5
Fazit und Ausblick

Ziel dieser Arbeit war es, AREA mit OpenGL ES 2.0 umzusetzen und ein Endprodukt zu erschaffen, welches POIs an ihrer relativ genauen Position auf dem Display markiert und ein Lösungkonzept für die Clusterbildung beinhaltet. In der implementierten Version werden auf der Kameraansicht von AREA POIs durch Kreise markiert und ein dynamischer Radar als Hilfsmittel bereitgestellt. Der Benutzer kann mit den angezeigten Kreisen interagieren, indem er diese auf dem Touchscreen berührt oder mit einer Wischbewegung anders positioniert. Die mit OpenGL ES 2.0 realisierte Version von AREA läuft auf dem verwendeten Nexus 4 flüssig oder zumindest mit keinen erkennbar großen Performanceproblemen.
Während der Umsetzung von AREA wurde schnell deutlich, dass das Zeichnen von mehreren Objekten gleichzeitig die CPU des Mobilgerätes stark beanspruchen kann. In den ersten Versuchen wurden auf eintretende Sensorereignisse alle Kreise neu angelegt und gezeichnet. Wird ein Kreis neu angelegt, so werden all seine Eckpunkte neu berechnet. Diese Methode hatte zur Folge, dass selbst die geringe Anzahl von etwa zehn Kreisen innerhalb von wenigen Minuten zum Absturz des Mobilgerätes führten. Es galt also, das Neuberechnen von Objekten zu vermeiden und somit bereits erstellte Objekte zu speichern und wiederzuverwenden. Es ist durchaus möglich, dass die implementierte Version auf leistungsschwächeren Geräten wesentlich schlechter abschneidet, als auf dem in dieser Arbeit verwendeten Nexus 4. Für weitere Implementationserweiterungen für AREA wäre empfehlenswert, dass noch vor der ersten Verwendung von AREA die Leistung des verwendeten Mobilgerätes ermittelt wird und entsprechend eine maximale Anzahl an gleichzeitig angezeigten POIs festgelegt wird. So könnte vermieden werden, dass leistungsschwächere Geräte bereits beim erstmaligen Start von AREA abstürzen.
Die mangelnde Unterstützung von Textrendering stellte in der Umsetzung von AREA eine große Hürde da. Ein Kreis ohne Beschriftung hätte dem Zweck von AREA, POIs grafisch darzustellen, nicht ausreichend genügt. Die implementierte Lösung zur Textrenderingproblematik funktioniert zwar tadellos, ist in ihrer Nutzung aber eingeschränkt, indem beispielsweise nur eine festgelegte Anzahl Zeichen definiert und grafisch realisiert werden kann. Eine Änderung der Schriftfarbe und -größe ist aber möglich und könnte für zukünftige Erweiterungen von AREA verwendet werden, um beispielsweise dem Benutzer eine Einstellungsoption anzubieten, in welcher er die Schriftfarbe und -größe manuell beinflussen könnte.
Bestimmte Sensorergebnisse, z.B die -90°, die als Wert für den Rotationswinkel ausgegeben werden, wenn das Mobilgerät in normaler senkrechter Haltung gehalten wird, lassen die Vermutung aufkommen, dass die Sensorergebnisse vom verwendeten Mobilgerät abhängig sind. Diese Vermutung gilt zu klären, falls eine plattformübergreifende Implementierung vorgesehen sein sollte. Meine Idee für weitere geplante Erweiterungen von AREA wäre, dass beim erstmaligen Verwenden von AREA auf einem Mobilgerät Konfigurationstests durchgeführt werden. In den Konfigurationstest wird der Benutzer aufgefordert, bestimmte Haltungen mit dem Mobilgerät

einzunehmen. Die erfassten Sensordaten werden ausgelesen und zu einem Muster zusammengesetzt, das für alle weiteren Berechnungen verwendet wird, um best mögliche Ergebnisse zu erzielen.
Weiter bietet AREA eine Vielzahl an Erweiterungsmöglichkeiten. Dem Benutzer könnte neben einer manuellen Auswahl der Schriftgröße und -farbe auch freie Auswahl über die Farbe und Form von POI, Radar und Sichtfeld angeboten werden. Dazu müssten die verschiedenen Formen aber vorher definiert werden. Dem Benutzer könnte die Möglichkeit geboten werden, für ihn uninteressante POIs zu verbergen und somit nicht mehr sichtbar zu machen. Die Implementierung einer Sprachausgabe könnte die Verwendung von AREA auch für Benutzer mit eingeschränkter Sehkraft interessanter gestalten.
Trotz der zu Beginn mühsamen Einarbeitung in die Konzepte von OpenGL ES und der Android Applikationsentwicklung, hat mir das Herumprobieren mit OpenGL ES 2.0 bereits nach den ersten Erfolgserlebnissen, wie das Zeichnen und Verschieben eines Kreises, viel Spass gemacht. Allgemein fand ich das Arbeiten mit OpenGL ES 2.0 sehr interessant und habe bereits geplant, weiter in eigenen Projekten mit OpenGL zu arbeiten.

Literaturverzeichnis

[BAH09] Benstead, L. ; Astle, D. ; Hawkins, K.: *Beginning OpenGL Game Programming.* Boston : Course Technology PTR, 2009

[Clo15] Clockworkcoders: *Uniform Variables.* `https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/uniform.php`. Version: Oktober 2015

[Dev15a] Developers: *Applying Projection and Camera Views.* `http://developer.android.com/training/graphics/opengl/projection.html`. Version: September 2015

[Dev15b] Developers: *Displaying Graphics with OpenGL ES.* `http://developer.android.com/training/graphics/opengl/index.html`. Version: September 2015

[Dev15c] Developers: *GLSurfaceView.Renderer.* `http://developer.android.com/reference/android/opengl/GLSurfaceView.Renderer.html`. Version: September 2015

[Dev15d] Developers: *OpenGL ES.* `http://developer.android.com/guide/topics/graphics/opengl.html`. Version: September 2015

[fra15] fractious: *Rendering Text in OpenGL on Android.* `http://fractiousg.blogspot.de/2012/04/rendering-text-in-opengl-on-android.html`. Version: September 2015

[Gmb15] GmbH, Statista: *Anzahl der Smartphone-Nutzer in Deutschland in den Jahren 2009 bis 2015.* `http://de.statista.com/statistik/daten/studie/198959/umfrage/anzahl-der-smartphonenutzer-in-deutschland-seit-2010/`. Version: September 2015

[Gol15] Golem.de: *Augmented Reality - AR.* `http://www.golem.de/specials/augmented-reality/`. Version: August 2015

[GPSR13] Geiger, P. ; Pryss, R. ; Schickler, M. ; Reichert, M.: Engineering an Advanced Location-Based Augmented Reality Engine for Smart Mobile Devices. Technical Report. University of Ulm. 2013.

[GSP+14] Geiger, P. ; Schickler, M. ; Pryss, R. ; Schobel, J. ; Reichert, M.: *Location-based Mobile Augmented Reality Applications: Challenges, Examples, Lessons Learned.* In: 10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps, Barcelona, Spain, April 3-5, 2014, pp. 383-394.

[Kod15] Kodzhabashev, Aleksandar: *Rendering Text in OpenGL ES 2 on Android.* `https://github.com/d3kod/Texample2`. Version: September 2015

[Ora15] Oracle: *Java docs.* `http://docs.oracle.com/javase/7/docs/api/java/nio/ByteOrder.html`. Version: September 2015

[Pro15] Programming, Computer: *An Efficient Way to Draw Approximate Circles in OpenGL.* `http://slabode.exofire.net/circle_draw.shtml`. Version: September 2015

[SPSR15] Schickler, M. ; Pryss, R. ; Schobel, J. ; Reichert, M.: *An Engine Enabling Location-based Mobile Augmented Reality Applications.* LNBIP 226, Springer, In: 10th International Conference on Web Information Systems and Technologies (Revised Selected Papers). pp. 363-378.

[Ulm15] Ulm, Universität: *Augmented Reality Engine Application.* `http://www.area-project.info`. Version: August 2015