ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Untersuchung und Analyse von OpenGL zur Anwendung und Optimierung in AREA

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Andreas Schmid
andreas-1.schmid@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Rüdiger Pryss

2014

Fassung 10. Dezember 2014

Satz: PDF-LaTeX 2$_\varepsilon$

Inhaltsverzeichnis

1

Einleitung

Mobile Systeme, wie beispielsweise Smartphones, unterstützen uns in der heutigen Zeit in fast allen Lebenslagen [RPR11, PLRH12, PMLR14]. Während die Anzahl der Möglichkeiten für neue Applikationen zunimmt [SSP+14, SHP+14, RLPL+13, IRLP+13, CNB+13, SSP+13, SRLP+13, PTKR10], steigen die benötigten Hardware Voraussetzungen, um die entworfenen Programme mit einer hohen Anzahl von Bildern pro Sekunde für den Benutzer flüssig darzustellen. Im Vergleich zu älteren Generationen sind heutige Smartphones mit leistungsstärkeren Prozessoren und Grafikprozessoren ausgestattet, die es den Entwicklern und den Benutzern ermöglichen, selbst Hardware-hungrige Anwendungen flüssig darzustellen. Immer häufiger können deshalb Entwickler Anwendungen schreiben, die bis an die Grenzen des Darstellbaren vorstoßen. Um die ganze Leistung der Hardware zu nutzen, können Entwickler direkt auf die Leistung des Grafikprozessors (kurz: GPU) zugreifen. Dies ermöglicht es, das Potential der GPU voll zu nutzen und somit eine bessere Leistung zu erreichen.

Viele Applikationen, die auf heutigen Smartphones verwendet werden, benutzen eine Vielzahl an unterschiedlichen Eingabesensoren, um ihre Funktion korrekt auszuführen. Ein Beispiel hierfür ist eine Art, die sich Augmented Reality Apps nennt. Diese verwenden Sensoren, um die Lage des Smartphones zu erkennen und danach Objekte auf dem Display darzustellen. Das Projekt AREA ist ein Augmented Reality App [GSP+14, GPSR13, Gei12], die dem Benutzer Informationspunkte (Points of Interest) auf dem Display ausgibt. Diese Punkte werden bei Bedarf aus dem Speicher oder dem Internet abgerufen.

Die vorliegende Arbeit behandelt die Umsetzung von AREA mit Open Graphics Library for Embedded Systems (kurz: OpenGL ES) [Khr14]. Mit dieser Grafik-Bibliothek soll überprüft werden, ob eine Umsetzung des Systems möglich ist und welche Probleme dabei entstehen. Weitergehend ist zu prüfen, ob OpenGL ES nützlich für die Optimierung der Anwendung ist.

1.1 Motivation

OpenGL ist eine Open Source Spezifikation zur Entwicklung von 2D und 3D Computergrafikanwendungen. Ein Ableger davon ist die Spezifikation OpenGL ES (Open Graphics Library for Embedded Systems), die für den Bereich der eingebetteten Systeme konzipiert ist. Mittels dieser Programmierschnittstelle lässt sich durch eine reduzierte Anzahl an Befehlen die Grafikkarte direkt ansprechen. OpenGL ES kann bei richtiger Implementierung den CPU-Overhead reduzieren, wodurch schwächere CPUs einen deutlichen Vorteil erhalten [App14]. Dies gilt sowohl für die Plattform IOS, wie auch für Android. Fraglich ist, ob der gegebene Code eine einfache Erweiterung zulässt und welche Problemstellen gefunden werden, die für ähnliche Projekte relevant sein könnten.

1.2 Aufbau der Arbeit

Bevor mit der Umsetzung der Erweiterung an AREA begonnen wird, werden die Grundlagen erläutert. Zu diesen zählt die Anwendung AREA, die verwendete Technik Augmented

Reality und die für die Erweiterung gewählte Programmierschnittstelle OpenGL. Danach werden ähnliche Anwendungen ermittelt und anschließend die Anforderungen an die Erweiterung gestellt. Im Kern der Arbeit wird die Implementierung von OpenGL ES in AREA erörtert und die Problemstellen aufgezeigt. Anschließend wird die Applikation vorgestellt und mit den bereits gestellten Anforderungen verglichen.

2

Grundlagen

In diesem Kapitel wird die Grundlage Augmented Reality (Erweiterte Realität, kurz: AR) und die darauf aufbauende Augmented Reality Engine Application (kurz: AREA) erläutert [Gei14]. Außerdem wird auf OpenGL eingegangen, welche eine Spezifikation und Programmierschnittstelle von Computergrafikanwendungen darstellt. Mit dem Ableger davon, OpenGL für eingebettete Systeme, soll AREA erweitert werden [Gro14a].

2.1 Augmented Reality

Augmented Reality ist im visuellen Anwendungsgebiet die Erweiterung der Realitätswahrnehmung, die das Sichtfeld des Benutzers mit Informationen erweitert. AR spricht theoretisch für alle Sinnes-Modalitäten, wird aber häufig nur mit visuellen Informationen in Verbindung gebracht. Diese Informationen können von reinem Text bis hin zu

Abbildung 2.1: Ein Head-Up-Display, wie es in Düsenflugzeuge eingesetzt wird. Grafik aus [Fau14].

komplexen 3D-Objekten reichen. Wichtig dabei ist, dass das Sichtfeld des Benutzers nicht ersetzt, sondern lediglich mit Informationen erweitert wird. Der Benutzer soll durch diesen Mix aus realen und generierten Informationen eine neue Realität erfahren.

Im Bereich der visuellen AR lassen sich nochmals Unterschiede aufzeigen, welche sich hauptsächlich durch ihre verwendeten Endgeräte klassifizieren lassen. Ein Beispiel im Bereich der Navigation stellen die sogenannten *projection displays* oder auch Head-Up-Displays dar [ABB+01]. Bis jetzt wurden diese größtenteils vom Militär für Kampfflugzeuge verwendet (siehe Abbildung 2.1). Ebenfalls zeigt der Automobilmarkt Interesse an diesem Gebiet. Viele Autohersteller entwickeln ihre eigene Version eines Head-Up-Displays für ihre Marken (siehe Abbildung 2.2). Die Head-Up-Displays sind durchsichtige Anzeigen, die vor das Sichtfeld des Benutzers platziert werden. Dort geben sie dem Nutzer relevante Informationen, ohne seine Aufmerksamkeit von wichtigen Aufgaben, wie dem Fahren oder dem Fliegen, abzulenken. Ein weiteres Endgerät welches für AR benutzt werden kann, ist das Smartphone. Smartphones besitzen eine Vielzahl an Sensoren und Eingabemöglichkeiten, um die AR für den Benutzer interaktiv zu gestalten. Die Kamera kann verwendet werden, um das Sichtfeld des Benutzers aufzunehmen und das Display ermöglicht es den aufgenommenen Video-Stream auszugeben. Im

Abbildung 2.2: Ein Head-Up-Display, wie es in einem Auto eingesetzt werden kann. Grafik aus [BMW14].

Display können relevante Informationen angezeigt werden, die aus den Sensordaten berechnet werden. Zum Beispiel kann eine Anwendung auf einem Smartphone (kurz: App) dem Benutzer Punkte von Interesse (Points of Interest, kurz: POI) anzeigen. POI können Landschaftsmarkierungen sein, sodass der Nutzer auf seinem Display angezeigt bekommt, in welcher Richtung sich dieser POI befindet.

2.2 AREA

AREA ist eine Smartphone-App, die vorhandene POI dem Nutzer in einer AR visualisiert [Gei14, Gei12]. AREA besteht aus zwei großen Komponenten. Die erste Komponente ist die Kartenansicht, welche die POI auf einer Karte in einer Vogelperspektive anzeigt. Diese POI reagieren auf einen Klick des Benutzers und öffnen darauf ein Informationsfenster, das weitere Informationen über den POI anzeigt. Der zweite Bestandteil ist die Kamera-Ansicht, die den AR-Teil der App darstellt. In diesem Teil wird das Sichtfeld durch die Kamera aufgenommen und durch die im Blickfeld des Benutzers befindlichen POI erweitert. Beim Klick auf diese angezeigten POI öffnet sich ähnlich wie in der Kartenansicht ein Informationsfenster, das weitere Informationen über den gewählten POI aufzeigt. Damit der Benutzer ein Gefühl dafür bekommt, wo sich weitere POI befinden, besitzt die Kamera-Ansicht ein Radar. In diesem wird angezeigt, in welcher Himmelsrich-

tung sich weitere POI befinden. Der Benutzer hat außerdem die Möglichkeit den Radius, in dem POI als kleine Punkte angezeigt werden, zu verändern. Dies erweist sich als hilfreich wenn sich zu viele POI in der Nähe des Benutzers befinden. AREA ist für die Software-Plattform Android und iOS implementiert, wodurch die einzelnen Komponenten auch in den Plattform-spezifischen Views realisiert sind. Diese Views sind bei Android und iOS die Oberflächenkomponenten, die für die Darstellung zuständig sind.

2.3 OpenGL ES

Für die neue Darstellung von AREA wird OpenGL ES 2.0 (Open Grafics Library for Embeded Systems, kurz: OpenGL ES) verwendet [Khr14]. OpenGL ES ist eine Vereinfachung der ursprünglichen Programmierschnittstelle und besitzt ähnlich wie OpenGL Befehlssätze, um 2D oder 3D Grafiken darzustellen. Der Unterschied liegt hauptsächlich in der Reduzierung der redundanten Befehlssätze und dem Wegfallen des Datentyps Double. Die betroffenen Funktionen sind in OpenGL ES mit einem Float-Parameter anzusprechen. OpenGL ES 1.0 kann in Android mit jeder Version verwendet werden [Goo14b] und OpenGL ES 2.0 ab dem API-Level 8, was für die Abdeckung von 77,5 Prozent der Geräte im Umlauf spricht [Goo14a]. Die Version 3.0 von OpenGL ES kann ab dem API-Level 18 verwendet werden, was die Zahl der kompatiblen Smartphones auf 22,5 Prozent senkt. OpenGL ES verwendet wie fast alle Grafikprogrammierschnittstellen eine Renderpipeline.

Die OpenGL ES 2.0 Renderpipeline beschreibt den Weg von einzelnen Vektoren oder Punkten bis hin zum fertig gezeichneten Bild [Bro13]. Die Hauptaufgabe besteht darin, die Punkte an die mathematisch korrekte Position zu verschieben und nach dem Rastern die zusammenhängenden Flächen, bestehend aus Vertices (Einzahl: Vertex) [MGS08, Gro14c], mit Farbe auszufüllen. Die zusammenhängenden Flächen liegen in Form eines Vertex-Arrays vor. Dieses enthält eine Gruppe von Vertices, welche das zu zeichnende Objekt beschreiben. Im ersten Schritt werden die einzelnen Punkte mithilfe des Vertex-Shaders an die gewünschte Position verschoben. Ein Shader ist ein Programm, das in OpenGL Shader Language (kurz: GLSL) geschrieben ist und dem Grafikprozessor (kurz:

GPU) vorschreibt, wie er die Transformation des einzelnen Vertex vornehmen soll. Ist die Position des Punktes bestimmt, muss er zuerst gerastert werden, um einem Pixel auf dem verwendeten Display zu entsprechen. Im Anschluss wird die Position an den Fragment-Shader weitergegeben. Der Fragment-Shader ist ebenfalls ein Programm für die GPU zur Berechnung eines oder mehreren Vertices. Der Unterschied ist, dass der Fragment-Shader keine Positionsbestimmung durchführt, sondern Teile des Displays in sogenannte Fragemente unterteilt. Aus den berechneten Vertex-Positionen werden dann die einzelnen Fragmente mithilfe dieses Programms eingefärbt. Als Beispiel können drei Vertices ein Dreieck beschreiben, dass im Fragment-Shader rot eingefärbt wird. Im Anschluss wird das erzeugte Bild gespeichert und in den Framebuffer geladen sowie auf dem Display angezeigt. Zwischen diesen einzelnen Schritten werden noch interne Schritte verarbeitet, die für den Entwickler aber von geringerer Bedeutung sind.

3

Verwandte Arbeiten

In diesem Kapitel werden ähnliche Arbeiten vorgestellt, die ebenfalls die Themen Augmented Reality und OpenGL behandeln. Alle vorgestellten Arbeiten sind in dem Google Play Store erhältlich und zeigen dem Benutzer, ähnlich wie AREA, POI in seiner Umgebung.

3.1 Wikitude - Wikitude SDK

Die erste vorgestellte Arbeit nennt sich Wikitude [Wik14b] (siehe Abbildung 3.1). Sie ist im Android Store unter gleichem Namen zu finden und wird ebenfalls als Software Developer Kit (kurz: SDK) für zahlreiche Plattformen angeboten [Wik14a]. Wikitude ist eine Augmented Reality App, die dem Benutzer per Kamera-Ansicht hilft Hotels, Restaurants oder beliebte Reiseziele für Ortsfremde zu finden. Interessante Orte werden

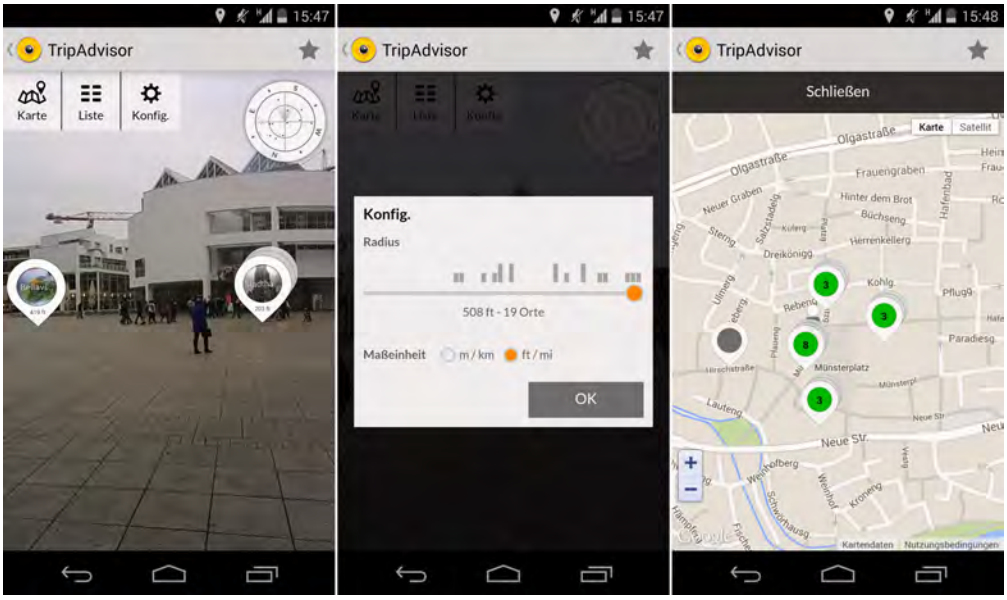

Abbildung 3.1: Verschiedene Ansichten der Wikitude App. Links ist die Kamera-Ansicht. Mitte ist die Auswahl des Radius. Rechts ist die Map-Ansicht.

wie in AREA als POI angezeigt und aus einer Onlinebibliothek geladen. Die Anwendung startet in einer Kamera-Ansicht, in welcher der Nutzer auswählen kann, welche POI er angezeigt haben möchte. Die App verwendet OpenGL ES 2.0 als Grafikschnittstelle, um die auf dem Display dargestellten Objekte zu zeichnen.

3.2 AugmentedMap - Metaio SDK

Eine weitere Arbeit, die ebenfalls OpenGL ES 2.0 verwendet, ist AugmentedMap [Aug14] (siehe Abbildung 3.2). Diese App visualisiert, wie Wikitude, verschiedene POI in der Umgebung des Benutzers. Auch diese Anwendung verwendet ein SDK als Grundlage für die Berechnung der Lage und der Anzeige. Das SDK mit dem Namen Metaio ist kommerziell erhältlich und ist für verschiedene Plattformen verfügbar. Im Vergleich zu Wikitude ist die Kamera-Ansicht sehr verwaschen. Außerdem gibt es in dieser App kein Infofenster, das nähere Informationen zu den einzelnen POI anbietet. Wird ein POI angeklickt, wird er lediglich auf dem Radar als ausgewählt markiert.

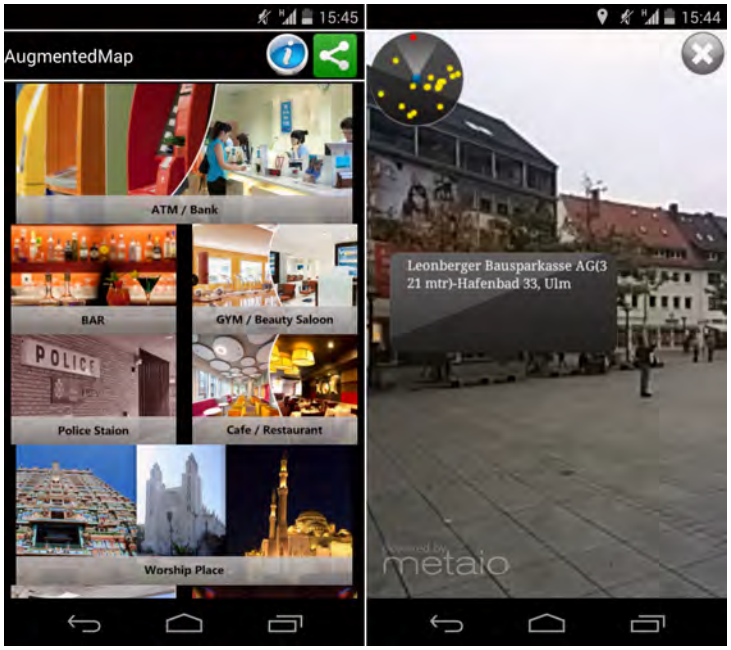

Abbildung 3.2: Die AugmentedMap App. Links können verschiedene Modi gewählt werden, die dann wie rechts in der Kamera-Ansicht dargestellt werden.

Beide Anwendungen bauen auf bereits existierende SDKs auf, die es den Entwicklern ermöglichen, mit vordefinierten Methoden auf die Sensordaten der Geräte zuzugreifen. Solche SDK können dem Entwickler viel Zeit und Arbeit ersparen, können ihn aber auch in seiner Arbeit einschränken. Für die Umsetzung in AREA wird kein bereits existierendes SDK mit OpenGL verwendet, um mögliche Problemstellen zu erkennen und gegebenenfalls Lösungen dafür zu entwickeln.

4

Projektanforderungen

Der Hauptaugenmerk dieser Arbeit liegt auf der Realisierung der Kamera-Ansicht von AREA in OpenGL ES 2.0. Es wird versucht alle bereits existierenden Funktionen des Originals umzusetzen. Im Folgenden werden die Anforderungen an die neue Umsetzung beschrieben.

4.1 Anforderungen an die Umsetzung mit OpenGL

Die Umsetzung soll POI dem Benutzer relativ zu seiner Position anzeigen. Zum einen in der Karten-Ansicht, bei der die POI aus der Vogelperspektive gezeichnet werden und zum anderen in einer Kamera-Ansicht. Diese Kamera-Ansicht soll POI anzeigen, sobald sie in das Sichtfeld der Kamera kommen. Die POI sollen an der korrekten Position auf dem Display dargestellt werden, relativ zu der Position des Benutzer. Die Daten der

Beschleunigungs-, GPS- und Magnetfeld-Sensoren sollen korrekt ausgewertet werden und eine stabile Bestimmung der Lage ermöglichen. Die Lage des Benutzers soll in Echtzeit erfasst und übernommen werden. Der Benutzer soll einen Radius wählen können, in dem die POI dargestellt werden. Klickt der Benutzer auf ein POI, sollen weitere Informationen zu dem gewählten POI erscheinen. Möchte der Benutzer sein Handy entweder im vertikalen Portrait-Modus oder im horizontalen Landscape-Modus verwenden, soll beides möglich sein und für ihn keine Einschränkungen gelten. Da die Auswertung der Sensordaten sehr häufig auftritt, soll die Berechnung und die Auswertung effizient ablaufen. Gleiches gilt für die Darstellung der POI. Die Anwendung soll stabil laufen und eine gute Wartbarkeit aufweisen. Der Code soll verständlich kommentiert sein, damit eventuell weitergehende Entwicklungen einfach realisierbar sind.

4.2 Einordnung der Anforderungen

Alle Anforderungen sind in Tabelle 4.1 aufgelistet und mit einem jeweiligen Typ versehen.

Tabelle 4.1: Tabellarische Aufzählung der Anforderungen.

Anforderung	**Typ**
POI auf Kamera-Ansicht anzeigen	funktional
POI auf Karten-Ansicht anzeigen	funktional
POI in Kamera-Ansicht nur anzeigen, wenn im Sichtfeld	funktional
POI in Kamera-Ansicht und Karten-Ansicht sollen auf Interaktionen reagieren	funktional
Sensordaten zur Positionierung des Smartphones auslesen (Beschleunigung, GPS, Magnetfeld)	funktional
Bei Bewegung des Smartphones Daten und POI in Echtzeit aktualisieren	funktional
Einstellbarer Radius für die Entfernung	funktional
Radar in Kamera-Ansicht mit POI aus der Umgebung und im Radius	funktional
POI Informationen anzeigen wenn ausgewählt	funktional
Portrait- und Landscape-Modus	funktional
Hohe Effizienz von Berechnungen	nicht-funktional
Hohe Effizienz beim Zeichnen der Anzeige	nicht-funktional
Hohe Stabilität	nicht-funktional
Hohe Genauigkeit	nicht-funktional
Gute Wartbarkeit	nicht-funktional
Einheitliche Spezifikation von POI	Implementierung
POI sollen intern erweiterbar sein	Implementierung
Einfache Einbindung in andere Anwendungen (Modularität)	Implementierung
Verständliche Kommentierung	Dokumentation

5

Architektur

In diesem Kapitel wird die Architektur der erweiterten Augmented Reality Engine Application (AREA) vorgestellt. Es wird auf die verwendeten Klassen eingegangen und die Kommunikation der einzelnen Komponenten angesprochen. Zum Schluss wird noch die neue Datenhaltung erläutert.

5.1 Architekturentwurf

Die Architektur der erweiterten AREA mit OpenGL besteht aus drei wesentlichen Bestandteilen. Ein Controller, der wie im ursprünglichen AREA Projekt die Sensordaten verwertet und entscheidet, ob ein POI gezeichnet werden muss. Es wird überprüft, in welche Richtung der Benutzer blickt und welchen Grad die vertikale Blickrichtung beträgt, um festzustellen, in welcher Lage sich das Smartphone befindet. Zusätzlich wird

überprüft, ob die im Blickwinkel liegenden POI in einem vom Benutzer ausgewählten Radius liegen. Liegen die POI innerhalb des gewählten Radius, werden sie gezeichnet. Teile des ursprünglichen *LocationControllers* sind etwas umgeschrieben, um eine genaue Position für die Objekte, die gezeichnet werden müssen, zu bestimmen. Dies ist notwendig, da OpenGL ein anderes Koordinatensystem verwendet als das ursprünglich von Android implementierte *Relative-Layout*.

Der zweite Teil ist das Model, das für die korrekte Erschaffung der Objekte in der View verantwortlich ist. In ihm sind wesentliche Grundbausteine vordefiniert. Diese können abgerufen werden, um verschiedene Formen vorzudefinieren. Die Formen werden dann erstellt und dem dritten Bestandteil, der View, übergeben.

In der View werden alle Daten, die von dem Controller und dem Model kommen, zu einem einheitlichen Darstellungscode gebündelt. Die View ist allein mit der Aufgabe beschäftigt eingehende Objekte in die Renderpipeline zu schieben, damit sie von der GPU auf dem Display gezeichnet werden können. Auf der View entsteht somit das endgültige Produkt, welches der Benutzer später zu sehen bekommt.

5.2 Kommunikationsablauf

Der wichtigste Teil von OpenGL ist der Renderer, der von verschiedenen Stellen Eingaben erhält, um das gewünschte Bild zu erschaffen. Damit der Renderer vernünftig arbeiten kann, muss eine Kommunikation zwischen den beiden Controllern (*LocationController* und *SensorController*) und der *AREAMainActivityOpenGL* von statten gehen. Diese Aktivität bündelt dann die Daten und übergibt sie dem Renderer (siehe Abbildung 5.1). Sobald sich die Position des Smartphones verändert, sendet der *SensorController* ein Update an den *LocationController*. Dieser holt sich dann alle POI aus dem Store, die innerhalb des gewählten Radius liegen. Er berechnet die Position des POI auf dem Display und speichert diese ab. Nun werden die POI an die *AREAMainActivityOpenGL* gesendet. In der *AREAMainActivityOpenGL* werden die POI auf ihre Position überprüft. Liegt ein POI innerhalb der Blickrichtung der Kamera wird aus dem POI ein *OGLObject* erstellt und in einer Liste zwischengespeichert. Dieses Objekt ist nun mit der genauen

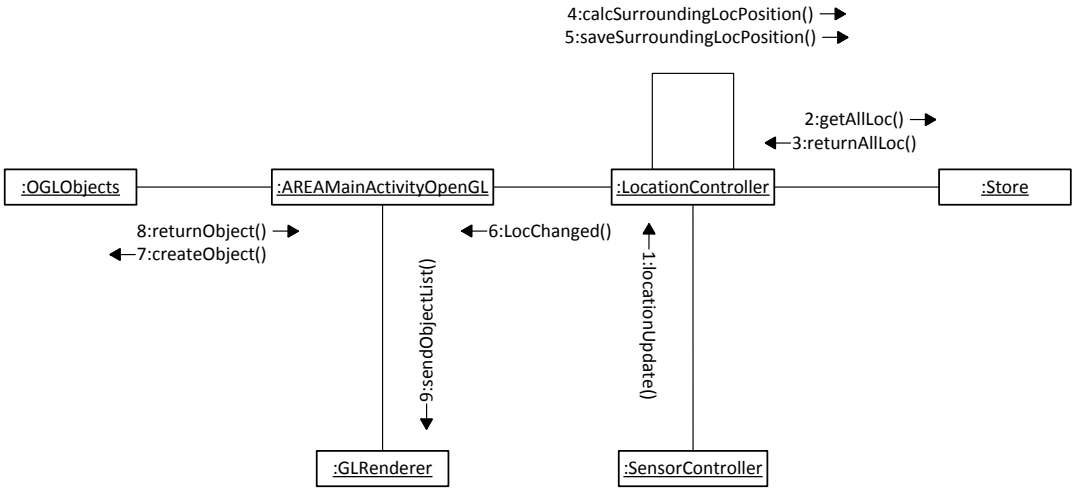

Abbildung 5.1: Kommunikationsdiagramm, welches den Ablauf beim Zeichnen eines Bildes bestimmt. Angestoßen wird diese Sequenz sobald sich die Position des Smartphones ändert.

Position und den objektspezifischen Werten erstellt und bereit gezeichnet zu werden. Ist die Liste vollständig, wird sie an den Renderer geschickt, um dort verarbeitet zu werden. Im Renderer werden dann die einzelnen Objekte gezeichnet.

5.3 Datenhaltung

AREA ist ursprünglich so erstellt worden, dass es eine einheitliche und einfache Schnittstelle besitzt, in der POI erstellt und angeboten werden können. Um so wenig wie möglich an dieser Eigenschaft zu verändern, wurden in dem ursprünglichen Code nur triviale Modifikationen vorgenommen. Diese Abänderungen ermöglichen es der ursprünglichen App, wie gewohnt verwendet werden zu können. Es wurden lediglich zwei Attribute eingefügt, die während der Positionsbestimmung aktualisiert werden. Die Klasse POI ist um die Attribute *GLPointPosX* und *GLPointPosY* ergänzt (siehe Abbildung 5.2). Diese dienen dem Renderer später als Angabe für die korrekte Position auf der *OpenGLSurfaceView*.

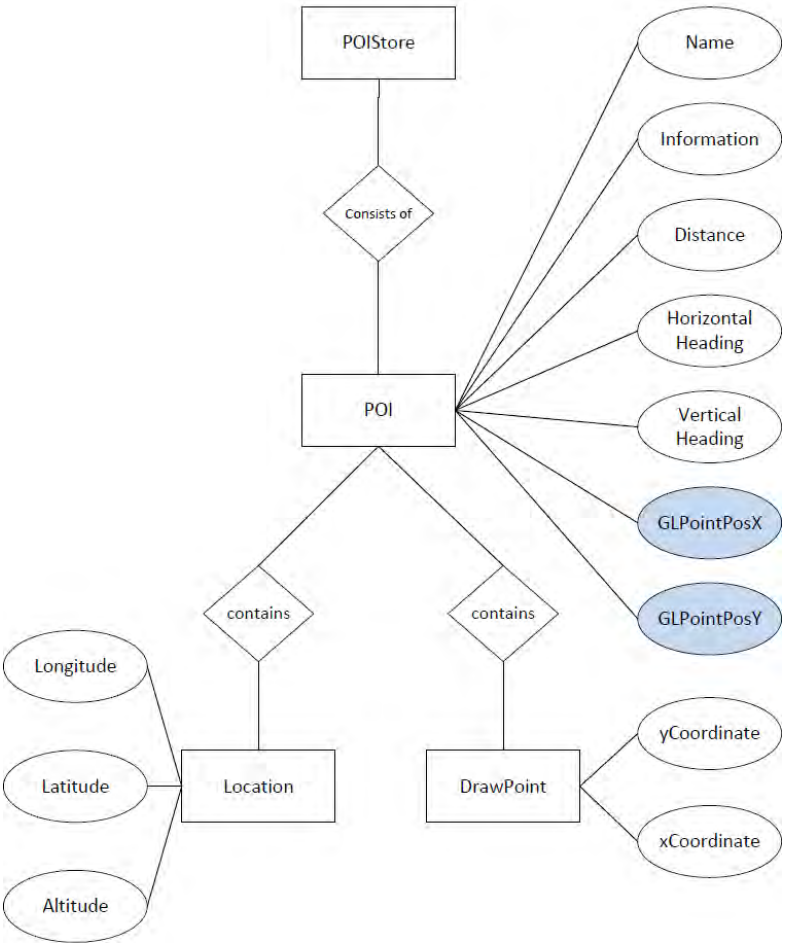

Abbildung 5.2: ER-Diagramm eines POI mit allen Attributen die es enthält. Blau dargestellt sind ergänzte Attribute, die für die spätere Berechnung der Position relevant sind.

6

Implementierung

Die Erweiterung von AREA ist auf der Grundlage von Android implementiert, welches mit Java programmiert ist. OpenGL ES 2.0 kann mit Java in den Android Code eingebettet werden, wobei ein kleiner Teil in OpenGL Shading Language (kurz: GLSL) geschrieben ist [Gro14b]. GLSL ist eine C-ähnliche Programmiersprache, die es ermöglicht sogenannte Shader für die Renderer zu schreiben. Als Entwicklungsumgebung ist „Eclipse IDE for Android Developers“ in der Version 23 verwendet worden. Für die Kompilierung und die Fehlersuche wird ein Nexus 5 Smartphone mit Android 4.4.4 verwendet. Der Emulator von Google konnte nach mehrmaligen Versuchen nicht reibungslos verwendet werden, weswegen alle Tests auf dem Nexus 5 ausgeführt wurden. AREA ist des Weiteren so erweitert, dass Benutzer mit einem OpenGL-inkompatiblen Smartphone auf die alte Darstellung zurückgreifen können.

In diesem Kapitel werden die wichtigsten Komponenten von AREA mit OpenGL ES 2.0 vorgestellt. Außerdem werden Probleme und verworfene Prototypen besprochen,

die während der Entwicklung zustande gekommen sind. Dazu gehört die prototypische Umsetzung der Kamera-Ansicht in OpenGL, der *LocationController* der die Umrechnung der Koordinaten verarbeitet und die *AREAMainActivity*, die den Fluss der Daten kontrolliert. Danach wird die Erstellung der Objekte besprochen, die im Renderer gezeichnet werden. Zum Schluss werden die Renderer erläutert.

6.1 Prototyp Kamera

Ein Grundbaustein von AREA ist die Kamera-view. Diese visualisiert, im Hintergrund der Anwendung, den Blickwinkel des Benutzers. Über dieser View wird das User Interface und die POI dargestellt. Zu Beginn der Entwicklung ist einen Prototyp erstellt worden, der testen sollte, ob mithilfe einer Umsetzung in OpenGL ein Nutzen entstehen kann (siehe Abbildung 6.1). Es wird der Kamera-Stream von Android aufgegriffen und daraus eine 2D-Textur erstellt. Diese Textur wird auf zwei rechteckige Dreiecke, die den kompletten Bildschirm abdecken, projiziert. Dreiecke sind notwendig, um ein Viereck in OpenGL darzustellen. Dies ist erforderlich, da OpenGL ES 2.0 keine komplexeren Objekte als Dreiecke zeichnen kann. OpenGL ES 2.0 ermöglicht es mit vier Vertices ein Rechteck zu zeichnen, wobei die verwendete Technik dieselbe ist. Auf dem Rechteck, kann die Textur des Kamera-Streams gezeichnet werden. Ein möglicher Fehler hierbei ist die falsche Positionierung der Textur. Da die Y-Achse der Textur in OpenGL ES 2.0 links unten beginnt und normale Bilder, die auf dem Speicher liegen, links oben beginnen, muss der Zeichenvorgang angepasst werden (siehe Abbildung 6.2). Im Regelfall verwenden Grafiken oder Texturen eine negative Y-Achse. Diese beginnt links oben bei (0,0) und erstreckt sich in Richtung rechts unten (x-Maximal,y-Maximal). Das Koordinatenfeld in OpenGL ES 2.0 für Texturen ist so aufgebaut, dass x und y links unten mit (0,0) beginnen und in Richtung rechts oben größer werden.

Der Test des Prototypen zeigte, dass im Vergleich zur nativen Kamera-Preview, das Android von Haus aus mitliefert, die Bildrate langsamer ist. Aufgrund dieser Beobachtung ist AREA mit OpenGL mit der nativen Android Kamera-Preview weiterentwickelt.

Abbildung 6.1: Der Kamera-Preview-Prototyp. Seine Aufgabe ist es, den Datenfluss der Kamera auf das Display zu zeichnen.

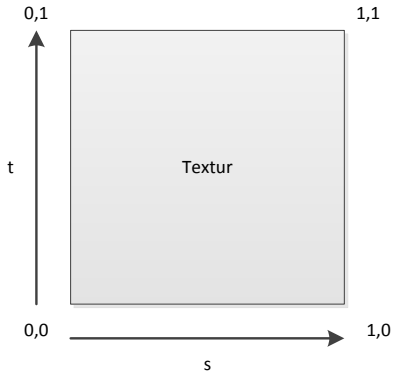

Abbildung 6.2: Koordinatensystem für Texturen in OpenGL. Es beginnt links unten bei (0,0) und expandiert in Richtung rechts oben bis (1,1). Die beiden Werte s und t bezeichnen die beiden Achsen der Textur. Im Regelfall verwenden Bilder oder Texturen ein anderes Koordinatensystem, bei dem die T-Achse (Y-Achse) gespiegelt ist.

6.2 LocationController und Datenhaltung

Der *LocationController* berechnet für alle POI die Abweichung in Grad vom Mittelpunkt des Sichtfeldes des Benutzers, in horizontaler wie auch in vertikaler Richtung. Das Koordinatensystem, mit dem die ursprüngliche Anwendung arbeitet, erstreckt sich von links unten bei (0,0) bis rechts oben bis (max,max). Da OpenGl ES 2.0 intern ein normiertes Koordinatensystem verwendet, welches sich aus der Mitte erstreckt, müssen die Koordinaten vor der Verwendung transformiert werden. Um die originale Anwendung so wenig wie möglich zu verändern, ist in die Routine des LocationControllers eine kleine Umrechnung implementiert, welche auf die ursprünglichen Koordinaten zugreift und diese in das OpenGL ES 2.0 Koordinatensystem umrechnet. Dadurch kann der *OGLLocationViewRenderer* ohne erheblichen Aufwand die Objekte an die richtige Position zeichnen (siehe Abbildung 6.3). Die x-Position und die y-Position werden innerhalb der POI-Klasse gespeichert. Somit ist es weiterhin möglich, diese Klassen für die Benutzung von AREA ohne OpenGL ES 2.0 zu verwenden.

Der ursprüngliche Punkt wird in das neue Koordinatensystem transformiert (siehe Listing 6.1). Die Umrechnung nimmt den unveränderten Achsenwert und normiert diesen. Danach wird er verdoppelt und an den Ursprung verschoben. Der Y-Achsenwert wird negiert um Rücksicht auf die gespiegelte Y-Achse von OpenGL ES 2.0 zu nehmen. Der Wert wird in den jeweiligen POI gespeichert und beim Erstellen der OGLObjects benutzt, um das Objekt korrekt zu platzieren.

Listing 6.1: Pseudocode der Konvertierung im *LocationController*.

```
1 GLPointX = PointX / originalDrawscreenSize;
2 GLPointX = ( GLPointX * 2 ) - 1;
3 poi.setGLX(GLPointX);
4
5 GLPointY = PointY / originalDrawscreenSize;
6 GLPointY = ( GLPointY * -2 ) + 1;
7 poi.setGLY(GLPointY);
```

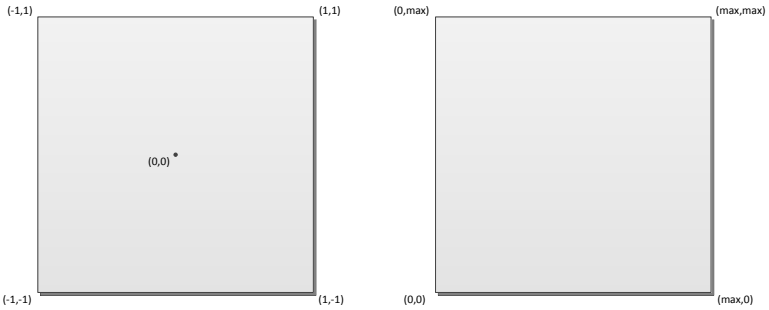

Abbildung 6.3: Gegenüberstellung der Koordinatensysteme. OpenGL ES 2.0 links, Android-RelativeLayout rechts. OpenGL besitzt ein normiertes Koordinatensystem, während Android von 0 bis zu einem maximalen Wert geht.

6.3 AREAMainActivityOpenGL

Die *AREAMainActivity* von AREA mit OpenGL ES 2.0 steuert die grundlegenden Aktionen der Anwendung. Sie ist zuständig für die Initialisierung der verschiedenen Renderer, welche die verschiedene Teile des User Interface zeichnen. Ein Renderer ist der *OGLLocationViewRenderer*, dessen Aufgabe es ist POI zu zeichnen, die im derzeitigen Blickfeld des Benutzers liegen. Der *OGLRadarPointsRenderer* zeichnet ein Teil des Radars in der linken oberen Ecke. Ein Renderer mit ähnlicher Aufgabe ist der *OGLRadarViewPortRenderer*, dessen Arbeit allein darin besteht, das Sichtfeld des Benutzers auf den Radar zu zeichnen. Eine weitere Möglichkeit wäre gewesen, alle Renderer in einem zu kombinieren. Sind diese jedoch separat, lassen sich einzelne Transformationen einfacher bewerkstelligen. Ein weiteres Argument hierfür ist, dass die Platzierung der POI einfacher vonstatten geht, wenn der *OGLLocationViewRenderer* die gleichen Maße wie die ursprüngliche Zeichenfläche besitzt. Das Radar zu manipulieren ist ebenfalls einfacher wenn er unabhängig von den anderen agiert. Um die korrekten Positionen der *OpenGLLocationView* zu berechnen, ist es notwendig die Höhe der *Android-ActionBar* zu erfassen und an den Renderer nach der Erstellung weiterzugeben. Der *OGLLocationViewRenderer* benötigt diese Höhe für die Positionierung, da Androids *Relative Layout*

und die *OpenGLSurface* nicht die exakt gleichen Offset-Positionen auf dem Display besitzen. Alle drei Renderer bekommen während der Laufzeit ständig neue Daten übermittelt. Diese Daten werden von dem *LocationController* und dem *SensorController* an die *AREAMainActivity* gesendet und von dieser nach der Verarbeitung an die einzelnen Renderer weitergegeben.

Die *AREAMainActivity* ist ebenfalls für die Verarbeitung der *Touch Events* verantwortlich. Berührt ein User das Display an einer bestimmten Stelle, überprüft die *AREAMainActivity* welches Element am wahrscheinlichsten ausgewählt ist. Die Bestimmung erfolgt über eine einfache Funktion, die berechnet, ob ein Element im Radius zu dem Event liegt. Liegt ein Element in dem Radius, wird der zuständige Dialog aufgerufen. Zu Testzwecken wurden die ursprünglichen POI neben den OpenGL-POI geladen. Es wurde festgestellt, dass alle drei Positionen einen unterschiedlichen Punkt besitzen (siehe Abbildung 6.4). Ein bis jetzt ungelöstes Problem ist, dass die gezeichneten POI eine leicht abweichende Position von der erwarteten Klickposition haben. Eine mögliche Lösung dieses Problems wäre, diese Punkte zu verschieben und somit die korrekte Position herzustellen. Vermutlich liegt der Kern des Problems an den vielen unterschiedlichen Koordinatensystemen und der von Android benutzten *TouchEvent*-Registrierung.

Während die *AREAMainActivity* auf eingehende *Touch Events* wartet, verarbeitet es währenddessen eingehende Listen aus POI, die von dem *LocationController* weitergegeben wurden. Diese Listen bestehen aus POI die im Blickfeld des Benutzers liegen, welche die korrekten Positionen für die OpenGL-Zeichenfläche besitzen. Die *AREAMainActivity* erstellt dann aus diesen Werten ein *OGLObject*, welches dann in einer weiteren Liste mit den restlichen POI gespeichert wird und an den zuständigen Renderer geschickt wird. Als Beispiel wird für den *OGLRadarPointsRenderer* eine Liste aus *OGLObjects* erstellt, die die Vorlage für die Punkte auf dem Radar sind. Für den *OGLLocationViewRenderer* wird eine Liste aus Dreiecken oder Kreisen erstellt. Diese Objekte stellen dann die POI dar.

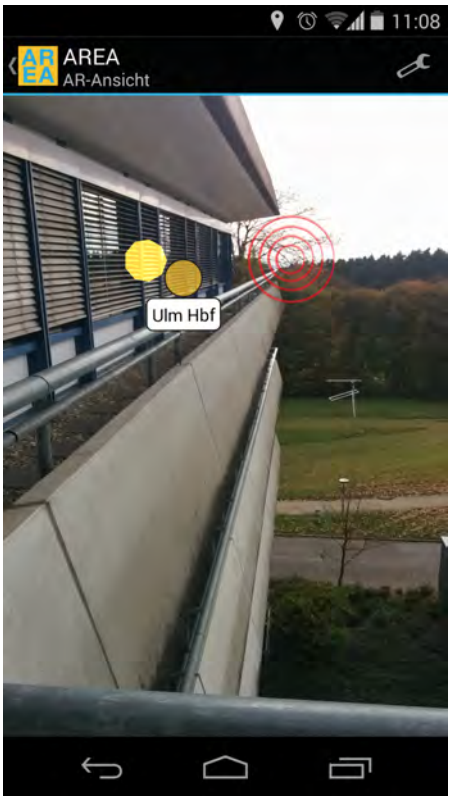

Abbildung 6.4: Rechter Punkt ist der ursprünglich verwendete POI. Der linke Punkt ist ein von OpenGL ES 2.0 gezeichneter Punkt. Die rote Markierung stellt den Punkt des *OnClickListener* dar.

6.4 OGLObject

Ein elementarer Bestandteil von AREA mit OpenGL ES 2.0 sind Objekte, die gezeichnet werden. Da wie schon angemerkt, OpenGL ES 2.0 von sich aus nicht in der Lage ist, komplexe Objekte oder Text darzustellen, muss der Entwickler jede Form selbst erstellen. OpenGL ES 2.0 kann maximal 3 Vertices zu einem Dreieck binden. Das gebundene Objekt wird danach gerastert und mit Farbe gefüllt. Wird ein Vertex verwendet, entsteht ein Punkt. Werden zwei Vertices verwendet, kann der Renderer eine Linie zeichnen und bei drei Vertices ein Dreieck. Soll ein Punkt oder eine Linie gezeichnet werden, muss im Shader beschrieben werden, wie groß oder wie breit die Linie sein soll. Wird keine Größe angegeben, geht der Renderer von einer Standardgröße von 0 aus. Linien oder Punkte der Größe 0 können nach dem Rastern nicht auf dem Display dargestellt werden. Umso komplexer eine Form ist, umso mehr Dreiecke werden für das Zeichnen benötigt.

Als Beispiel wird ein rotes Dreieck initialisiert (siehe Listing 6.2). Zuerst wird ein Float Array definiert, das mit beliebig vielen Vertices gefüllt wird. Jeder Vertex besitzt die Werte X, Y, Z welche ausschlaggebend für die Position im Raum sind. Die restlichen Werte R, G, B, A sind für die Farbe und den Alpha Wert zuständig. Ist das Array gesetzt, wird ein *FloatBuffer* initialisiert, der zuerst wie ein *ByteBuffer* erstellt werden muss und im Speicher adressiert wird. Es wird die Länge angegeben, die für diesen Buffer benötigt wird und die Reihenfolge, mit der dieser gelesen werden soll. Anschließend wird er mit dem Array befüllt. Das Attribut *numberOfVertices* dient dem Renderer später beim Zeichnen zur Angabe wie viele Vertices dieser Buffer enthält.

Listing 6.2: Pseudocode eines OGLObject.

```
1 float[] triangleVertices = {
2 // X,Y,Z                R,G,B,A
3  0.0f, 0.1f, 0f,         1f, 0f, 0f, 1f,
4 -0.1f, 0.0f, 0f,         1f, 0f, 0f, 1f,
5  0.1f, 0.0f, 0f,         1f, 0f, 0f, 1f
6 };
7
8 vertexData = ByteBuffer.allocateDirect(triangleVertices.length *
```

```
9  BYTES_PER_FLOAT).order(ByteOrder.nativeOrder()).asFloatBuffer();
10
11 vertexData.put(triangleVertices);
12 numberOfVertices = triangleVertices.length/8;
```

In AREA mit OpenGL ES 2.0 wurde eine spezielle Klasse entworfen, in der im Konstruktor zwei Argumente übergeben werden. Das erste Argument ist ein *Integer*, das einer ID entspricht und klar spezifiziert was für eine Art von Objekt erstellt werden soll. Es sind einfache Formen implementiert bis hin zu komplexen Kreisen. Noch komplexere Objekte hätten ebenfalls schon in dieser Klasse realisiert werden können. Um einzeln mit diesen Objekten arbeiten zu können, sie separat zu bewegen und getrennt der Größen anzupassen, sind Grundformen einfacher zu handhaben. Wenn als Beispiel ein komplett fertiger Radar vordefiniert worden ist, wäre er nur als ein ganzes Objekt manipulierbar. Der Radar besteht aber aus mehreren einzelnen Teilen, die unabhängig transformiert werden müssen. Kreise sind zum Teil komplexere Objekte mit vielen Vertices. Diese Kreise müssen wie alle vorherigen Objekte ebenfalls aus einzelnen Dreiecken erstellt werden. Das bedeutet, dass mit OpenGL nur annähernd perfekte Kreise gezeichnet werden können. Soll ein Kreis mit weniger Dreiecken gezeichnet werden, kann das über das zweite Argument realisiert werden.

Das zweite Argument ist ebenfalls ein *Integer*, das angibt, mit wie vielen Vertices das Objekt vorzugsweise aufgebaut werden soll. In den simplen Fällen, wie dem Punkt bis hin zum Viereck, kann dieses Argument nichts beeinflussen. Dieser Parameter ist bei Objekten, wie dem Kreis, in die Erstellung mit einbezogen. Somit kann ein Kreis mit einer genauen Anzahl an Punkten gezeichnet werden, was der Performance zugute kommt. Da der Renderer jedes mal wenn er die Szene neu zeichnet jedes Dreieck berechnen muss, ist eine hohe Anzahl an Vertices schlecht für die Bildrate. Kommt dann noch hinzu, dass es zu einer größeren Anzahl an Objekten auf der View kommen kann, ist eine komplexe Form für ein häufig zu zeichnendes Objekt ungeeignet. Während der Implementierungsphase wurde versucht, die POI mittels Kreise zu zeichnen. Dies ergab in der Darstellung ein Einbruch in der Bildrate, so dass eine flüssige Verarbeitung der Daten nicht möglich war. Daher sind die POI derzeit prototypisch als Dreiecke oder

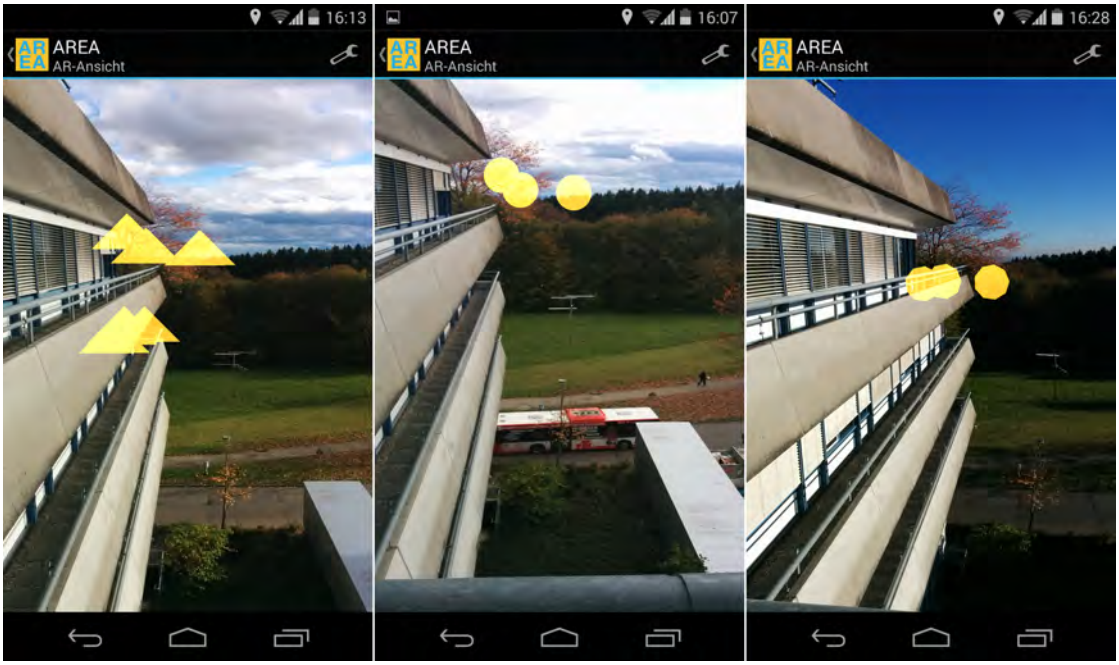

Abbildung 6.5: Drei unterschiedliche Darstellungen der POI. Links sind die POI als Dreiecke dargestellt. In der Mitte sind die POI als fast perfekte Kreise gezeichnet. Rechts ist der Kreis mit wenigen Vertices gezeichnet, was weniger Aufwand benötigt.

Kreise mit wenigen Vertices gezeichnet. Dadurch kommt es selbst bei einer großen Anzahl an Objekten zu keinem Einbruch in der Bildrate (siehe Abbildung 6.5). Diese Umsetzung ist wie beschrieben auf einem Nexus 5 getestet. Bei leistungsschwächeren Smartphones könnte sich dieser Unterschied in der Bildrate noch stärker bemerkbar machen.

Das *OGLObjekt* besitzt außerdem mehrere Matrizen, die für die Positionierung, Drehung und Skalierung benutzt werden können. Es ist möglich, auf jedem Objekt diese einzeln aufzurufen. Mithilfe dieser Matrizen können während der Laufzeit vor jedem einzelnen Zeichenvorgang die Objekte als Beispiel entgegen der Rotation des Smartphones gedreht werden. Dies könnte vonnöten sein um das Objekt während der Laufzeit im Lot zu halten. Das mag zwar bei einem Kreis keinen Unterschied machen, sollten aber ein Objekt gezeichnet werden, das eine Struktur aufweist, würde dies sofort bemerkt werden.

6.5 OpenGl Renderer

Die Renderer sind das Kernstück von AREA mit OpenGL ES 2.0. In ihnen fließen alle vorbereiteten Daten zusammen und werden auf dem Display dargestellt. Ihre Hauptarbeit ist das Zeichnen von Objekten und um diese Aufgabe ausführen zu können, muss ein Renderer eine funktionierende Renderpipeline besitzen. Eine Renderpipeline ist eine Abfolge von Schritten, die die GPU tätigt, um am Ende ein darstellbares Bild zu erhalten. OpenGL ES 2.0 stellt viele Elemente bereit, die dem Entwickler helfen die Daten so zu verarbeiten wie er es wünscht. Seine Aufgabe besteht im Wesentlichen darin die Renderpipeline so einzustellen, dass sie genau so zeichnet, wie er es möchte. Diese App verwendet drei relativ ähnliche Renderer, die von der *AREAMainActivityOpenGL* angestoßen werden und gleichzeitig zeichnen. Der *OGLLocationViewRenderer* ist der ursprünglichen *LocationView* nachempfunden und besitzt die gleichen Größen. Dieser Renderer ist für die POI verantwortlich. Der *OGLRadarPointsRenderer* und der *OGLRadarViewPortRenderer* sind im oberen linken Eck platziert. Sie zeichnen das Radar mit seinen POI-Punkten und die Blickrichtungsmarkierung. (siehe Abbildung 6.6)

Ein Renderer muss in OpenGL ES 2.0 drei Methoden implementieren, damit er funktionieren kann. In dem *OGLLocationViewRenderer* besitzt die *onCreate-Methode* mehrere Funktionen. Als erstes initialisiert sie benötigte Identitätsmatrizen, die später für die Berechnung der endgültigen Position der zu zeichnenden Objekte benötigt werden. Danach liest sie einen Vertex-Shader und einen Fragment-Shader ein. Sind diese Shader initialisiert, wird ein Programm erstellt, mit dem die beiden Shader verknüpft werden. Dieses Programm besteht immer aus einem Vertex-Shader und einem Fragment-Shader. Würde ein Shader in diesem Programm fehlen, wüsste OpenGL zum Beispiel nicht, in welcher Farbe er die Pixel zeichnen, oder an welcher Stelle er die Pixel rot färben soll. Ist das Programm fertig, werden Attribute an das Programm gebunden. Diese Attribute stellen die Verbindung zwischen Vertex-Shader und Fragment-Shader dar. So wird zum Beispiel die Position, die der Vertex-Shader berechnet, an den Fragment-Shader weitergegeben und in diesem benutzt, um zu bestimmen welche Pixel rot eingefärbt werden sollen. Neben den Attributen wird auch eine Matrix initialisiert. Diese dient ähnlich wie das Positions-Attribut dazu, die endgültige Position der Vertices zu bestimmen.

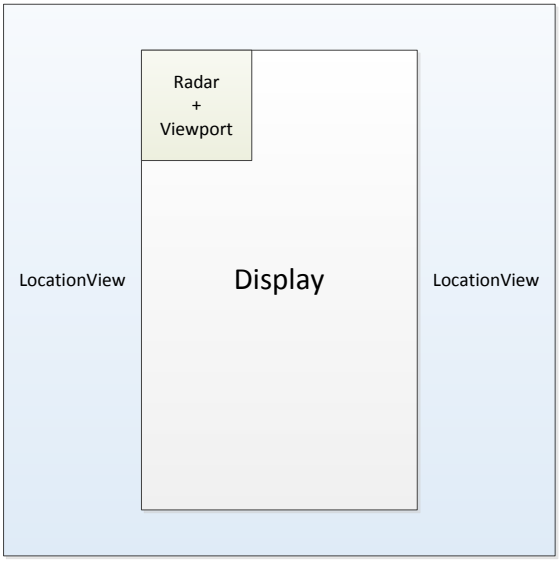

Abbildung 6.6: Platzierung der verschiedenen Renderer.

Die zweite Methode die ein Renderer implementiert haben muss, ist die *onSurfaceChanged*-Methode. Diese Methode wird jedes mal aufgerufen, sollte sich die Länge oder Breite des Displays ändern. Mindestens einmal bei der Erstellung des Viewports, dem Zeichenbereich des Renderers, wird diese Methode verwendet. Ihr werden Länge und Breite des neuen Displays übergeben. Diese Werte können dann verwendet werden, um die perspektivische Verzerrung auszugleichen, die bei verschiedenen Seitenverhältnissen entsteht. Dies ist notwendig, da der Zeichenbereich von OpenGL, in dem die Objekte gezeichnet werden, normiert ist. In den Renderern ist eine mathematische Funktion implementiert, die genau diese Verzerrung berechnet und dann in die *ProjektionsMatrix* speichert. Diese Projektionsmatrix wird beim Zeichenvorgang benutzt um die gerade besprochene Verzerrung auszugleichen (siehe Abbildung 6.7).

Die dritte und letzte Methode die benötigt wird, ist die *onDrawFrame-Methode*. Sie wird jedes mal aufgerufen, wenn das Bild neu gezeichnet werden soll. Beim Initialisieren kann festgelegt werden, ob das Bild jedes mal erneuert werden soll, sobald sich ein beliebiger Wert geändert hat, oder ob der Vorgang erneut angestoßen werden soll sobald der

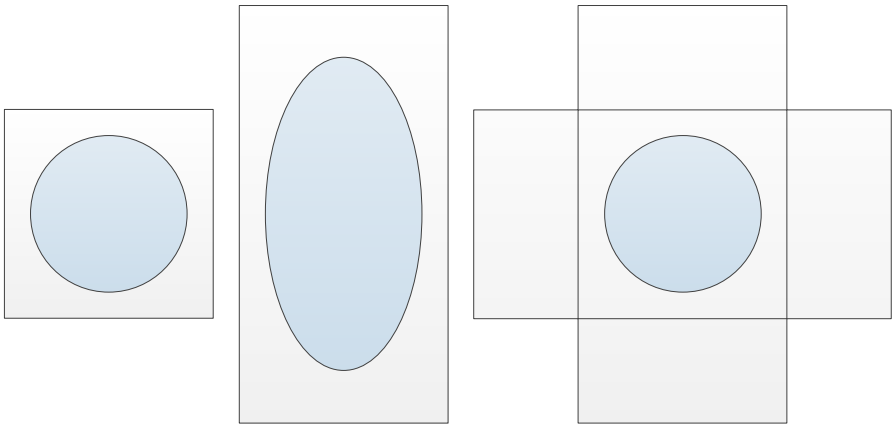

Abbildung 6.7: Unterschiedliche Display Formen und die entstehenden Verzerrungen. Links ist das Display quadratisch und ohne perspektivische Verzerrung. Mitte ist das Display rechteckig, wodurch eine perspektivische Verzerrung entsteht. Rechts wird die Form des Kreises durch eine zusätzliche Matrix entzerrt, sodass es egal bei welchen Maßen, immer korrekt dargestellt wird.

Renderer fertig mit seiner Arbeit ist. In dieser Anwendung ist es so implementiert, dass das Bild automatisch neu gezeichnet wird, sobald der Renderer fertig mit Zeichnen ist. Da die Sensorwerte sich konstant ändern, hätte die erstere Option keinen Unterschied erzeugt. Wird die Methode aufgerufen, wird zuerst eine Kopie der Liste erstellt, die die *OGLObjects* enthält. Dies ist notwendig um den Zeichenvorgang und die sich ständig ändernde Liste der zu zeichnenden Objekte zu entkoppeln. Ansonsten kann es zu einem Fehler während der Laufzeit kommen. Als nächstes wird der Bildschirm geleert und die bereits erwähnte *ProjektionsMatrix* mit der *Viewmatrix* multipliziert. Die *Viewmatrix* beschreibt die Position der Kamera im Raum. Werden diese multipliziert ergibt dies die Position der Kamera in dem neuen entzerrten Raum. Zum Schluss wird für jedes Objekt der Zeichenbefehl aufgerufen.

7

Vorstellung der Anwendung

In diesem Kapitel wird die Umsetzung in OpenGL ES 2.0 auf Android für AREA vorgestellt. Die wichtigsten Funktionen und einzelne Screenshots der Anwendung werden hierbei besprochen, was den derzeitigen Stand der Entwicklung zeigen soll.

7.1 Karten-Ansicht

Ein Hauptteil der Anwendung stellt die Kamera-Ansicht dar. Er kann über die Karten-Ansicht aufgerufen werden, welcher die POI aus Vogelperspektive anzeigen. Dieser Teil ist nicht mit OpenGL ES umgesetzt, das Hauptaugenmerk liegt auf der Umsetzung der Kamera-Ansicht in OpenGL ES. Um den Augmented Reality Teil der Anwendung zu starten wartet die Anwendung darauf, dass der Benutzer den Kamera Button im oberen rechten Eck anklickt (siehe Abbildung 7.1). Wird der Button ausgewählt, wird überprüft,

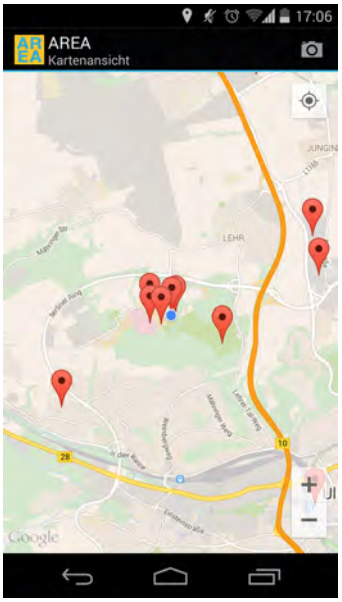

Abbildung 7.1: Die Karten-Ansicht von AREA. In diesem Teil der Anwendung werden die POI in der Vogelperspektive dargestellt.

ob das Gerät für OpenGL ES 2.0 fähig ist. Ist das Smartphone in irgendeiner Art nicht in der Lage OpenGL ES 2.0 darzustellen, wird die bereits implementierte Version der Kamera-View gewählt.

7.2 Kamera-Ansicht

Ist die Darstellung der Kamera-Ansicht in OpenGL möglich, wird die neue Aktivität gestartet. Dem Benutzer wird die Kamera-Ansicht angezeigt, in welcher mehrere Interface-Objekte dargestellt werden. Zum einen findet er den Radar, der visualisiert, in welcher Richtung das Smartphone derzeit schaut und in welcher horizontalen Richtung sich die POI befinden (siehe Abbildung 7.2). Auf dem Radar werden POI als Punkte dargestellt, die umso näher in der Mitte des Kreises gezeichnet werden, desto näher sie sich in der Nähe des Benutzers befinden. Es werden nur POI angezeigt, die sich innerhalb des vom Benutzer festgelegten Radius befinden. Der Benutzer kann diesen mit einem Klick auf den Radar oder dem Konfigurations-Button einstellen. Das einstellbare Minimum des

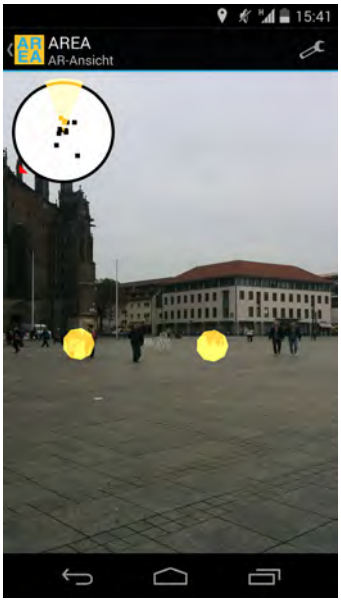

Abbildung 7.2: AREA Kamera-Ansicht. Zu sehen sind POI als gelbe Kreise, der Radar mit POI-Punkten und die Nordnadel am Radar als rotes Dreieck.

Radius liegt bei 1000 Metern, das Maximum bei 15000 Metern. Auf dem Radar ist ein Kreisausschnitt eingefärbt, der angibt, in welche Richtung der Benutzer blickt. Ebenfalls auf dem Radar zu sehen ist die Nordnadel, die immer in Richtung Norden zeigt. Sie ist ein rotes Dreieck, das sich am äußeren Rand des Radars bewegt. Sie reagiert auf die horizontale Bewegung des Benutzers, sodass die Nadel nach unten beziehungsweise hinter den Benutzer zeigt, wenn der Benutzer sich von Norden abwendet. Dreht der Benutzer sein Smartphone in eine Richtung, in der POI liegen, werden diese als Objekte auf dem Display dargestellt. Diese Objekte sind als Kreise oder Dreiecke angezeigt. Diese POI werden an der korrekten horizontalen und vertikalen Position gezeichnet. Diese POI werden gleich wie die Radarpunkte nur dann gezeichnet, wenn sie sich innerhalb des ausgewählten Radius befinden. Bei einem Klick auf ein POI öffnet sich das zugehörige Dialogfenster.

Das Dialogfenster des ausgewählten POI legt sich über die Kamera-Ansicht und zeigt dem Benutzer Informationen zu dem gewählten Punkt (siehe Abbildung 7.3). Liegen mehrere POI dicht beieinander, wählt die Anwendung den obersten Punkt als ausge-

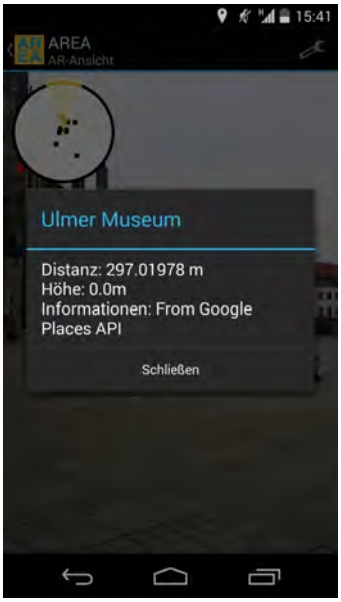

Abbildung 7.3: AREA Info Box. In ihr wird die Hintergrundinformation des gewählten POI angezeigt.

wähltes Objekt. Um ein anderes Objekt im Hintergrund auszuwählen, muss entweder der Radius variiert werden oder des Benutzers eigene Position verändert werden.

8

Abgleich der Anforderungen

Ziel der Arbeit war es herauszufinden, wie gut sich die AREA Kamera-Ansicht in OpenGL ES 2.0 umsetzten lässt. In diesem Kapitel werden die in Kapitel 4 aufgeführten Anforderungen evaluiert und bewertet.

8.1 Bewertung der Anforderungen

Dieses Vorhaben ist, wie schon beschrieben, zum Teil gelungen. Die ursprüngliche View wurde durch eine *OpenGL-SurfaceView* ersetzt und lässt eine stabile Darstellung der POI zu. Sie ist modular aufgebaut um eventuell einer weiteren Entwicklung den Weg zu vereinfachen. Die originale Datenstruktur wurde nur minimal erweitert und der Arbeitsablauf der Anwendung ist fast identisch zur Blaupause. Lediglich die Darstellung von Texten konnte bis zu dem Abgabetermin nicht realisiert werden. Ebenfalls existieren

kleine Fehler in der Platzierung der POI und der *Event-Listener*. Dies ist vermutlich zurückzuführen auf die unterschiedlichen Maße der einzelnen Displays und Views. Alle Daten werden in Echtzeit verarbeitet und gezeichnet. Ebenfalls kann der Benutzer wie gewohnt die Größe des Radar-Radius verändern. Lediglich die textuelle Wiedergabe des Radius fehlt. Die Kamera-Ansicht passt sich an den Neigungswinkel des Smartphones an, wodurch der Benutzer die Anwendung entweder im Landscape-Modus oder im Portrait-Modus verwenden kann. Alle verwendeten Klassen sind so aufgebaut, dass sie mit etwas Kenntnis von OpenGL einfach erweiterbar und einfach zu lesen sind.

Im Folgenden sind alle bereits besprochenen Anforderungen an die Umsetzung in tabellarischer Form zusammengefasst (Siehe Tabelle 8.1).

Tabelle 8.1: Tabellarische Aufzählung der Anforderungen.

Anforderung	**Bewertung**
POI auf Kamera-Ansicht anzeigen	+
POI auf Karten-Ansicht anzeigen	++
POI in Kamera-Ansicht nur anzeigen, wenn im Sichtfeld	++
POI in Kamera-Ansicht und Karten-Ansicht sollen auf Interaktionen reagieren	+
Sensordaten zur Positionierung des Smartphones auslesen (Beschleunigung, GPS, Magnetfeld)	++
Bei Bewegung des Smartphones Daten und POI in Echtzeit aktualisieren	++
Einstellbarer Radius für die Entfernung	++
Radar in Kamera-Ansicht mit POI aus der Umgebung und im Radius	++
POI Informationen anzeigen wenn ausgewählt	++
Portrait- und Landscape-Modus	++
Hohe Effizienz von Berechnungen	+
Hohe Effizienz beim Zeichnen der Anzeige	-
Hohe Stabilität	+
Hohe Genauigkeit	-
Gute Wartbarkeit	+
Einheitliche Spezifikation von POI	+
POI sollen intern erweiterbar sein	+
Einfache Einbindung in andere Anwendungen (Modularität)	+
Verständliche Kommentierung	+

9

Zusammenfassung und Ausblick

In diesem Kapitel wird das Projekt zusammengefasst und erörtert welche Aspekte in einer weiteren Arbeit verbessert werden könnten.

9.1 Zusammenfassung

Das Ziel dieser Arbeit ist es, die Augmented Reality Engine Application (kurz: AREA) mit OpenGL zu erweitern. Es wird für die Implementierung in Android OpenGL ES 2.0 verwendet. Was gezeigt wird ist, dass die Umsetzung von OpenGL in AREA ein Projekt mit teils unvorhergesehenen Problemstellen ist. Es wird zuerst ein Grundverständniss von OpenGL und Android vorausgesetzt, welches sich hauptsächlich auf die Koordinatensysteme der jeweiligen Bibliotheken bezieht und dem korrekten Aufbau der Renderpipeline in OpenGL ES 2.0. Die größten Probleme macht die Anpassung der

Zeichenflächen, die bei den beiden Bibliotheken von unterschiedlichen Standardgrößen ausgehen. Ein Problem das besteht, ist die korrekte Positionierung der POI und der Darstellung von Text. Ein Vorteil, den OpenGL ES 2.0 mit sich bringt ist, dass die Objekte die gezeichnet werden modular erweiterbar und besser manipulierbar sind. Als Beispiel können bestimmte POI als komplexe Modelle gezeichnet werden und während der Laufzeit unterschiedliche Texturen erhalten. Außerdem ist nun die Grundlage geschaffen um tiefer gehende Veränderungen im Zeichenablauf darzustellen.

9.2 Ausblick

Die Implementierung von OpenGL ES 2.0 in AREA lässt sich noch an vielen Stellen optimieren. Dazu gehören eine verbesserte Umsetzung der *OGLObjects*-Listen, die gezeichnet werden. Dies könnte man größtenteils in sogenannte *VertexBufferObjects* aufbauen, die dann im Speicher der GPU zwischengespeichert werden und nicht mehr einzeln initialisiert werden müssen. Durch diese Umsetzung wäre eine Performance-Verbesserung mit großer Wahrscheinlichkeit garantiert. Ebenfalls könnten die einzelnen Renderer in einem kompakten zusammengeführt werden. Dabei müsste beachtet werden, dass alle Positionswerte und Größen neu bestimmt werden müssten. Eine weitere Möglichkeit wäre alle in der Umgebung liegenden POI direkt in einem 3D-Raum zu zeichnen, sodass sie an der korrekten Position um den Benutzer platziert werden. Während der Laufzeit müsste dann dem Renderer nur eine Matrix mit der Blickrichtung des Smartphones übergeben werden. Dies hätte den Vorteil, dass während der Laufzeit kaum Datenverkehr zwischen den Sensoren und dem Renderer entsteht und alle POI im Speicher hochperformant bereit liegen, um gezeichnet zu werden.

Was aber immer noch ein zentraler Punkt weiterführender Entwicklung wäre, ist die korrekte Positionierung der POI und die Umsetzung von Text mithilfe von OpenGL ES 2.0.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ABB+01] AZUMA, R. ; BAILLOT, Y. ; BEHRINGER, R. ; FEINER, S. ; JULIER, S. ; MACINTYRE, Blair: Recent advances in augmented reality. In: *Computer Graphics and Applications, IEEE* 21 (2001), Nov, Nr. 6, S. 34–47. http://dx.doi.org/10.1109/38.963459. – DOI 10.1109/38.963459. – ISSN 0272–1716

[App14] APPLE: *OpenGL Performance Optimization*. https://developer.apple.com/library/mac/technotes/tn2093/_index.html. Version: 2014. – Accessed: 2014-10-28

[Aug14] AUGMENTEDMAP: *AugmentedMap Android Appstore*. https://play.google.com/store/apps/details?id=com.shiksha.augmentedmap&hl=de. Version: 2014. – Accessed: 2014-10-28

[BMW14] BMW: *Grafik: Auto Head-Up-Display*. http://www.bmw.com/com/en/newvehicles/6series/coupe/2004/allfacts/_shared/img/ergonomics_hud.jpg. Version: 2014. – Accessed: 2014-10-28

[Bro13] BROTHALER, Kevin: *OpenGL ES 2 for Android - A Quick-Start Guide*. 1. Aufl. Dallas, Texas : Pragmatic Programmers, LLC, 2013. – ISBN 978–1–937–78534–5

[CNB+13] CROMBACH, Anselm ; NANDI, Corina ; BAMBONYE, Manassé ; LIEBRECHT, Martin ; PRYSS, Rüdiger ; REICHERT, Manfred ; ELBERT, Thomas ; WEIERSTALL, Roland: Screening for mental disorders in post-conflict regions using computer apps - a feasibility study from Burundi. In: *XIII Congress of*

European Society of Traumatic Stress Studies (ESTSS) Conference, 2013, 70–70

[Fau14] FAUL, Mark: *Grafik: Jet Head-Up-Display*. `http://markfaul.files.wordpress.com/2009/09/f-16_hud_020410_091.jpg`. Version: 2014. – Accessed: 2014-10-28

[Gei12] GEIGER, Philip: *Entwicklung einer Augmented Reality Engine am Beispiel des iOS*, University of Ulm, Bachelorarbeit, September 2012

[Gei14] GEIGER, Phillip: *AREA - Augmented Reality Engine Application*. `http://www.area-project.info/`. Version: 2014. – Accessed: 2014-10-28

[Goo14a] GOOGLE: *Android - API Levels*. `http://developer.android.com/about/dashboards/index.html`. Version: 2014. – Accessed: 2014-10-28

[Goo14b] GOOGLE: *Android - OpenGL ES*. `http://developer.android.com/guide/topics/graphics/opengl.html`. Version: 2014. – Accessed: 2014-10-28

[GPSR13] GEIGER, Philip ; PRYSS, Rüdiger ; SCHICKLER, Marc ; REICHERT, Manfred: Engineering an Advanced Location-Based Augmented Reality Engine for Smart Mobile Devices / University of Ulm. Ulm : University of Ulm, 2013 (UIB-2013-09). – Technical Report

[Gro14a] GROUP, Khronos: *OpenGL Homepage*. `https://www.opengl.org/`. Version: 2014. – Accessed: 2014-10-28

[Gro14b] GROUP, Khronos: *OpenGL Shading Language*. `https://www.opengl.org/documentation/glsl/`. Version: 2014. – Accessed: 2014-11-03

[Gro14c] GROUP, Khronos: *Vertex Specification*. `https://www.opengl.org/wiki/Vertex_Specification`. Version: 2014. – Accessed: 2014-11-03

[GSP+14] GEIGER, Philip ; SCHICKLER, Marc ; PRYSS, Rüdiger ; SCHOBEL, Johannes ; REICHERT, Manfred: 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, 2014, 383–394

[IRLP+13] ISELE, Dorothea ; RUF-LEUSCHNER, Martina ; PRYSS, Rüdiger ; SCHAUER, Maggie ; REICHERT, Manfred ; SCHOBEL, Johannes ; SCHINDLER, Arnim ; ELBERT, Thomas: Detecting adverse childhood experiences with a little help from tablet computers. In: *XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference*, 2013, 69–70

[Khr14] KHRONOS: *Khronos OpenGL ES 2.0*. `https://www.khronos.org/opengles/2_X/`. Version: 2014. – Accessed: 2014-10-28

[MGS08] MUNSHI, Aaftab ; GINSBURG, Dan ; SHREINER, Dave: *OpenGL(R) ES 2.0 Programming Guide*. 1. Addison-Wesley Professional, 2008. – ISBN 0321502795, 9780321502797

[PLRH12] PRYSS, Rüdiger ; LANGER, David ; REICHERT, Manfred ; HALLERBACH, Alena: Mobile Task Management for Medical Ward Rounds - The MEDo Approach. In: *1st Int'l Workshop on Adaptive Case Management (ACM'12), BPM'12 Workshops*, Springer, September 2012 (LNBIP 132), 43–54

[PMLR14] PRYSS, Rüdiger ; MUNDBROD, Nicolas ; LANGER, David ; REICHERT, Manfred: Supporting medical ward rounds through mobile task and process management. In: *Information Systems and e-Business Management* (2014), March

[PTKR10] PRYSS, Rüdiger ; TIEDEKEN, Julian ; KREHER, Ulrich ; REICHERT, Manfred: Towards Flexible Process Support on Mobile Devices. In: *Proc. CAiSE'10 Forum - Information Systems Evolution*, Springer, 2010 (LNBIP 72), 150–165

[RLPL+13] RUF-LEUSCHNER, Martina ; PRYSS, Rüdiger ; LIEBRECHT, Martin ; SCHOBEL, Johannes ; SPYRIDOU, Andria ; REICHERT, Manfred ; SCHAUER, Maggie: Preventing further trauma: KINDEX mum screen - assessing and reacting towards psychosocial risk factors in pregnant women with the help

of smartphone technologies. In: *XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference*, 2013, 70–70

[RPR11] ROBECKE, Andreas ; PRYSS, Rüdiger ; REICHERT, Manfred: DBIScholar: An iPhone Application for Performing Citation Analyses. In: *CAiSE Forum-2011*, CEUR Workshop Proceedings, June 2011 (Proceedings of the CAiSE'11 Forum at the 23rd International Conference on Advanced Information Systems Engineering Vol-73)

[SHP+14] SCHLEE, Winfried ; HERRMANN, Jochen ; PRYSS, Rüdiger ; REICHERT, Manfred ; LANGGUTH, Berthold: Moment-to-moment variability of the auditory phantom perception in chronic tinnitus. In: *13th Int'l Conf on Cochlear Implants and Other Implantable Auditory Technologies*, 2014

[SRLP+13] SCHOBEL, Johannes ; RUF-LEUSCHNER, Martina ; PRYSS, Rüdiger ; REICHERT, Manfred ; SCHICKLER, Marc ; SCHAUER, Maggie ; WEIERSTALL, Roland ; ISELE, Dorothea ; NANDI, Corina ; ELBERT, Thomas: A generic questionnaire framework supporting psychological studies with smartphone technologies. In: *XIII Congress of European Society of Traumatic Stress Studies (ESTSS) Conference*, 2013, 69–69

[SSP+13] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; NIENHAUS, Hans ; REICHERT, Manfred: Using Vital Sensors in Mobile Healthcare Business Applications: Challenges, Examples, Lessons Learned. In: *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, 2013, 509–518

[SSP+14] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; MAIER, Fabian ; REICHERT, Manfred: Towards Process-Driven Mobile Data Collection Applications: Requirements, Challenges, Lessons Learned. In: *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, 2014, 371–382

[Wik14a] WIKITUDE: *Wikitude Android Appstore*. `https://play.google.com/store/apps/details?id=com.wikitude&hl=de`. Version: 2014. – Accessed: 2014-10-28

[Wik14b] WIKITUDE: *Wikitude Homepage*. http://www.wikitude.com/. Version: 2014. – Accessed: 2014-10-28

Name: Andreas Schmid Matrikelnummer: 745913

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Andreas Schmid