Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

DotViewer: Darstellung gigantischer Punktwolken auf Android Geräten

Jakob Krause
Matrikelnummer: 4573260
jakobkrause@zedat.fu-berlin.de

Betreuung und Erstgutachten:
Prof. Dr. Marco Block-Berlitz

7. Juli 2016

Eidesstattliche Erklärung

zur Bachelorarbeit

Name: _____ Vorname: _____

Ich versichere, die Bachelorarbeit selbständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben.

Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Berlin, den _____

Inhaltsverzeichnis

1 Einleitung und Motivation **3**
- 1.1 Archaeocopter und Archaeonautic Projekt . . . 3
- 1.2 Ziele und Aufbau der Arbeit . . . 4

2 Theorie und verwandte Arbeiten **5**
- 2.1 Einführung in die 3D-Rekonstruktion . . . 5
  - 2.1.1 Aktive Rekonstruktion . . . 5
  - 2.1.2 Passive Rekonstruktion . . . 6
  - 2.1.3 VisualSFM . . . 6
- 2.2 Minimum Bounding-Box . . . 6
- 2.3 Repräsentationen von Punktwolken . . . 7
  - 2.3.1 QSplat Verfahren . . . 7
  - 2.3.2 Octree . . . 8
  - 2.3.3 Multiresolution Octree . . . 9
  - 2.3.4 Moving-Least-Squares Surfaces . . . 9
  - 2.3.5 Tetraedertree . . . 11
- 2.4 Bestehende Applikationen . . . 12
  - 2.4.1 KiwiViewer . . . 12
  - 2.4.2 LiMo . . . 12
  - 2.4.3 iOS App mit Knn-Baum . . . 12
  - 2.4.4 HuMoRS . . . 13
- 2.5 Proxy Entwurfsmuster . . . 14
- 2.6 Singleton Entwurfsmuster . . . 14
- 2.7 Representational State Transfer . . . 15

3 Frameworks und Architektur **17**
- 3.1 Java und Android . . . 17
- 3.2 Google Protocol Buffers . . . 18
- 3.3 Open-Graphics-Library for Embedded-Systems . . . 18
  - 3.3.1 Vertex . . . 18
  - 3.3.2 Shader . . . 19
  - 3.3.3 Pipeline . . . 19
- 3.4 NanoHTTPD . . . 19
- 3.5 la4j . . . 20
- 3.6 DEFLATE . . . 20
- 3.7 Volley . . . 20

4 dotViewer **21**
4.1 Server . . . 21
4.1.1 Multiresolution Tree . . . 21
4.1.2 RESTful API . . . 22
4.1.2.1 MRT-Proxy Anfrage . . . 23
4.1.2.2 Punktanfrage . . . 24
4.2 Client . . . 24
4.2.1 Netzwerkverkehr . . . 25
4.2.2 Rendering . . . 26
4.2.2.1 Shader . . . 26
4.2.2.2 Scene Klasse . . . 26
4.2.2.3 RemotePointClusterGL . . . 27
4.2.2.4 DrawableCache Klasse . . . 28
4.2.3 Benutzer Interface . . . 28

5 Experimente und Auswertungen **30**
5.1 Durchführung . . . 30
5.2 Auswertung . . . 30

6 Zusammenfassung und Ausblick **36**
6.1 Zusammenfassung . . . 36
6.2 Ausblick . . . 36

Literaturverzeichnis **37**

1 Einleitung und Motivation

Das Interesse an der Darstellung von großen Punktwolken ist durch das Aufkommen von günstigen 3D-Scannern und der 3D-Rekonstruktion in den letzten Jahrzehnten stark gewachsen. Aufgrund der technologischen Entwicklungen in diesem Bereich ist es möglich von kleinen Objekten, wie einem Dinosaurierschädel bis hin zu ganzen Städten, Modelle in hoher Detailstufe digital zu erfassen. In der Denkmalpflege werden Objekte mittlerweile aus Routine abgescannt und archiviert.

Für gewöhnlich sind die erzeugten Modelle enorm groß und daher nicht ohne weiteres darstellbar. Die Modelle bestehen meist aus einer ungeordneten Sammlung von dreidimensionalen Punkten mit Farbinformationen. Desktop-Lösungen verlassen sich zur Lösung des Problems auf Level-of-Detail (LOD) Konzepte kombiniert mit Out-of-core Verfahren und klugen Caching Strategien, um der hohen Datenmenge Herr zu werden. Durch das Aufkommen von leistungsstarken Smartphones entstand eine neue Plattform zum Darstellen von Modellen.

Daraus entstand eine Nachfrage durch Forschungsgruppen für eine mobile Applikation zum Betrachten großer Modelle. Die Applikation kann beispielsweise bei 3D-Rekonstruktion schnelles Feedback liefern, vereint mit den Vorzügen eines mobilen Gerätes. Dadurch können schnell unvollständige oder schwer zu erfassende Bereiche des Modells beim Scannen erkannt werden. Des Weiteren kann die Applikation zur Präsentation von Modellen eingesetzt werden.

Die Herausforderung bestand darin, trotz der Limitierungen eines Tablets durch seinen geringen Arbeitsspeicher und der relativ schwachen GPU, Punktwolken mit mehreren Millionen Punkten flüssig darzustellen. Dabei sollte es möglich sein, jederzeit neue Punkte in die bestehende Darstellung hinzuzufügen. Zur Zeit existieren nur bedingt geeignete Softwarelösungen für das Problem.

1.1 Archaeocopter und Archaeonautic Projekt

Das Archaeocopter Projekt, welches 2012 von Dr. Benjamin Ducke und Prof. Dr. Marco Block-Berlitz ins Leben gerufen wurde, hat es sich zum Ziel gesetzt, ein unbemanntes Flugobjekt (UAV) zu entwickeln, welches durch halb-autonome Flüge Archäologen bei ihrer Arbeit durch Luftaufnahmen unterstützt. Aus diesen Aufnahmen lassen sich durch 3D-Rekonstruktion Modelle generieren. Diese Technik wird auch im Denkmalschutz angewandt.

Die Idee war es, aktuelle Verfahren aus Computervision und künstlicher Intelligenz zusammen mit UAVs mit Kameras für die Datenerhebung einzusetzen. Das innovative Verfahren lässt sich auch unter Wasser einsetzen. Offiziell ging das Projekt im September

Abbildung 1.1: Ein UAV beginnt mit dem Filmen einer Landschaft. Aus dem Videomaterial kann später ein dreidimensionales Modell erstellt werden.

2012 mit Unterstützung von Prof. Dr. Raúl Rojas von Berlin's Freier Universität an den Start. Diese Arbeit ist in Zusammenarbeit mit dem Projekt entstanden.

1.2 Ziele und Aufbau der Arbeit

Ziel dieser Arbeit ist eine mobile Applikation für das Android Betriebssystem zu entwickeln, welche die folgenden Anforderungen erfüllen soll:

1. flüssige Darstellung von Punktwolken mit mehreren Millionen Punkten
2. neue Punkte können zur Laufzeit in die Darstellung hinzugefügt werden
3. Implementierung in Java
4. modularer Aufbau
5. einfache Nutzung

Dafür wird in Kapitel 2 ein Einblick in den Stand der Forschung mitsamt bestehender Arbeiten aus dem Feld gegeben. Bestehende Ansätze und Datenstrukturen werden mit den Anforderungen verglichen, um festzustellen, ob sie verwendbar sind. Im folgendem Kapitel werden in der Arbeit verwendete Technologien vorgestellt. Der Aufbau sowie die Implementierung der Applikation wird in Kapitel 4 beschrieben. In Kapitel 5 wird die Bildrate und Bildqualität der Applikation anhand zweier Beispielmodelle evaluiert. Im letzten Kapitel wird die Arbeit zusammengefasst. Des Weiteren werden mögliche Verbesserungen vorgestellt.

Der Quellcode wird nach Abschluss dieser Arbeit über die Entwicklungsplattform Github[1] frei zugänglich gemacht.

[1] http://www.github.com

2 Theorie und verwandte Arbeiten

In diesem Kapitel werden Grundlagen zum Verstehen der Arbeit und des Kontextes vorgestellt. Es wird eine kurze Einführung in die 3D-Rekonstruktion gemacht. Des Weiteren werden Punktwolken sowie deren Repräsentationen diskutiert. Anschließend werden bestehende Arbeiten diskutiert. Zum Abschluss werden genutzte Entwurfsmuster und Architekturstile vorgestellt.

2.1 Einführung in die 3D-Rekonstruktion

Die Idee, aus einer Folge von Bildern ein 3D-Modell zu errechnen, ist eines der Kernthemen der Computervision. Verwendungen dieser Technik sind vielfältig und finden sich in Wissenschaft und Wirtschaft wieder. Anwendungen existieren zum Beispiel in der Robotik [10], in welcher mit Hilfe eines Stereokamerasystems die Position des Roboters innerhalb seiner Umgebung feststellbar ist. Ein weiteres Feld ist die Archäologie und der Denkmalsschutz. Ein Beispiel dafür ist das Archaeocopter Projekt.

Allgemein kann man zwischen aktiver und passiver Rekonstruktion [12] unterscheiden. Die beiden Verfahren werden im Anschluss kurz vorgestellt.

2.1.1 Aktive Rekonstruktion

Bei aktiver Rekonstruktion wird aktiv mit einem Sensor das Objekt abgetastet, um die Struktur zu ermitteln. 3D-Scanner sind ein Vertreter dieser Gattung. Im Grunde sind sie der Kamera ähnlich. Genau wie diese besitzen sie ein Sichtfeld. Allerdings liefern sie statt Farbwerte, Abstandswerte von ihrem Sichtfeld. Die Abstandswerte können unter anderem mit Hilfe einer Time-of-Flight Camera oder der Triangulierungsmethode ermittelt werden.

Bei einer Time-of-Flight Kamera [16] wird ein modulierter Lichtstrahl versendet. Aus der Dauer bis der Strahl nach der Reflexion seinen Ausgangspunkt erreicht, kann die Entfernung vom Objekt ermittelt werden. Diese Methode ist bei nahen und feinen Objekten ungenau, weil die Zeit nur zu einer gewissen Genauigkeit gemessen werden kann. Die Auflösung erreicht etwa 320x240 Pixel. Vorteilhaft sind die hohen Bildraten von bis zu 160 Bildern pro Sekunde.

Bei der Triangulierungsmethode wird von einem Laser ein Punkt auf das Objekt projiziert. Dieser Punkt wird von einer Kamera erfasst. In Abhängigkeit von der Entfernung erscheint der Laserpunkt im Sichtfeld der Kamera. Um das Verfahren zu beschleunigen, kann statt einem Punkt eine Linie verwendet werden (siehe Abb. 2.1).

Diese Methode ist sehr genau und daher für Skulpturen gut geeignet.

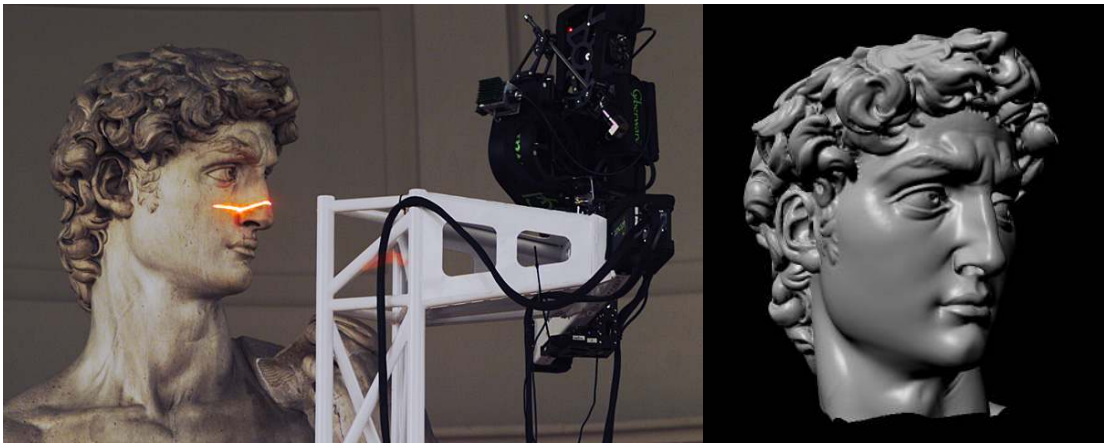

Abbildung 2.1: Links: Anwendung der Triangulierungsmethode mit Hilfe einer Scanline beim „The Digital Michelangelo“ Projekt, Rechts: Das Ergebnis des Scanvorganges (Abb. aus [18])

2.1.2 Passive Rekonstruktion

Unter passiver Rekonstruktion versteht man Methoden, welche nicht aktiv eine Szene abtasten, sondern vorhandene photometrische Informationen (z.B. Fotos) nutzen, um die Tiefe zu berechnen. Das Stereo-Verfahren [12] ist eines der ersten in diesem Feld. Ausgehend von zwei auf der x-Achse verschobenen Bildern einer Szene gilt es, Punktpaare zwischen den beiden Bildern zu finden. Um das zu vereinfachen, sucht man nach einer Abbildung von Punkten aus Bild 1 zu Bild 2. Aufgrund der Verschiebung der Bilder auf der x-Achse kann man durch Epipolargeometrie die Tiefe der Punkte berechnen.

2.1.3 VisualSFM

Bei VisualSFM[1] handelt es sich um ein 3D-Rekonstruktionssystem. Es verfügt über eine grafische Benutzeroberfläche und zeichnet sich durch Skalierbarkeit auf Grundlage der Nutzung von nVidia oder ATI Grafikkarten des Computers aus. Entwickelt wurde das Programm von dem Studenten Changchang Wu[2], welcher mittlerweile bei Google beschäftigt ist. VisualSFM wird von dem Archaeocopter Projekt zur 3D-Rekonstruktion verwendet.

2.2 Minimum Bounding-Box

In der Geometrie versteht man unter einer Minimum Bounding-Box ein Rechteck (2D) oder Box (3D) welches eine Menge von Punkten umschließt. Dabei ist die Fläche bzw. das Volumen minimal.

[1] http://ccwu.me/vsfm/. Aufg. am 24.05.2016
[2] http://ccwu.me/. Aufg. am 24.05.2016

2.3 Repräsentationen von Punktwolken

Punkt basierende geometrische Oberflächen können als Stichprobenmenge einer kontinuierlichen Oberfläche verstanden werden. Dabei entstehen dreidimensionale Raumkoordinaten $p_i \in \mathbb{R}^3$. Oft existieren noch weitere Daten zu dem Punkt, wie eine Normale n_i oder eine Farbe c_i. Eine Punktwolke S ist eine Menge solcher Punkte. Im folgendem wird eine Auswahl von Ansätze vorgestellt.

2.3.1 QSplat Verfahren

Erste Vorschläge zur Darstellung von sehr großen Punktmengen wurden durch das QSplat Verfahren gemacht [7]. Splat ist das englische Wort für Farbklecks.

Die Punktmenge wird durch eine Hierarchie auf Basis von Kugeln modelliert. Jede Kugel repräsentiert einen Knoten in einem Binärbaum. Jeder Knoten enthält den Mittelpunkt seiner Kugel, den Radius, die Normale und die Breite des Normalen Kegels und optional eine Farbe. Die Datenstruktur wird zu Beginn erstellt. Der Konstruktionsalgorithmus kann entweder auf einer Punktwolke oder besser einem triangulierten Modell angewendet werden. Bei Letzterem ist es leichter, die Normalen zu berechnen. Für den Radius der Kugeln wird die Länge der längsten anliegenden Kante gewählt. Mit Hilfe folgendem Algorithmus kann nun die Datenstruktur erstellt werden:

```
1  BuildTree(vertices[begin..end]) {
2    if (begin == end)
3      return Sphere(vertices[begin])
4    else
5      midpoint = PartitionAlongLongestAxis(vertices[begin..end])
6      leftsubtree  = BuildTree(vertices[begin..midpoint])
7      rightsubtree = BuildTree(vertices[midpoint+1..end])
8      return BoundingSphere(leftsubtree, rightsubtree)
9  }
```

Sobald die Datenstruktur aufgebaut ist, kann die Punktmenge mit folgendem Algorithmus gezeichnet werden:

```
1  TraverseHierarchy(node) {
2    if (node not visible)
3      skip this branch of the tree
4    else if (node is a leaf node)
5      draw a splat
6    else if (benefit of recursing further is too low)
7      draw a splat
8    else
9      for each child in children(node)
10       TraverseHierarchy(child)
11 }
```

Zuerst wird getestet, ob die Kugel des Knotens sichtbar ist. Dafür wird getestet, ob die Kugel in dem View-Frustum liegt. Falls nicht, muss der Knoten samt seiner Kinder

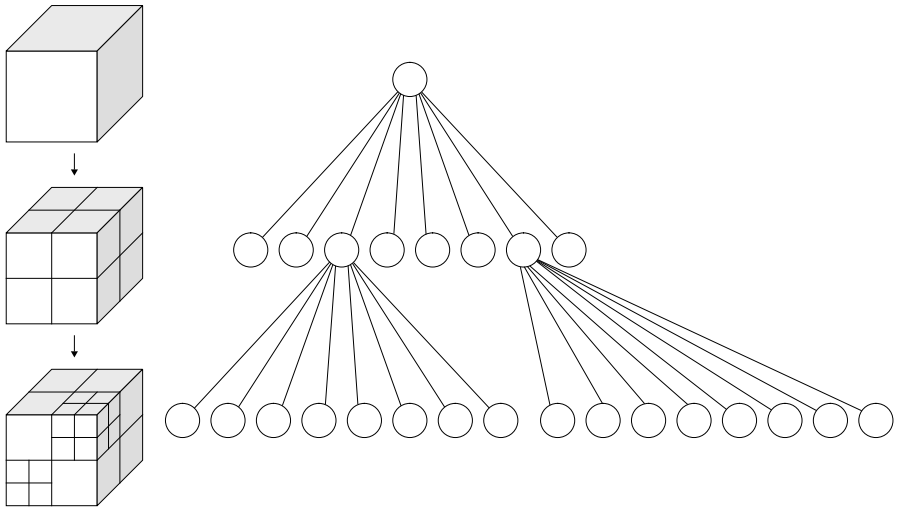

Abbildung 2.2: In jedem Level zerlegt der Octree den Raum in weitere feinere Würfel (Abb aus. [19])

nicht beachtet werden und die Rekursion wird abgebrochen. QSplat führt auch Backface-Culling aus mit Hilfe der Normalen Kegel.

Anhand der Größe der projizierten Kugel auf die View-Ebene wird entschieden, ob die Rekursion fortgesetzt wird. Überschreitet diese einen gewissen Schwellenwert, wird die Rekursion beendet und der aktuelle Knoten gezeichnet.

Das Verfahren kann modifiziert auch zum Streaming von Punktwolken genutzt werden [9] .

Für HD-Displays ist der Algorithmus zu rechenintensiv, weil die Berechnung pro Pixel von der CPU erledigt wird.

2.3.2 Octree

Octrees sind eine Datenstruktur, um dreidimensionale Daten hierarchisch zu untergliedern. Sie wurde 1980 von Donal Maegher beschrieben [1]. Octrees sind analog im Dreidimensionalen zu Quadtrees im Zweidimensionalen.

Jeder Knoten repräsentiert einen Würfel, welcher alle in den Knoten eingefügten 3D-Punkte beinhaltet. Jeder innere Knoten besitzt immer 8 Kinder. Diese unterteilen den Würfel des Knotens in 8 gleich große Oktanten usw. (siehe Abb. 2.2)

Die eigentlichen Punktdaten sind in den äußeren Knoten gespeichert. Äußere Knoten können auch leer sein.

Der Octree unterstützt das Einfügen und Löschen von Punkten.

2.3.3 Multiresolution Octree

Die Datenstruktur entstammt aus der Arbeit "Interactive Editing of Large Point Clouds" [11]. Sie unterstützt Einfügen, Löschen und bietet unterschiedlich detaillierte Darstellungen des Ausgangsmodells. Im Folgenden wird die Datenstruktur vorgestellt.

Der Multiresolution Octree (MRT) kann als eine spezielle Form des Octree verstanden werden. Wie beim Octree enthalten die äußeren Knoten alle Punkte. Die Tiefe ergibt sich aus der Eigenschaft, dass kein Blatt mehr als n_{max} Punkte beinhalten darf. Ist n_{max} nach einer Einfüge-Operation überschritten, wird das Kind geteilt und die bestehenden Punkte werden auf die 8 neuen Kinder verteilt.

Die inneren Knoten sollen eine vereinfachte bzw. gröbere Version ihrer Kinder liefern. Dafür haben diese eine dreidimensionale Rasterung gespeichert. Das Raster unterteilt den Würfel in k^3 gleich große Rasterzellen (z.B. k=128). Jede Rasterzelle hat zusätzlich ein Gewicht und eine Farbe als RGB-Wert gespeichert. Das Raster selbst ist nicht als einfaches Array gespeichert, sondern als Hashtabelle, um Speicherplatz zu sparen. Auf diese Art werden nur die Zellen gespeichert, welche Punkte enthalten.

Zellen mit hohem Gewicht werden beim Rendern größer gezeichnet. Sobald ein weiterer Punkt in die gleiche Zelle fällt, wird das Gewicht inkrementiert.

Der Farbwert entspricht dem des zuerst hinzugefügten Punktes der Zelle.

Einfügen eines Punktes Beim Einfügen können zwei Fälle auftreten.

1. Fall : Der Punkt liegt außerhalb der Wurzel. Die bestehende Wurzel muss so lange erweitert werden, bis sie den neuen Punkt mit einschließt. Die Breite sowie Höhe der Wurzel verdoppelt sich dabei in jedem Schritt.

Sobald der Punkt in der Wurzel liegt, tritt der 2. Fall ein.

2. Fall : Der Punkt liegt innerhalb der Wurzel Zuerst wird der Punkt der Rasterung hinzugefügt. Das heißt, das Gewicht in der entsprechende Rasterzelle wird um eins erhöht und die Farbe des Punktes wird gegebenenfalls gespeichert. Dann wird ermittelt, in welchem der Kinder der Punkt liegt. Nun wird der Vorgang beim Kind wiederholt, bis ein äußerer Knoten erreicht wird. Falls die maximale Anzahl Punkte n_{max} überschritten wurde, muss der Knoten gespalten werden. Alle bisher gespeicherten Punkte und der neue Punkt werden nun auf die neuen Kinder verteilt.

2.3.4 Moving-Least-Squares Surfaces

Das Moving-Least-Squares (MLS) Verfahren wurde von Levin 1998 vorgestellt [5]. Das Verfahren wurde erstmals 2001 durch Alexa zur Darstellung von Punktwolken benutzt [8]. Die Arbeit wird im Folgenden kurz vorgestellt. Die Grundidee der Arbeit ist, dass die gegebene Menge Punkte S indirekt eine Oberfläche S_A definiert. Es wird eine Projektion $\phi : U \to \mathbb{R}^3$ vorgestellt, welche einen beliebigen Punkt aus der Umgebung U von S auf eine Oberfläche S_A, welche das Objekt lokal beschreibt, projiziert. Alle Punkte, die auf

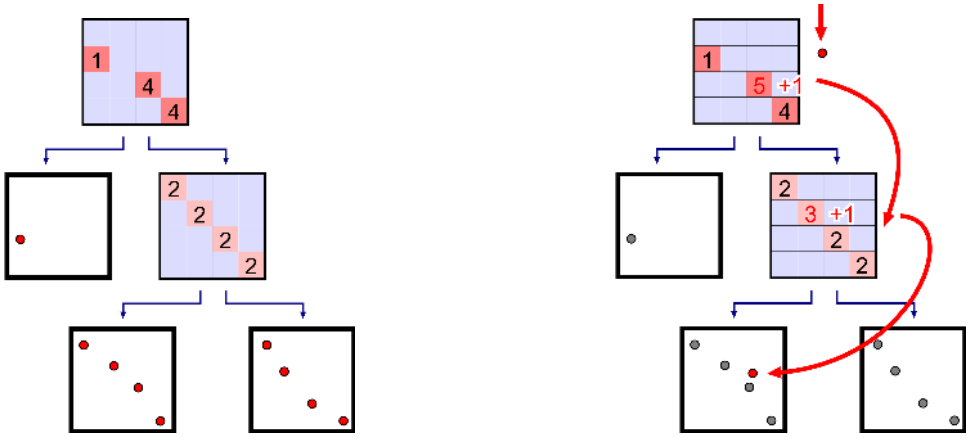

Abbildung 2.3: Abb. Links: Die Rasterung ist schematisch für die zweite Dimension dargestellt. Die inneren Knoten beinhalten die Gewichte. Die äußeren Knoten enthalten die Punkte. Rechts: Das Schema nach dem Einfügen eines weiteren Punktes in den Quadranten rechts unten. Auf jedem Level wird ein Gewicht erhöht. (Abb. aus [11])

sich selbst abbilden, ergeben die abgeschätzte Oberfläche S_A:

$$S_A := \{x \in \mathbb{R}^3 | \phi(x) = x\}$$

U kann man als eine Vereinigung von Kugeln mit Radius r_k, dessen Mittelpunkt ein Punkt aus S ist, beschreiben:

$$U := \bigcup_i \{x \in \mathbb{R}^3 | \|x - p_i\| < r_K\}$$

Die Projektion eines Punktes r wird in 3 Schritten ermittelt.

1. Ermittle eine Referenzebene $H = \{x | \langle n, x\rangle - D = 0, x \in \mathbb{R}^3\}, n \in \mathbb{R}^3, \|n\| = 1$ durch Minimierung des Ausdrucks,

$$\sum_{i=1}^{N} (\langle n, p_i\rangle - D)^2 \Theta(\|p_i - q\|)$$

wobei q die Projektion von r auf H ist. Bei Θ handelt es sich um eine monoton, radial, fallende Funktion mit positivem Wertebereich. Normalerweise gilt:

$$\Theta(d) = e^{\frac{d^2}{h^2}}$$

h ist meist der durchschnittliche Abstand von benachbarten Punkten und hat einen direkten Einfluss darauf, wie glatt die Oberfläche erscheint.

2. Mit Hilfe von H kann nun ein zweidimensionales Polynom zum Abschätzen der Umgebung von S_A in der Nähe von r gefunden werden.

$$\sum_{i=1}^{N}(g(x_i, y_i) - f_i)^2\Theta(\|p_i - q\|))$$

Hier ist f_i der kürzeste Weg von H zu p_i. Bei $x_i$und $y_i$handelt es sich um die Koordinaten von Punkt q_i.

3. Die Projektion von r ist letztendlich wie folgt definiert:

$$\phi(r) = q + g(0, 0)n$$

Neben dem Darstellen von Punkten kann das Verfahren auch eingesetzt werden, um weitere Punkte zu generieren.

Beim Rendern wird eine Datenstruktur wie bei dem QSplat-Verfahren verwendet. Es unterscheiden sich die Blätter, in denen zusätzlich zur Position, Radius, Normalen und Farbe auch eine Referenzebene H sowie die Koeffizienten des Polynoms gespeichert werden.

Wenn bei dem Rendervorgang ein Blatt erreicht wird und wenn mehr als ein Punkt benötigt wird, werden mit Hilfe des Polynoms weitere Punkte für die Umgebung generiert.

2.3.5 Tetraedertree

Die Arbeit [14] stellt eine offline Datenstruktur zum Verwalten von triangulierten Punkten vor. Die Datenstruktur unterteilt den Raum in einzelne Tetraeder. Durch die Tetraeder können Punkte durch baryzentriche Koordinaten repräsentiert werden.

In der Vorverarbeitung wird Top-Down eine Diamanten Hierarchie aufgebaut, wobei jeder Diamant aus einer Menge von Tetraedern besteht. Die Eingabe erfolgt offline als eine Menge von Dreiecken. Zu Beginn wird die Bounding-Box in 6 Tetraeder unterteilt, wobei die Diagonale von allen Tetraedern geteilt wird. Sobald ein eingefügtes Dreieck zwischen mehrere Tetraeder (Blätter) fällt, wird es am Rand des Tetraeders abgeschnitten. Die neuen Dreiecke werden dann in ihre entsprechenden Blätter hinzugefügt. Sobald in einem Blatt mehr als eine vorher festgelegte Menge Dreiecke enthalten ist, wird es anhand einer Ebene, welche durch den Mittelpunkt der längsten Kante und seiner gegenüberliegenden Kante verläuft, geteilt. Im Anschluss werden die Dreiecke auf die zwei Kinder verteilt. Der nächste Schritt vereinfacht die Kinder Buttom-Up. Jeder Diamant wird dabei unabhängig voneinander abgearbeitet. Dafür muss beachtet werden, das Vertices, welche an den Außenwänden liegen, nicht verändert werden dürfen. Bei Vertices an den Innenwänden muss auch der korrespondierende Vertex beim Nachbarn mit einbezogen werden. Ist beides nicht der Fall, kann beliebig vereinfacht werden. Da die Diamanten unabhängig voneinander vereinfacht werden, ist es einfacher die Datenstruktur zu parallelisieren.

2.4 Bestehende Applikationen

Die Idee, Punktwolken auf mobilen Geräten zu betrachten, ist nicht neu. Daher existieren schon einige Vorschläge zum Lösen des Problems. Im Folgenden werden einige dieser Möglichkeiten vorgestellt.

2.4.1 KiwiViewer

KiwiViewer[3] ist eine Open-Source Applikation zur Erkundung von Punktwolken. Die App ist für Android sowie iOS verfügbar. Multi-Touch Gestensteuerung wird unterstützt. KiwiViewer bedient sich der Point-Cloud-Library[4] (PCL) für seine Kernfunktionen. Bei der Point-Cloud-Library handelt es sich um eine Open-Source Bibliothek zum verarbeiten von Punktwolken. Die Punktdaten werden entweder über eine SD-Karte, E-Mail oder URL geladen. Die Applikation lässt sich sehr einfach bedienen.

Abgrenzung Die App speichert die geladenen Punkte direkt auf dem Tablet. Punktwolken werden nicht vereinfacht. Daher treffen Modelle mit mehreren Millionen Punkten schnell an die Grenzen der GPU.

2.4.2 LiMo

Bei LiMo[5] handelt es sich um eine von OGSystems entwickelte Android Applikation zum Betrachten von LiDAR Daten. LiMo wurde für den professionellen Gebrauch entwickelt und ermöglicht das Beobachten von Gebäuden sowie Naturszenen. Daten können auch über einen eigenen Webservice gestreamt werden, weshalb die App auch zu Monitoring Zwecken eingesetzt werden kann. Die Applikation unterstützt bis zu 5 Millionen Punkte.

Abgrenzung Die Anwendung ist auf LiDAR Daten beschränkt. LiDAR ist primär zum Scannen von großen Objekten, wie Brücken geeignet und verfügt daher nicht über die Genauigkeit, welche beispielsweise bei einer Skulptur benötigt wird. Die Applikation ist auf 5 Millionen Punkte beschränkt.

2.4.3 iOS App mit Knn-Baum

Diese Applikation wurde von der Visual Computing Gruppe[6] des CRS4 (Center for Advanced Studies, Research and Development in Sardinia) entwickelt. Die Applikation verwendet einen Knn-Baum zum Verwalten der Punktmenge [13]. Ähnlich wie beim MRT werden in den inneren Knoten vereinfacht Abbilder der Kinder gespeichert. Der Ansatz verwendet eine Client-Server Struktur. Die Datenstruktur selbst wird auf einem Server gespeichert. Die Knoten werden auf Anfrage eines Clients übertragen. Übertragende

[3] http://www.kiwiviewer.org/. Aufg. am 19.05.2016
[4] http://pointclouds.org/. Aufg. am 22.05.2016
[5] https://play.google.com/store/apps/details?id=com.ogs.limo&hl=en. Aufg. am 27.05.2016
[6] http://www.crs4.it/vic/. Aufg. am 11.06.2016

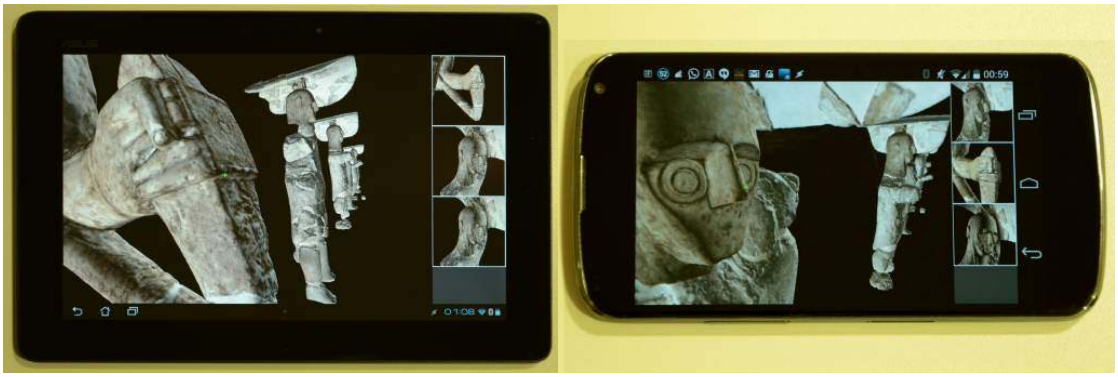

Abbildung 2.4: Darstellung einer Szene von hochauflösenden Statuen auf einem Tablet (Asus TF201) sowie Smartphone (LG Nexus 4). Am rechten Bildrand sieht man die vorausberechneten Vorschaubilder. (Abb. aus [15])

Knoten werden mit Hilfe eines LRU-Cache gespeichert. Die Implementierung erfolgte in C++ und OpenGl ES. Für die Netzwerkkommunikation wird Http-Pipelining sowie eine Wavletkompression verwendet.

Abgrenzung Die Datenstruktur ist nicht dynamisch, also die Menge der Punkte muss von Anfang an feststehen. Die Implementierung erfolgte in C++ und für iOS. Die Datenstruktur bietet den Vorteil, dass sie eine geringe Tiefe besitzt. Die Benchmarks sind beeindruckend, da Punktwolken bis zu 180 Millionen Punkte dargestellt werden können. Die Anzahl gleichzeitig angezeigter Punkte liegt allerdings bei 1 Millionen Punkte auf den Testgeräten. Der Quellcode ist nicht frei verfügbar.

2.4.4 HuMoRS

Das HuMoRs (Huge models Mobile Rendering System) [15] ist eine weitere Entwicklung der Visual Computing Gruppe des CRS4. Die Applikation beherrscht das Darstellen von Modellen mit mehr als 70 Millionen Punkten bzw. Dreiecken. Die Anwendung läuft auf Android und wurde mit mehreren Geräten sowie Versionen getestet. Besonderes Augenmerk wurde in der Arbeit auf die Bedienbarkeit gelegt. Der Rotationspunkt des Modells wird dynamisch berechnet. Des Weiteren werden zu jedem Modell Thumbnails generiert, um zu interessanten Abschnitten des Modells springen zu können. Die Anwendung folgt dem gleichen Client-Server Ansatz wie die iOS Applikation aus Sektion 2.4.3. Als Datenstruktur wird ein Tetraederbaum (siehe Sektion 2.3.5) verwendet. Diese ermöglicht eine Kompression, welche einen Vertex mit Position, Farbe und Normale auf 64Bit komprimiert.

Auf der Clientseite wird ein LRU-Cache verwendet, um den Netzwerkverkehr niedrig zu halten. Die Applikation wurde in C++ mit Hilfe des Qt-Frameworks programmiert.

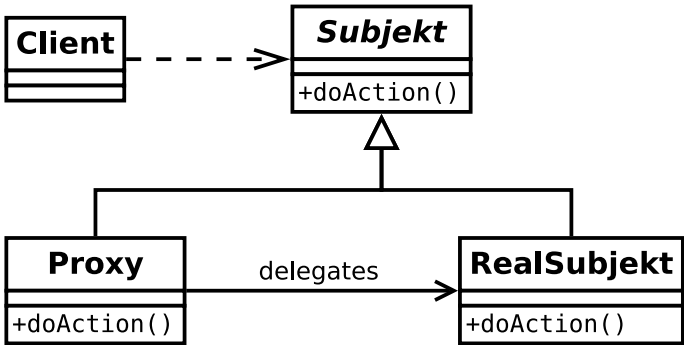

Abbildung 2.5: Das Diagramm zeigt, wie das reale Subjekt durch ein Proxy Objekt repräsentiert wird, auf das der Client zugreift.

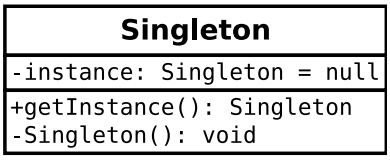

Abbildung 2.6: UML-Diagramm eines Singleton Objektes, welches die einzige Instanz beherbergt.

Abgrenzung Die Arbeit bietet gute Ansätze, besonders in Bezug auf die Bedienung. Die verwendete Datenstruktur bietet eine detaillierte Darstellung samt hoher Bildfrequenz. Allerdings müssen auch hier alle Punkte vor Aufbau der Datenstruktur bekannt sein.

2.5 Proxy Entwurfsmuster

Bei einem Proxy (siehe Abb. 2.5) [21] handelt es sich um ein Objekt, welches als Schnittstelle zu etwas Anderem agiert. Diese sind zum Beispiel eine große Datei oder eine andere teure Ressource.

Ein klassisches Beispiel sind Platzhalter für noch nicht fertig geladene Bilder auf Webseiten.

2.6 Singleton Entwurfsmuster

Man spricht von einem Singleton, wenn von einer Klasse nur eine einzige oder beschränkte Anzahl Instanzen erlaubt sind. Typische Vertreter sind Objekte, die einen Zustand oder globale Variablen speichern wie das Android Kontext Objekt.

2.7 Representational State Transfer

Representational State Transfer (REST) ist ein Programmierparadigma für serviceorientierte verteilte Systeme. REST wurde im Jahr 2000 von Roy Fielding in seiner Doktorarbeit [6] wie folgt beschrieben.

REST definiert sich über eine Menge von Beschränkungen bei der Kommunikation von Komponenten. Das prominenteste Beispiel ist das World Wide Web. Die Beschränkungen werden im Folgenden kurz vorgestellt.

Client-Server Modell

Nach Andrews [2] ist der Client ein auslösender Prozess und der Server ein reagierender Prozess. Der Client stellt Anfragen, auf welche der Server reagiert. Der Client kann entscheiden, wann er mit dem Server interagiert. Der Server wiederum muss auf Anfragen warten und dann auf diese reagieren. Oft ist ein Server ein nicht endender Prozess, welcher auf mehrere Clients reagiert.

Zustandslosigkeit

Jede Anfrage vom Client muss alle Informationen enthalten, welche notwendig sind, um die Anfrage zu verarbeiten. Des Weiteren darf kein gespeicherter Kontext auf dem Server vorliegen, auf welchen Bezug genommen wird. Alle Zustände werden auf dem Client gespeichert.

Caching

Serverantworten müssen implizit oder explizit als cachebar gekennzeichnet sein. Die Idee ist, den Netzwerkverkehr effizienter zu machen. Bemerkenswert dabei ist, dass dadurch ganze Interaktionen wegfallen können.

Einheitliche Schnittstelle

Ein integraler Bestandteil einer REST-Architektur ist eine einheitliche Schnittstelle. Das vereinfacht die Systemarchitektur und verbessert die Sichtbarkeit von Interaktionen. Sie ist durch vier weitere Eigenschaften beschrieben.

1. Adressierbarkeit von Ressourcen Jede Information, die über einen URI kenntlich gemacht wurde, wird als Ressource gekennzeichnet. Die Ressource selbst wird in einer Repräsentation übertragen, welche sich von der internen Repräsentation unterscheidet. Jeder REST-konforme Dienst hat eine eindeutige Adresse, den Uniform Resource Locator (URL).

2. Repräsentationen zur Veränderung von Ressourcen Wenn ein Client die Repräsentation einer Ressource mitsamt seinen Metadaten kennt, reicht dies aus, um die Ressource zu modifizieren bzw. zu löschen.

3. Self-descriptive messages Jede Nachricht beschreibt, wie ihre Informationen zu verarbeiten ist, z.B. durch Angabe ihres Internet Media Types (MIME-Type).

4. Hypermedia as the Engine of Application State Bei Hypermedia as the Engine of Application State (HATEOAS) navigiert der Client ausschließlich über Hypermedia, welche vom Server bereitgestellt wird. Abstrakt betrachtet stellen HATEOAS-konforme REST-Services einen endlichen Automaten dar, dessen Zustandsveränderungen durch die Navigation mittels der bereitgestellten URI erfolgt. Durch HATEOAS ist eine lose Bindung gewährleistet und die Schnittstelle kann verändert werden.

Mehrschichtige Systeme

Der Client soll lediglich die Schnittstelle kennen. Schichten dahinter bleiben ihm verborgen.

3 Frameworks und Architektur

Im Folgenden werden die Technologien, Begriffe und Frameworks vorgestellt, welche in der Arbeit verwendet werden.

3.1 Java und Android

Bei Java [3] handelt es sich um eine 1995 von Sun Microsystems veröffentliche Programmiersprache. Entwickelt wurde die Sprache von James Gosling. Java ist objektorientiert, nebenläufig und plattformunabhängig. Letzteres wird erreicht, indem Java Code, genauer gesagt Java Byte Code, auf einer virtuellen Maschine (JVM) interpretiert wird. Moderne Implementierungen der JVM unterstützen die sogenannte Just-in-Time-Kompilierung. Das bedeutet, dass eine Übersetzung in den Maschinencode während der Laufzeit vorgenommen wird. Des Weiteren kümmert sich die Laufzeitumgebung von Java um das Speichermanagement. Nach dem Tiobe-Index zu urteilen ist Java eine der populärsten Programmiersprachen der Welt[1].

Android[2] ist ein mobiles Betriebssystem, welches momentan von Google weiter entwickelt wird. Es wurde von 33 Mitgliedern der Open Handset Alliance entwickelt. Ziel war es, einen offenen Standard für mobile Geräte zu schaffen[3].

Android baut auf dem Linux Kernel auf und ist für eine Bedienung über Touchdisplays ausgelegt. Daher werden Eingaben hauptsächlich über Gesten und Tippen am Display vorgenommen. Android verfügt über Ableger für Fernseher (Android TV), Autos (Android Auto) sowie Smartwatches (Android Wear). Android hat seit mehreren Jahren einen dominanten Marktanteil[4] bei mobilen Geräten und ist das mit Abstand meist genutzte Betriebssystem weltweit[5].

Java ist die bevorzugte, wenn auch nicht einzig mögliche Programmiersprache für Android. Allerdings kommt bei Android keine Standard JVM zum Einsatz, sondern eine modifizierte Version. Bis zur Version Android 4.4 kam die virtuelle Maschine Dalivk zum Einsatz. Diese wurde vollständig in Android 5.0 ersetzt durch Android Runtime (ART)[6].

[1] http://www.tiobe.com/tiobe_index?page=Java. Aufg. am 17.05.2016
[2] https://www.android.com/. Aufg. am 19.05.2016
[3] http://www.openhandsetalliance.com/press_110507.html. Aufg. am 23.05.2016
[4] http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Aufg. am 21.05.2016
[5] https://en.wikipedia.org/wiki/Usage_share_of_operating_systems. Aufg. am 21.05.2016
[6] http://developer.android.com/about/versions/android-5.0-changes.html. Aufg. am 23.05.2016

3.2 Google Protocol Buffers

Bei Google Protocol Buffers[7] handelt es sich um eine plattformunabhängige Datenstruktur zum Serialisieren von Daten.

Die Datenstruktur wird in einem einheitlichen Schema festgelegt, zum Beispiel:

```
1 message Person {
2   required string name = 1;
3   required int32 id = 2;
4   optional string email = 3;
5 }
```

Das Schema definiert einen Namespaces, über welchen dann die eigentlichen Daten ausgelesen werden können. In dem Beispiel hat der Datentyp „Person“ drei Felder. Jedes dieser Felder ist getypt und mit einem Schlüsselwort versehen, welches angibt, ob das Feld Pflicht ist.

Es besitzt also jede „Person“ einen String für den Namen, aber nicht immer einen String für die Mail.

Aus dem Schema können für jede unterstützte Programmiersprache (Java, C++, Python, JavaNano, Ruby, Objective-C und C#) Klassen generiert werden, welche die Daten (de)serialisieren. Das macht es möglich, Clients in anderen Sprachen zu schreiben.

Protocol Buffers sind komplett abwärtskompatibel. Die Schemata können problemlos erweitert werden.

Ein weiterer Vorteil ist der merklich verringerte Overhead im Vergleich zu JSON oder XML. Nachteilig ist, dass die übertragenen Daten nicht ohne Weiteres für einen Menschen lesbar sind. Protocol Buffers werden intensiv intern bei Google selbst eingesetzt. Ein weiterer prominenter Nutzer ist Blizzard beim BATTLE.NET[8].

3.3 Open-Graphics-Library for Embedded-Systems

Open-Graphics-Library for Embedded-Systems (OpenGL ES) [17] bietet ein offenes Interface für Grafikhardware. Dieses besteht aus einer Sammlung von Prozeduren und Funktionen, die es den Programmierern ermöglichen Shaderprogramme, Objekte und Operationen zu spezifizieren, um dreidimensionale Farbbilder zu produzieren. Viele der Funktionen von OpenGL ES implementieren das Zeichnen von geometrischen Objekten, wie Punkte oder Linien. OpenGL ES setzt voraus, dass die Grafikhardware über einen Framebuffer verfügt.

3.3.1 Vertex

Unter einem Vertex versteht man eine Datenstruktur, welche einen zwei- oder dreidimensionalen Punkt im Raum beschreibt.

[7] https://developers.google.com/protocol-buffers/. Aufg. am 20.05.2016
[8] https://news.ycombinator.com/item?id=11444846. Aufg. am 23.05.2016

3.3.2 Shader

Bei Shadern handelt es sich um vom Nutzer geschriebene Programme, welche an unterschiedlichen Stationen der Rendering-Pipeline aufgerufen werden. Shader werden in der C nahen OpenGL Shading Language geschrieben.

3.3.3 Pipeline

Der Vorgang des Renderings lässt sich vereinfacht als Datenverarbeitung von aufeinanderfolgenden Stufen darstellen [20]. Die Stufen werden im Folgenden kurz vorgestellt.

Den Anfang macht die Vertex Spezifikation. In diesem Schritt wird ein Stream von Vertices für OpenGL ES vorbereitet. Dazu muss festgelegt werden, was für ein Grundobjekt (Primitive) die Daten darstellen. Beispiele wären Punkte, Linien oder Dreiecke. Darauf folgt der Vertex Shader. Dieser erhält einzelne Elemente aus dem Vertexstream und gibt nach dem Ausführen des Shaders ein einzelnes Vertex zurück. Der Shader wird vom Benutzer programmiert. Typischerweise wird in diesem Schritt die Projektionsmatrix auf die Punkte angewendet.

Als nächster Schritt kann eine Tessellation angewendet werden. In dieser optionalen Stufe werden Patches (Primitves für Tesselation) in mehrere kleinere Primitives zerlegt.

Darauf folgt der Geometry Shader. Dieser optionale Shader erhält als Eingabe einen Primitive und gibt keine oder mehre Primitives zurück.

Im Anschluss wird das Vertex Post-Processing ausgeführt. In diesem Schritt werden Teile außerhalb des Kamerafensters verworfen. Dieser Prozess nennt sich Clipping. In dieser Stufe werden die dreidimensionalen Koordinaten zu zweidimensionalen Kamerakoordinaten umgerechnet.

In der nächsten Stufe folgt das Primitve Assembly. Hier werden die erstellten Primitives zu einer Sammlung von finalen kleineren Primitives zerteilt. Zum Beispiel werden aus einer Liste von Vertices vom Primitive Typ GL_LINE_STRIP mit 8 Mitgliedern 7 neue Primitives vom Typ Line-Base.

Im Anschluss werden die Primitives in diskrete Elemente unterteilt (gerastert). Diese Elemente nennt man Fragmente, welche an den Fragment Shader weitergereicht werden. Die Ausgabe des Shaders ist ein Farb-, Tiefen- und sogenannter Stencilwert. Im letzten Schritt werden für die Ausgabewerte des Shaders eine Handvoll Tests durchgeführt. Ein Beispiel ist der Tiefentest (Depth-Test), um zu vermeiden, dass verdeckte Objekte gezeichnet werden. Andere Tests sind der Scissor-Test, der Stencil-Test und der Pixel-Ownership-Test.

3.4 NanoHTTPD

NanoHTTPD[9] bezeichnet sich selbst als einen schlanken HTTP Server, welcher darauf ausgerichtet ist, sich einfach in bestehende Anwendungen einbetten zu lassen.

Das Projekt ist Open-Source und wird aktiv auf Github entwickelt.

[9] https://github.com/NanoHttpd/nanohttpd. Aufg. am 24.05.2016

3.5 la4j

La4j[10] ist eine, aus einem Studentenprojekt entstandene, offene Java Bibliothek. Die Bibliothek liefert Objekte (Matrizen und Vektoren) und Algorithmen für Lineare Algebra.

3.6 DEFLATE

Bei DEFLATE [4] handelt es sich um einen Kompressionsalgorithmus von Phil Katz aus dem Jahre 1993. Er kombiniert die Kompressionsalgorithmen LZ77 oder LZSS mit einer Huffman-Kodierung. Der Algorithmus zeichnet sich durch eine solide Kompression in kurzer Zeit aus. Der Algorithmus findet zum Beispiel im PNG-Format Anwendung.

3.7 Volley

Bei Volley [11] handelt es sich um eine HTTP-Bibliothek zum Verwalten von Netzwerkanfragen. Volley bietet automatisches Koordinieren von Anfragen. Es ermöglicht mehrere nebenläufige Netzwerkverbindungen, Anfragen, Priorisierung und einiges mehr. Die Bibliothek ist frei und wird unter diesem Link[12] entwickelt.

[10] http://la4j.org/. Aufg. am 10.06.2016
[11] http://developer.android.com/training/volley/index.html. Aufg. am 24.05.2016
[12] https://android.googlesource.com/platform/frameworks/volley/. Aufg. am 24.05.2016

4 dotViewer

Mobile Geräte verfügen über vergleichsweise begrenzte Ressourcen bezogen auf die Rechenleistung von CPU und GPU und Arbeitsspeicher. Punktwolken mit mehr als einer Million Punkte bringen die GPU schnell an ihre Grenzen. Daher wurde ein Ansatz gewählt, der es ermöglicht, Punkte nach Bedarf anzuzeigen und zu vereinfachen. Zum Ermitteln der Punkte kommt ein Multiresolution Octree (siehe Sektion 2.3.3) zum Einsatz. Dadurch wird die GPU effizient eingesetzt.

Die Defizite beim Speicher werden durch eine Client-Server Architektur ausgeglichen (siehe Abb. 4.1). Das eigentliche Modell wird von einem Server verwaltet. Der Client hat nur den momentan benötigten Satz Punkte gespeichert. Um Netzwerkverkehr niedrig zu halten, werden die Punkte von Client in einem Cache nach dem Least-Recently-Used Prinzip gespeichert. Zum Rendern werden die Punkte als Vertex Buffer Object in den Speicher der GPU geschrieben und anschließend gezeichnet.

Als Programmiersprache wurde Java gewählt aufgrund des guten Kompromisses aus Portabilität und Performance sowie der guten Integration in das Android Betriebssystem.

Als Entwicklungsumgebung wurde für den Client Android-Studio und für den Server Intellij IDEA verwendet.

4.1 Server

Der Server hat drei Aufgaben. Zum ersten erstellt er aus gegebenen Punkten einen MRT. Seine zweite Aufgabe ist es, Punktanfragen zu bedienen und als letztes ein Proxy Objekt (siehe Kapitel 2.5) des MRT an die Clients zu verteilen.

Zur Interaktion stellt der Server ein RESTful Interface auf Basis des HTTP-Protokolls bereit.

4.1.1 Multiresolution Tree

Der Multiresolution Tree besteht aus vier Klassen (siehe Abb. 4.2). Die Klasse „MultiResolutionTree“ dient als Schnittstelle für alle äußeren Komponenten. Dadurch ist eine sinnvolle Kapselung gewährleistet. Sie beherbergt einen Zeiger auf den Wurzelknoten des MRTs. Des Weiteren legt sie einen Index von den Knoten an, um schnellen Zugriff zu ermöglichen. Zusätzlich existiert eine Factory-Methode zum Erstellen von Protocol Buffer Objekten.

Die Implementierung des MRT ist analog zu der Beschreibung in Kapitel 2.3.3. Jeder Knoten hat die absolute Koordinate seines Mittelpunktes als ID gespeichert. Erwähnenswert sind die Entscheidungen bei der Raster Klasse. Die Rasterwerte sind in einer internen Default Hash Map vermerkt. Diese bildet einen dreidimensionalen Vektor auf

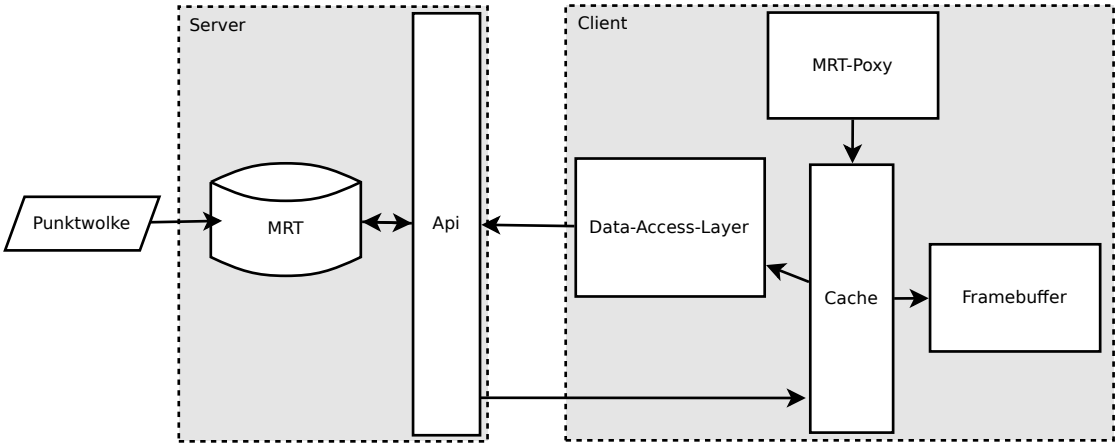

Abbildung 4.1: Das Datenflussdiagramm zeigt den groben Aufbau von Client und Server sowie die Interaktion mit der Api durch den Data-Access-Layer des Servers.

Farbwert und Gewicht ab. Charakteristisch ist bei einer Default Hash Map, dass bei Abfrage von nicht existierenden Einträgen ein Standartwert zurückgegeben wird. Dadurch wird im Vergleich zu einem n^3Array Speicherplatz gespart. Der eigentliche Rastervorgang wird durch die folgende Hashfunktion erreicht:

$$H(p) = (\left\lfloor \frac{x(p)}{cellLength} \right\rfloor, \left\lfloor \frac{y(p)}{cellLength} \right\rfloor, \left\lfloor \frac{z(p)}{cellLength} \right\rfloor)$$

p ist hier der Positionsvektor des Punktes relativ zum Ursprung des Würfels. Die Variable$cellLength$ entspricht der Länge einer Gitterzelle. Also:

$$cellLength = \frac{cube.length}{k}$$

Einfach gesprochen werden die Koordinaten auf ein Vielfaches der $cellLength$ abgerundet.

Damit schnelle Zugriffe bei Anfragen auf die entsprechenden Knoten bzw. Punkte möglich sind, existiert eine weitere Hash Map als Index. Diese bildet Ids auf die entsprechenden Knoten ab. Der Index wird nach einer festen Anzahl Einfügeoperationen aktualisiert.

4.1.2 RESTful API

Unter einer RESTful API versteht man ein Webinterface, welches die Beschränkungen von REST (siehe Kapitel 2.7) einhält.

Die API wurde mit Hilfe des NanoHTTPD Frameworks (siehe Kapitel 3.4) implementiert. NanoHTTPD wurde ausgewählt, weil es schlank und einfach ist.

Ressourcen werden über GET-Anfragen mit folgender Form abgerufen:

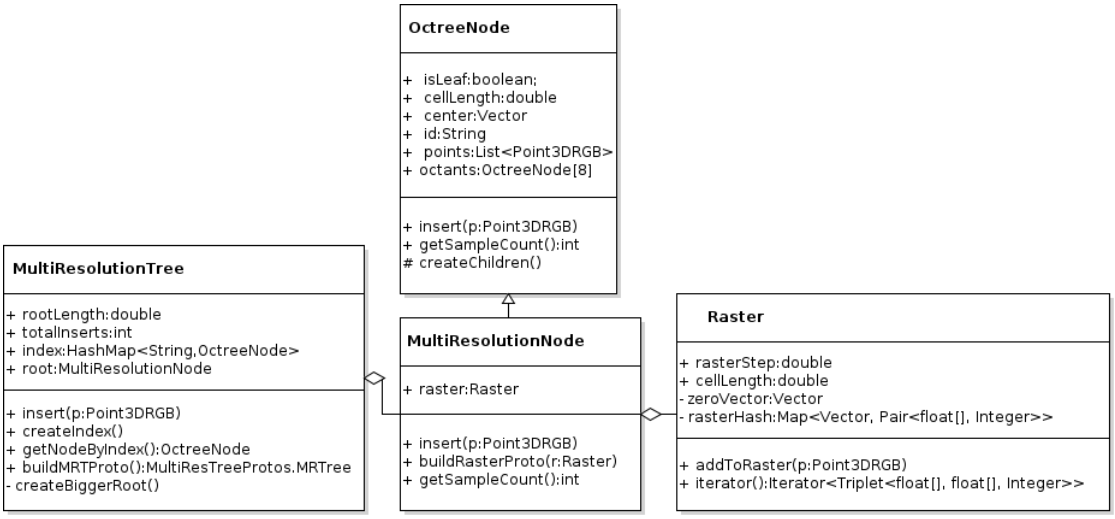

Abbildung 4.2: Das UML-Diagramm zeigt den Aufbau des Multiresolution Trees.

```
1 GET SERVER_IP:PORT/?parameter
```

Parameter	Werte	Erklärung
mode	„proxy“, „samples“	Definiert den Typ der Anfrage
id	String	Id für Punktdaten

Prinzipiell wird bei jeder Anfrage eine Fallunterscheidung an dem Parameter „mode“ gemacht. Die Anfragetypen werden im Folgenden vorgestellt.

4.1.2.1 MRT-Proxy Anfrage

Da die Ermittlung gebrauchter Punkte auf dem Clients stattfindet, muss auch dieser in Kenntnis über die Struktur des MRTs gesetzt werden. Aus diesem Grund stellt der Server ein MRT-Proxy zur Verfügung. Der MRT-Proxy ist im Prinzip gleich dem Multiresolution Octree, allerdings haben seine Knoten keine Punkte oder Rasterung gespeichert, sondern nur Ids, um beim Server die entsprechenden Punkte anzufragen. Diese können entweder Originalpunkte aus Blättern sein oder Punkte aus der Rasterung.

Eine Anfrage wäre zum Beispiel:

```
1 GET 192.168.2.1:8080/?mode=tree
```

Zum Versenden wird aus der Datenstruktur ein Protocol Buffer Objekt erstellt. Diese kann dann problemlos serialisiert werden. Das Protocol Buffer Objekt ist wie folgt definiert.

```
1 package DataAccesLayer;
2
3 option java_outer_classname = "MultiResTreeProtos";
4
```

```
5  message MRTree{
6    required MRNode root = 1;
7    message MRNode {
8      required string id = 1;
9      repeated double center = 2 [packed=true];
10     required double cellLength = 3;
11     required int32 pointCount = 4;
12     required bool isLeaf = 5;
13     repeated MRNode octant = 6;
14   }
15 }
```

4.1.2.2 Punktanfrage

Der 2. Typ Anfragen liefert Punkte an die Clients aus. Bei dieser Anfrage wird vom Client immer eine Id als Parameter mitgesendet. Diese Id passt auf einen Knoten des MRTs.

Falls es sich um einen inneren Knoten handelt, wird die Rasterung zu einer Liste von Punkten exportiert. Bei einem Blatt wird lediglich auf die vorhandene Punktliste zugegriffen.

Der Server greift über die „MultiResolutionTree“ Klasse auf den entsprechenden Knoten zu und generiert ein Protocol Buffer Objekt. Die Spezifikation des Objektes lautet:

```
1  package DataAccesLayer;
2
3  option java_outer_classname = "RasterProtos";
4
5  message Raster{
6    repeated Point3DRGB sample = 1;
7    message Point3DRGB{
8      repeated float position = 1;
9      repeated float color = 2;
10     required int32 size = 3;
11   }
12 }
```

Es handelt sich also um eine Liste von Punkten mit Positions-, Farb- und Gewichtswerten (size).

Alle Protocol Buffer Objekte werden, bevor sie über das Netzwerk versendet werden, serialisiert und durch den DEFLATE Algorithmus komprimiert, um den Netzwerkverkehr möglichst niedrig zu halten.

4.2 Client

Bei dem Client handelt es sich um eine Android Anwendung. Der Client ist verantwortlich für das Darstellen der Punktwolke und reagiert auf Eingaben des Users.

Der Client verfolgt eine eventbasierte Architektur. Wird vom Nutzer eine Translation (Zwei-Finger Geste) oder Rotation (Slide Geste) an dem Punktmodell ausgeführt, kann

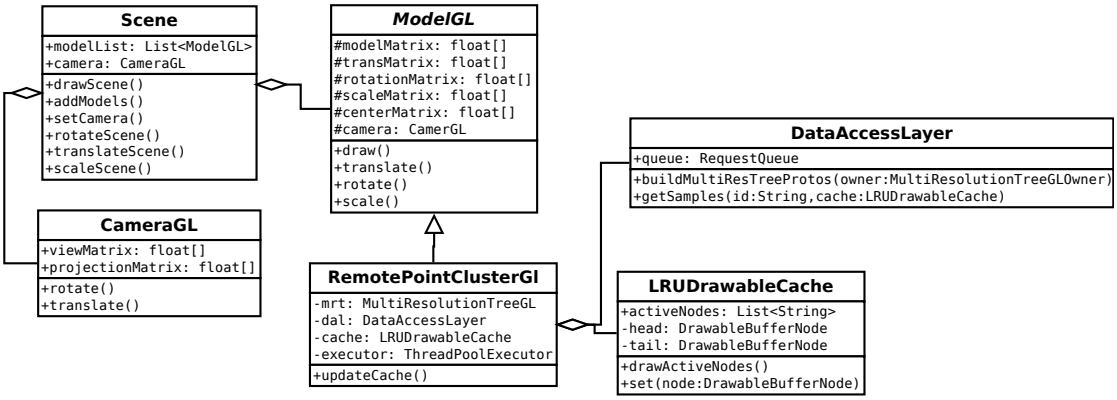

Abbildung 4.3: Das UML-Diagramm zeigt den Aufbau des Multiresolution Trees.

sich die Menge der zu zeichnenden Punkte ändern.

Die Menge der momentan zu zeichnenden Punkte wird im Folgenden als aktive Punkte bezeichnet. Die Knoten, welche die aktiven Punkte beinhalten, werden als aktive Knoten bezeichnet.

4.2.1 Netzwerkverkehr

Bei der „DataAcessLayer“ Klasse handelt es sich um ein Singleton (siehe Kapitel 2.6), welches als Schnittstelle für jeglichen Netzwerkverkehr fungiert. Diese verfügt über Funktionen zum Anfordern von Punkten oder des Proxys. Für die asynchronen Anfragen wird das Volley Framework verwendet (siehe Kapitel 3.7). Das Framework stellt eine Prioritätswarteschlange für HTTP-Anfragen zur Verfügung. Volley ermöglicht es, die Callback Funktion, welche nach Eintreffen einer Antwort aufgerufen wird, zu überschreiben, um eigene Logik zu definieren. Die zwei Methoden der Klasse werden im Folgendem vorgestellt.

buildMultiResTreeProtos(owner:MultiResTreeOwner) Die Methode dekoriert eine Instanz, welche das „MultiResTreeOwner“ Interface implementiert (z.B. „RemotePointClusterGL“) mit dem MRT-Proxy.

Als erster Schritt wird eine HTTP-Anfrage für den MRT Proxy an den Server gestellt. Beim Empfangen der Antwort werden die Daten deserialisiert und im Anschluss mit Hilfe einer Factory Methode zum MRT-Proxy umgewandelt. Die als Parameter übergebene Instanz erhält einen Zeiger auf den Proxy.

getSamples(id:String, cache:LRUDrawableCache) Die Methode fordert Punktdaten vom Server an und speichert diese in einem Cache Objekt, welches später zum Zeichnen der Punkte verwendet wird. Beim Erhalten der Antwort werden die Daten zuerst dekomprimiert (DEFLATE) und deserialisiert. Im Anschluss werden die Farb-, Positions- und

Gewichtswerte in einen nativen Buffer geschrieben, damit OpenGL bei Bedarf darauf zugreifen kann.

4.2.2 Rendering

In diesem Abschnitt wird beschrieben, wie die Punktdaten ausgewählt und verarbeitet werden.

4.2.2.1 Shader

Die Shader sind als Textdatei in den Android Ressourcen gespeichert. Bei der Initialisierung der OpenGL View-Instanz werden sie kompiliert und gelinkt. Im Folgenden werden die Shader vorgestellt.

```
1  # vertex_shader.glsl
2  attribute vec3 a_Position;
3  attribute vec3 a_Color;
4  attribute float a_Size;
5  uniform mat4 u_Matrix;
6  varying vec4 v_Color;
7
8  void main() {
9    v_Color = vec4(a_Color, 1.0);
10   gl_Position = u_Matrix * (vec4(a_Position, 1.0));
11   gl_PointSize = min(4.0, sqrt(a_Size));
12 }
```

In dem Vertex Shader werden die Postionen der Punkte mit der Projektionsmatrix multipliziert und an die nächste Stufe der OpenGL Pipeline weitergegeben. Des Weiteren wird die Größe der zu zeichnenden Punkte berechnet. Es wird die Wurzel gezogen, um sehr große Werte abzuschwächen. Durch die min-Funktion wird eine obere Grenze von 4 eingeführt. Das ist nötig, denn bei sehr detaillierten Objekten kann das Gewicht schnell sehr groß werden. Die Farbwerte werden einfach an den Fragment Shader durch eine varying Variable weitergereicht.

```
1  precision mediump float;
2  varying vec4 v_Color;
3  void main() {
4    gl_FragColor = v_Color;
5  }
```

Der Fragment Shader empfängt die Farbwerte und gibt sie weiter an die vorgegebene Variable „gl_FragColor“ und legt damit den Farbwert des Fragmentes fest.

4.2.2.2 Scene Klasse

Die „Scene“ Klasse (siehe Abbildung 4.3) ist ein Singleton welches die Schnittstelle für alle renderingrelevanten Aktionen bildet. Alle zu zeichnenden Objekte sowie die Kamera sind in dieser Klasse gespeichert. Nutzereingaben werden von dieser Klasse entgegengenommen und entsprechend verarbeitet.

drawScene() Beim Aufruf der Methode wird die draw-Methode von jedem Element der Liste ausgelöst.

4.2.2.3 RemotePointClusterGL

Die „RemotePointClusterGL“ kümmert sich um das Ermitteln aktiver Punkte mit Hilfe des MRT, dafür sind Verweise auf den Cache und dem MRT-Proxy gespeichert. Des Weiteren greift die Klasse auf die „DataAccesLayer“ Instanz zu, um entweder den MRT Proxy zu aktualisieren oder neue Punkte anzufordern. Bei Initialisierung der Instanz wird der Proxy vom Server erfragt. Der gespeicherte LRU-Cache beinhaltet die Punktdaten.

Immer wenn sich die Kameraposition ändert, wird die updateCache-Methode aufgerufen, um den aktiven Knoten zu ermitteln.

updateCache() Die Methode bestimmt die momentan aktiven Punkte bzw. Knoten mit Hilfe des Proxy Objektes.

Knoten (mit ihren Punkten) werden zu den aktiven Knoten hinzugefügt, wenn folgende Kriterien erfüllt sind:

- die Punkte sind sichtbar
- die Auflösung der Punkte ist ausreichend oder schon maximal

Um das zu erreichen, wird der Proxy mit Hilfe des folgenden Algorithmus traversiert.

```
1  public List<String> getIdsViewDependent(){
2    List<String> ids = new LinkedList<>();
3    _getIdsViewDependent(root, ids);
4    return ids;
5  }
6
7  private void _getIdsViewDependent(OctreeNodeGL currentNode, List<String
       > ids) {
8    if ((currentNode.isLeaf ||
9    currentNode.getDetailFactor(this.owner) < DETAIL_THRESHOLD)){
10     ids.add(currentNode.id);
11     return;
12   }
13   for (OctreeNodeGL node : currentNode.octants ) {
14     if (node.isVisible(owner) && node.pointCount > 0)
15       _getIdsViewDependent(node, ids);
16   }
17 }
```

Beim Test auf Sichtbarkeit wird geprüft, ob die projizierte Box (von einem Knoten) sich mit der View-Ebene schneidet. In jedem Schritt wird der Detail-Factor ermittelt und mit einem festgelegten Schwellenwert verglichen. Der Schwellenwert ist experimentell ermittelt und kann abhängig von der Leistung des Gerätes gewählt werden. Sobald der Detail-Factor klein genug ist, gilt der Knoten als ausreichend aufgelöst und wird gezeichnet. Der Detail-Factor berechnet sich wie folgt:

```
1 getDetailFactor(MRTNode k, Camera c){
2   float[] edgePointsOnViewPlane = projectPoints(edgePoints(k), c)
3   float[] minimumBoundingBox = getMinimumBoundingBox(
        edgePointsOnViewPlane)
4   return getArea(minimumBoundingBox) * 1/(projectPoint(k.center.z, c))
5 }
```

Die einfließende z-Koordinate sorgt dafür, dass von dem Betrachter entfernte Knoten weniger detailliert dargestellt werden.

Der komplette Vorgang findet in einem eigenem Thread statt, damit der Nutzer nicht warten muss, um neue Eingaben zu machen. Für die Verwaltung von Threads kommt die von Java mitgelieferte „ThreadPoolExecutor“ Klasse zum Einsatz.

Nachdem alle nötigen Knoten festgestellt wurden, wird geprüft, ob sich diese schon im Cache befinden. Falls nicht, werden diese vom Server angefordert.

draw() Zum Zeichnen des Modells wird die draw-Methode aller aktiver Knoten im Cache aufgerufen.

4.2.2.4 DrawableCache Klasse

Diese Klasse speichert und cached die Punktdaten vom Server. Des Weiteren sendet sie Punktdaten als Vertex Buffer Object (VBO) an die GPU. Der Vorteil von VBOs ist, dass Punktdaten nicht bei jedem Frame neu übermittelt werden müssen. Stattdessen werden die Daten direkt auf der Grafikkarte gespeichert. Dadurch wird die Performance stark verbessert. Schlussendlich wird in dieser Klasse das Zeichnen durch OpenGL initiiert.

Der Cache besitzt eine Prioritätswarteschlange von Knoten nach dem LRU-Prinzip. Die Knoten besitzen die Punktdaten und einen Vermerk, ob diese schon im Speicher der GPU gelandet sind. Die Positions-, Farb- und Gewichtsdaten werden in einem einzigen Buffer pro Knoten ineinander abgespeichert, um die Performance zu verbessern [1].

Beim Zeichnen wird zuerst geprüft, ob die Daten des Knotens schon als VBO auf der GPU sind. Falls ja, werden die Punkte mit der folgenden Methode gezeichnet. Ansonsten werden die Punkte erst übermittelt.

```
1     public void draw(){
2         glDrawArrays(GL_POINTS, 0, pointCount);
3     }
```

4.2.3 Benutzer Interface

Die Applikation unterstützt drei Gesten zum Erkunden der Modelle. Das einfache Scrollen mit einem Finger wird als Rotation interpretiert. Modelle können pro Bewegung nur um

[1] https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TechniquesforWorkingwithVertexData/TechniquesforWorkingwithVertexData.html. Aufg. am 13.05.2016

eine Achse rotiert werden, um die Eingabe zu vereinfachen. Zwei-Finger-Scrollen wird als Zoom verstanden und das Zusammenziehen von zwei Fingern (Pinch) als Skalierung.

Für das Verarbeiten der Scroll-Gesten mit einem oder zwei Fingern werden Eingaben an die „GestureListener“ Klasse weitergeleitet. Diese Klasse implementiert das von Android mitgelieferten „GestureDetector.OnGestureListener“ Interface. Bei Pinch Eingaben kommt die „ScaleListener“ Klasse zum Einsatz. Diese implementiert das vorgegebene „ScaleGestureDetector.OnScaleGestureListener“ Interface. Die Listener Klassen besitzen Zeiger auf das „Scene“ Singleton. Die Call-Back Methoden der Listener Klasse rufen dann die entsprechenden Transformationsmethoden auf.

Des Weiteren bietet die Applikation in seinem Actionbar, dem schwarzen Menübalken am oberen Bildschirm, einen Zugang zum Settings-Menü. Dort kann die IP des Servers sowie ein Schwellenwert für die Detailstufe eingetragen werden.

5 Experimente und Auswertungen

In diesem Kapitel soll die Performance des dotViewers evaluiert werden. Performance ist hier definiert als die Anzahl der berechneten Bilder (Frames) pro Sekunde (FPS). Umso mehr Bilder pro Sekunde berechnet werden, desto performanter und flüssiger ist die Darstellung. Die Darstellungsqualität selbst gibt Aufschluss darüber, wie gut die Datenstruktur das Grundmodell vereinfacht.

5.1 Durchführung

Zur Messung werden unterschiedliche Modelle in verschiedenen Vergrößerungsstufen um 360 Grad gedreht. Die Punkte jedes Modells werden vollständig in die Datenstruktur eingefügt, bevor gemessen wird. Zum Messen der FPS wurde eine einfache Methode in den Mainloop von OpenGL hinzugefügt, welche den Wert in der Konsole ausgibt. Als Erstes wird eine Felslandschaft mit ca. 1.6 Millionen Punkten analysiert. Das Modell wurde vom dem Archaeocopter Projekt via 3D-Rekonstruktion errechnet. Als Zweites wird eine Baustelle und Umgebung mit 20 Millionen Punkten getestet. Das Modell stammt von der Jacobs University Bremen gGmbH und wurde von Dorit Borrmann, Jan Elseberg, Hamid Reza Houshiar und Andreas Nüchter aufgenommen. Angeboten wird das Modell in dem Robotic 3D Scan Repository[1]. Als Server kommt ein Thinkpad T460 mit einem Intel i5 Prozessor und 16GB RAM zum Einsatz. Bei dem Client handelt es sich um ein LG G3 Smartphone mit einem 4-Kern Qualcomm Snapdragon 80 Chip. Das Smartphone verfügt über eine Auflösung von 1440x2560 Pixel und 2 GB RAM und ist damit mit einem Tablett vergleichbar. Der Netzwerkverkehr läuft über das WLAN.

Die Messwerte und die Darstellungsqualität selbst geben Aufschluss darüber, wie gut die Datenstruktur das Grundmodell vereinfacht.

5.2 Auswertung

Die Felslandschaft ist mit 1.5 Millionen Punkten für ein Landschaftsmodell wenig detailliert. Der Aufbau des MRTs auf dem Server dauert 3 Minuten. Das Modell wird von der Applikation mit mindestens 30FPS in jeder Perspektive dargestellt. Die verschiedenen Detailstufen werden kaum spürbar nachgeladen (siehe Abb. 5.2 und Abb. 5.1). Verzögerungen durch den Netzwerkverkehr beim Aufbau des Bildes übersteigen nicht die 5-Sekunden-Marke. Sobald Flächen mit unterschiedlichen Sampleraten nebeneinander liegen, kann es zu sichtbaren Grenzen kommen. Das kann passieren, sobald eine geraster-

[1] http://kos.informatik.uni-osnabrueck.de/3Dscans/

te Fläche (innerer Knoten) neben einer Fläche mit rohen Punktdaten (Blatt) liegt. Der Effekt ist in Abb. 5.2 im oberen Bild zu beobachten.

Das zweite Modell stellt eine weitläufige Baustelle dar. Das Modell enthält urbane Objekte, z.B. Häuser und Brücken. Das Einfügen der 20 Millionen Punkte in den MRT dauert 13 Minuten. Die Darstellung des Geländes wird durch die Applikation sinnvoll vereinfacht. Die Performance bleibt bei jeder Einstellung über 18FPS. Auffallend sind die recht hohen Latenzen von bis zu 10 Sekunden beim Nachladen der einzelnen Oktanten. Diese sind allerdings nicht verwunderlich Anbetracht der hohen Menge an Punkten, welche vom Server an die Clients gesendet werden. Bei nahen Darstellungen werden Details sichtbar. Es kann auf kurzer Distanz zu verdunkelten Bereichen kommen, sobald der Punktabstand größer als 1 Pixel beträgt (siehe Abb. 5.3).

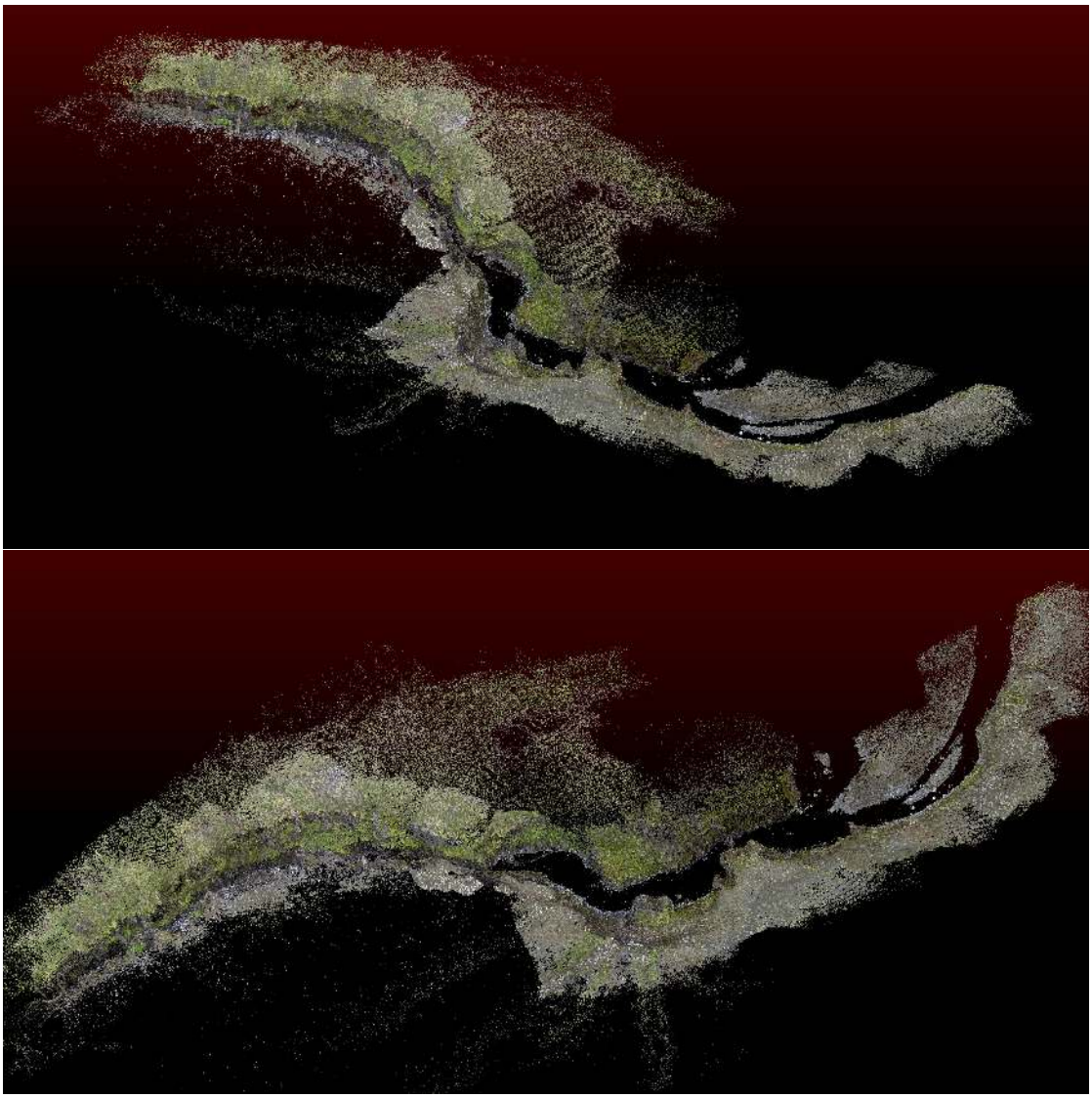

Abbildung 5.1: Visualisierung einer Felsschlucht aus der ferner Distanz. Details wurden hier vereinfacht. Die dicker gezeichneten Punkte repräsentieren Bereiche mit einer hohen Dichte an Punkten.

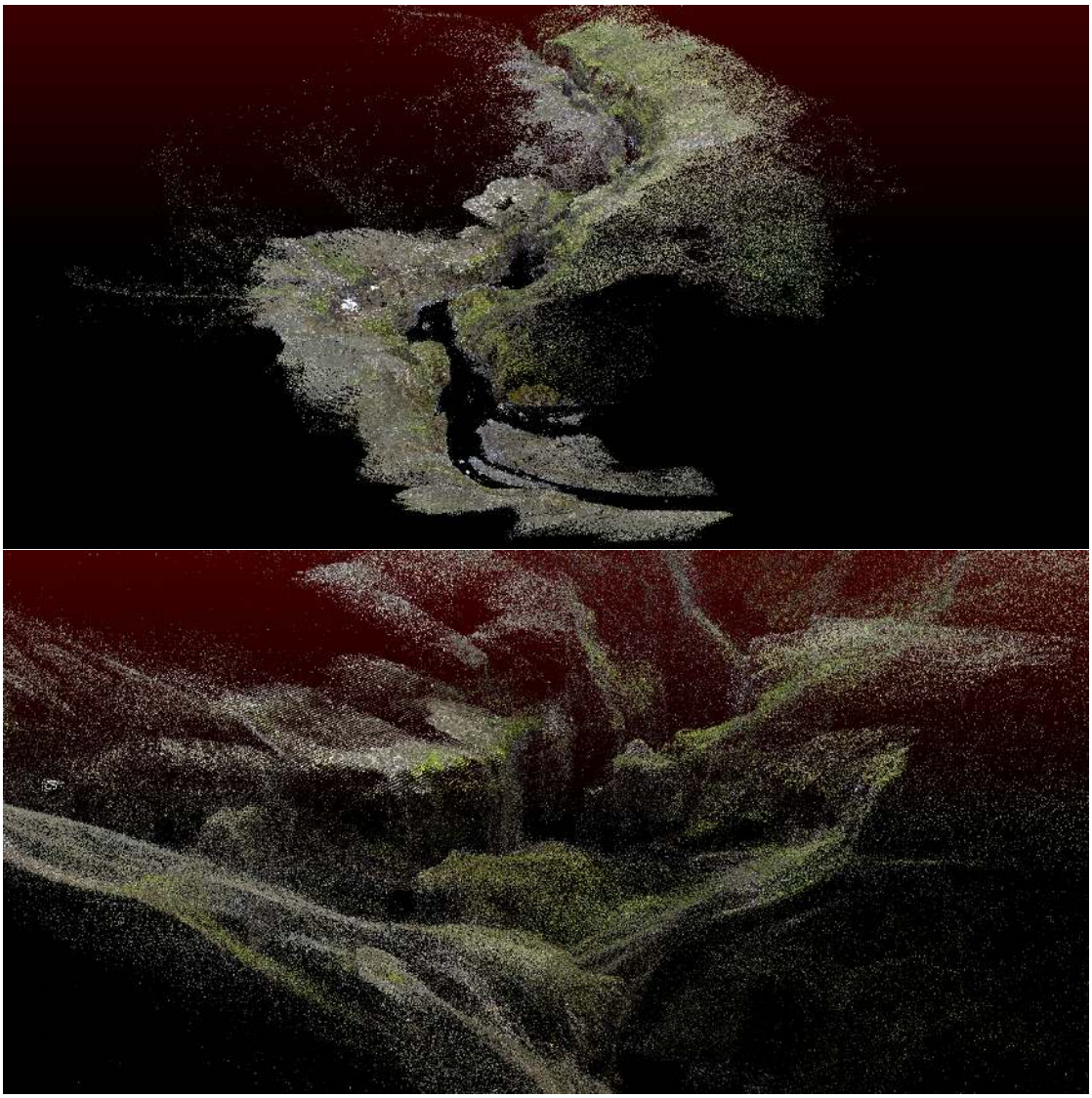

Abbildung 5.2: Oben: Bild aus mittlerer Perspektive. Eine Kante zwischen zwei Oktanten ist oben rechts im Bild deutlich erkennbar. Unten: Die Schlucht in naher Aufnahme. Punkte welche in der Ferne liegen werden gröber Dargestellt.

Abbildung 5.3: Oben: Vereinfachte Darstellung der Baustelle. Viele Details wurden durch einfache Punkte ersetzt. Unten: Nahaufnahme der Sandkuhle. Auffallend ist die dunkle Verfärbung des hinteren Bereiches.

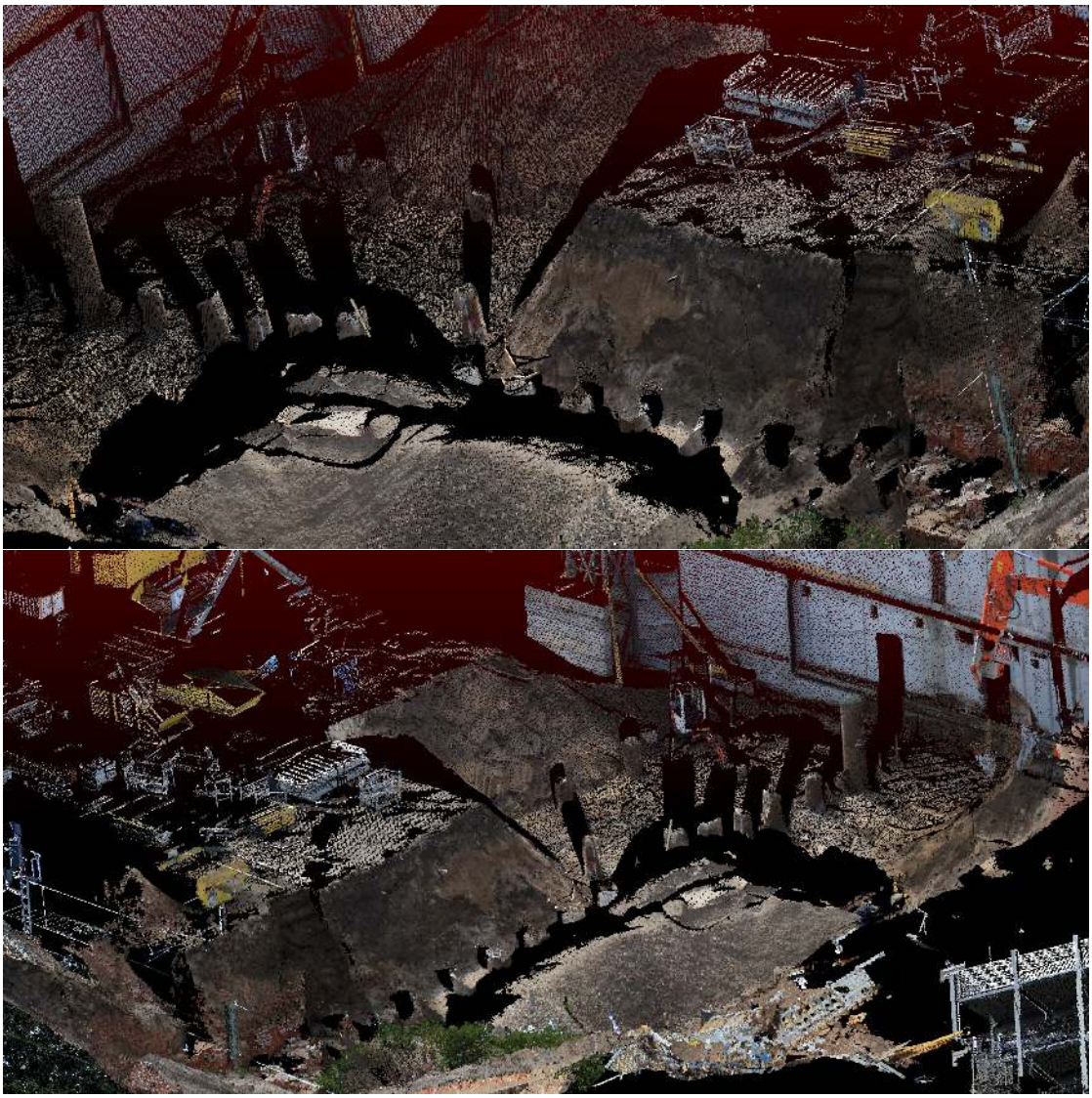

Abbildung 5.4: Beide Bilder zeigen die Baustelle auf mittlerer Distanz. Die Bildfrequenz liegt bei den Szenen bei ungefähr 15FPS. Das Nachladen zwischen den Zoomstufen ist stark spürbar, wenn die Daten nicht gecached sind.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

In der vorliegenden Bachelorarbeit wurde eine Implementierung zum Betrachten von großen Punktmengen auf Android Geräten vorgestellt.

Im Kapitel Grundlagen und verwandte Arbeiten wurde eine Einführung in die 3D-Rekonstruktion gegeben. Des Weiteren wurden unterschiedliche Repräsentationen von Punktwolken diskutiert. Im Folgenden wurden bestehende Lösungen für mobile Geräte diskutiert. Aktuelle Arbeiten konnten große Punktmengen mit mehr als 100 Millionen Punkten bewältigen. Allerdings wurden offline Datenstrukturen für die Punktverwaltung verwendet. Daher mussten die Punktwolken beim Erstellen schon vollständig vorliegen.

Aufbauend auf diesem Schwachpunkt wurde im Kapitel dotViewer eine Applikation vorgestellt, welche zum einen in der Lage ist, Punktdaten online zu verarbeiten sowie Punktwolken mit über 20 Millionen Punkten darzustellen. Die theoretische Grundlage für die verwendete Datenstruktur war der in Kapitel 2 vorgestellte MRT. Als Architektur wurde das Client-Server Modell mit einer RESTfull API gewählt, um Speicher und GPU vom Tablett zu entlasten.

In Kapitel 5 wurde die Applikation durch Messen der FPS und durch Beobachten der Darstellung evaluiert. Die Framerate ist dabei nicht unter 15FPS gesunken, was völlig ausreichend für das Betrachten ist. Durch die teilweise unterschiedlich aufgelösten Rasterungen kann es zu sichtbaren Kanten kommen. Des Weiteren können Bereiche dunkler wirken, wenn der Punktabstand ein wenig größer ist als die Pixeldichte. Dieser Effekt ist nicht verwunderlich, da zwischen den Punkten nicht interpoliert wird. Dadurch füllen sich die leeren Bereiche mit der Hintergrundfarbe und die Farbe wirkt verfälscht.

6.2 Ausblick

Viele Komponenten der Anwendung sind einfach gehalten, um den angepeilten Zeitrahmen nicht zu sprengen. Der Server ist nicht an eine Datenbank gekoppelt, um Daten auf die Festplatte auszulagern. Dadurch wäre die Größe der Modelle theoretisch nur vom Festplattenspeicher begrenzt und die Applikation könnte vergleichbare Punktenmengen verarbeiten wie HuMoRs. Ein weiteres Problem ist die Kompression vor dem Netzwerkverkehr. Für die Punktdaten lässt sich sicherlich eine besser angepasste Kompression finden. Andere Projekte benutzen hierfür zum Beispiel eine Wavletkompression [13] . Die Steuerung ist auch nicht optimal gelöst. Intelligentere Mechanismen, wie sie bei HuMoRs vorhanden sind, wären eine sinnvolle Ergänzung.

Es würde sich auch anbieten, den Server um weitere HTTP-Methoden zu erweitern.

Die PUT-Methode könnte zum Hinzufügen von Punkten genutzt werden. Dadurch wäre der Server über eine plattformunabhängige Schnittstelle gut ansprechbar .

Die Client[1] und Server[2] wird nach Abgabe dieser Arbeit frei zugänglich durch die Entwicklungsplattform Github sein. Aufgrund seiner modularen Architektur sind weitere Komponenten unkompliziert nachreichbar.

[1] https://github.com/garlicPasta/dotViewer
[2] https://github.com/garlicPasta/dotServer

Literaturverzeichnis

[1] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[2] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Comput. Surv.*, 23(1):49–90, March 1991.

[3] James Gosling and Henry McGilton. The java language environment. *Sun Microsystems Computer Company*, 2550, 1995.

[4] P. Deutsch. Deflate compressed data format specification version 1.3, 1996.

[5] David Levin. The approximation power of moving least-squares. *Mathematics of Computation of the American Mathematical Society*, 67(224):1517–1531, 1998.

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[7] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[8] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Point set surfaces. In *Proceedings of the Conference on Visualization '01*, VIS '01, pages 21–28, Washington, DC, USA, 2001. IEEE Computer Society.

[9] Szymon Rusinkiewicz and Marc Levoy. Streaming qsplat: A viewer for networked visualization of large, dense models. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, pages 63–68, New York, NY, USA, 2001. ACM.

[10] Denis Klimentjew. *Grundlagen und Methodik der 3D-Rekonstruktion und ihre Anwendung für landmarkenbasierte Selbstlokalisierung humanoider Roboter*. PhD thesis, 2008.

[11] Michael Wand, Alexander Berner, Martin Bokeloh, Philipp Jenke, Arno Fleck, Mark Hoffmann, Benjamin Maier, Dirk Staneker, Andreas Schilling, and Hans-Peter Seidel. Special section: Point-based graphics: Processing and interactive editing of huge point clouds from 3d scanners. *Comput. Graph.*, 32(2):204–220, April 2008.

[12] Stawros Ladikos et al. *Real-Time Multi-View 3D Reconstruction for Interventional Environments*. PhD thesis, Technische Universität München, 2011.

[13] Marcos Balsa Rodriguez, Enrico Gobbetti, Fabio Marton, Ruggero Pintus, Giovanni Pintore, and Alex Tinti. Interactive Exploration of Gigantic Point Clouds on Mobile Devices. In David Arnold, Jaime Kaminski, Franco Niccolucci, and Andre Stork, editors, *VAST: International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*. The Eurographics Association, 2012.

[14] Marcos Balsa Rodríguez, Enrico Gobbetti, Fabio Marton, and Alex Tinti. Compression-domain seamless multiresolution visualization of gigantic triangle meshes on mobile devices. In *Proceedings of the 18th International Conference on 3D Web Technology*, pages 99–107. ACM, 2013.

[15] Marcos Balsa Rodriguez, Marco Agus, Fabio Marton, and Enrico Gobbetti. HuMoRS: Huge models mobile rendering system. In *Proc. ACM Web3D International Symposium*, pages 7–16. ACM Press, New York, NY, USA, August 2014.

[16] Larry Li. Time-of-flight camera–an introduction. *Technical White Paper, May*, 2014.

[17] Jon Leech. Opengl es version 3.1. 2015.

[18] Michelangeloproject. http://graphics.stanford.edu/projects/mich/, 2016. Aufgerufen am 2016-03-03.

[19] Octree wikipedia. https://en.wikipedia.org/wiki/Octree, 2016. Aufgerufen am 2016-03-03.

[20] Opengl pipeline. https://www.opengl.org/wiki/Rendering_Pipeline_Overview, 2016. Aufgerufen am 2016-03-03.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.