Institut für Formale Methoden der Informatik
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3328

Effiziente Darstellung von Kartendaten auf Mobilgeräten

Zeno-Oliver Groß

Studiengang:	Informatik
Prüfer:	Prof. Dr. Stefan Funke
Betreuer:	Prof. Dr. Stefan Funke
begonnen am:	24. April 2012
beendet am:	25. Oktober 2012
CR-Klassifikation:	I.3.3, H.3.3

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Die Darstellung von Kartenmaterial ist ein Zentraler Teil von Softwarelösungen zur Navigation. Auf mobilen geräten wie Smartphones benötigt solche Software in der Regel eine Verbindung zum Internet, da das Kartenmaterial auf einen Server abgelegt ist. Alle Berechungen finden dort statt und werden dann in Form von Bildern von dem jeweiligen Gerät abgerufen. Dieses Konzept ist beispielsweise in Google Maps [GMa] oder auch in der Standardansicht von OpenStreetMap mit der Software mapnik [Map, OSMd] verwirklicht.

Da eine Internetanbindung bei vielen Mobilgeräten nicht immer vorhanden oder gerade im Hinblick auf die Akkulaufzeit gewollt ist, wäre eine Offlinelösung notwendig. Hier besteht die Schwierigkeit, dass die Berechnungsgeschwindigkeit der gängigsten Mobilgeräte immernoch den Desktop-PCs hinterherhinkt. Dies trifft nicht nur auf Prozessoren zu sondern auch auf die Speicheranbindung zu SD-Karten. Aus diesen Gründen wird in dieser Arbeit ein Verfahren inklusive Dateiformat vorgestellt um Kartendaten für eine effiziente Suche und die daran anschließende Darstellung zu organisieren. Als grundlegendes Kartenmaterial sollen hier die Daten aus dem OpenStreetMap-Projekt [OSMf] dienen. Für das bequeme Einlesen dieser Daten wurde mit osmpbf eine Bibliothek für das kompakte PBF Datenformat von OpenStreetMap [OSMh]. Organisiert werden die Daten im „OSM Vector Map“-Format, das ein Gitter und einen R-Baum für die räumliche Indizierung verwendet. Für dieses Format wurden dann mit `osmvmap`, `spatiallight` und `osmvrender` Programmbibliotheken implementiert um solche Dateien zu generieren, einzulesen and anschließend zu visualisieren. Darauf aufbauend wurde eine Smartphone Applikation auf Android-Basis, „OSMVMapView“, für die Anzeige der Karte geschrieben. Nachdem die genannten Strukturen, Formate und Bibliotheken in dieser Arbeit vorgestellt sind, werden die Ergebnisse dargestellt und kurz ausgewertet. Anschließend werden noch Punkte gezeigt an denen angeknüpft werden kann.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Präliminarien stellt zunächst die zu Grunde liegende Plattform der Implementierung vor. Anschließend werden das Projekt hinter OpenStreetMap, die Struktur der Kartendaten und die Dateiformate beschrieben.

Kapitel 3 – osmpbf Bibliothek führt zunächst in die Google Protocol Buffers ein und stellt darauf aufbauend das PBF Dateiformat und die `osmpbf` Bibliothek vor.

Kapitel 4 – OSM Vector Map beschreibt den Aufbau sowie die Implementierung durch Programmbibliotheken des gleichnamigen Formates.

Kapitel 5 – Kartenrenderer stellt zunächst die Grundlagen sowie das Konzept der Programmbibliotheken `osmvmap` und `spatiallight` für die Visualisierung des Kartenmaterials vor und geht anschließend auf die Implementierung der Android Application OSMVMapView ein, welche die erwähnten Bibliotheken nutzt.

Kapitel 6 – Ergebnisse zeigt Renderings einiger Kartenausschnitte sowie Auflistungen zu den Laufzeiten der Visualisierung unter mehreren Kriterien und wertet diese aus.

Kapitel 7 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

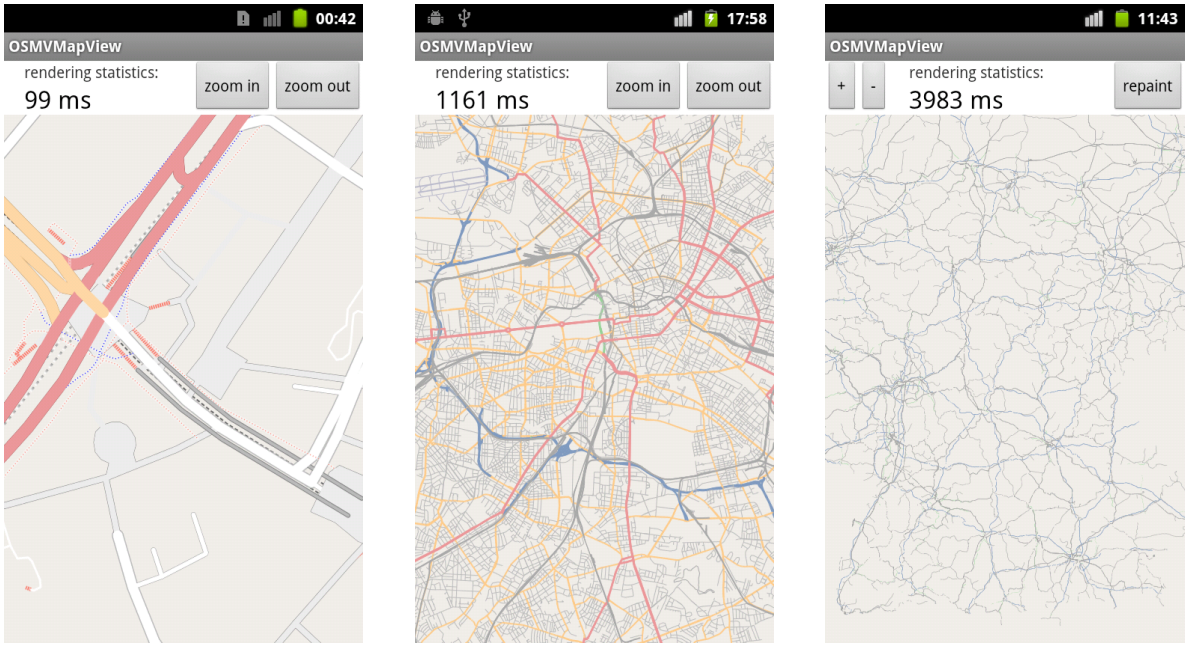

Abbildung 1.1: Screenshot von OSMVMapView

2 Präliminarien

2.1 Android

Android ist ein Betriebssystem basierend auf Linux. Es wurde von der Open Handset Alliance, dessen Gründer und Hauptmitglied Google ist, für den Einsatz auf mobilen Geräten wie Mobiltelefone, Tablets und kleinen Notebooks entwickelt.

Die aktuelle Version von Android zum Zeitpunkt dieser Arbeit ist 4.1. Da diese nicht auf allen Geräten eingesetzt werden kann oder Hersteller keine Unterstützung dafür anbieten, reicht die Vielfalt der Versionsnummern bis 1.5, wobei die gängigste 2.3 ist. Eine aktuelle Statistik über die eingesetzten Versionen von Android ist in [andb] zu finden. Die meisten Applikationen sind aufwärtskompatibel, sodass im Rahmen dieser Arbeit Android 2.3 eingesetzt wurde und die Implementierung darauf aufbaut.

Applikationen für diese Plattform sind in der Regel in Java implementiert und die sogenannte DalvikVM führt den Java-Bytecode aus. Es gibt aber auch die Möglichkeit native Bibliotheken zu nutzen. Hierfür wird das „Java Native Interface“, eine Schnittstelle zwischen der Java-Applikation und der nativen Bibliothek, benötigt. Ab Android in der Version 2.3 können Applikationen vollständig nativ implementiert werden, wobei die in Android üblichen grafischen Bedienelemente noch nicht unterstützt werden [Anda].

2.1.1 Apps

Eine Anwendung für die Android Platform (App) besteht aus mehreren Komponenten. Die für die Anwendung hier wichtigste Komponente ist die „Activity“. Sie stellt eine Visuelle Oberfläche dar mit der ein Benutzer interagieren kann. Dies ist z.B. eine Kalenderansicht, eine Liste von E-Mails oder die Bedienelemente einer Kameraanwendung. Activities werden durch Programme bereitgestellt und können von anderen Programmen genutzt werden. So ist es beispielsweise möglich ein aufgenommenes Bild einer Kamera-Activity in das eigene Programm einzubinden und so das geschossene Foto weiter zu verwenden. Activities stellen also nicht nur eine einfache Benutzerobefläche dar, sondern bieten auch alle damit verbundene Funktionalität und jede App dabei besteht aus mindesten einer, der „main activity“. Weitere Informationen sind in der Dokumentation „Android Developers“ [Anda] zu finden.

2.1.2 Native Bibliotheken

Auf der Android Plattform gibt es nicht nur die Möglichkeit Applikationen in Java zu entwickeln sondern auch eine Mischung aus Java und nativen Bibliotheken, also Binärcode in Maschinensprache des jeweiligen Gerätes. Die Verwendung dieser Schnittstelle im Falle der Android Plattform ist in [Anda] näher beschrieben.

Der Vorteil von nativem Programmcode ist, dass dieser ohne Verarbeitung direkt ausgeführt werden kann, was in der Regel im Vergleich zu Java-basierten Anwendungen einen Geschwindigkeitsvorteil bietet. Aus diesem Grund wurde im Rahmen dieser Arbeit die Funktionalität für die Visualisierung von Kartendaten als native Bibliothek implementiert und wird von einer Android App namens „OSMVMapView“ für die Berechnungen verwendet.

2.2 OpenStreetMap

2.2.1 Allgemeines

OpenStreetMap („OSM“) ist ein gemeinnütziges Projekt mit dem Ziel eine freie Landkarte der Erde bereitzustellen. Der Begriff „frei“ ist im gleichen Sinne wie „freie Software“ zu verstehen, also der freie Zugriff auf die Kartendaten und die daraus resultierenden Karten in Bildform. Darüber hinaus ist es jedem erlaubt die Kartendaten zu verändern und beliebig Daten hinzuzufügen oder zu entfernen.

Das Konzept hinter OpenStreetMap ist das gleiche wie bei der Wikipedia und hat die gleichen Stärken und Schwächen: Die Kartendaten werden auf freiwilliger Basis eingetragen („gemapt“) und aktualisiert [OSMa]. Das bedeutet, die Korrektheit und Genauigkeit des Kartenmaterials kann nicht garantiert werden. So zeigt sich, dass beispielweise viele Städte in Deutschland genauer und vor allem aktueller eingetragen sein könnten als es in Konkurenzprodukten der Fall ist. Genauso findet man auch Gebiete, vor allem in ländlichen Räumen, mit sehr wenigen Eintragungen.

2.2.2 Strukturen

Die Karten in OpenStreetMap setzen sich aus vier Grundelementen zusammen: Punkte, Linien, Flächen und Relationen. Da Linien und Flächen beide aus Punkten bestehen, werden nur drei Daten-Grundelemente („data primitive“) abgespeichert. Für die Identifikation eines data primitive wird jedes mit einer eindeutige Kennung („ID“) versehen. Zusätzlich können jedem Element beliebig viele Attribute zugewiesen werden. Die Daten-Grundelemente gliedern sich in:

Node Ein Node („Knoten“) repräsentiert einen Punkt auf der Karte, angegeben durch seine Koordinaten in Längen- und Breitengraden.

Diese Knoten können beispielsweise für sich alleine stehen um wichtige Punkte auf der Karte zu markieren oder können zusammengefasst Linienzüge oder Flächen ergeben.

Way Ein Way („Weg“) ist ein zusammengesetztes Element. Hier werden Nodes - referenziert über die jeweilige ID - in einer Liste zu einem Knotenzug zusammengefasst. Unterschieden werden hier offene Ways, geschlossene Ways, Flächen („Areas“) und eine Kombination aus geschlossenem Way und Fläche (entspricht etwa einer umrandeten Fläche).

Bei einem offenen Way ist der Anfangs- und Endpunkt unterschiedlich und dient so zur modellierung von Linienzügen. Geschlossene Ways, Flächen und die Kombination daraus besitzen denselben Anfangs- und Endpunkt. Sie unterscheiden sich jedoch in ihren Attributen, ob das resultierende Polygon umrandet oder gefüllt wird.

Relation Ein weiteres zusammengesetztes Element ist mit der Relation gegeben. Anders als bei einem Way kann hier jedes andere Element referenziert werden. Eine Referenz wird hierbei „Member“ (Mitglied) genannt. Jedes Member besitzt eine kontextabhängige Rolle, wodurch seine Funktion innerhalb der Relation definiert wird.

Mit diesem Element können komplexere Objekte wie z.B. ausgehöhlte Flächen modelliert oder einfache Gruppen gebildet werden.

Jedes Objekt der Karte wird mit Hilfe dieser Grundlementen und deren zugehörigen Attributen („Tags“) modelliert. Spezifiziert werden hier allgemeine Eigenschaften wie Bezeichnung, Art des Polygons sowie spezielle Eigenschaften wie Straßenart, Schienetyp, Brückenebene oder Beschaffenheit einer Fläche. Ein Tag ist hierbei ein Zeichenkettenpaar aus Schlüssel und Wert („Key“ und „Value“), im weiteren Verlauf angegeben durch `[Schluessel = Wert]`.

Beispielsweise definiert so ein Way mit den Tags `[highway = pedestrian]` und `[area = yes]` einen öffentlichen Platz. Besitzt ein Way die Tags `[railway = rail]`, `[bridge = yes]` und `[layer = 2]`, so ist eine Eisenbahnbürcke auf zweiter Ebene gemeint.

Eine Auflistung der Keys, den möglichen Values, deren Bedeutungen und den Regeln welche Tags welchen Objekten zugewiesen werden sollten ist auf dem OpenStreetMap-Wiki [OSMb] zu finden.

2.2.3 Formate

Zur Zeit dieser Ausarbeitung gibt es zwei Formate zum Abspeichern und für den Austausch von OSM-Daten. Das am meisten verwendete Format ist das von `.osm`-Dateien. Hier werden die Daten als XML-Schema abgelegt. Das zweite Format ist das „Protocolbuffer Binary Format“. Wie der Name bereits suggeriert, ist es ein Binärformat, das mit Hilfe der Google Protocol Buffers serialisiert wird. Dateien dieses Typs besitzen die Endung `.osm.pbf`.

Im Folgenden wird nur das XML-Format kurz vorgestellt. Auf das PBF Format wird im Abschnitt 3.2 genauer eingegangen.

Listing 2.1 Beispiel eines OSM Datensatzes im XML-Format

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
  <bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900"
   maxlon="12.2524800"/>
  <node id="1831881213" version="1" changeset="12370172" lat="54.0900666"
   lon="12.2539381" user="lafkor" uid="75625" visible="true"
   timestamp="2012-07-20T09:43:19Z">
    <tag k="name" v="Neu Broderstorf"/>
    <tag k="traffic_sign" v="city_limit"/>
  </node>
  ...
  <node id="298884272" lat="54.0901447" lon="12.2516513" user="SvenHRO" uid="46882"
   visible="true" version="1" changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
  <way id="26659127" user="Masch" uid="55988" visible="true" version="5"
   changeset="4142606" timestamp="2010-03-16T11:47:08Z">
    <nd ref="292403538"/>
    ...
    <nd ref="261728686"/>
    <tag k="highway" v="unclassified"/>
    <tag k="name" v="Pastower Straße"/>
  </way>
  <relation id="56688" user="kmvar" uid="56190" visible="true" version="28"
   changeset="6947637" timestamp="2011-01-12T14:23:49Z">
    <member type="node" ref="294942404" role=""/>
    ...
    <member type="way" ref="4579143" role=""/>
    ...
    <member type="node" ref="249673494" role=""/>
    <tag k="name" v="Küstenbus Linie 123"/>
    <tag k="network" v="VVW"/>
    <tag k="operator" v="Regionalverkehr Küste"/>
    <tag k="ref" v="123"/>
    <tag k="route" v="bus"/>
    <tag k="type" v="route"/>
  </relation>
  ...
</osm>
```

Das XML-Format enthält neben allgemeinen Informationen, wie die verwendete API Version und dem Namen des Programmes, das die XML-Daten generiert hat, je Elementtyp einen zusammenhängenden Block:

Zuerst folgt ein Block mit Nodes, die jeweils Koordinaten im WGS-84 Referenzsystem [WGS00]. Falls vorhanden, ist innerhalb jeder Node einen Block aus Tags enthalten. Als nächstes folgt ein Block aus Ways jeweils mit einem Block aus Referenzen zu Nodes und ein Block aus Tags. Zuletzt folgt ein Block aus Relations und jede Relation besitzt einen Block aus Mitgliedern und danach einen Block aus Tags. Ein Beispiel für dieses Format findet sich in Listing 2.1

Dieses Format besitzt einige Vorteile [OSMg]: Es ist von Menschen leicht lesbar und anpassbar, es ist Platform- und Architekturunabhängig, durch die Flexibilität des XML-Formates ist der Aufwand für einen konkreten Parser sehr gering und es kann gut komprimiert werden. Jedoch gibt es hierbei auch nicht zu vernachlässigende Nachteile [OSMg]: Die resultierenden Dateien werden sehr groß (planet.osm unkomprimiert: 171GB), in der regel müssen die Daten erst vollständig dekomprimiert werden bevor man sie nutzen kann und das Einlesen kann sehr viel Zeit in Anspruch nehmen.

3 osmpbf Bibliothek

Das PBF Format wird verwendet um OpenStreetMap Daten platzsparend abzuspeichern. Es ist als Alternative zum XML Format konzipiert. Im Vergleich zum XML Format soll es dadurch möglich sein solche Daten deutlich schneller einzulesen und zu schreiben. Grundlage für dieses Format sind die von Google entwickelten Protocol Buffers [OSMh].

Ein schneller Zugriff auf OSM Daten ist schon bei der Vorverarbeitung wichtig. Da zum Zeitpunkt dieser Arbeit noch keine freie C++ Bibliothek zum Einlesen und Schreiben solcher Daten existierte, wurde im Rahmen dieser Arbeit die osmpbf Bibliothek erstellt.

In diesem Kapitel soll zunächst kurz auf die Google Protocol Buffers eingegangen werden. Darauf aufbauend soll als nächstes das PBF Format erklärt und abschließend die osmpbf Bibliothek vorgestellt und deren API näher erläutert werden.

3.1 Google Protocol Buffers

Die Google Protocol Buffers sind ein Werkzeug zur kompakten Serialisierung von Datenstrukturen. Durch die Sprach- und Platformübergreifende Implementierung und die Erweiterbarkeit kann es für die Serialisierung von Daten beispielsweise im Bereich der Kommunikation oder zum speichern von Daten genutzt werden.

Ähnlich wie bei XML-Schemata werden zuerst Container, sog. „messages“ (Nachrichten), definiert. Aus dieser (Typ-)Definition wird dann Code erzeugt und in das eigene Projekt eingebunden, wodurch dann Datenströme eingelesen und herausgeschrieben werden können. Darüber hinaus ist diese Definition beliebig erweiterbar, ohne die Kompatibilität zu älterer Software einbüßen zu müssen.

3.1.1 Verwendung

Angelehnt an die Dokumentation von Google in [PBD] soll nun die Verwendung der Protocol Buffers vorgestellt werden:

Zunächst wird die Struktur der Informationen, die serialisiert werden sollen, als Protocol Buffer Message Typen in Dateien mit der Endung `.proto` definiert. Ein einfaches Beispiel einer message-Definition zum speichern von Adressdaten findet sich in Listing 3.1.

Jede message besteht aus Einträgen mit jeweils einer eindeutigen Nummer und dem jeweiligen Datentyp. Jedem Eintrag wird ausserdem zugewiesen, ob dieser optional (`optional`),

notwendig (`required`) oder auch mehrfach aneinander gereiht (`repeated`) ist. Datentypen können ganze Zahlen, Fließkommazahlen, Wahrheitswerte, Zeichenketten, Binärdaten, Aufzählungstypen oder auch andere messages sein. Es ist ausserdem möglich den optionalen Einträgen einen Standardwert zuzuweisen oder mehrfach aneinander gereihte Einträge zusätzlich zu kompaktieren.

Listing 3.1 Beispiel einer message-Definition zum speichern von Adressdaten

```
1  message Person {
2          required string name = 1;
3          required int32 id = 2;
4          optional string email = 3;
5
6          enum PhoneType {
7                  MOBILE = 0;
8                  HOME = 1;
9                  WORK = 2;
10         }
11
12         message PhoneNumber {
13                 required string number = 1;
14                 optional PhoneType type = 2 [default = HOME];
15         }
16
17         repeated PhoneNumber phone = 4;
18 }
```

Sind alle messages definiert wird der Protocol Buffer Compiler mit dieser Eingabe ausgeführt und der davon erzeugte Programmcode kann nun benutzt werden.

Ist C++ als Ausgabesprache des Protocol Buffer Compilers gewählt, wird für jede message eine Klasse erstellt. Mit der jeweiligen Klasse kann auf die Einträge einzeln zugegriffen und die Serialisierung und Deserialisierung ausgeführt werden. Beispiele dafür sind in den Listings 3.2 und 3.3 zu finden. Wie weiter oben erwähnt können später nahezu problemlos Einträge hinzugefügt werden. Software die mit einer ältere Version der message-Definition erzeugt wurde funktioniert auch auf Eingabedaten aus der aktuelleren Definition. Unbekannte Einträge werden dabei überlesen [PBD].

Listing 3.2 Beispiel für das Anlegen und Serialisieren einer message des Types „Person“

```
1 Person person;
2 person.set_name("John Doe");
3 person.set_id(1234);
4 person.set_email("jdoe@example.com");
5 fstream output("myfile", ios::out | ios::binary);
6 person.SerializeToOstream(&output);
```

Listing 3.3 Beispiel für das Einlesen und die Ausgabe einer message des Types „Person“

```
1 fstream input("myfile", ios::in | ios::binary);
2 Person person;
3 person.ParseFromIstream(&input);
4 cout << "Name: " << person.name() << endl;
5 cout << "E-mail: " << person.email() << endl;
```

Die nähere Funktionsweise wie Daten mit den Protocol Buffers encodiert werden findet sich in [PBD]. Mit dieser Grundlage wird im nächsten Abschnitt das osm.pbf Format vorgestellt.

3.2 osm.pbf Format

Das PBF Format stellt ein Binärformat dar, um OpenStreetMap Daten zu speichern. Es ist als Ersatz zum XML Format konzipiert und die resultierenden Dateien sind bis zu 70% kleiner (vergleiche bzip gepackte `Planet.osm` Datei) und der Zugriff ist bis zu 6 mal schneller [OSMh]. Die Abkürzung PBF steht für „Protocolbuffer Binary Format“, woraus auf das zugrunde liegende Dateiformat, die Google Protocol Buffers, geschlossen werden kann.

Das Format definiert einen Aufbau aus Blöcken, sogenannten „Blobs“. Da diese Blöcke unabhängig von einander sind ist auf dieser Ebene ein wahlfreier Zugriff möglich. So können nicht benötigte oder unlesbare Blöche übersprungen werden. Ein Blob besteht aus:

1. Größenangabe der folgenden BlobHeader message - 32 Bit langer Integer ohne Vorzeichen in „Network Byte Order“
2. Blob-Kopf in Form einer serialisierten Protocol Buffer message des Typs `BlobHeader`
3. Blob-Daten in Form einer serialisierten Protocol Buffer message des Typs `Blob`

In der BlobHeader message sind immer der Typ der im Blob enthaltenen Daten und die Größe des Blobs gespeichert. Darüber hinaus ist es möglich zusätzliche Metadaten, wie z.B. ein Index über die OpenStreetMap Daten des Blobs, abzulegen. Die Blob message besteht aus mehreren optionalen Feldern, die angeben ob und wie die OSM Daten komprimiert vorliegen. Je Kompressionsverfahren ist ein Eintrag vorgesehen, der nur dann (komprimierte) OSM Daten enthält, wenn die jeweilige Kompression benutzt wurde. Eine Aufschlüsselung der beiden beschriebenen message Typen aus [OSMh] ist im Listing 3.4 zu sehen. Für die serialisierte Form der BlobHeader message und der Blob message gibt es ein Größenbeschränkung um so robust zu erkennen, ob die Eingabedaten beschädigt sind [OSMh]. Dabei muss die Länge von der BlobHeader message unterhalb von 64 KiB und die Länge von der Blob message unterhalb von 32 MiB liegen.

Die Daten in jedem Blob sind die Binärdaten einer serialisierten Protocol Buffer message. Zur Zeit dieser Arbeit sind zwei Arten von Blob-Daten definiert: OSM Kopfdaten und OSM Datensätze. OSM Kopfdaten sind als „HeaderBlock“ message und OSM Datensätze sind als „PrimitiveBlock“ message gespeichert. Aus Gründen der Einfachheit wird im weiteren

Listing 3.4 Blob Protocol Buffer message Definition

```
1  message BlobHeader {
2          required string type = 1;
3          optional bytes indexdata = 2;
4          required int32 datasize = 3;
5  }
6
7  message Blob {
8          optional bytes raw = 1; // No compression
9          optional int32 raw_size = 2; // Only set when compressed, to the uncompressed size
10         optional bytes zlib_data = 3;
11         //optional bytes lzma_data = 4; // PROPOSED.
12         //optional bytes OBSOLETE_bzip2_data = 5; // Deprecated.
13 }
```

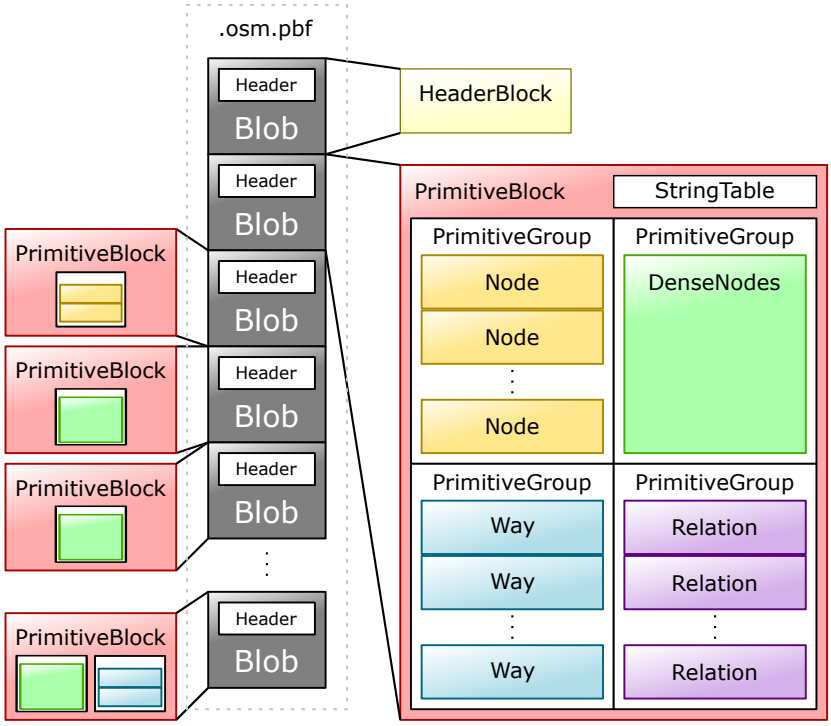

Abbildung 3.1: Schema des PBF Formates

Verlauf auf den Zusatz „message“ verzichtet. Zur Illustration ist in Abbildung 3.1 eine schematische Darstellung des PBF Formates zu sehen.

In jeder PBF Datei muss vor den OSM Daten ein HeaderBlock stehen. In diesem Datenblock werden notwendige und optionale Eigenschaften zum sicheren Einlesen der Daten festgehalten. Zu den notwendigen Eigenschaften zählen zur Zeit „OSMSchema-V0.6“ und „DenseNodes“. Die erstere bedeutet, dass die Daten im OSM-Schema der Version 0.6 abgespeichert sind und die letztere weist darauf hin, dass die Daten der Knoten in kompakter

Listing 3.5 HeaderBlock und PrimitiveBlock Protocol Buffer message Definition

```
message HeaderBlock {
        optional HeaderBBox bbox = 1;
        /* Additional tags to aid in parsing this dataset */
        repeated string required_features = 4;
        repeated string optional_features = 5;

        optional string writingprogram = 16;
        optional string source = 17; // From the bbox field.
}

message PrimitiveBlock {
        required StringTable stringtable = 1;
        repeated PrimitiveGroup primitivegroup = 2;

        // Granularity, units of nanodegrees, used to store coordinates in this block
        optional int32 granularity = 17 [default=100];
        // Offset value between the output coordinates coordinates and the granularity grid
            in unites of nanodegrees.
        optional int64 lat_offset = 19 [default=0];
        optional int64 lon_offset = 20 [default=0];

        // Granularity of dates, normally represented in units of milliseconds since the 1970
            epoch.
        optional int32 date_granularity = 18 [default=1000];


        // Proposed extension:
        //optional BBox bbox = 19;
}
```

Form vorliegen. Mit den optionalen Eigenschaften kann beispielsweise eine Sortierung der Daten oder zusätzliche Metadaten angekündig werden. Neben diesen Eigenschaften sind auch optionale Einträge zu einem Rechteck minimaler Größe („bounding box“), das alle folgenden geographischen Daten umschließt, oder der Name des Programmes, das diesen Datensatz erstellt hat, zu finden. In Listing 3.5 ist die Protocol Buffer Definition der HeaderBlock message zu finden.

In einem PrimitiveBlock werden die OSM Daten in Gruppen („PrimitiveGroup“ message) aus je einer Art von Daten-Grundelemente (siehe 2.2) gespeichert. Die message Definition hierzu ist im Listing 3.5 dargestellt.

Alle in diesem Block verwendeten Zeichenketten sind in einer Tabelle abgelegt und bei den jeweiligen Attributen wird mit einer Indexnummer auf die entsprechende Stelle in der Tabelle verweisen anstatt extra eine Zeichenkette zu speichern. Jede Tabelle besitzt an der Stelle 0 (also auch Index = 0) einen Eintrag mit einer leeren Zeichenkette. Der Index 0 dient später als Trennungszeichen oder steht für eine „ungültige“ Zeichenkette.

Neben der Zeichenkettentabelle sind auch Einträge für die Auflösung der Zeitstempel und Werte der Verschiebung der Geokoordinaten sowie deren Auflösung enthalten. So

Listing 3.6 PrimitiveGroup Protocol Buffer message Definition

```
1 message PrimitiveGroup {
2         repeated Node   nodes = 1;
3         optional DenseNodes dense = 2;
4         repeated Way    ways = 3;
5         repeated Relation relations = 4;
6         repeated ChangeSet changesets = 5;
7 }
```

sind beispielsweise in den Nodes nur relative Angaben der Längen- und Breitengraden gespeichert. Den Absoluten Wert des Breitengrades errechnet sich dann mit

$$LAT = 0,000000001 \cdot (\texttt{lat\_offset} + (\texttt{granularity} \cdot \texttt{lat}))$$

und analog dazu ergibt sich der Längengrad aus

$$LON = 0,000000001 \cdot (\texttt{lon\_offset} + (\texttt{granularity} \cdot \texttt{lon}))$$

.

Die Grundelemente werden in sogenannten PrimitiveGroups gespeichert (siehe Listing 3.6). In einer solchen message darf nur eine Sorte von Grundelementen enthalten sein. In einem PrimitiveBlock darf also Beispielsweise eine PrimitiveGroup ausschließlich aus Nodes, eine aus Ways und eine aus Relations vorhanden sein. Aus der Dokumentation in [OSMh] geht nicht hervor ob es erlaubt ist mehrere PrimitiveGroups mit dem gleichen Grundelementtyp in einen PrimitiveBlock zu speichern. Laut [OSMh] stellt Osmosis die Referenzimplementierung dar. In dieser treten bei der Serialisierung keine Gruppen mit Grundelementen einer Art mehrfach in einem PrimitiveBlock auf (siehe [OSMe]). Aus diesem Grund wird hier davon ausgegangen, dass dies in der Praxis nicht vorkommen soll.

Die Eigenschaften und Attribute, wie sie in 2.2 vorgestellt wurden, sind direkt als Einträge in den jeweiligen message Definitionen enthalten (siehe Listing 3.7). Tags werden hierbei in zwei Feldern aufgeteilt, eines aus Keys und eines aus Values. Die zu einem Key an einer Stelle des Key-Feldes gehörende Value steht dabei an der gleichen Stelle im Value-Feld. Gleiches gilt für die Member der Relations: es gibt drei Felder, eines für die Rollenbezeichnung, eines für die ID und eines für den Typ.

Es wird davon ausgegangen, dass die Referenzen je Way bzw. je Relations tendenziell ähnliche IDs aufweisen. Daher werden die jeweiligen Felder per Delta-Kodierung gepackt. Die so resultierenden kleineren Zahlen können bei der Serialisierung dichter gepackt gespeichert werden.

Neben den herkömlichen Nodes gibt es beim PBF Format zusätzlich eine kompaktere Form, die sogenannten „dense nodes“ (siehe Listing 3.7). Hierbei werden die Daten von vielen Nodes platzsparender abgespeichert. Nodes bestehen aus einer ID, einem Paar von Geokoordinaten und mehreren Tags. Bei den dense nodes werden diese Eigenschaften aneinandergereiht gespeichert und (bis auf die Tags) mit der Delta-Kodierung gepackt. Die

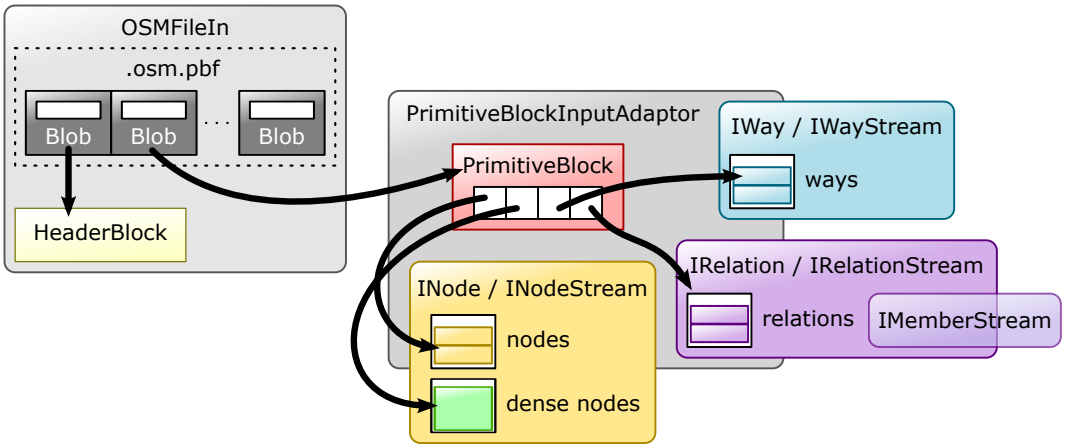

Abbildung 3.2: Vereinfachtes Schema der Eingabe API der osmpbf Bibliothek

Tags werden als Key-Value-Paare in einem Feld (Eintrag `keys_vals`) nach der folgenden Vorschrift abgespeichert:

$$((\texttt{<key id>}\ \texttt{<value id>})^*\ 0)^*$$

Die „keyid“ bzw. „valueid“ entspricht einem Index der Zeichenkettentabelle und eine „0“ ist das Trennsymbol, das die Key-Value-Paare des nächsten Nodes ankündigt.

Die restlichen Strukturen dienen dazu weitere Metadaten wie Zeitstempel, Version usw. abzuspeichern. Diese sind im Rahmen dieser Arbeit zwar nicht relevant, es können jedoch nähere Informationen dazu in [OSMh] gefunden werden.

Nachdem das PBF Dateiformat vorgestellt wurde, wird im Folgenden näher auf die osmpbf Bibliothek eingegangen.

3.3 Eingabe API und Implementierung

Die Protocol Buffer message Definition zum PBF Dateiformat ist allein nicht ganz ausreichend um die Daten direkt auszulesen zu können. Wie im vorherigen Abschnitt beschrieben benötigen beispielsweise die dense nodes einen zusätzlichen Vorverarbeitungsschritt. Die osmpbf Biliothek ist so gestealtet möglichst einfach und effizient auf die OSM Daten zugreifen zu können.

Die Eingabe API besteht hierbei aus zwei Teilen. Der Dateizugriff auf die Blobs und die Dekodierung der darin enthaltenen Daten. Eine vereinfachtes Schema der Eingabe API ist in Abbildung 3.2 dargestellt. Die wichtigsten Klassen und deren Aufgaben werden in den folgenden Abschnitten vorgestellt.

Listing 3.7 Node, DenseNodes, Way und Relation Protocol Buffer message Definitionen

```
message Node {
        required sint64 id = 1;
        required sint64 lat = 7;
        required sint64 lon = 8;
        repeated uint32 keys = 9 [packed = true]; // Denote strings
        repeated uint32 vals = 10 [packed = true];// Denote strings
        optional Info info = 11; // Contains metadata
}

message Way {
        required int64 id = 1;
        // Parallel arrays.
        repeated uint32 keys = 2 [packed = true];
        repeated uint32 vals = 3 [packed = true];

        optional Info info = 4;

        repeated sint64 refs = 8 [packed = true]; // DELTA coded
}

message Relation {
        enum MemberType {
                NODE = 0;
                WAY = 1;
                RELATION = 2;
        }
        required int64 id = 1;

        // Parallel arrays.
        repeated uint32 keys = 2 [packed = true];
        repeated uint32 vals = 3 [packed = true];

        optional Info info = 4;

        // Parallel arrays
        repeated int32 roles_sid = 8 [packed = true];
        repeated sint64 memids = 9 [packed = true]; // DELTA coded
        repeated MemberType types = 10 [packed = true];
}

message DenseNodes {
        repeated sint64 id = 1 [packed = true]; // DELTA coded

        //repeated Info info = 4;
        optional DenseInfo denseinfo = 5;

        repeated sint64 lat = 8 [packed = true]; // DELTA coded
        repeated sint64 lon = 9 [packed = true]; // DELTA coded

        // Special packing of keys and vals into one array. May be empty if all nodes in this
            block are tagless.
        repeated int32 keys_vals = 10 [packed = true];
}
```

3.3.1 Dateizugriff

Die unterste Schicht bildet die Klasse `BlobFileIn`, sie ist für den Dateizugriff zuständig. Ein Beispiel für ein einfaches einlesen und dekodieren der enthaltenen Daten ist in Listing 3.8 zu sehen. Zuerst wird eine Instanz von `BlobFileIn` angelegt, dann der Dateiname übergeben und schließlich die Datei geöffnet. Für das Einlesen der Daten eines Blobs ist ein Puffer nötig. Hierbei kann dieser selbst angelegt und später freigegeben werden oder es wird einfach die Klasse `BlobDataBuffer` benutzt, die selbst einen Puffer bereitstellt und diesen verwaltet. Mit einem Aufruf von `readBlob(...)` wird der Blob an der aktuellen Stelle der Datei eingelesen, decodiert und dessen Daten in einen Puffer geschrieben. Der Typ der herausgelesenen Daten wird dabei von dieser Methode zurückgegeben bzw. oder der Instanz der `BlobDataBuffer` Klasse übergeben.

Intern verwendet `BlobFileIn` für den Dateizugriff die in Unix definierte `mmap`-Funktion. Hierbei wird die Datei in einen Speicherbereich des Arbeitsspeichers abgebildet und der Zugriff ist genauso gestaltet, wie auf andere Teile des Arbeitsspeichers. Im Hintergrund kümmert sich das Betriebssystem darum die gewünschten Teile des Inhalts einzulesen und als Cache noch im Speicher zu behalten.

Eine Schicht zwischen dem Dateizugriff und dem Einlesen der Daten aus den Blobs bildet die Klasse `OSMFileIn`. Beim öffnen einer Datei wird zusätzlich der erste Blob eingelesen und (falls vorhanden) die OSM-Kopfdaten analysiert. Diese werden auf Korrektheit überprüft und die dort definierten Eigenschaften, die nötig sind die darauf folgenden Daten einzulesen, werden mit den Eigenschaften der Bibliothek abgeglichen. Mit dieser Klasse können die eingelesenen Daten der Blobs direkt an die `PrimitiveBlockInputAdaptor` Klasse übergeben werden. Diese ist für die Dekodierung der Informationen einer serialisierten PrimitiveBlock message zusändig.

Ein Beispiel wie `OSMFileIn` zusammen mit `PrimitiveBlockInputAdaptor` verwendet wird ist in Listing 3.9 zu finden.

3.3.2 Zugriff auf Grundelemente

Die obersten Schichten bilden die Klassen `PrimitiveBlockInputAdaptor`, `INode*`, `IWay*` und `IRelation*`. Die `PrimitiveBlockInputAdaptor`-Klasse stellt Schnittstellen zur Verfügung um auf die Attribute der Grundelemente aus den dekodierten Daten zugreifen zu können. Jedem Typ von Grundelement wird eine Schnittstelle zugeordnet, so wird den Nodes (inkl. dense nodes) die Klassen `INode` und davon abgeleitet `INodeStream` zugeordnet. So ist dann (nach einer Vorverarbeitung der dense nodes) wahlfreier und stets sequentieller Zugriff auf die Nodes möglich. Ähnliches gilt auch für Ways und Relations.

Ein Beispiel für den Zugriff auf Nodes ist mit Listing 3.10 zu sehen.

Listing 3.8 Beispiel für einfaches einlesen einer Datei bestehend aus Blobs

```
1  #include <osmpbf/blobfile.h>
2
3  osmpbf::BlobFileIn inFile("path/to/file.blob");
4
5  inFile.open();
6
7  osmpbf::BlobDataBuffer buffer;
8
9  do {
10         inFile.readBlob(buffer);
11
12         // write some statistics
13         switch (buffer.type) {
14         case osmpbf::BLOB_OSMHeader:
15                 std::cout << "found OSM Header" << std::endl;
16                 break;
17         case osmpbf::BLOB_OSMData:
18                 std::cout << "found OSM Data" << std::endl;
19                 break;
20         case osmpbf::BLOB_Invalid:
21         default:
22                 std::cout << "invalid BLOB type" << std::endl;
23                 break;
24         }
25
26         std::cout << "current buffer size in bytes: " << buffer.totalBytes << std::endl;
27         std::cout << "size of extracted blob data in bytes: " << buffer.availableBytes <<
               std::endl;
28
29         /* do something interesting with blob data */
30
31 } while (buffer.type != BLOB_Invalid);
32
33 inFile.close();
34 buffer.clear(); // not neccessary, if the instance of this buffer is destroyed soon
```

Listing 3.9 Anwedungsbeispiel für die Klassen OSMFileIn und PrimitiveBlockInputAdaptor

```
#include <osmpbf/osmfile.h>
#include <osmpbf/primtiveblockinputadaptor.h>

osmpbf::OSMFileIn inFile("path/to/file.osm.pbf");

if (!inFile.open())
	return;

osmpbf::PrimitiveBlockInputAdaptor pbi;

while (inFile.parseNextBlock(pbi)) {
	if (pbi.isNull())
		continue;

	/* do some interesting stuff with the extracted data */
	std::cout << "found " << pbi.waysSize() << " ways" << std::endl;
	std::cout << "found " << pbi.nodesSize() << " nodes" << std::endl;
	std::cout << "found " << pbi.relationsSize() << " relations" << std::endl;
}

inFile.close();
```

Listing 3.10 Bespiel für den Zugriff auf Daten der Nodes mit Hilfe eines INodeStream

```
void dump(osmpbf::INode & node) {
	std::cout << "[Node]" <<
		"\nid = " << node.id() <<
		"\nlat = " << node.lati() <<
		"\nlon = " << node.loni() <<
		"\ntags ([#] <key> = <value>):" << std::endl;
	if (node.tagsSize())
		for (int i = 0; i < node.tagsSize(); i++)
			std::cout << '[' << i << "] " << node.key(i) << " = " << node.value(i)
			    << std::endl;
	else
		std::cout << " <none>" << std::endl;
}

void dumpNodes(osmpbf::PrimitiveBlockInputAdaptor & pbi) {
	for (osmpbf::INodeStream node = pbi.getNodeStream(); !node.isNull(); node.next())
		dump(node);
}
```

4 OSM Vector Map

Eine effiziente Darstellung von OSM-Daten für eine vorgegebene Region erfordert eine schnelle Suche von Objekten, die in diesen Bereich liegen. Möchte man einen Kartenausschnitt heraus vergrößert darstellen, so sind in der Regel nicht alle Details notwendig. Beispielsweise sind für eine solche Ansicht nur Autobahnen und Bundesstraßen relevant und kleinere Straßen innerhalb von Städten oder Pfade können weggelassen werden, da sie die Darstellung unter Umständen überladen könnten. Durch Vergrößerungsgrade bzw. -stufen werden diese Ansichten beschrieben, welche eine Suche komplizierter gestaltet.

Das PBF Format ist auf geringen Speicherverbrauch und schnellen (sequentiellen) Zugriff ausgelegt. Die Daten sind im Allgemeinen jedoch weder sortiert noch existiert eine Art Index für eine schnelle Suche. Nun wäre es denkbar die Daten im PBF Format geographisch sortiert zu speichern und diese nach Lage und Vergrößerungsgrad zu indizieren. Hierbei sind bei der Darstellung die relevanten Linienzüge nicht direkt verfügbar und deren Bestandteile müssen zunächst aus den PBF Daten extrahiert werden, was gerade für dense nodes einen hohen Aufwand bedeuten kann (siehe 3.2). Für eine Unterscheidung der Objektunterarten wie z. B. Straßen- oder Schienentypen muss jedem Linienzug ein Darstellungsstil (Renderingstil) zugeordnet sein. Diese Zuordnung kann wegen der großen Menge an Objekten nur unter Umständen im Hauptspeicher gehalten werden. Um diese Linienzüge schließlich akkurat auf der Ebene darzustellen müssen die Geokoordinaten in Ebenenkoordinaten umgewandelt werden, was einen hohen Rechenaufwand bedeuten kann.

Mit „OSM Vector Map“ wird nun in diesem Kapitel ein Format für vektorbasierte Karten vorgestellt, das Lösungen für die genannten Probleme bietet.

Als erstes wird kurz erläutert welche Daten für die Darstellung wichtig sind und wie die restlichen aussortiert werden. Anschließend werden die Renderingstile erklärt und wie sie zustande kommen. Danach wird die Organisation der Daten in der OSM Vector Map und die zu Grunde liegenden Datenstrukturen vorgestellt und zum Abschluss wird auf die Serialisierung der Kartendaten eingegangen.

4.1 Relevante Daten

Für eine Kartendarstellung werden nicht alle OSM Grundelemente und davon nicht alle Daten benötigt. Beispielsweise Informationen über die Historie der Grundelemente können weggelassen werden. Grundelmente wie einige Relations, die ausschließlich zur Gruppierung verwendet werden und keine visuelle Repräsentation besitzen sind ebenfalls irrelevant.

Filter	Kriterium
`KeyOnlyTagFilter`	Key und beliebiger Value
`StringTagFilter`	Key und Zeichenkette für Value
`MultiStringTagFilter`	Key und mindestens eine übergebene Zeichenkette als Value
`BoolTagFilter`	Key und Wahrheitswert für Value: - zulässige Werte für „wahr“: „true“, „yes“ und „1“ - zulässige Werte für „falsch“: „false“, „no“ und „0“
`IntTagFilter`	Key und ganze Zahl für Value
`AndTagFilter`	Übereinstimmung aller enthaltenen Filter
`OrTagFilter`	Übereinstimmung mindestens einem der enthaltenen Filter

Tabelle 4.1: Tagfilter und ihre Bedeutung: Hierbei wird für ein Grundelement der erste Tag gesucht mit dem das Kriterium übereinstimmt.

Verwendet man darüber hinaus mehrere Vergrößerungsstufen sind auf vielen Stufen nicht alle Details nötig.

Für die Darstellung des Kartenmaterials müssen nun aus den Grundelementen und deren Eigenschaften Polygone, Linienzüge und Punkte konstruiert werden, die nur noch Informationen über die Koordinaten, den jeweiligen Renderingstil und, sofern nötig, eine Bezeichnung enthalten. Diese Informationen werden hier vollständig aus den Attributen und den Geokoordinaten der Grundelmente abgeleitet.

Nun ist noch zu klären welche Grundelmente für die Darstellung der jeweiligen Zoomstufe herangezogen werden. Hierbei kann man sich mit Regeln für Renderingstile behelfen, wie sie in Mapnik verwendet werden. Dort werden Grundelmente nur dann dargestellt, wenn diese zu einem Renderingstil zugeordnet werden können. Das gleiche Prinzip wurde hier angewandt und wird im übernächsten Abschnitt zusammen mit den Renderingstilen erläutert.

4.1.1 Filter

Im weiteren Verlauf sind sogenannte Tagfilter von besonderer Bedeutung. Diese sind ein Hilfsmittel um Grundelemente nach ihren Attributen (Tags) auszusortieren. Der Einfachheit halber wurden Tagfilter als Teil der osmpbf Bibliothek implementiert. In der Tabelle 4.1 sind alle Filter mit ihrer jeweiligen Bedeutung aufgeführt.

4.2 Regeln und Renderingstile

Bei Kartendarstelltungen gibt es oft mehrere Vergrößerungsstufen. In dieser Arbeit wird in Anlehnung an Mapnik [Map] eine minimale Stufe 0 definiert, auf dieser sollen dann die wenigsten Details sichtbar sein. So sind beispielsweise auf niedrigen Zoomstufen Landstraßen

von geringerer Bedeutung und würden die Kartendarstellung mit Details überladen. Da es auf Karten eine bestimmte Rangfolge von Objekten gibt, um beispielsweise Brücken über den darunterliegenden Straßen dargestellt sind, sind Zeichenebenen eingeführt.

Für jede Zeichenebene wird nun eine Filterregel definiert, die bestimmen soll welche Grundelmente auf dieser Ebene dargestellt werden. Auf jeder Ebene werden anschließend Regeln definiert, die bestimmen sollen welchen Stil das jeweilige Grundelement für diese Ebene erhält. Diese Regeln setzen sich zusammen aus den Stilinformationen, einem Tagfilter und einer Zoomstufenbeschränkung.

Ein konkretes Beispiel dieser Definitionen ist in Listing 4.1 dargestellt. Hier wird eine Ebene definiert auf denen Objekte mit einem „highway“-Key dargestellt werden. Dieser Ebene werden dann zwei Stile mit demselben Filter aber für unterschiedliche Zoomstufenbereiche definiert.

Ein Renderingstil setzt sich aus drei Unterkategorien zusammen: Linien-, Füll- und Beschriftungsstil. Im Rahmen dieser Arbeit sind der Linien- und der Füllstil (bis auf die Einbindung externer Grafiken) vollständig unterstützt. Aus Gründen der Volständigkeit wurde eine message-Definition für Beschriftungsstile beigefügt.

So ist es möglich für jede Ebene Regeln aufzustellen, die dann den (für die jeweilige Zoomstufe) relevantenObjekten Renderingstile zuweist. Diese Stile werden bei der Kartendarstellung über Kennungsnummern (IDs) referenziert. Nun wird jedem Objekt ein sogenanntes „StyleIdTuple“ zugewiesen. Dieses ist nach folgender Vorschrift aufgebaut:

$$(\texttt{<style id>}^* \; 0)^*$$

Die 0 trennt die Ebenen voneinander ab, so kann ein Objekt in einer Ebene mehrere Stile besitzen. Bei der Darstellung wird das Objekt dann in einer Ebene nacheinander mit jedem Stil gezeichnet. So ist es möglich gerade kompliziertere Objekte wie z.B. Schienen akkurat darzustellen. Prinzipiell ist ein solches StyleIdTuple mit einem „Objekttyp“ vergleichbar: beispielsweise können alle Autobahnen dann denselben StyleIdTuple besitzen.

Für diese Arbeit wurde der OSM-Stil von Mapnik für die wichtigsten Verkerswege übernommen. Es besteht aber weiterhin die Möglchkeit die Implementierung um andere Objekte wie Flüsse, Seen, Wälder und vieles mehr zu erweitern oder sogar einen Parser für die Mapnik Stildefinitionen einzubinden.

4.3 Mercator-Projektion

Für die Darstellung der Linienzüge und Polygone ist eine Abbildung von Geokoordinaten auf Koordinaten in der Ebene nötig. Die Mercator-Projektion ist eine Form der Zylinderprojektion und ist die Standardprojektion von Geokoordinaten inOpenStreetMap [OSMc]. In [Osb08] ist eine detailierte Beschreibung und eine Herleitung der in dieser Arbeit verwendeten Formeln zu finden.

Listing 4.1 Beispiel für eine Stildefinition; Auszug aus der Standarddefinition; argbi() ist hierbei eine Hilfsfunktion, die den Opazitätswert setzt

```
[...]

LayerRule * layer;
StyleRule * style;
osmvmap::RenderStyle * renderStyle;

/*
 * LAYER fill-highway
 */
layer = addLayerRule(new osmpbf::KeyOnlyTagFilter("highway"));

// style roads - motorway 5-11
{
        osmpbf::MultiStringTagFilter * multiFilter = new
            osmpbf::MultiStringTagFilter("highway");
        *multiFilter << "motorway" << "motorway_link";

        style = addStyleRule(layer, multiFilter);

        style->minZoom = 5; style->maxZoom = 6;

        renderStyle = addRenderStyle(style);
        renderStyle->mutable_line_style()->set_color(argbi(0x809bc0));
        renderStyle->mutable_line_style()->set_width(0.5);

        style = addStyleRule(layer, multiFilter);

        style->minZoom = 7; style->maxZoom = 8;

        renderStyle = addRenderStyle(style);
        renderStyle->mutable_line_style()->set_color(argbi(0x809bc0));
        renderStyle->mutable_line_style()->set_width(1);

        style = addStyleRule(layer, multiFilter);

        [...]
}

[...]
```

Bei dieser Art der Koordinatentransformation sind die projizierten Koordinaten entlang der Zylinderachse verzerrt, sodass eine winkeltreue Abbildung entsteht. Dadurch werden kleine Objekte unverzerrt Dargestellt, wohingegen Richtungen und Flächengrößen mit zunehmendem Breitengrad immer stärker variieren. Für die Anwendung hier ist die Winkeltreue jedoch ausreichend.

Die Umrechnung von Geokoordinaten für die normale Lage, d.h. die Achse der Abbildungszylinder entspricht der Erdachse, ist gegeben durch

$$\begin{aligned} x &= \lambda - \lambda_0 \\ y &= \ln\left[\tan\left(\frac{\pi}{4} + \frac{\phi}{2}\right)\right] \end{aligned}$$

wobei λ die geographische Länge, λ_0 die Verschiebung zum Nullmeridian und ϕ die geographische Länge ist.

4.4 Organisation der Daten

Die Daten in der OSM Vector Map müssen so organisiert sein, dass eine schnelle Suche möglich ist, die Renderingstile und deren Referenzen möglichst vollständig im Arbeitsspeicher gehalten werden können und gleichzeitig die Linienzüge platzsparend abgespeichert werden.

Hierbei wird genauso wie im PBF Format auf die Google Protocol Buffer Bibliothek zurückgegriffen, um die Datenstrukturen mit dessen Hilfe zu serialisieren. Falls nicht anders angegeben ist in diesem Format vor jeder message eine Grössenangabe der message in einem 32 bit Integer im Big-Endian Format gespeichert.

Eine Datei im OSM Vector Map (OSMVMap) Format ist folgendermaßen aufgebaut:

VMapHeader message Kopfdaten bestehend aus Referenzen zu den Kartendaten, einer Bounding Box, der minimalen Zoomstufe für diesen Datensatz, der Anzahl an Renderingstilen und der Anzahl an StyleIdTuples

Renderingstile aneinander gereihte messages des Typs „RenderStyle“

StyleIdTuples aneinander gereihte messages des Typs „StyleIdTuple“

StringPool alle Zeichenketten, die in diesem Datensatz verwendet werden, aneinander gereiht

Kartendaten Je Zoomstufe folgt ein Datensatz mit den dort darzustellenden Objekten. Die Datensätze sind nach der Zoomstufe aufsteigend angeordnet.

Zur weiteren Illustration ist dieses Dateiformat in Abbildung 4.1 dargestellt.

Die Datensätze bestehen aus zwei serialisierten Datenstrukturen, einem Gitter und einem R-Baum. Diese sind in 4.5 genauer erklärt. Die Kartendaten in den oben genannten Strukturen setzen sich aus „PolyLines“ zusammen. Eine PolyLine ist ein Linienzug bestehend aus

Listing 4.2 Definition einer PolyLine message

```
message PolyLine {
        // rendering info
        optional bool is_closed = 1 [default = false];

        optional sint32 z_order = 2 [default = 0];

        optional uint32 style_id_tuple_ref = 4 [default = 0];

        optional uint32 label_id = 6 [default = 0];

        // nodes
        repeated sint64 node_X = 10 [packed = true]; // DELTA CODED
        repeated sint64 node_Y = 11 [packed = true]; // DELTA CODED
        repeated bool node_multiple_used = 12 [packed = true];
}
```

Koordinatenpaaren und für die Darstellung wichtige Zusatzinformationen. Verteilt werden diese auf das Gitter und den R-Baum je nach Darstellungsart.

Wird auf einen (OSM-)Datensatz ein Gitter gelegt so überlappen Linienzüge und Flächen möglicherweise mit mehreren Gitterzellen. Um möglichst wenig Redundanz bei den Zeichenoperationen zu erhalten müssen betreffende Linienzüge und Flächen auseinander geschnitten werden. Beim Schneiden einer beliebigen Fläche an einem Rechteck (hier eine Gitterzelle) kann es vorkommen, dass die Fläche in mehrere Teilflächen aufgespalten werden muss. Das Resultat sind kleine Flächen, die zusammen mehr Speicher benötigen als die Ursprüngliche.

Dies kann zwar umgangen werden indem Flächen vollständig zu jeder überlappenden Gitterzelle hinzugefügt werden, wodurch die jeweilige Fläsche unter Umständen mehrfach gezeichnet wird. Um dieses Problem zu lösen werden die Flächen nun in einen R-Baum eingefügt. Die Suche in einem R-Baum ist im schlimmsten Falle langsamer als die Suche in einem Gitter [Gut84]. Wie sich in Kapitel 6 herausstellen wird, ist jedoch in den meisten Fällen die Zeit für die Suche gegenüber der Dauer von Zeichenoperationen deutlich geringer.

PolyLines werden aus den Informationen der relevanten Ways und Nodes erzeugt. Eine PolyLine entspricht prinzipiell einem Way, dessen Knoten mit Hilfe der Mercator-Projektion auf ein ebenes Koordinatensystem umgewandelt werden. Die Zusatzinformationen für das Zeichnen der PolyLines bestehen aus einer Referenz auf einen StyleIdTuple, einer weiteren zu einer Beschriftung und einer Angabe ob der Linienzug geschlossen ist. Durch Letztere kann ausserdem ein Koordinatenpaar eingespart werden.

Um die Darstellung der PolyLines später weiter zu beschleunigen gibt es zusätzlich je Knoten (Koordinatenpaar) eine Angabe, ob dieser direkt mit anderen PolyLines verbunden ist. Wenn beispielsweise der Detailgrad reduziert werden soll, kann ein Renderer mit dieser Information solche Knoten weglassen, die nicht zu anderen Pfaden gehören.

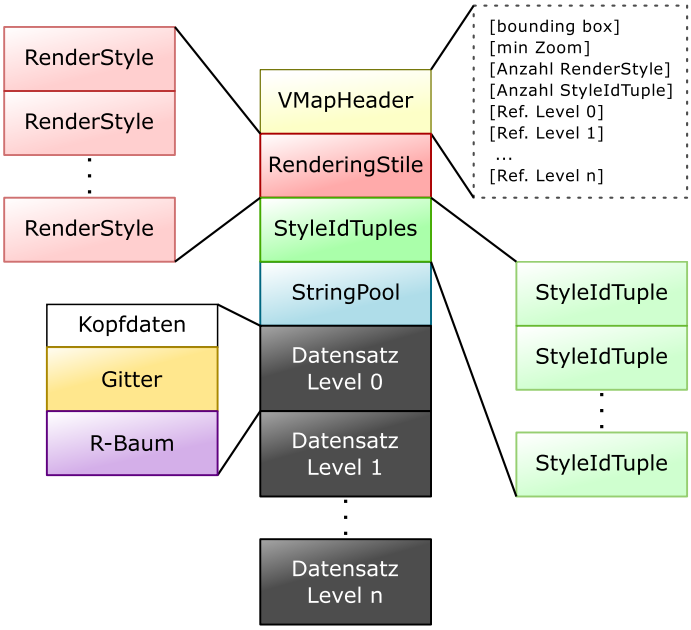

Abbildung 4.1: Schematische Darstellung des OSM Vector Map Formates

4.5 Datenstrukturen

Für die schnelle Suche auf geographischen Daten wie Straßenkarten sind Indexstrukturen nötig. Bei dieser Arbeit wurden zwei solche Strukturen, ein Gitter und ein R-Baum, implementiert, auf welche die Datenobjekte je nach Typ verteilt werden.

4.5.1 Gitter

Eine einfache Methode Daten räumlich zu indizieren ist mit Hilfe eines Gitters. Auf einem definierten Koordinatenbereich wird ein Gitter mit einheitlicher Kachelgröße gelegt und die zu indizierenden Objekte werden dann auf die Kacheln verteilt. Im einfachsten Fall besitzt jede Kachel einen sogenannten „Bucket“, in dem die Objekte beispielsweise als Liste abgelegt sind. Da sich Objekte über mehrere Kacheln hinweg erstrecken können gibt es einige Strategien, wie damit verfahren werden soll:

1. Objekt in jeder überlappenden Kachel abspeichern
2. Objekt auseinander schneiden (ist nicht immer trivial möglich)
3. Buckets auf mehrere Kacheln erweitern

In dieser Arbeit wurde das auftrennen der Objekte gewählt, da es sich bei den Objekten, die im Gitter gespeichert werden, um Linienzüge handelt. Alle weiteren Objekte werden separat behandelt.

In serialisierter Form besitzt das Gitter eine zweistufige Struktur. Die erste ist ein Gitter aus Referenzen und die zweite ein Feld aus Buckets. Die Referenzen zeigen dabei auf die entsprechenden Stellen im Feld. Das Feld aus Buckets ist dicht gepackt, d.h. leere Buckets erhalten im Referenzengitter den Wert 0 und sind im Feld nicht vorhanden.

4.5.2 R-Baum

Ein R-Baum ist eine räumliche Indexstruktur. Sie wurde von A. Guttman in [Gut84] eingeführt und ähnelt einem B-Baum (siehe [BM70]). Ein R-Baum ist darauf ausgelegt eine schnelle Suche auf mehrdimensionalen Objekten zu ermöglichen. Dazu gehören nicht nur Punkte sondern auch im zweidimensionalen Fall Polygone, im dreidimensionalen Fall Körper und beliebige Objekte höherer Dimensionalität. Eine Suchanfrage ist dabei immer ein (mehrdimensionales) Intervall bzw. ein an den Koordinatenaxen ausgerichtetes „Rechteck“.

Ein R-Baum ist ein höhenbalancierter Baum und trägt die zu indizierenden Informationen nur in seinen Blättern. Eine schematische Darstellung dieser Datenstruktur ist in Abbildung 4.2 zu finden. Jeder Information wird ein „minimum bounding rectangle“ (MBR) zugeordnet. Dies bezeichnet das kleinste mehrdimensionale Intervall, das die Ausdehnung der jeweiligen Informationen vollständig überdeckt.

Ein Blattknoten für N-dimensionale Daten (`<data>`) besitzt dann Einträge der Form

$$(MBR, \texttt{<data>}) \text{ mit } MBR = ([a_0, b_0], [a_1, b_1,], ...[a_{N-1}, b_{N-1}]) \ .$$

Alle anderen Knoten sind ähnlich aufgebaut, ersetzt wird hierbei der `<data>`-Eintrag durch eine Referenz auf die jeweiligen Kindknoten und das MBR beschreibt dabei das Intervall geringster Ausdehnung, das alle Intervalle des Kindknotens einschließt.

Bei dieser Konstruktion gibt es ausserdem einige Regeln:

- Alle Blattknoten sind auf derselben Tiefe angesiedelt.
- Die Anzahl der Einträge der Knoten (mit Ausnahme des Wurzelknotens) sind nach oben und nach unten hin beschränkt: M ist die maximale Anzahl und $m \leq \frac{M}{2}$ die minimale Anzahl.
- Ist der Wurzelknoten kein Blattknoten muss dieser mindestens 2 Kindknoten besitzen.

Für die Anwendung hier ist die Suchanfrage und der Aufbau des Baumes wichtig. Eine Suchanfrage ist dabei ähnlich wie bei einem B-Baum: Sei T der zu untersuchende Knoten des R-Baumes. Setzte zu beginn T auf den Wurzelknoten.

Suche in einem Blatt Ist T ein Blattknoten, überprüfe je Eintrag E_i das MBR mit dem Suchintervall auf Überlapp. Trifft dies auf ein E_i zu, so gehört dieser zum Ergebnis dazu.

Suche in einem Unterbaum Ist T kein Blattknoten, überprüfe je Eintrag R_i das MBR mit dem Suchintervall auf Überlapp. Führe die Suchanfrage auf jedes zutreffende R_i erneut mit $T := R_i$ aus.

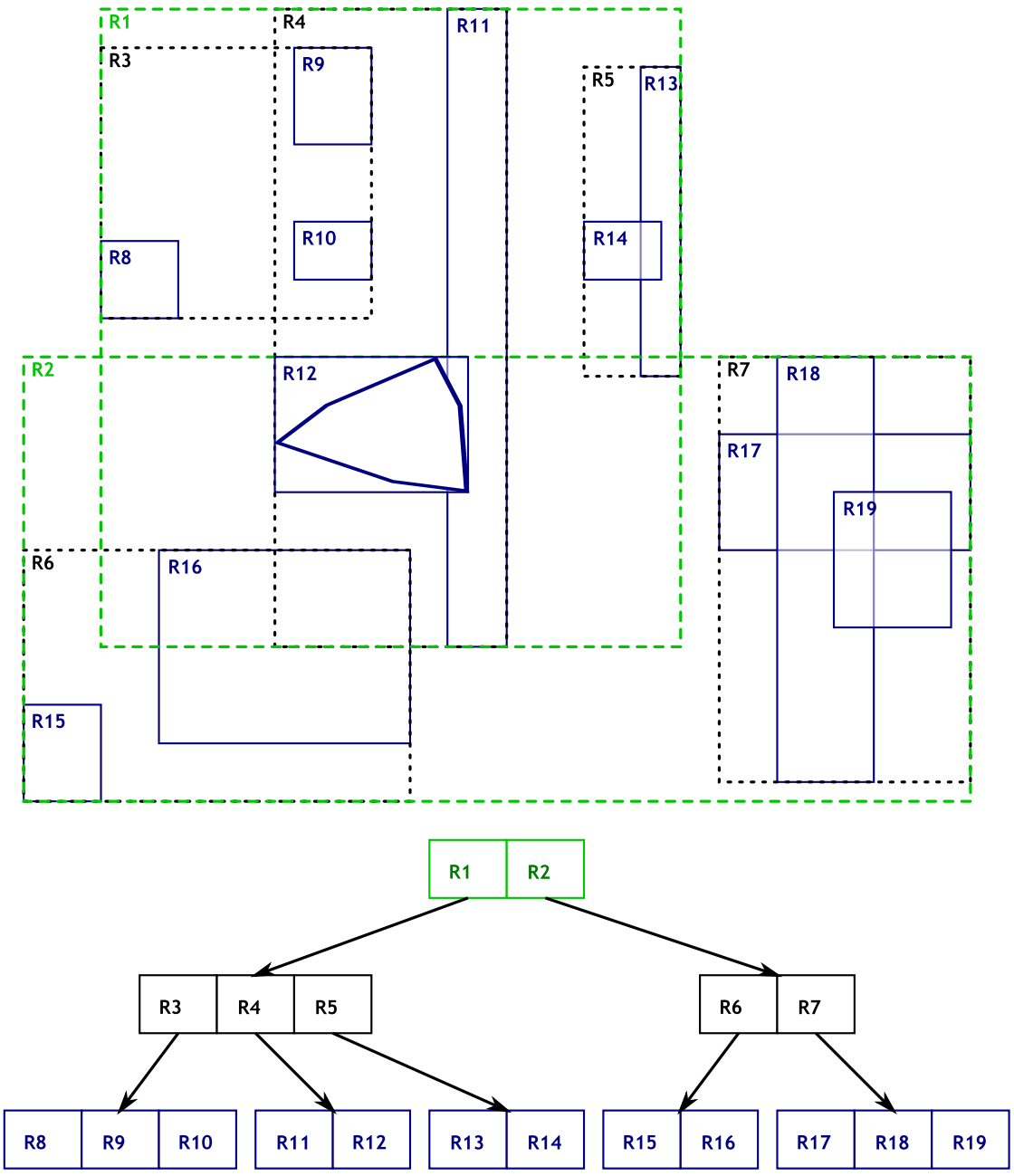

Abbildung 4.2: Schematische Darstellung eines R-Baumes, angelehnt an [Gut84].

Beim Aufbau des Baumes kann jeder Eintrag einzeln in den Baum eingefügt werden oder aber andererseits kann der Baum von den Blattknoten ausgehend bis zur Wurzel konstruiert werden. Durch letzteres Verfahren ist es nach [RL85] möglich bei statischen R-Bäumen die Suchanfrage zu beschleunigen, wenn die Einträge geschickt verteilt und die Knoten ebenso geschickt zusmmengefasst sind. Im weiteren Verlauf wird zunächst die Einfügeoperation aus [Gut84] vorgestellt und anschließend auf den `PACK`-Algorithmus aus [RL85] eingegangen.

Die Einfügeoperation `Insert` kann in zwei Teile aufgeteilt werden: Die Suche nach einem geeigneten Blattknoten für den neuen Eintrag und die Propagation der Änderung bis zur Wurzel durch gegebenenfalls nötige Spaltung von Knoten und Anpassung der MBR. Ist E der einzufügende Eintrag, so ist MBR_E das dazugehörige MBR:

1. Führe `ChooseLeaf` mit E aus und erhalte daraus mit L den Blattknoten für E.
2. Besitzt L genug Platz für E so füge E in L ein und erhalte L'. Anderenfalls führe `SplitNode` aus und erhalte die neuen Blattknoten L' und L''. Diese enthalten (zusammen) E und die Einträge aus L.
3. Führe `AdjustTree` auf L' und gegebenenfalls L'' aus.
4. Wurde im vorherigen Schritt der Wurzelknoten in R' und R'' aufgespalten, erstelle einen neuen Wurzelknoten R mit Referenzen auf R' und R''.

Die Methode für die Suche eines passenden Blattknotens, `ChooseLeaf`, für einen neuen Eintrag E ist in [Gut84] folgendermaßen dargestellt:

1. Setze den Knoten N auf den Wurzelknoten.
2. Ist N ein Blattknoten, gib N als Ergebnis zurück.
3. Anderenfalls wähle aus den Einträgen in N einen Eintrag E_i aus, dessen MBR durch eine Einfügeoperation am geringsten an Fläche zunimmt. Bei gleicher Zunahme zweier Einträge wähle denjenigen mit kleinerer Fläche aus.
4. Setzte N auf den in E_i referenzierten Kindknoten von N und gehe zu Punkt 2.

An dieser Stelle kann eine Optimierung durchgeführt werden, wodurch sich die Überlappung der Unterbäume des R-Baumes verringern lässt und somit die Suche beschleunigt wird. Eine Variante die Einfügeoperation zu optimieren ist in [BKSS90] vorgestellt, bei der nicht nur die Zunahme des Flächeninhalt sondern auch der Umfang als Maß für die Überlappung betrachtet wird.

Wurde nun ein passender Blattknoten gefunden, muss der Baum von dort bis zum Wurzelknoten auf die Änderung mit Hilfe der Methode `AdjustTree` aus [Gut84] angepasst werden. Als Eingabe erhält diese Funktion den angepassten Blattknoten L' und, falls eine Aufspaltung erfolgte, L'':

1. Setze $N_0 := L'$ und gegebenenfalls $N_1 := L''$.
2. Ist N_0 der Wurzelknoten, setzte $R' := N_0$ (und $R'' := N_1$) und halte.

3. Sei P der übergeordnete Knoten von N_0. Passe $MBR(P)$ an, sodass es alle Einträge von N_0 dicht umschließt.
4. Wurde N_1 gesetzt, erzeuge einen neuen Eintrag F. Setze die Referenz von F auf N_0, berechne MBR_N0 und setze $MBR_F := MBR(N_0)$. Besitzt P genügen Platz für F, füge F zu P hinzu. Anderenfalls rufe `SplitNode` auf und erhalte die neuen Knoten P' und P''.
5. Setze $N_0 := P'$ und $N_1 = P''$ und fahre mit Punkt 2 fort.

In `Insert` wie auch in `AdjustTree` wird die Teilungsoperation `SplitNode` aufgerufen, wenn der jeweilige Knoten nicht mehr genügend Platz für einen neuen Eintrag bietet. In [Gut84] sind zwei Varianten dargestellt, eine mit linearer Laufzeit und eine mit quadratischer Laufzeit. Da die letztere die besseren Ergebnisse liefert, wird sie im Folgenden grob umrissen:

Als Eingabe erhält die Methode eine Liste aus Einträgen eines Knotens inklusive den neuen Eintrag. Aus dieser Liste werden zwei Elemente herausgesucht, die am meisten Platz einnehmen, wenn man sie in einen Knoten zusammenfassen würde. Diese werden dann in den jeweils eine von zwei resultierenden Listen L_1 und L_2 eingefügt. Für jeden der restlichen Einträge N_i wird die Zunahme der Fläche d_1 und d_2 berechnet, wenn N_i zu L_1 bzw. L_2 hinzugefügt würde. Das N_i, welches die größte Präferenz zu einer der beiden Listen besitzt, wird ausgewählt. Dieser Eintrag wird dann zu derjenigen Liste hinzugefügt, deren Fläche nach dem Einfügen am geringsten zunimmt. Schließlich erhält man zwei Listen von Einträgen, die dann zu Knoten zu weiteren Verarbeitung zusammengefasst werden.

In [BKSS90] wird eine Variante der `SplitNode` vorgestellt. Hierbei werden die auf die beiden Listen L_1 und L_2 zu verteilenden Einträge zuerst sortiert und dann Verteilungen der Knoten ermittelt und jeweils ein Wert für den Überlapp berechnet:

$$\texttt{overlap-value} = \text{area}(MBR(L_1) \cap MBR(L_2))$$

Gewählt wird die Verteilung mit dem niedrigsten Wert für `overlap-value`. Haben zwei Verteilungen den gleichen Wert so entscheidet ein „Flächen-Wert“:

$$\texttt{area-value} = \text{area}(MBR(L_1)) - \text{area}(MBR(L_2)) \ .$$

Außerdem wird durch erneutes Einfügen von Einträgen versucht die Verteilung der Einträge zu verbessern und den Überlapp zu verringern.

Die zweite Variante den R-Baum aufzubauen ist von der Blättern zur Wurzel. Dies eignet sich für die Anwendung in dieser Arbeit ganz besonders, da der Baum statisch bleiben kann. Angelehnt an [RL85] soll der nun folgende `PACK`-Algorithmus zur Erzeugung des Baumes vorgestellt werden: Eingabe der (rekursiven) Methode ist eine Liste von Datenobjekten, `DLIST`.

1. Enthält `DLIST` weniger als M (maximal Anzahl an Einträgen je Knoten) Elemente, füge alle Elemente von `DLIST` zu einem neuen Knoten hinzu, gebe diesen zurück und halte.
2. Enthält `DLIST` mehr als M Elemente, sortiere diese (beispielsweise nach der ersten Koordinate). Lege leere Liste aus Elementen, `NLIST`, an.

3. Solange `DLIST` nicht leer ist, fasse das erste Element von `DLIST` und bis zu $M-1$ weitere, die am nächsten zum ersten liegen, zu einem neuen Knoten $N1$ zusammen und füge diesen zu `NLIST` hinzu.
4. Rufe `PACK` rekursiv mit dem Parameter `NLIST` auf. Gib die Rückgabe von `PACK` als Resultat zurück und halte.

Eine daran angelehnte Variante dieser Methode wurde im Rahmen der Arbeit implementiert. Sie weist den Einträgen für die Blattknoten einen Index der Z-Kurve (Lebesgue-Kurve) zu und sortiert diese dann danach. Anschließend werden nacheinander jeweils M Einträge zu Blattknoten zusammengefasst. Auf den höheren Ebenen werden die Knoten nach ihrer Erstellungsreihenfolge sortiert und ebenso jeweils zu bis zu M Knoten zusammengefasst.

In [KF93] wurde dieses Verfahren als Alternative zum Packed Hilbert R-Baum vorgestellt und ein ähnlich gutes Verhalten gezeigt. Die Z-Kurve ist eine sogenannte raumfüllende Kurve, sie wurde hier gewählt, da der Index durch bitweises Verschränken der Koordinaten berechnet werden kann.

4.6 Serialisierung

Nachdem die Datenstrukturen vorgestellt wurden, soll in diesem Abschnitt nun auf die Serialisierung des OSMVMap Formates und von Gitter und R-Baum eingegangen werden. Die Serialiserung der beiden Datenstrukturen ist dabei in Abbildung 4.3 veranschaulicht.

Wie in Abbildung 4.1 zu sehen war stehen nach den Kopfdaten des OSMVMap Formates die Renderingstile nacheinander als serialisierte RenderStyle messages gefolgt von den StyleIdTuple messages und den Zeichenketten. Danach folgen die Kartendaten mit Gitter und R-Baum.

Je Vergrößerungsgrad gibt es einen Datensatz mit Gitter und R-Baum. Die Kachelgröße w_K des Gitters auf höchster Zoomstufe K ist vorgegeben und verdoppelt sich bis zur niedrigsten mit jeder zweiten Stufe. Für den R-Baum wird bei der Serialisierung für alle Zoomstufen einheitlich eine maximale Anzahl an Unterbäumen je Knoten verwendet.

Damit die Datenobjekte in der richtigen Reihenfolge gezeichnet werden, müssen sie vor dem Einfügen in das Gitter nach ihrem „level“-Tag sortiert werden. Dieser gibt an auf welcher Ebene sich ein Objekt befindet, was beispielsweise für Brücken und Tunnels wichtig ist. Die Datenobjekte für Gitter und R-Baum bilden jeweils in Bytefolgen serialisierte PolyLine messages. Für das Gitter wurden betreffende Linienzüge aufgetrennt und um keine Löcher entstehen zu lassen an den Trennstellen jeweils ein weiterer Knoten als Rand hinzugefügt.

Nachdem Gitter und R-Baum für eine Vergrößerungsstufe aufgebaut und die Kopfdaten des Datensatzes heraus geschrieben sind, wird zuerst das Gitter und dann der R-Baum serialisiert. Wie in Abschnitt 4.5.1 erwähnt bildet das Gitter eine zweistufige Struktur aus Referenzengitter und Datenblöcken. Der R-Baum ist in Level-Order abgelegt, sodass bei kleinen Knoten mit den Referenzen ein Teil des Baumes mit im Dateisystemcache liegen kann.

Um das Erzeugen der Datenstrukturen, die Serialisierung und die Deserialisierung zu vereinfachen wurde dazu die Bibliothek `spatiallight` implementiert. Die mit ihr erzeugten Datensätze bestehen aus einem Gitter, einem R-Baum oder aus beidem, wobei die hier vorgestellen Methoden zum Aufbau und zur Serialisierung verwendet wurden. Die Deserialisierung (und Darstellung) der Datensätze wird dann in Kapitel 5 vorgestellt.

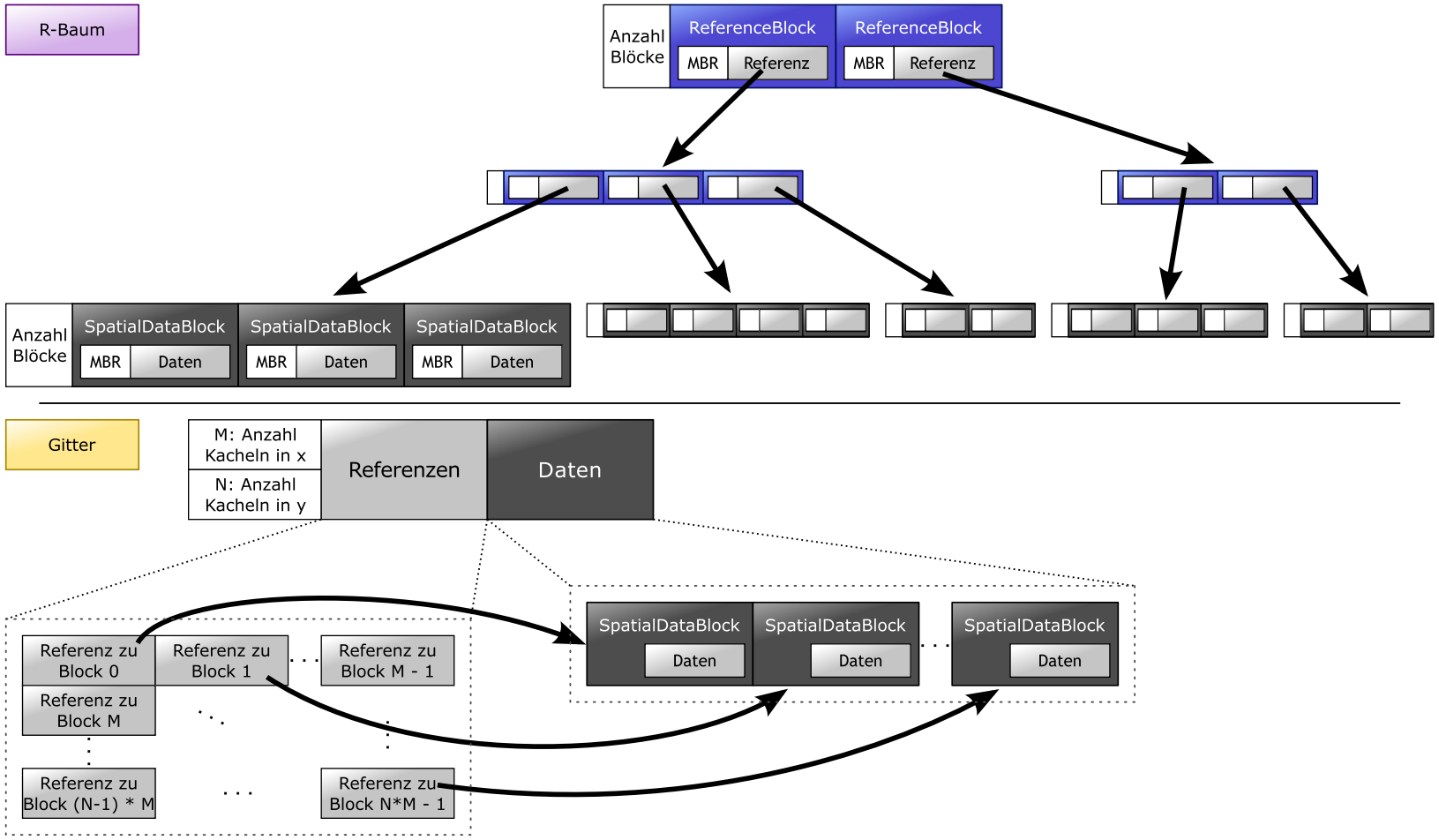

Abbildung 4.3: Schematische Darstellung eines Datensatzes als Gitter und R-Baum: Der R-Baum ist in Levelorder serialisiert. Die Referenzen und die Anzahl an Kacheln/Blöcke sind jeweils vorzeichenlose 64-bit Integer. Im R-Baum ist bei jeder Blockanzahl der Knotentyp in den niederwertigsten Bit codiert.

5 Kartenrenderer

Im vorherigen Kapitel wurde mit OSMVMap ein Format für Kartenmaterial vorgestellt, dessen Darstellung auf Vektorgrafiken basiert. Nun soll die Darstellung von diesem Katrenmaterial auf einem Mobilgerät angesprochen werden. In diesem Kapitel wird zunächst das Grundkonzept erläutert, anschließend die Vektorgrafik-Bibliothek „cairo“ vorgestellt um schließlich die Implementierung der Renderingbibliothek `osmvrender` und des Androidprogramms „OSMVmapView“ genauer darzulegen.

5.1 Konzept

Um ein felxibles und erweiterbares Grundgerüst aufzubauen, wurde die Aufgabe der Darstellung des Kartenmaterials auf Module verteilt:

1. Die Eingabeschicht für das Kartenmaterial, hier im OSMVMap Datenformat
2. Eine generische Renderingschicht, die aus vektorbasierten Kartendaten und Renderingstilen ein Bild in einen ebenso generischen Puffer synthetisiert
3. Die Ausgabeschicht, die das erzeugte Bild darstellt, in eine Datei schreibt, etc.
4. Die Zwischenschicht, welche die „Kommunikation“ zwischen Eingabe, Rendering und Ausgabe verwaltet und die nötigen Puffer bereitstellt.

Für eine schnelle Bildsynthese des Kartenmaterials ist nicht nur eine effiziente Organisation der Daten sondern auch eine Bibliothek für schnelles Rendern von Linienzügen und Polygonen nötig. Eine Möglichkeit wäre es OpenGL ES direkt zu nutzen. Bei komplizierteren Linienstilen wie gestrichelten Linien oder Verbindungstücken zwischen Linien müssen solche Elemente nachgebildet werden, was die Komplexität der Implementierung stark erhöhen würde. Dieses Problem wäre durch den Einsatz einer Vektorgrafikbibliothek gelöst. Ein Beispiel dafür ist die Grafikbibliothek cairo. Sie ermöglicht nicht nur effizientes Rendern in den Hauptspeicher sondern kann unter Umständen auch vorhandene Grafikhardware nutzen.

5.2 cairo Grafikbibliothek

Als Grundlage für die Bildsynthese von Kartenmaterial wurde in dieser Arbeit die oben erwähnte cairo Bibliothek ausgewählt. Es ist eine OpenSource Bibliothek, die unter anderem bei der Darstellung des GTK-Toolkits, des Gnome-Desktops, des Firefox (insbesondere die Mobilvariante) und vieles mehr eingesetzt wird. Sie besitzt eine große Zahl an Backends für das Erzeugen von Bildern aus Vektorgrafiken, vom einfachen Rendern in den Hauptspeicher bis hin zur Unterstützung der beschleunigten OpenGL und OpenGL ES 2.0 Bibliothek. Im weiteren Verlauf wird kurz auf das grundlegende Konzept eingegangen, weitere Informationen sind auf der Homepage des Projektes unter [Cai] zu finden.

Das Modell der cairo Bibliothek besteht aus zwei Teilen:

nouns Abstrakte Objekte wie Pfade, Masken, ein cairo-Kontext und einer Quelle sowie dem Ziel von Zeichenoperationen

verbs Methoden zur Manipulation der „nouns“

Jede Operation wird auf einen „cairo-Kontext“ angewendet bzw. diesem zugewiesen. Es repräsentiert eine Instanz der cairo Bibliothek und beinhaltet den aktuellen Zustand und alle nötigen Darstellungsattribute. Ein cairo-Kontext wird dabei aus einem `surface` erzeugt. Dies ist das Ziel von Zeichenoperationen und kann ein Puffer im Hauptspeicher, der Framebuffer der Grafikkarte oder auch eine Datei sein. Die einfachste Art eines `surface` ist der Typ `image_surface`, der einen Bereich im Hauptspeicher darstellt.

Die Quelle für Zeichenoperationen ist dabei die „Information“, welche auf das Ziel übertragen werden soll. Dies könnte eine einfache Farbe, ein Muster oder auch ein anderes `surface` sein. Eine Maske bestimmt dann auf welche Bereiche des Zieles die Quelle übertragen wird. Pfade bestehen hier aus Linienzügen und Kurven, die (ähnlich wie die Maske) Bereiche definieren, die gezeichnet werden sollen. Manipuliert werden die „nouns“ beispielsweise durch Zeichenoperationen, das setzen von Farben, die Erstellung von Pfaden und vieles mehr.

Mit cairo ist es außerdem möglich Text darzustellen und die Ausgabe kann mit Verschiebung, Skalierung, Rotation oder beliebigen Matrizen transformiert werden.

5.3 Implementierung

Das in Abschnitt 5.1 vorgestellte Konzept wurde mit der Implementierung der Renderingbibliothek `osmvrender` umgesetzt. Die vier Module sind dabei auf die Klassen `VMapInputAdaptor`, `RenderContext`, `VMapViewDelegate` und `VMapViewController` verteilt, die jeweils für die Eingabe, Bildsynthese, Ausgabe und Verwaltung zuständig sind. Zur Illustration ist in Abbildung 5.1 das Schema der Renderbibliothek dargestellt.

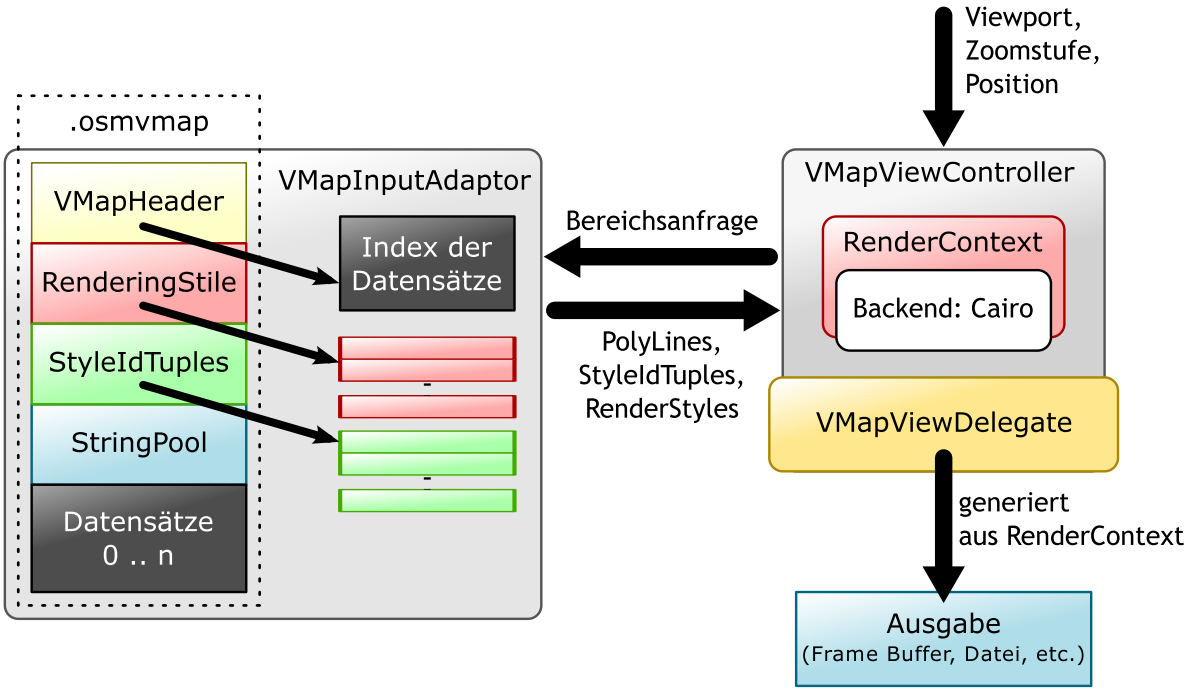

Abbildung 5.1: Schema der Renderbibliothek: der VMapViewController ist die zentrale Schnittstelle für das Rendern des Kartenmaterials; eine Instanz von VMapInputAdaptor stellt das Kartenmaterial bereit; auf einem RenderContext-Objekt wird ein Bild erzeugt; eine VMapViewDelegate Instanz kümmert sich um die Ausgabe

5.3.1 View Controller

Die Zentrale Verwaltung für die Darstellung der Kartendaten bildet der sogenannte „View Controller“. Dieser nimmt Parameter für die Anzeige entgegen, delegiert die Suche von Objekten im aktuellen Kartenausschnitt und stößt die Erzeugung eines Bildes (sowie die Anzeige dessen) an. Parameter für die Anzeige sind hierbei Größe des Ausgabebildes, die Geoposition des Kartenmittelpunktes und die Vergrößerungsstufe.

Der View Controller ist in der Klasse `VMapViewController` implementiert. Dieser wird bei der Initialisierung eine Eingabe- und Ausgabeschicht übergeben. Bereichsanfragen für den zu aktualisierenden Kartenausschnitt übergibt der View Controller der Eingabeschicht und stößt unter Umständen eine Vereinfachung der erhaltenen Daten an. Diese werden anschließend anhand der StyleIdTuples und des jeweiligen Renderingstiles auf ein `surface` der cairo-Bibliothek gerendert. Die Vereinfachung der Daten und die Kriterien dafür sind im nächsten Abschnitt über die Eingabeschicht ausführlicher erklärt.

Wird dem View Controller die Anfrage gestellt die Kartenansicht zu erzeugen geht dieser wie folgt vor:

1. Berechne aus der Zoomstufe z, dem Kartenmittelpunkt p, den Abmessungen (w, h) der Ausgabe und einem vom Ausgabegerät abhängigen Faktor f_G ein zweidimensionales Intervall $([x_l, x_h], [y_l, y_h])$ aus

$$\begin{aligned} f_z &:= 2^{31-z} \cdot f_G \\ x_l &:= p_x - \frac{w}{2} \cdot f_z; \\ x_h &:= p_x + \frac{w}{2} \cdot f_z; \\ y_l &:= p_y - \frac{h}{2} \cdot f_z; \\ y_h &:= p_y + \frac{h}{2} \cdot f_z; \end{aligned}$$

2. Übergebe dieses Intervall als Suchanfrage der Eingabeschicht.
3. Gibt es keine Ergebnisse in diesem Bereich, halte an.
4. Extrahiere anderenfalls aus den Ergebnissen die PolyLines, skaliere deren Koordinaten mit $1/f_z$, vereinfache deren Knotenzüge und füge sie schließlich in eine gemeinsame Liste L ein.
5. Iteriere über alle Ebenen E_i. Iteriere über alle PolyLines $P_j \in L$. Enthält das zur P_j gehörende StyleIdTuple für die Ebene E_i Einträge R_k, rendere P_j für alle R_k mit den dazugehörenden Renderingstilen.

Anmerkung: Ist die Anzahl der Vergößerungsstufen bei der Initialisierung bekannt, kann der jeweilige Faktor f_z vorausberechnet werden.

5.3.2 Eingabeschicht

Für das Einlesen, Decodieren und Bereitstellung der Daten ist die Klasse `VMapInputAdaptor` verantwortlich. Eine Instanz dessen erhält als Eingabe einen in Linux üblichen `file descriptor`, welcher einer Referenz auf einen generischen Datenstrom entspricht.

Bei der Initialisierung werden zunächst die Kopfdaten eingelesen, daraus ein Index über die Datensätze angelegt und schließlich werden die Renderingstile und die StyleIdTuple decodiert. Um Dateizugriffe einzusparen werden diese Strukturen vollständig im Hauptspeicher gehalten. Die Parameter der Renderingstile werden beim Einlesen zusätzlich in ein an cairo angepasstes Format umgewandelt. Nachdem dieser Schritt abgeschlossen ist kann nun auf die Informationen der Datensätze, Renderingstile, StyleIdTuples und Zeichenketten zugegriffen werden.

Um auf einen Datensatz für eine Vergrößerungsstufe zugreifen zu können liefert `VMapInputAdaptor` eine Instanz der `spatiallight::binary::InputAdaptor`-Klasse (im weiteren Verlauf kurz: `MapSource`). Wie der Name suggeriert gehört diese zur Eingabeschicht

der spatiallight-Bibliothek. Einer MapSource kann eine Anfrage in Form eines zweidimensionalen Intervalls gestellt werden. Die Instanz sammelt zunächst über das Referenzengitter (siehe 4.5.1) die mit dem Intervall überlappenden Gitterzellen ein und fügt die Referenzen in eine Warteschlange ein. Anschließend wird im R-Baum nach der ersten Überlappung eines Blattknoteneintrags mit der Anfrage gesucht und der Aktuelle Zustand gespeichert. War die Anfrage erfolgreich, so können nun iterativ alle Ergebnisse abgefragt werden.

Ein Ergebnis ist dabei eine message des Typs SpatialDataBlock, bzw. einer Instanz von ISpatialData um auf die serialisierten PolyLines der message zugreifen zu können. Zur Vorbereitung für die Darstellungsschicht müssen die PolyLines dekodiert werden. Diese Aufgabe übernimmt die statische Methode extractPolyLines der VMapInputAdaptor-Klasse, vereinfacht dabei die eingelesenen Linienzüge und Polygone und erstellt schließlich eine Liste aus den für die Darstellung nötigen Objekten.

Die Vereinfachung der PolyLines erfolgt nach folgenden Regeln:

- Anfangs- und Endpunkt dürfen nicht weggelassen werden.
- Punkte, deren Koordinaten in mehr als einer PolyLine verwendet werden, dürfen ebenfalls nicht entfernt werden.
- Iteriert man über die Konten einer PolyLine und ist die euklidsche Distanz zwischen dem aktuell betrachteten und dem nächsten Punkt geringer als $2,5$ Pixel so kann der nächste übersprungen (bzw. entfernt) werden.
- Besteht eine PolyLine nur aus zwei Punkten und beträgt die euklidsche Distanz zwischen diesen weniger als $0,9$ Pixel, lasse die PolyLine ganz weg. Dies gilt auch für schon vereinfachte PolyLines.
- Ist das MBR einer PolyLine Außerhalb des darzustellenden Bereiches, lasse diese weg.

Wie später in den Ergebnissen (Kapitel 6) gezeigt wird, verringert sich der Detailgrad durch diese Vereinfachung unmerklich, wogegen sich die Geschwindigkeit der Bildsynthese deutlich steigert.

5.3.3 Renderer

Das Modul für die vektorbasierte Bildsynthese entspricht der abstrakten Klasse AbstractRenderContext, welche cairo für die eigentliche Erzeugung des Resultates verwendet. Die Implementierung der Basisklasse verfügt über die Funktionalität PolyLines zu rendern und das Renderziel sowie der dazugehörige cairo-Kontext sind beliebig. Für die Bilderzeugung sind PolyLines und Renderingstile durch die Eingabeschicht so gestaltet, dass die jeweiligen Koordinaten der Knoten und Eigenschaften wie Liniendicke oder Farbe direkt an cairo übergeben werden.

In der Anwendung hier wird eine Instanz der ImageRenderContext-Klasse verwendet. Sie verfolgt einen einfachen Ansatz dessen Ziel der Zeichenoperationen ein image_surface ist. Es ist auch denkbar jeden anderen Typ zu verwenden, auch einen zum beschleunigten

Rendern mit Hilfe von OpenGL ES 2.0. Letzteres erfordert jedoch einen komplexeren Aufbau der resultierenden Android App und die nötige Unterstützung des jeweiligen Gerätes. Die Entscheidung für den simplen Ansatz kam dadurch zustande, dass die Erzeugung der Bilder mit einem `image_surface`ausreichend schnell ist (siehe Kapitel 6) und dieser Ansatz auf allen Versionen von Android ab 2.2 funktioniert.

5.3.4 View Delegate

Um die Anzeige bzw, Ausgabe der erzeugten Bildinformationen einer `AbtractRenderContext`-Instanz kümmert sich der sogenannte „View Delegate“. Dieser ist in der vollständig abstrakten Klasse `AbstractViewDelegate` definiert und jede Abgeleitete Klasse bestimmt die für die jeweilige Anwendung geeignete Funktionalität der Ausgabe. Dabei müssen in dieser mindestens die folgenden beiden Methoden implementiert sein:

display Dieser Methode wird eine Instanz einer `AbstractRenderKontext`-Klasse mit dem synthetisierten Bild und dessen Position auf der „Ausgabeebene“ übergeben. Mit dieser Funktion wird die Ausgabe erzeugt.

setViewPort Mit dieser Methode wird angegeben welche Abmessungen die Ausgabe (mindestens) besitzen soll.

In dieser Arbeit wurde ein View Delegate mit Hilfe von OpenGL ES 1.0 implementiert. Dieser wird in der Android App OSMVMapView eingesetzt, die im nächsten Abschnitt vorgestellt wird.

5.4 Die Android App OSMVMapView

Für die Darstellung des Kartenmaterials auf Mobilgeräten mit Android als Betriebssystem wurde die App OSMVMapView entwickelt. Sie benutzt eine native Bibliothek für die Suche, die Berechnungen und die Bildsynthese von OSMVMap-Daten. Eine schlichte Benutzeroberfläche zeigt dabei den Aktuellen Kartenauschnitt, drei Knöpfen zur Einstellung der Vergrösserungsstufe und zum Anstoßen der Bildsynthese und eine Ausgabe der verwendeten Zeit für die Darstellung. Ein Screenshot davon ist Abbildung 5.2 dargestellt.

Die Benutzeroberfläche wurde in Java programmiert, wobei aus Gründen der Einfachheit für die Anzeige eine abgeleitete Klasse von `GLSurfaceView` verwendet wird. Das Rendering wird dabei in einem zweiten Thread mit einen `NativeRenderer`-Objekt durchgeführt. Diese stellt die Schnittstelle zu einer einfachen Bibliothek dar, welche Funktionen von `osmvrender`, `osmvmap` und `spatiallight` benutzt. Für die Ausgabe wird OpenGL ES 1.0 verwendet und wird angesteuert von einem View Delegate von Typ `GLESTexViewDelegate`. Hierbei erfolgt die Darstellung über eine Textur, die auf ein Rechteck gelegt, angezeigt wird. Die Eingabedaten bezieht die App aus einer Datei im OSMVMap Format, die bei der Initialisierung der nativen Bibliothek geladen wird.

Testergebnisse zur Geschwindigkeit und Genauigkeit der Darstellung mit dieser App sind in Kapitel 6 zu finden.

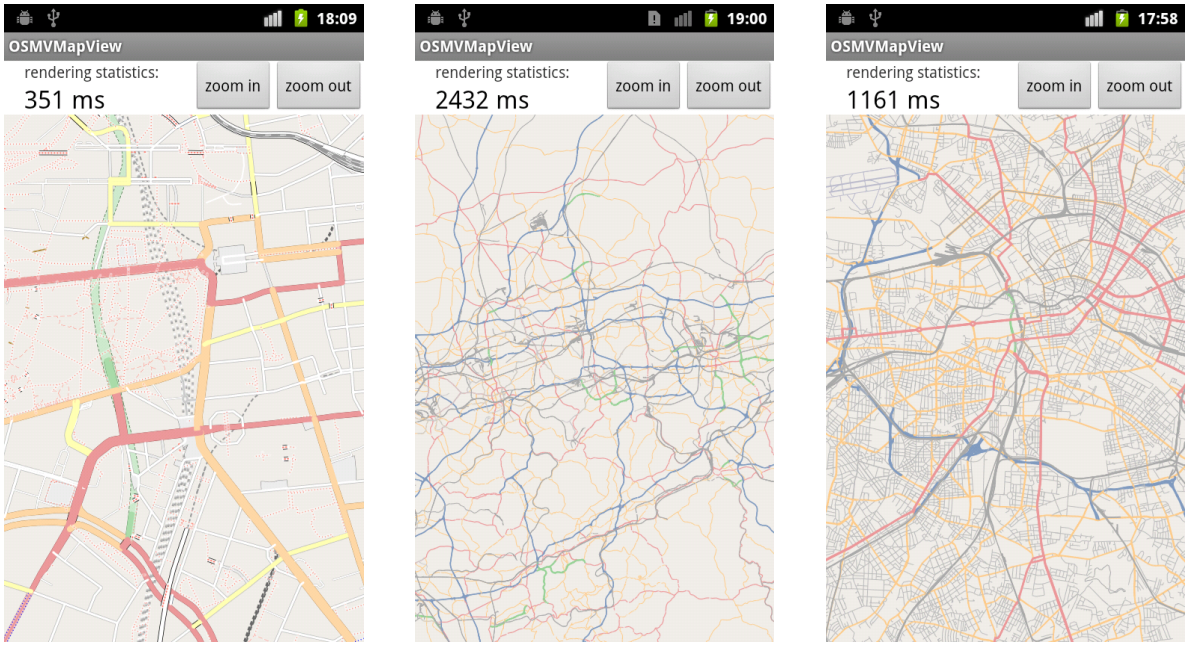

Abbildung 5.2: Screenshot von OSMVMapView

6 Ergebnisse

Nachdem die Renderingbibliothek, deren Implementierung sowie eine Anwendung vorgestellt wurden, sollen in diesem Kapitel Testergebnisse gezeigt werden. Als Testgeräte dienten ein Mobiltelefon des Typs Motorola Milestone und ein Tabletcomputer des Typs Point of View ProTab 25XXL.

Das Motorola Milestone ist mit einer ARM Cortex A8 CPU mit 550MHz Taktrate und 256MB Arbeitsspeicher ausgestattet, wovon im Normalbetrieb etwa 50MB für Andwendungen zur Verfügung stehen. Die Bildschirmauflösung beträgt 480 x 854 Pixel und als Betriebsystem ist Android in der Version 2.3.7 installiert. Der Tabletcomputer besitzt ebenfalls eine CPU von Typ ARM Cortex A8 jedoch mit bis zu 1,2GHz Taktfrequenz und mit 1GB einen deutlich größeren Hauptspeicher. Die Bildschirmauflösung beträgt 1024 x 600 und das installierte Betriebssystem ist Androiud 4.0.4.

Jeweils zwei Resultate des Kartenrendereres auf den Testgeräten sind in den Abbildungen 6.1 und 6.2 zu sehen. Oberhalb der Kartedarstellung ist jeweils die Zeit angezeigt, die benötigt wurde den aktuellen Ausschnitt darzustellen.

In den folgenden Abschnitten werden zunächst die Gesamtlaufzeiten auf den Testgeräten miteinander verglichen, dann folgt ein Vergleich verwendeten Datenstrukturen, anschließend wird der Einfluss durch die Vereinfachung von PolyLines gezeigt und schließlich werden die Auswirkungen der Parameter wie Kachelgröße des Gitters und Branchingfaktor des R-Baumes dargestellt.

6.1 Gesamtlaufzeiten

Laufzeiten für die Visualisierung der Kartendaten sind in den Tabellen 6.1 und 6.2 aufgelistet. Hier wurden die beiden Testgeräte miteinander verglichen. Als Grundlage dienten jeweils Datensätze von Baden Württemberg und Deutschland, deren minimale Kachelgröße w_K 500 Meter und der Maximale Branchingfaktor des R-Baumes 32 beträgt. Als Mittelpunkt des Kartenausschnittes wurde der Schlossplatz in Stuttgart gewählt, da diese Region in beiden Datensätzen vorhanden ist und der Detailgrad dort hoch ist. Bei den Testläufen ist zu beachten, dass es bei den Laufzeiten um die Höchstwerte handelt, die auftreten, wenn der Dateisystemcache nicht vollständig ausgenutzt werden kann. Die Ergebnisse zeigen bei beiden Geräten zunächst eine stark bzw. schwach zunehmende Laufzeit bedingt durch die Größe des jeweiligen Datensatzteiles und schließlich eine schnelle Abnahme der Laufzeit durch den kleiner werdenden Kartenausschnitt. Das Motorola Milestone hinkt dabei stets dem Leistungsstärkeren Point of View ProTab 25XXL hinterher, obwohl die

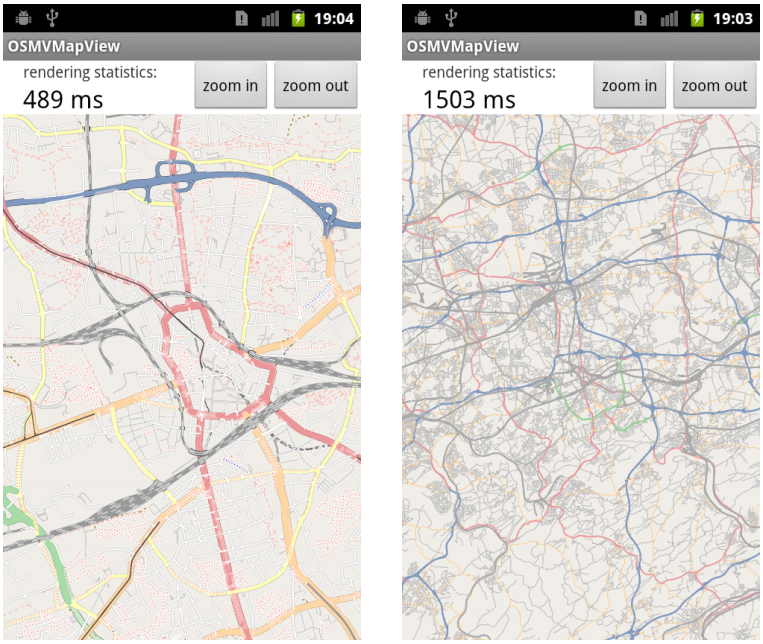

Abbildung 6.1: Kartendarstellung auf dem Motorola Milestone; Kartenmittelpunkt ist Bochum

Bildschirmauflösung beim ProTab 25XXL etwas höher ist und dadurch mehr dargestellt werden kann. Diese Auflistung zeigt auch, dass der Ansatz eines allein auf Software basierten Renderingverfahrens die Laufzeit zwar stark bremst, sich dies mit bis zu 2 Sekunden aber im Rahmen hält.

6.2 Datenstrukturen

Wie in Kapitel 4 bereits erwähnt, wird in OSMVMAP für die räumlische Indizierung ein Hybrid-Ansatz aus Gitter und R-Baum verwendet. Da im R-Baum ohne zusätzliche Verarbeitung nicht nur Flächen sondern auch einfache Linienzüge gespeichert werden können, wurden nun R-Baum und die Kombination aus Gitter und R-Baum miteinander verglichen.

Tabelle 6.3 zeigt für zwei unterschiedliche Datensätze die Zeiten, die für eine Suchanfrage benötigt wurden. Als Testgerät wurde das Motorola Milestone eingesetzt. Die Ergebnise zeigen, dass mit R-Baum effektiver Objekte für den aktuellen Kartenausschnitt gefunden werden, d.h. es werden keine überschüssigen Objekte gefunden. Nach der Suche im Gitter müssen zusätzlich alle Objekte gefiltert werden, um diejenigen zu entfernen, die nicht innerhalb des Kartenausschnittes liegen.

Jedoch dauert es in der Regel beim R-Baum länger als bei der Hybridlösung und die Zeiten für das Rendering sind zu ähnlich, um einen Ansatz mit dem R-Baum alleine zu

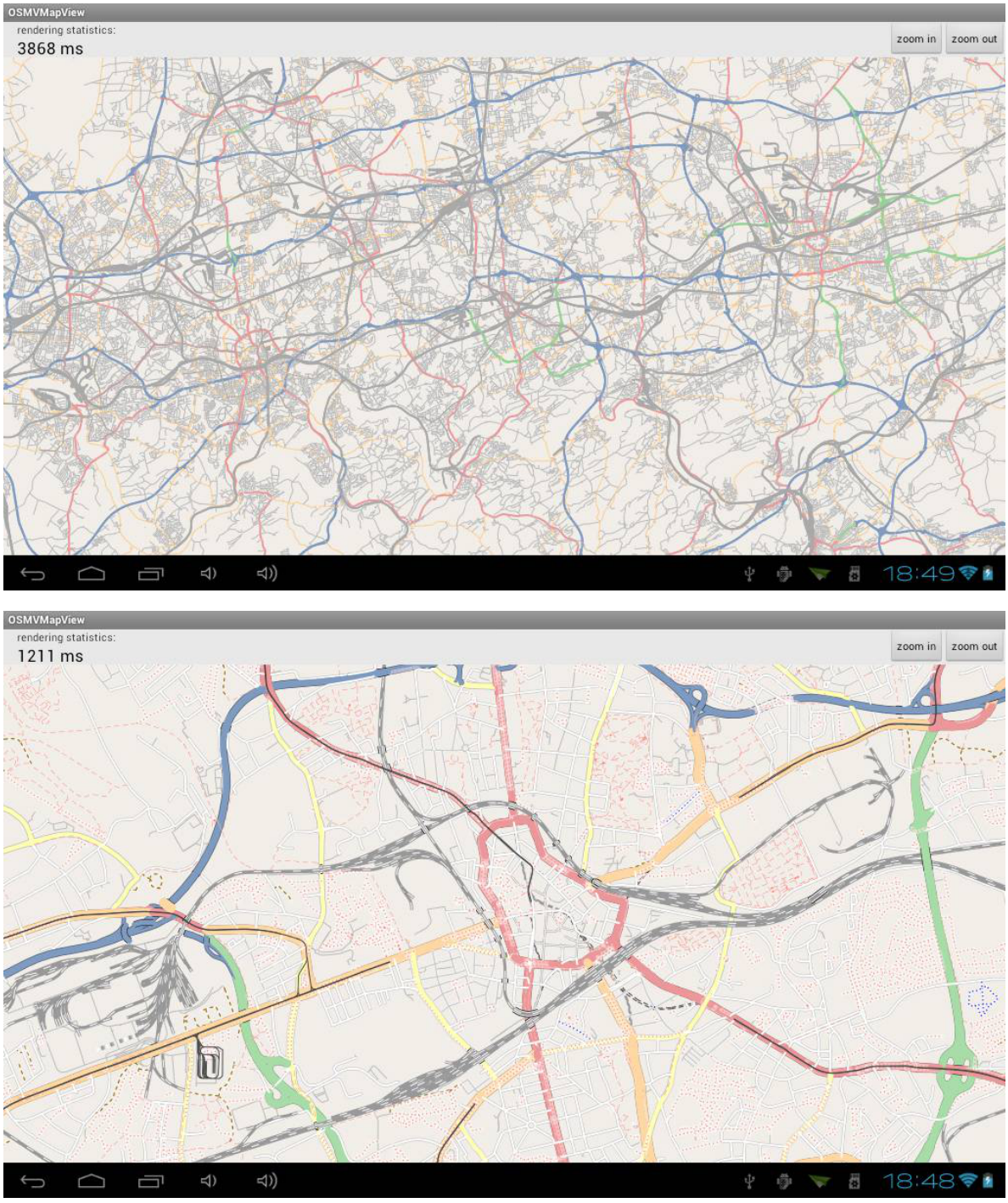

Abbildung 6.2: Kartendarstellung auf dem Point of View ProTab 25XXL; Kartenmittelpunkt ist Bochum

Motorola Milestone

	Baden-Württemberg			Deutschland		
Zoomstufe	gesamt	Rendering	#Obj.	gesamt	Rendering	#Obj.
5	256 ms	61 ms	1322	1404 ms	672 ms	13986
6	707 ms	411 ms	7868	4330 ms	2053 ms	37905
7	1152 ms	728 ms	14021	3170 ms	1450 ms	24810
8	1120 ms	658 ms	11702	1624 ms	680 ms	12111
9	1147 ms	682 ms	11136	1297 ms	701 ms	11144
10	2520 ms	1671 ms	25893	2470 ms	1652 ms	26033
11	1371 ms	1017 ms	13015	1846 ms	1053 ms	12920
12	912 ms	620 ms	5415	1263 ms	636 ms	5381
13	1382 ms	1110 ms	5334	2151 ms	1175 ms	5307
14	922 ms	628 ms	2294	1512 ms	613 ms	2280
15	455 ms	312 ms	917	1007 ms	302 ms	878
16	281 ms	145 ms	269	758 ms	123 ms	254
17	276 ms	69 ms	75	811 ms	70 ms	74
18	206 ms	43 ms	28	818 ms	56 ms	28

Tabelle 6.1: Auflistung der Gesamtlauf-, Renderingzeiten für die Darstellung bei Datensätzen von Baden-Württemberg und Deutschland.

Point of View ProTab 25XXL

	Baden-Württemberg			Deutschland		
Zoomstufe	gesamt	Rendering	#Obj.	gesamt	Rendering	#Obj.
5	214 ms	83 ms	1322	1344 ms	555 ms	12348
6	637 ms	373 ms	7868	2345 ms	1389 ms	28557
7	1032 ms	652 ms	13205	2432 ms	1570 ms	30643
8	929 ms	651 ms	12284	1180 ms	764 ms	14363
9	1174 ms	867 ms	14694	1168 ms	873 ms	14616
10	2506 ms	1998 ms	32653	2442 ms	1999 ms	32580
11	1396 ms	1183 ms	15690	1338 ms	1145 ms	15536
12	888 ms	773 ms	6770	845 ms	763 ms	6683
13	1451 ms	1350 ms	6418	1448 ms	1341 ms	6410
14	807 ms	743 ms	2696	758 ms	714 ms	2581
15	414 ms	389 ms	1076	427 ms	398 ms	1105
16	230 ms	209 ms	376	206 ms	190 ms	367
17	105 ms	96 ms	130	93 ms	86 ms	120
18	59 ms	55 ms	37	60 ms	55 ms	39

Tabelle 6.2: Auflistung der Gesamtlauf-, Renderingzeiten für die Darstellung bei Datensätzen von Baden-Württemberg und Deutschland.

Zoom-stufe	R-Baum		Gitter & R-Baum		
	Suche	#Obj.	Suche	Filter	#Obj.
5	196 ms	1282	124 ms	66 ms	9916
6	469 ms	7677	110 ms	198 ms	26469
9	518 ms	10519	254 ms	209 ms	23348
12	234 ms	5313	188 ms	104 ms	12295
15	274 ms	847	157 ms	20 ms	1942
18	289 ms	30	160 ms	5 ms	544

Renderingzeiten

Zoom-stufe	R-Baum alleine	Gitter & R-Baum
5	56 ms	66 ms
6	369 ms	399 ms
9	635 ms	684 ms
12	595 ms	620 ms
15	274 ms	278 ms
18	63 ms	42 ms

Tabelle 6.3: Vergleich der Suchzeit und Renderzeit für R-Baum und Hybridansatz; Datensatz: Baden-Württemberg; Kartenmittelpunkt: Schlossplatz in Stuttgart; Testgerät: Motorola Milestone

rechtfertigen. Dies betrifft vor allem die höheren Zoomstufen, da die Suchzeit beim R-Baum nicht signifikant abnimmt.

6.3 Vereinfachung der PolyLines

Gerade bei niedrigen Vergrößerungsgraden kann auf viele Details verzichtet und die Darstellung darurch beschleunigt werden. Getestet wurde auch hier mit deinem Datensatz von Baden-Württemberg auf dem Motorola Milestone.

Tabelle 6.4 zeigt einen Vergleich der beiden Fälle mit und ohne einen zusätzlichen Vereinfachungsschritt der gefundenen PolyLines. Bei den niedrigen Zoomstufen werden so bis zu 2/3 der Daten eingespart, woraus eine deutlich niedrigere Renderingzeit resultiert. Die entstehende Verzögerung durch den Mehraufwand wirkt sich effektiv nur auf die höheren Zoomstufen aus und ist dort zu vernachlässigen.

Es werden hierbei weniger Details angezeigt, da echt weniger PolyLines dargestellt sind. Abbildung 6.3 zeigt jedoch, dass die Unterschiede zwischen dem Original auf der linken Seite und der detailärmeren Fassung auf der rechten Seite auch im Differenzbild nicht signifikant sind.

Abbildung 6.3: Vergleich der Ergebnisse bei Vereinfachung der PolyLines; oben links: ohne Vereinfachung; oben rechts: mit Vereinachung; unten: Differenzbild (invertiert und leicht erhöhter Kontrast)

	ohne Vereinfachung		mit Vereinfachung		
Zoomstufe	Rendering	#Obj.	Rendering	Vorverabeitung	#Obj.
5	583 ms	9916	66 ms	77 ms	1305
6	1726 ms	26469	399 ms	204 ms	7761
9	1495 ms	18160	684 ms	201 ms	10811
12	706 ms	5460	620 ms	99 ms	5257
15	285 ms	845	278 ms	17 ms	845
18	43 ms	27	42 ms	4 ms	27

Tabelle 6.4: Vergleich der Renderzeiten bei Vereinfachung der PolyLines; Datensatz: Baden-Württemberg; Kartenmittelpunkt: Schlossplatz in Stuttgart; Testgerät: Motorola Milestone

6.4 Parameter bei der Erstellung

Wird ein Datensatz generiert, so bestimmen zwei Parameter nicht nur die Dateigröße sondern auch die Laufzeit der Bildsynthese (siehe 4.6). Der eine Parameter ist die Kantenlänge w_K einer Gitterkachel der größten Vergrößerungsstufe k und der andere ist die maximale Anzahl M an Unterbäumen eines Knotens im R-Baum.

Erwartungsgemäß würde eine kleinere Kachelgröße dazu führen, dass die durchschnittliche Anzahl der Objekte pro Kacheln sinkt und die Objektanzahl steigt, da Linienzüge über mehrere Kacheln öfter auseinander geschnitten werden müssen. Bei kleinen Kacheln sollte sich auch die Suchzeit verringern, da es wenig überschüssige Objekte gibt. Um dies zu prüfen, wurde bei einer Testreihe die Kachelgröße auf 1km gesetzt und dann schrittweise halbiert. Tabelle 6.5 mit den Ergebnissen der Testläufe zeigt hierbei jedoch keine großen Verbesserungen. Wie erwartet zeigt sich in Tabelle 6.6, dass mit kleiner werdenden Kacheln die Dateigröße steigt.

Bei einem R-Baum ist mit M, der Maximalen Anzahl an Unterbäumen je Knoten, ein Parameter um die Breite und Tiefe des resultierenden Baumes zu beeinflussen. Je tiefer der Baum umso mehr Knoten wären abzuspeichern, was die Dateigröße stark beeinflussen sollte. Sind in einem Knoten viele Unterbäume bzw. in den Blättern Datenobjekte vorhanden sollte sich die Suchzeit entsprechend verlängern. Die Ergebnisse der Testläufe in den Tabellen 6.5 und 6.6 bestätigen die Annahmen nur zum Teil. So verbessert sich die Zugriffszeit zwar leicht bei hoher Vergrößerungsstufe, die Dateigröße verändert sich jedoch nicht signifikant.

Baden-Württemberg

Zoomstufe	$w_K = 250m$	$w_K = 500m$	$w_K = 1000m$	$M = 256$	$M = 512$
5	201 ms	199 ms	256 ms	364 ms	361 ms
6	715 ms	724 ms	707 ms	1177 ms	1173 ms
9	1133 ms	1110 ms	1147 ms	1462 ms	1470 ms
12	913 ms	904 ms	912 ms	903 ms	953 ms
15	569 ms	510 ms	445 ms	475 ms	594 ms
18	219 ms	206 ms	207 ms	258 ms	372 ms

Deutschland

Zoomstufe	$w_K = 500m$	$w_K = 1000m$	$M = 512$	$M = 1024$
5	1404 ms	1548 ms	3509 ms	2981 ms
6	4330 ms	4795 ms	6923 ms	6995 ms
9	1297 ms	1269 ms	1447 ms	1478 ms
12	1263 ms	1104 ms	940 ms	973 ms
15	1007 ms	925 ms	601 ms	637 ms
18	818 ms	669 ms	287 ms	339 ms

Tabelle 6.5: Vergleich der Gesamtzeit für unterschiedlichen Kachelgrößen bei zwei Datensätzen von Baden-Württemberg und Deutschland. Beide Male ist der Kartenmittelpunkt der Schlossplatz in Stuttgart. Testgerät ist das Motorola Milestone

	$w_K = 250m$	$w_K = 500m$	$w_K = 1000m$	$M = 512$	$M = 1024$
Baden-Württemberg	709 MB	597 MB	544 MB	651 MB	651 MB
Deutschland	4,4 GB	3,7 GB	3,3 GB	4,0 GB	4,0 GB

Tabelle 6.6: Vergleich der resultierenden Dateigrößen bei zwei Datensätzen von Baden-Württemberg und Deutschland

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Methode vorgestellt Kartendaten effizient auf einem Mobilgerät zu visualisieren.

Als Plattform wurde Android gewählt, während die Wahl des Kartenmaterials auf das des OpenStreetMap-Projektes fiel. Hierbei liegen die Daten in zwei Formaten vor: das XML Format und das PBF Format. Für eine schnellere Verarbeitung dieser Daten fiel die Entscheidung auf das PBF Format. Um dieses Format einlesen zu können wurde eine Probgrammibliothek names `osmpbf` implementiert und in dieser Arbeit vorgestellt.

Mit dem Ziel eine effiziente Suche in Kartendaten und eine ebenso effiziente Bildsynthese zu ermöglichen wurde mit OSM Vector Map (OSMVMap) ein Format eingeführt um auf Vektorgrafik basierte Karten abzuspeichern. Hierzu wurde zunächst erläutert, dass nur ein bestimmter Teil der OSM-Daten für die Darstellung relevant sind. Ein Kriterium dafür ist eine Art Detailgrad durch die Vergrößerungsstufe der Kartenansicht vorgegeben. Beispielsweise erfordert die Darstellung eines großen Kartenausschnittes auf einem kleinen Bildschirm keine vollständige Ansicht aller Straßenarten. Genauso besitzen bereits viele andere Objekte keine visuelle Repräsentation und können weggelassen werden. Hierbei wurden mit Renderingstilen und Filtern Regeln auf Basis von Mapnik-Stilen eingeführt und deren Benutzung vorgestellt. Anschließend wurde die Organisation der Daten im OSMVMap Format und die speziell die darin enthaltenen Datenstrukturen für eine räumliche Indizierung erklärt. Bei diesem Ansatz sollen Linienzüge im Gitter und Flächen im R-Baum abgespeichert werden um ein Auseinanderschneiden von Flächen zu vermeiden. Beispielsweise bei Seen und Flüssen kann es mitunter oft größere und vor allem konvexe Flächen geben. Schließlich wurde dann die Serialisierung mit Hilfe der für diese Arbeit implementierten Programmbibliotheken `osmvmap` und `spaitallight` erklärt.

Darauf wurde der Kartenrenderer mit dem Grundkonzept bestehend aus 4 Modulen vorgestellt: die Eingabeschicht für das Kartenmaterial im OSMVMap Format, eine generische Renderingschicht für die Bildsynthese mit der cairo-Bibliothek, die Ausgabeschicht („View Delegate“) und eine Zwischenschicht („View Controller“), welche die Kommunikation unter den anderen Modulen steuert. Nachdem die jeweilige Funktionsweise und Implementierung erläutert war, wurde die Android App OSMVMapView vorgestellt. Sie wurde im Rahmen dieser Arbeit implementiert und deren Grundzüge sind hier erklärt worden.

Zuletzt wurden die Ergebnisse aus den Berechnungsläufen gezeigt. Die angewandten Datenstrukturen und Techniken wurden auf bis zu zwei Geräten getestet: Es zeigte sich, dass der Ansatz aus R-Baum und Gitter in der Regel schneller ist als der R-Baum alleine. Darüber hinaus konnte gezeigt werden, dass eine Vereinfachung von Linienzügen gerade bei niedrigem Detailgrad nicht nur die Berechnungszeit verkürzt sondern auch die Unterschiede zum

Original kaum merkbar sind. Zuletzt stellte sich heraus, dass sie Größe der Gitterkacheln kaum die Laufzeit jedoch die Dateigröße stark beeinflusst. Umgekehrt zeigte sich dies beim R-Baum für den Parameter der maximalen Kindknotenzahl: Mit abnehmender maximaler Kindknotenzahl vergrößert sich die resultierende Datei und die Suchzeit nimmt zu einem gewissen Grad ab.

Ausblick

Da für diese Arbeit eine grundlegende Implementierung der Kartendarstellung auf Mobilgeräten geschaffen wurde, gibt es eine Vielzahl an Punkten zum anknüpfen. Zunächst wäre da eine effektivere Art Renderingstile für die Vorverarbeitung zu definieren. Es ist denkbar hier ein Format beispielsweise auf XML-Basis zu definieren und einen Parser dafür zu implementieren. Darüber hinaus könnte der Ansatz der beschleunigten Bildsynthese mit OpenGL ES 2.0 weiter verfolgt werden, da die cairo-Bibliothek dies anbietet. In Hinblick auf aktuellere Mobilgeräte mit schnellerer Speicheranbindung wie z.B. der in Kapitel 6 vorgestellte Tablet-PC die Organisation der Daten im OSMVMap Format überdacht werden um eine speicherplatzsparende Variante zu finden.

Literaturverzeichnis

[Anda] Android Developers. URL http://developer.android.com/. (Zitiert auf den Seiten 9 und 10)

[andb] Dashboards - Android Developers. URL http://developer.android.com/about/dashboards/index.html. (Zitiert auf Seite 9)

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pp. 322–331. ACM, New York, NY, USA, 1990. (Zitiert auf den Seiten 36 und 37)

[BM70] R. Bayer, E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. In *SIGFIDET Workshop*, pp. 107–141. 1970. (Zitiert auf Seite 34)

[Cai] cairographics.org. URL http://www.cairographics.org/. (Zitiert auf Seite 42)

[GMa] Google Maps. URL http://maps.google.com. (Zitiert auf Seite 7)

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pp. 47–57. ACM, New York, NY, USA, 1984. doi:10.1145/602259.602266. (Zitiert auf den Seiten 32, 34, 35, 36 und 37)

[KF93] I. Kamel, C. Faloutsos. On packing R-trees. In *Proceedings of the second international conference on Information and knowledge management*, CIKM '93, pp. 490–499. ACM, New York, NY, USA, 1993. (Zitiert auf Seite 38)

[Map] mapnik Project. URL http://mapnik.org. (Zitiert auf den Seiten 7 und 28)

[Osb08] P. Osborne. The Mercator Projections, 2008. URL http://mercator.myzen.co.uk/mercator.pdf. (Zitiert auf Seite 29)

[OSMa] About - OpenStreetMap Wiki. URL http://wiki.openstreetmap.org/wiki/About. (Zitiert auf Seite 10)

[OSMb] Category:En:Tags - OpenStreetMap Wiki. URL http://wiki.openstreetmap.org/wiki/Category:En:Tags. (Zitiert auf Seite 11)

[OSMc] Mercator - OpenStreetMap Wiki. URL http://wiki.openstreetmap.org/wiki/Mercator. (Zitiert auf Seite 29)

[OSMd] OpenStreetMap. URL http://www.openstreetmap.org. (Zitiert auf Seite 7)

[OSMe] openstreetmap / osmosis - GitHub. URL https://github.com/openstreetmap/osmosis. (Zitiert auf Seite 20)

[OSMf] OpenStreetMap Wiki. URL http://wiki.openstreetmap.org. (Zitiert auf Seite 7)

[OSMg] OSM XML - OpenStreetMap Wiki. URL http://wiki.openstreetmap.org/wiki/OSM_XML. (Zitiert auf Seite 13)

[OSMh] PBF Format - OpenStreetMap Wiki. URL http://wiki.openstreetmap.org/wiki/PBF_Format. (Zitiert auf den Seiten 7, 15, 17, 20 und 21)

[PBD] Developer Guide - Protocol Buffers- Google Developers. URL https://developers.google.com/protocol-buffers/docs/overview. (Zitiert auf den Seiten 15, 16 und 17)

[RL85] N. Roussopoulos, D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, SIGMOD '85, pp. 17–31. ACM, New York, NY, USA, 1985. (Zitiert auf den Seiten 36 und 37)

[WGS00] Department of Defense World Geodetic System 1984. Technical Report TR 8350.2, 3rd Edition, NIMA – National Imagery and Mapping Agency, 2000. (Zitiert auf Seite 12)

Alle URLs wurden zuletzt am 19.10.2012 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Zeno-Oliver Groß)