Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Vitalij Kagadij

3-D Rekonstruktion von Gebäuden aus Bildern

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Vitalij Kagadij

3-D Rekonstruktion von Gebäuden aus Bildern

Eingereicht am: 28. Juli 2014

Vitalij Kagadij

Thema der Arbeit
3-D Rekonstruktion von Gebäuden aus Bildern

Stichworte
Punktwolke, RANSAC, Computergraphik

Kurzzusammenfassung
Im einundzwanzigsten Jahrhundert besitzen wir solche Technologien, wie zum Beispiel Google Maps, welches 3-D Modelle für Rekonstruktion der berühmten Gebäude auf einer digitalen Karte verwendet wird. Es ist eine sehr zeitaufwändige Aufgabe so ein Modell zu erschaffen. Das Ziel, dieser Bachelorarbeit, ist es diesen Prozess zu automatisieren. Es wird ein Konzept entworfen, das aus Fotos- bzw. Videoaufnahmen von Gebäuden in ein 3-D Modell umgesetzt wird.
Vitalij Kagadij

Title of the paper
3-D reconstruction of buildings from images

Keywords
Point cloud, RANSAC, computer graphics

Abstract
Such technology like Google Maps uses 3-D models to display the famous buildings on the map. To create that kind of a model is a time consuming task. The aim of this bachelor thesis is to automate this process. It is to design a concept, that constructs a 3D model from video or photo.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einführung

3-D Rekonstruktion von Gebäuden ist ein aktives Forschungsthema in Computer Vision und in digitaler Photogrammetrie (Die relevante Konferenzen sind: International Symposium on Visual Computing(http://www.isvc.net/); International Scientific and Technical Conference From imagery to map: digital photogrammetric technologies(http://conf.racurs.ru/conf2014/eng/); International Conference on Computer Graphics Theory and Applications(grapp.visigrapp.org/)). Dreidimensionale Gebäude sind sehr wichtig für solche Bereiche wie Stadtplanung, Bau-, Umwelt-, Kommunikationsmanagement, Tourismus und virtuelle Touren von Städten. Manuelle Erstellung von 3-D Objekte ist eine sehr zeitaufwändige und komplizierte Aufgabe, deswegen ist es notwendig, ein automatisiertes Prozess zu entwickeln, welches die Rekonstruktion von 3-D Gebäuden aus Bildern zu beschleunigen ermöglicht.

Der aktuelle Stand der Dinge in Automatisierung von Rekonstruktion von Gebäuden befindet sich noch auf einem niedrigen Niveau. Viele Algorithmen und Systeme wurden schon entworfen, um das Problem zu lösen. Eine vielseitige Lösung wurde aber noch nicht vorgeschlagen, da die Problematik sehr Komplex ist. Die Rekonstruktion selber impliziert die Verarbeitung bei verschiedenen Ebenen: low-level Verarbeitung (feature extraction), middle-level Verarbeitung (representation und description von Gebäudemodell) und high-level Verarbeitung (matching und reasoning). Jede von diesen Ebenen ist kompliziert und gewichtig genug, um eine Thema der Bachelorarbeit zu sein. Das Grundwissen zu diesen Themen kann man aus dem Buch von Szeliski (2010) erwerben.

1.1 Motivation

Die mobilen Technologien entwickeln sich heutzutage mit einer exponentiellen Geschwindigkeit. Ein Handy oder ein Tablett haben schon die Leistung eines durchschnittlichen Laptops. Außerdem können fast alle mobile Geräte Fotos und Video aufnehmen und verfügen über eine Internetverbindung. Es ist bequemer ein kleines Smartphone mitzunehmen, welches ebenso alles filmen, photographieren und sogar Musik abspielen kann, anstatt einer großen Kameras.

Heutzutage besitzt fast jeder ein Smartphone. Wie schön wäre es, z. B. für einen 3-D Designer, mit einem Handy durch die Gegend zu laufen, Gebäude aufzunehmen und per Knopfdruck

aus diesen Aufnahmen ein für die Bearbeitung vorbereitetes 3-D Modell generieren zu lassen. Wahrscheinlich ist die Rechnerleistung eines mobilen Gerätes nicht dafür ausreichend. Das Gerät könnte sich aber ruhig mit einem Server verbinden lassen, wo die ganze Berechnungen dann stattfindet. Auf dem Server läuft in diesem Fall eine Software, welche aus zugelieferten Aufnahmen eine Punktwolke erzeugt und ein 3-D Netz aus diesen Punkten konstruiert.

Das Ziel dieser Bachelorarbeit ist es ein Konzept für eine Technologie zu entwerfen, die es ermöglicht, aus einer Punktwolke-Datei, ein 3-D Netz automatisch zu generieren. Punktwolke(.ply) ist eine Datenstruktur, die man nach der Triangulation von mehreren Fotoaufnahmen zur Verfügung kriegt. Es existieren schon einige gute Lösungen in diesem Bereich (z. B. «Clustering Views for Multi-view Stereo» (CMVS) oder «Patch-based Multi-view Stereo Software» (PMVS) von Yasutaka Furukawa), mit deren Hilfe man aus einer Menge von Bildern die Punktwolke-Datei(.ply) am Ausgang hat. Deswegen möchte ich die Technologie nicht neu entwickeln, sondern werde sofort mit .ply-Datei arbeiten.

Meine Idee ist es einen Prototyp zu implementieren, der im Eingang die Punktwolke-Datei kriegt, die Punkte bearbeitet und auf der Basis von diesen Punkten ein 3-D Netz eines Gebäudes konstruiert.

1.2 Related work

Man versucht schon seit langem eine Lösung für das 3-D Rekonstruktion Problem zu finden. Es ist aber sehr komplex und eine einzige Lösung zu finden ist es sehr kompliziert. Die zahlreichen Verfahren werden vorgeschlagen, welche die verschiedensten Ebenen dieses Problemes lösen.

Wie ich schon erwähnt habe, gibt es eine sehr gute Lösungen für das structure-from-motion Problem. Die Software von [Furukawa] benutzt die "multi-view stereo Algorithmenünd wandelt die Eingangsaufnahmen in eine Punktwolke-Datei um. Am Ausgang kriegt man ein sehr gutes Ergebnis. Die ganze Umgebung wird mittels 3-D Punkten abgebildet. Die sind zwar ziemlich dicht, doch bei Vergrößerung könnte man die Lücken sehen. Außerdem kann man mit der Punktwolke nicht viel anfangen, weil es meistens mit 3-D Objekten gearbeitet wird. Ich möchte diese Punktwolke-Datei als Basis für mein Konzept nehmen.

Es gibt einige Verfahren, die sich mit Rekonstruktion von Umgebung aus Luftbildern beschäftigen. Das sind zum Beispiel die Publikationen von [Suveg und Vosselman (2003)] und [Baillard und Zisserman (2000)]. Hier geht es zwar auch um die Stadtrekonstruktion, aber der ist Ansatz ganz anders. Die Luftbilder können nicht jederzeit problemlos und auch nicht von jedem einfach so genutzt werden. Außerdem werden bei Googele Maps die gesamten Stadtteile in 3-D rekonstruiert. Es wird deswegen schwierig ein einzelnes Gebäude raus zu filtern und es

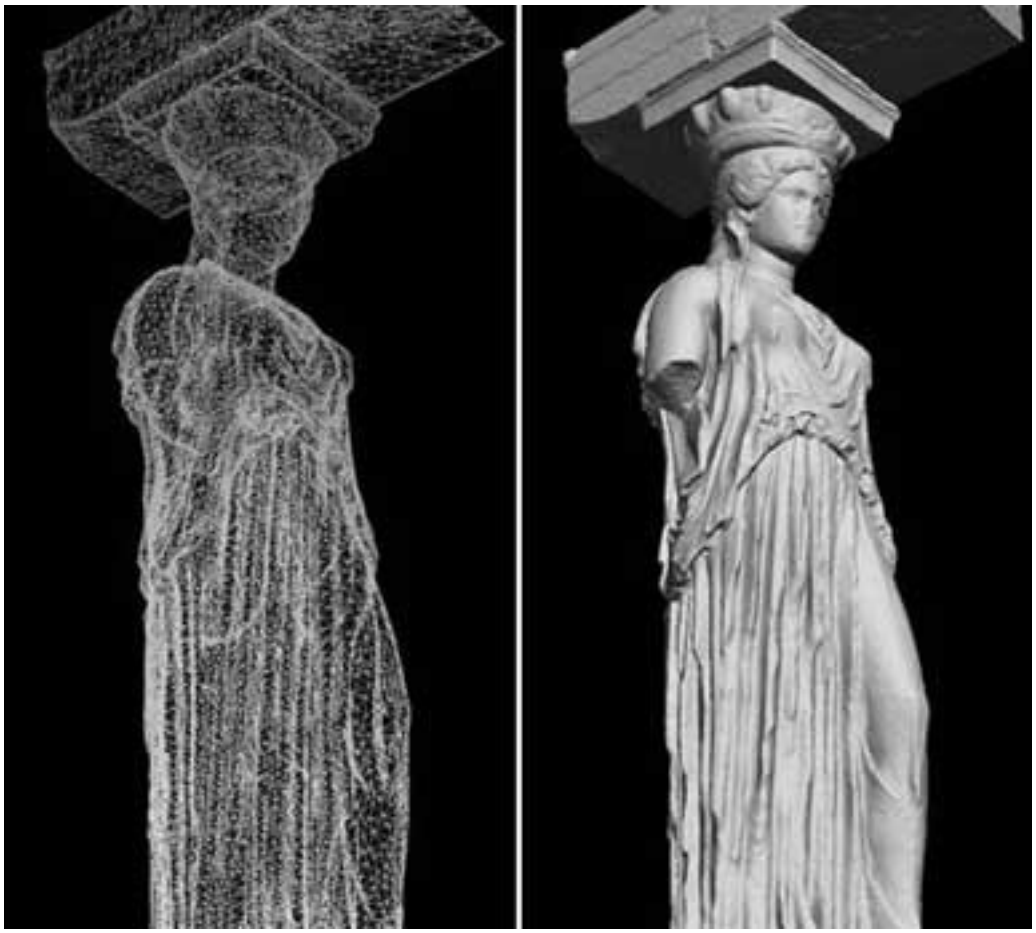

Abbildung 1.1: Punktwolke und 3-D Netz

von dem anderen zu unterscheiden. Eine Texturierung von diesem Gebäude würde hier gar nicht in Frage kommen.

[Larsen (2010)] hat ein sehr gutes Verfahren entworfen. Der wichtigste Beitrag in diesem Projekt, ist die Entwicklung eines Prozesses zur automatischen Fassade-Rekonstruktion, welches zu einem groben Modell solche Details, wie Türen und Fenster hinzufügt. Zu erst wird ein 3-D Modell des Gebäudes erstellt, was auch das Ziel meiner Bachelorarbeit ist.

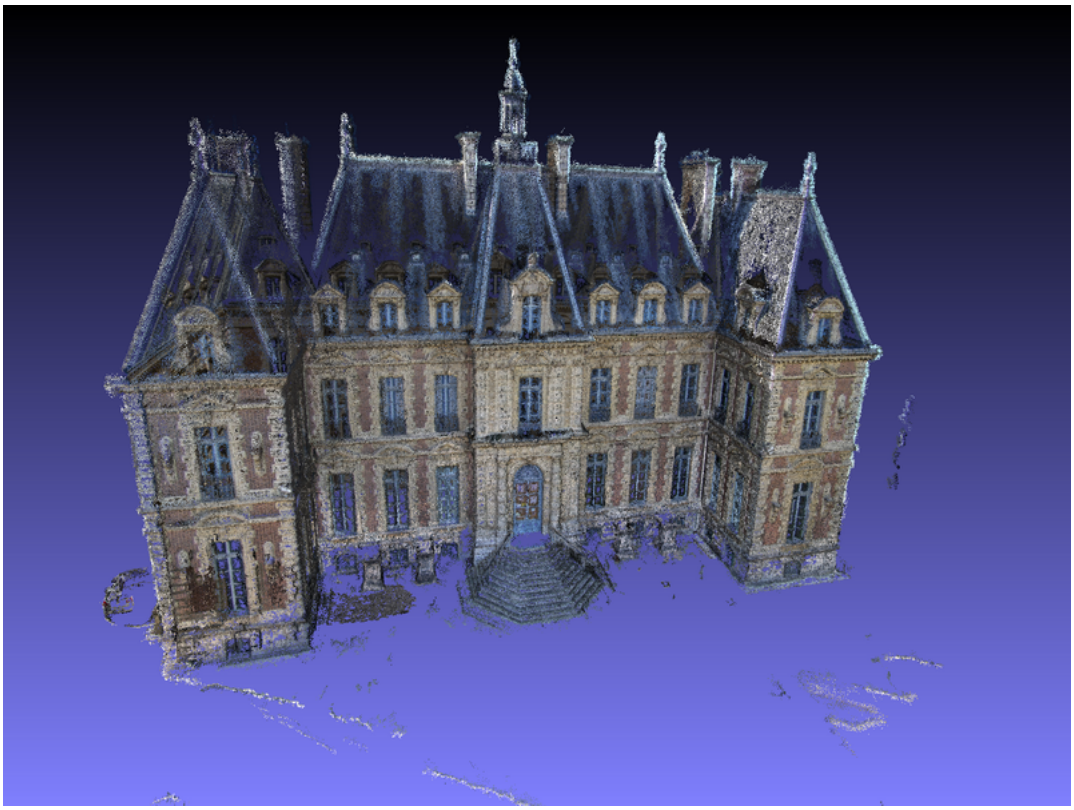

Abbildung 1.2: Ein mit dem Punktwolke dargestellte Gebäude

1.3 Aufbau dieser Arbeit

In dem Kapitel «Grundlagen» beschäftige ich mich mit Computer und Stereo Vision. Ich zeige, wie man aus einer 2-D Aufnahme eine 3-D Information kriegen kann. Danach erzähle ich über die Punktwolken, welche der Anfangspunkt meiner Arbeit sind.

In dem nächsten Kapitel «Entwicklungsumgebung und benutztes Software» wird erläutert, welches Software ich für die Umsetzung meiner Idee benutzte.

Dann kommt das Kapitel «Design», wo ich das Konzept meiner Entwicklung erläutere. Hier werden alle theoretischen Grundlagen und die von mir benutzten Algorithmen und Verfahren beschrieben.

In dem Kapitel «Implementierung» zeige ich, wie ich mein theoretisches Konzept praktisch umgesetzt habe.

Am Ende kommt das Kapitel «Evaluation», in dem ich meine Ergebnisse bewerten werde. Ich setze mich mit dem auseinander, was ich erreicht habe und was noch in der Zukunft gemacht werden könnte.

2 Grundlagen

In diesem Kapitel möchte ich einige Grundlagen von 3-D Rekonstruktion erläutern. Was stellt man unter einer 3-D Rekonstruktion vor? Um es einfacher auszudrucken, verwandelt man ein nicht 3-D Gegenstand in ein 3-D Gegenstand. In unserem Fall nehmen wir die 2-D Fotoaufnahmen, auf welchen ein Gebäude dargestellt wird und machen daraus ein 3-D Objekt, also rekonstruieren es aus 2-D in 3-D. Die 2-D und 3-D Aufnahmen bezeichnet man in der Computerwelt als«Computer Vision».

2.1 Computer Vision

2.1.1 Was ist Computer Vision?

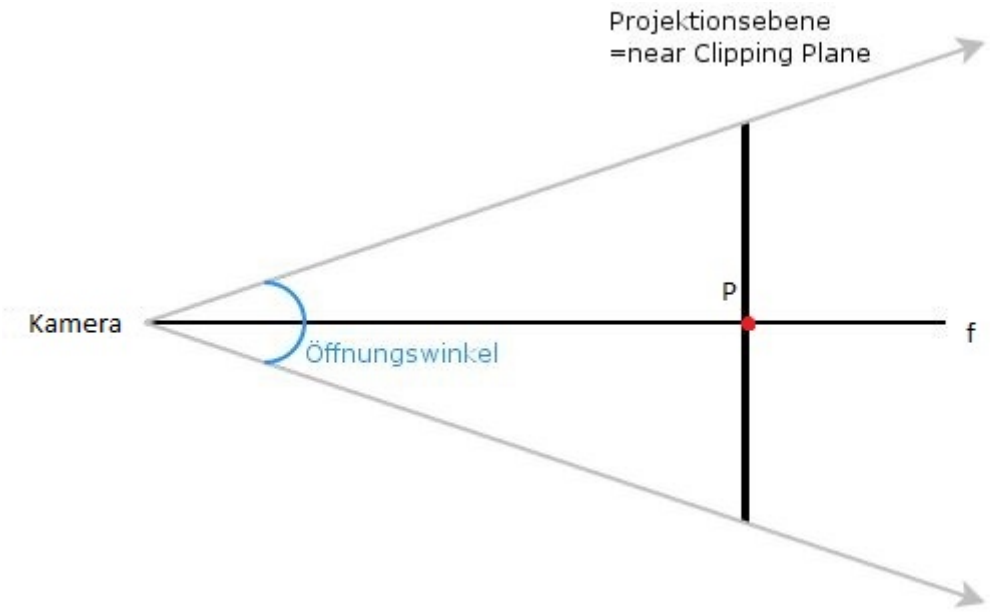

Abbildung 2.1: Projektion

Als Menschen sind wir in der Lage die 3-D Welt, wo wir leben, so wahr zu nehmen, dass sie für uns auch 3-D bleibt. Unsere Sehorgane sind so aufgebaut, dass unser Gehirn für jeden Punkt der Welt, außer der Position, auch die Tiefe berechnen kann. So betrachten wir das Bild, was wir sehen, in 3-D Koordinaten.

Das ist aber nicht der Fall für digitale Geräte. Eine Fotokamera zum Beispiel macht die Aufnahmen der Welt in 2-D. Betrachten wir die Abbildung 2.1. Alle Punkte, die in der realen Welt auf der Linie f liegen, werden auf einen einzigen Punkt P der Projektionsebene projiziert. Es heißt, dass bei 2-D Darstellung eine riesige Menge von Information verloren geht. Die Aufgabe von Computer Vision ist unter anderem, die Tiefeinformation aus Paaren von Bildern, zu extrahieren.

2.1.2 Tiefeinformation

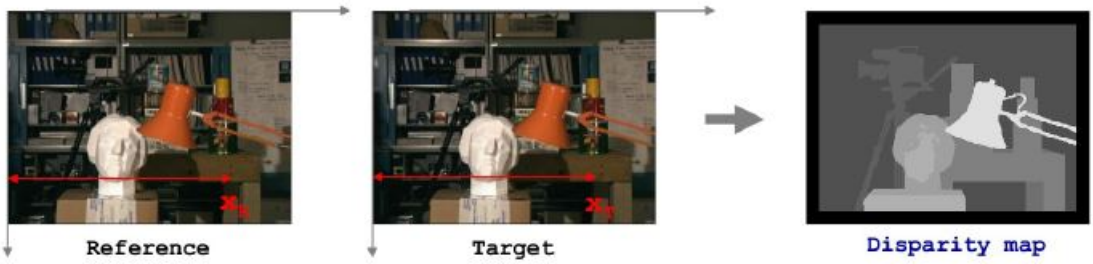

Abbildung 2.2: Disparity map

Tiefeinformation ist eine Bezeichnung für einen Punkt im Bild, welche die Entfernung von der Kamera bis zu diesem Punkt berechnet. Diese Information gibt uns schon bestimmte 3-D Vorstellung. Hat man die Tiefeinformation von jedem Punkt einer Fotoaufnahme, so kann man ein *Disparity map* bilden(Abbildung 2.2). Das ist ein Bild, was die Informationen über die Abstände der Oberflächen einer Szene von dem Betrachtungspunkt erhält. Die Punkte auf diesem Bild werden mit schwarz-weißem Gradient dargestellt. Je näher dieser Punkt zum Betrachter ist, desto weißer wird er auf dem Disparity map abgebildet. Die Erstellung von einem Bild mit Tiefeinformationen ist der erste Schritt, der gemacht werden muss, wenn man mit 3-D Rekonstruktion anfängt. Mehr über Disparity map und über Computer Vision allgemein kann man bei [Szeliski (2010)] nachlesen.

2.1.3 Stereo-Paar

Stereovision ist ein Teil der Computervision, was sich damit beschäftigt, die Tiefeinformation aus Bildern zu extrahieren. Um diese Information zu bekommen, braucht man zwei Aufnahmen von einem Objekt - Stereo-paar. Genauso wie der Mensch zwei Augen braucht, um zwei Abbildungen von der Welt und Umgebung zu erzeugen(Abbildung 2.3).

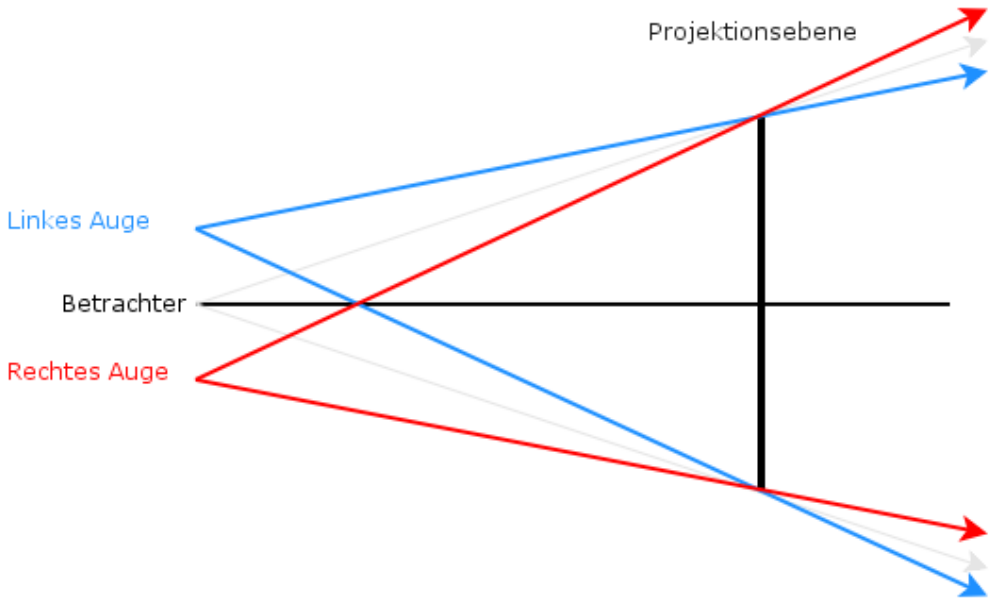

Abbildung 2.3: Stereovision

Stereo-Paar ist ein Paar von planaren Bildern des gleichen Objektes, welches kleine Unterschiede zwischen den Bildern enthält, die dafür dienen, den Volumeneffekt darzustellen. Dieser Effekt ergibt sich aus der Tatsache, dass die Objekte der Szene sich in verschiedenen Abständen von dem Betrachter befinden. Wenn man die aus verschiedenen Blickwinkeln betrachtet, haben diese unterschiedliche Winkelverschiebungen(Parallaxe). Ein klassisches Stereo-Paar besteht aus zwei Bildern, die horizontal nebeneinander angeordnet sind, in einem Abstand was allgemein dem Menschlichen Augenabstand entspricht.

Um ein Stereo-Paar zu kriegen, braucht man zwei Aufnahmeeinrichtungen, die gleichzeitig betrieben werden. Es gibt Stereo-Kameras, die speziell dafür geeignet sind. Außerdem gibt es Sondervorsätze für Objektive an den traditionellen Fotokameras. Jedoch könnte man mit einer normalem Kamera oder mit einem Smartphone ein Stereo-Paar von einem statischem Objekt aufnehmen, indem man zwei Aufnahmen mit einer Versetzung macht.

2.1.4 Korrespondenzproblem

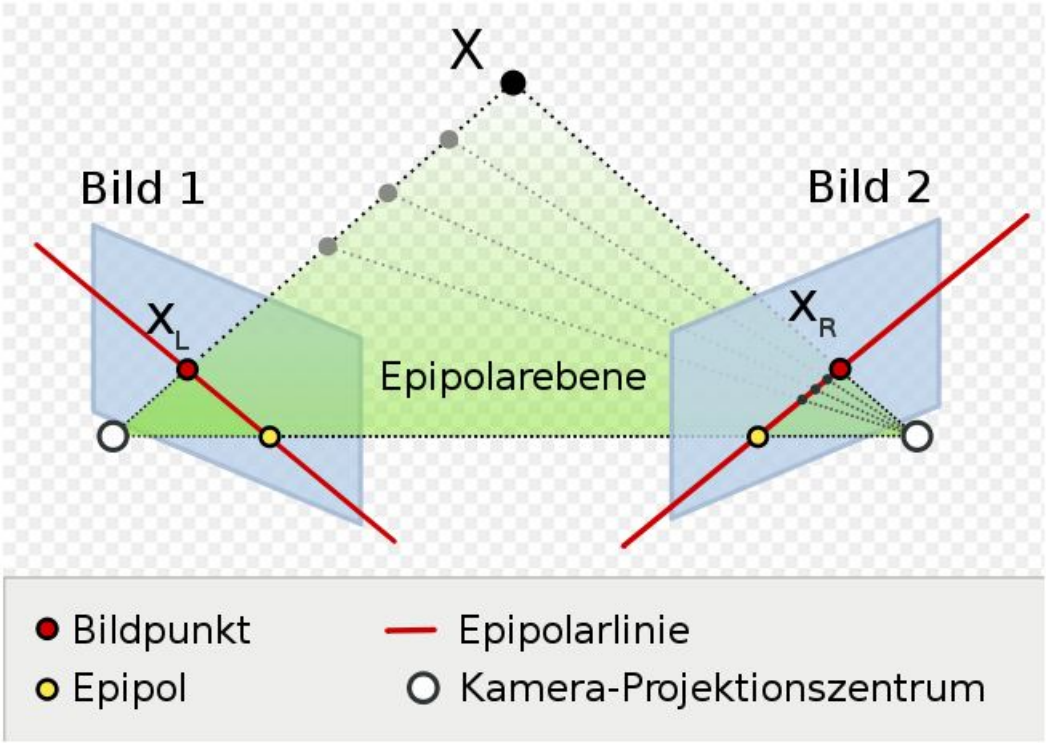

Abbildung 2.4: Epipolargeometrie

Hat man ein Stereo-Paar, ist jetzt die Aufgabe die korrespondierenden Punkte auf den beiden Bildern zu finden - Korrespondenzpaar. Ein Punkt oder besser gesagt eine Menge aus Punkten im linken Bild kann überall in rechtem Bild sein. Die Aufgabe, ein korrespondierender Punkt zu finden, ist nicht trivial. Deswegen wäre es schön diese Menge zu begrenzen. Und hier kommt die Epipolargeometrie zur Hilfe.

Epipolargeometrie ist ein mathematisches Modell aus der Geometrie, das die geometrischen Beziehungen zwischen verschiedenen Kamerabildern des gleichen Objekts darstellt[wiki]. Betrachten wir die Abbildung 2.4. Die Ursprungsorte der beiden Kamerakoordinatensysteme und ein beliebiger Punkt im 3-D Raum, bestimmen eine Epipolarebene. Jeder Punkt auf dieser Ebene wird auf dieselben beiden Epipolarlinien in den beiden Bildern projiziert. So schränkt sich, bei bekannter Epipolargeometrie der Suchbereich im zweiten Bild auf eine Linie ein. Gleichzeitig verringert das die Anzahl von falschen Zuordnungen korrespondierender Punkte durch die Suchraumeinschränkung.

Abbildung 2.5: Korrespondierende Punkte

Die Korrespondenzen müssen nicht für jeden Punkt des Bildes gefunden werden. Es wäre auch fast unmöglich, da die meisten Punkte sehr schwierig zu identifizieren sind. Auf dem blauen Himmel, zum Beispiel, sind alle Punkte gleich blau. Dafür müssen zuerst die Bildmerkmale gefunden werden. Das sind die Stellen auf dem Bild, die sich von ihrer Umgebung schroff unterscheiden. Die Merkmale sind zum Beispiel:

- weiße Flecken auf dunklem Hintergrund(Kontraständerung)
- Kanten, Ecken und andere gut strukturiert Objekte
- Regionen, deren Farbe sich von anderen Objekten sehr gut unterscheiden lässt

Für solche Bildmerkmale ist es viel einfacher die korrespondierende Punkte zu finden und die zu machen.

Es gibt mehrere Verfahren, die eine Lösung für das Korrespondenzproblem anbieten, wie zum Beispiel SIFT und SURF Algorithmen. Das Hauptziel ist es in der Pixelmenge auf den beiden Bildern ein Stereo-Paar zu finden, die paarweise denselben Punkt der 3-D Welt abbilden. Abbildung 2.5 zeigt zwei Bilder mit gefundenen Korrespondenzpunkten. Mit dieser Kenntnis kann man mit Hilfe von Triangulation anfangen, die Tiefeinformation zu berechnen.

2.1.5 Triangulation

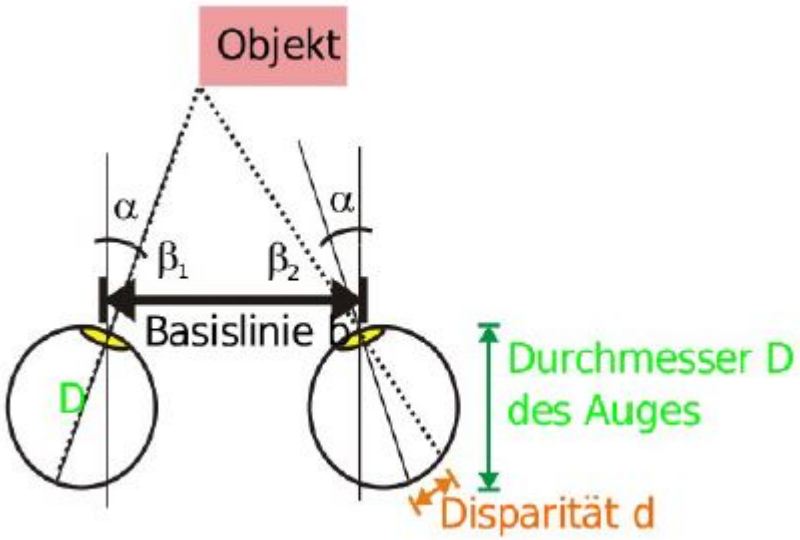

Abbildung 2.6: Triangulation

Das Problem der Bestimmung der Position eines 3-D Punktes aus einer Menge von entsprechenden Korrespondenzpunkten und bekannten Kamerapositionen wird Triangulation genannt. Die Idee ist einfach und aus Geometrie bekannt. Drei Punkte repräsentieren einen Dreieck. Hier sind zwei Punkte die zusammengehören und Korrespondenzpunkte heißen. Der dritte Punkt ist ein Punkt in der 3-D Welt. Die Abbildungen des dritten Punktes sind diese entsprechende zwei Korrespondenzpunkte.

Sehen wir die Abbildung 2.6 an. Der Dreieck wird durch zwei Winkel $\beta 1$ und $\beta 2$ und einer Seite b bestimmt. Die Winkel sind eine Kombination, welche sich aus der Fokussierungsrichtung α und Disparität d zusammensetzt. Die Seite ist die Basislinie, also der Abstand zwischen Linsenzentren.

Wenn man den Dreieck kennt, lässt sich die Höhe dieses Dreiecks berechnen. In unserem Fall ist das der Abstand zwischen der Basislinie und dem entsprechenden Punkt in der 3-D Welt - die Tiefeinformation(Z in Abbildung 2.7). Bei der Triangulation von Stereo-Paaren ist der Abstand zu einem bestimmten Punkt, von Disparität abhängig.

Die Disparität ist der Unterschied der x-Koordinaten zwei korrespondierender Punkte. Je näher ein Punkt an der Kamera liegt, desto größer ist die Disparität und umgekehrt.

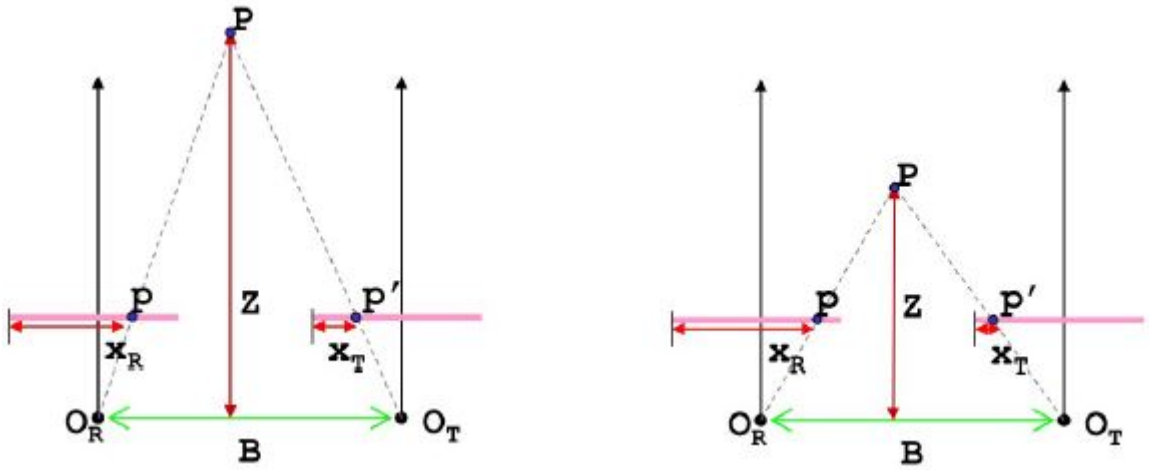

Abbildung 2.7: Triangulation

2.2 Punktwolken

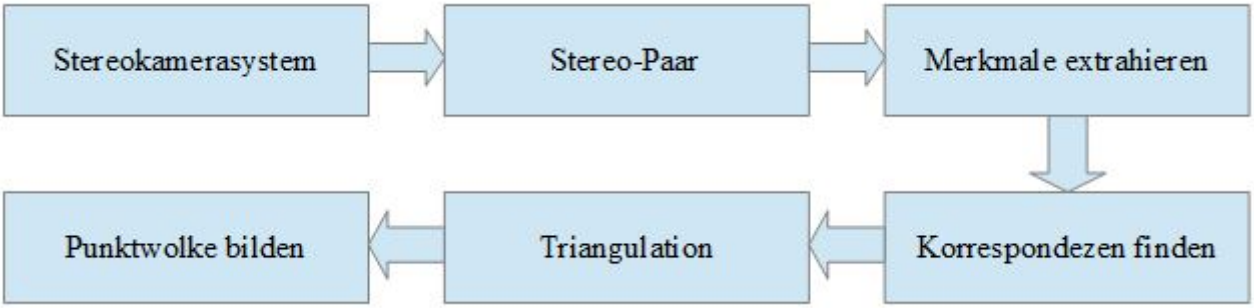

Abbildung 2.8:

Nachdem man die Tiefinformationen einer Szene berechnet hat und Disparity map vorhanden ist, kann man mit der Transformation des Objektes in 3-D Raum anfangen. Da die 3-D Koordinaten von allen Punkten schon bekannt sind, kann es daraus eine Datenstruktur namens Punktwolke gebildet werden.

Mit dem Begriff Punktwolke oder Punkthaufen wird eine Menge von Punkten eines Vektorraums bezeichnet, die eine unorganisierte räumliche Struktur ("Wolke") aufweist[Otepka u. a. (2013)]. Eine Punktwolke ist durch die enthaltenen Punkte beschrieben, die jeweils durch ihre Raumkoordinaten erfasst sind. Zu den Punkten können zusätzlich Attribute, wie z. B. geometrische Normalen, Farbwerte oder Messgenauigkeiten, erfasst werden.

Abbildung 2.9: Ein mit Punktwolke beschriebener Szene

Das erste Ziel der 3-D Rekonstruktion ist das Erstellen einer Punktwolke-Datenstruktur aus den Bilderpaaren. Damit abstrahiert man die 2-D Ebene und erhält dann eine räumliche Darstellung. Die Abbildung 2.8 zeigt noch mal die Schrittsequenz für die Erstellung von einer Punktwolke-Datei aus einem Bilderpaar.

Eine Software, welche alle diese Schritte ausführt und aus einer Reihe von Fotoaufnahmen die Punktwolke zur Verfügung stellt, ist zum Beispiel «osm bundler». Da werden die tools von Furukawa benutzt. Ich arbeite mit dieser Software, um eine Punktwolke-Datei für meinen Prototypen zu entwickeln.

3 Design

In diesem Kapitel geht es um das Konzept meines Projektes. Hier wird das theoretische Wissen erläutert, welches für die Umsetzung eines Prototyps sowie der Algorithmen, notwendig ist.

3.1 Architektur

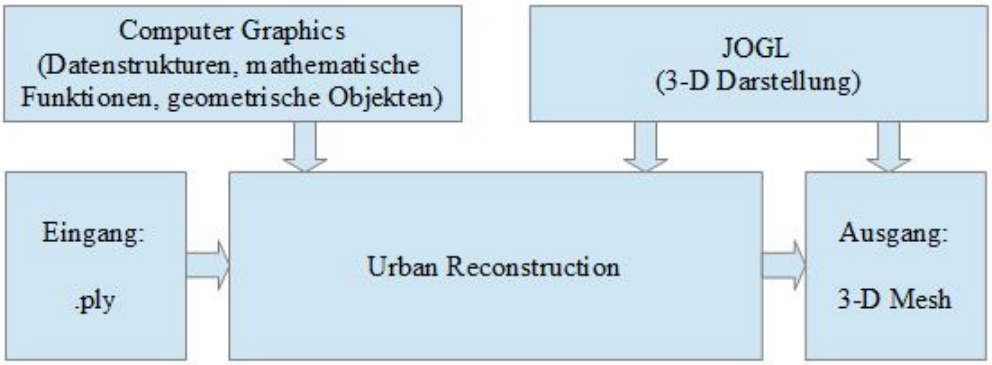

Abbildung 3.1: Komponentenarchitektur

Die Abbildung 3.1 stellt die Architektur des Prototypes dar. Die Hauptkomponente der «Urban Recontruction» - sind das Kern meines Projektes, aus welchem ich meine Idee implementiere. Die zwei fremden Komponente werde ich benutzen - «JOGL» für die 3-D Darstellung von Meshes und «Computer Graphics» auch mit der Sammlung von Grundlagen für die Arbeit mit geometrischen Objekten und verschiedenen Datenstrukturen, wie zum Beispiel den Vektoren und den Punktwolken.

3.2 Arbeit mit Punktwolken

Wie ich schon erwähnt habe, arbeite ich in meinem Prototypen mit einer Punktwolke-Datei. Das ist die Eingangsdatei für mein Projekt, mit der ich die 3-D Rekonstruktion anfange. Davor muss aber diese Datei noch vorbereitet werden.

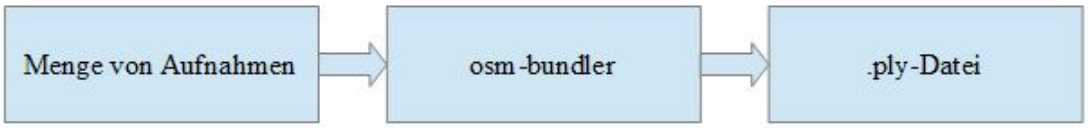

Abbildung 3.2: .ply Generierung

Ich benutze die Software osm bundler, um die Punktwolke zu generieren. Sobald ich eine Menge von Fotoaufnahmen von einem Objekt habe, kann ich diese in die Software exportieren und das Prozess der Konvertierung laufen lassen. Die Ausgangsdatei von osm bundler ist die .ply-Datei.

```
1  ply
2  format ascii 1.0
3  element vertex 5875
4  property float x
5  property float y
6  property float z
7  property float nx
8  property float ny
9  property float nz
10 property uchar diffuse_red
11 property uchar diffuse_green
12 property uchar diffuse_blue
13 end_header
14 -0.179383 2.42424 -12.0903 -0.532327 -0.753292 0.386237 251 217 162
15 -0.256097 2.35224 -11.9081 -0.47614 -0.604125 0.639003 249 193 148
16 -0.133479 2.37441 -12.0134 -0.589312 0.455402 0.667324 251 204 154
17 -0.0756632 2.45501 -12.2107 -0.711215 0.344272 0.612903 251 218 159
18 -0.0756632 2.45501 -12.2107 -0.711215 0.344272 0.612903 251 218 159
19 -0.045207 2.48288 -12.2936 -0.755079 0.269463 0.5977 251 218 157
```

Abbildung 3.3: Der Inhalt von .ply-Datei

Der Inhalt von .ply-Datei ist die Information über die Punkte, die das fotografierte Objekt bestimmen. Die Abbildung 3.3 demonstriert einen Ausschnitt aus einer .ply-Datei. Zuerst kommt das Header, wo es beschrieben wird, welche Eigenschaften eine Punkt hat. Diese werden in Zahlen unten dargestellt. Jede Zeile, die nach dem Bezeichner *end_header* kommt, beschreibt einen Punkt der Punktwolke. Die Zahlen werden hier von Links nach Rechts in Dreiergruppen geteilt. Die erste Gruppe repräsentiert die Koordinaten des Punktes im 3-D

Raum. Danach stehen die drei Zahlen für den Normalvektor. Die letzte Gruppe beschreibt die Farbe des Punktes in RGB Format.

Damit ich mit den Daten aus .ply-Datei arbeiten kann, müssen die zuerst in eine Datenstruktur umgewandelt werden, die mein Projekt versteht. Dafür brauche ich einen Konverter. Die Datenstruktur mit der ich arbeite heißt *PointCloud* und speichert eine Liste von Punkten ab, die wie folgt beschrieben sind:

- jeder Punkt hat einen dreidimensionalen Vektor, welches die Position des Punktes speichert(Koordinaten)

- jeder Punkt hat einen dreidimensionalen Vektor, welches den Normalvektor des Punktes speichert

- jeder Punkt hat einen dreidimensionalen Vektor, welches die Information zu der Farbe des Punktes in RGB Format speichert

Die Aufgabe des Konverters ist in diesem Fall die .ply-Datei Zeile nach Zeile abzuarbeiten und die Zahlen, die den Punkt beschreiben, in den entsprechenden Felder der den Punkt repräsentierte Datenstruktur abzuspeichern. Alle Punkte werden in ein *PointCloud*-Objekt zusammengefasst.

Das Konzept des Konverters impliziert die Arbeit mit Streams und Files. Wird die .ply-Datei einmal in eine interne Datenstruktur konvertiert, so kann man die überall im Programm weiter benutzen.

3.3 RANSAC Algorithmus

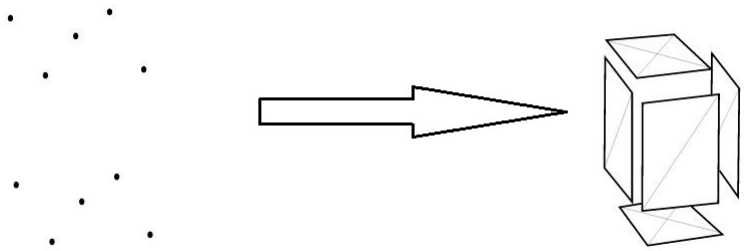

Abbildung 3.4: Polygon aus Punktwolke

Der erste Schritt zu einem 3-D Darstellung ist gemacht - die Punktwolke ist vorhanden und für die Bearbeitung vorbereitet. Diese Datenstruktur repräsentiert aber nur eine chaotische

ungeordnete Menge von Punkten. Demnächst muss diese Menge in einer bestimmten Weise strukturiert werden.

Das Endziel ist ein 3-D Modell, welches in der Computer Grafik auch *Mesh* oder *Polygonnetz* genannt wird. Die bestehen aus untereinander mit Kanten verbundenen Punkten. Dafür werden meistens die Dreiecksnetze verwendet. Die Aufgabe jetzt, ist aus der Punktwolke einen Dreiecksnetz zu bilden, um die Polygone zu bestimmen, durch welche ein Gebäude repräsentiert wird.

Die Gebäude bestehen meistens aus vier Wänden mit einem Dach. Ist der Dach flach, so wird aus einem Gebäude ein Würfel. Das heißt für uns, dass wir die Punktwolke nach den Punkte durchsuchen sollen, welche die Wände des Gebäudes bestimmen. Diese Aufgabe lässt sich mit dem RANSAC-Algorithmus lösen.

RANSAC steht für **ran**dom **sa**mple **c**onsensus, was in Deutsch etwa «übereinstimmung mit einer zufälligen Stichprobe» bedeutet. Das ist ein Algorithmus, welches zur Schätzung eines Modells innerhalb einer Reihe von Messwerten dienen sollte, auch mit Ausreißern und groben Fehlern. Mit anderen Worten kann man mit Hilfe von RANSAC die mathematischen Strukturen identifizieren, wie zum Beispiel Linien oder Kreise. In unserem Fall verwende ich diesen Algorithmus, um die Ebenen zu entdecken.

Das Prinzip von RANSAC ermöglicht uns die beste Ebenen in einer 3-D Punktwolke zu finden. Gleichzeitig reduziert der Algorithmus die Anzahl von Iterationen, sogar in dem Fall, wenn die Menge von Punkten sehr groß ist. Es werden hierzu 3 Punkte zufällig gewählt und die Parameter von der entsprechenden Ebene werden kalkuliert. Danach identifiziert der Algorithmus, nach einer vorgegebenen Schwelle, alle Punkte der gesamten Punktwolke, die zu berechneter Ebene gehören. Die Prozedur wird N-mal wiederholt; jedes Mal wird das neue Ergebnis mit dem Gespeichertem verglichen. Falls das neue Ergebnis besser ist, wird das Alte durch das Neue ersetzt.

Der Algorithmus braucht als Input vier Sätze, welche sind:

- Eine 3-D Punktwolke(*point_cloud*)

- Eine Toleranzschwelle von dem Abstand t zwischen der gewählten Ebene und anderen Punkten. Dieser Wert ist verbunden mit altimetrischer Genauigkeit von der Punktwolke

- Ein *forseeable_support* ist eine maximale wahrscheinliche Anzahl von Punkten, die zu einer Ebene gehören. Die ist von der Punktedichte abgeleitet

- Die minimale Wahrscheinlichkeit α mindestens eine gute Ebene nach N Versuchen zu finden.

Die Abbildung 3.5 demonstriert den Pseudocode von RANSAC Algorithmus für die Identifizierung der Ebene.

1. *bestSupport* = 0; *bestPlane*(3,1) = [0, 0, 0]
2. *bestStd* = ∞; *i* = 0
3. ε = 1 – *forseeable_support*/length(*point_list*)
4. *N*=round (log (1 – α)/log (1 – (1 – ε) ^3))
5. **while** (*i* <= *N*)
6. *j* = pick 3 points randomly among (*point_list*)
7. *pl* = pts2plane(*j*)
8. *dis* = dist2plan(*pl*, *point_list*)
9. *s* = find(abs(*dis*)<= *t*)
10. *st* = Standard_deviation (*s*)
11. **if** (length(*s*) > *bestSupport* **or** (length(*s*) = *bestSupport* **and** *st* < *bestStd*)) **then**
12. *bestSupport* = length (*s*)
13. *bestPlan* = *pl*; *bestStd* = *st*; **endif**
14. *i* = *i*+1; **endwhile**

Abbildung 3.5: RANSAC Pseudocode [Tarsha-Kurdi u. a. (2008)]

In diesem Pseudocode ϵ steckt der Prozentsatz der akzeptierten fehlerhaften Beobachtungen. Die Funktion *pts2plane* berechnet die Parameter der Ebene aus gewählten Punkten. Diese benutzt die Formel der Fläche:

$$A^*x + B^*y + C^*z + D = 0$$

wo A, B und C bestimmen die Norm der Ebene und x, y und z sind die Koordinaten des Punktes, welche zu dieser Ebene gehören.

Die Funktion *dist2plan* berechnet den Abstand zwischen den Punkten der Punktwolke und gefundener Ebene(kann positiv und negativ sein). Die Formel ist:

$$d = P - X^*norm$$

wo P der angegebene Punkt ist, X irgendein Punkt der Ebene ist und norm der Normalvektor der Ebene ist.

Anzahl von Wiederholungen *N* kann wie folgend berechnet werden[cite8]:

$$N = \frac{\log(1-\alpha)}{\log(1-(1-\varepsilon)^s)}$$

wo *s* die minimale Anzahl von Punkten ist, die für die Bestimmung des Modells notwendig sind(im Fall der Ebene sind das drei Punkte).

Um alle Wände des Gebäudes (Seiten des Würfels) zu finden, wird der Algorithmus mehrmals aufeinander angewendet. In jeder Iteration wird die Menge von betrachteten Punkten aus der Punktwolke entfernt. Die Operation wird wiederholt, bis die Anzahl von gebliebenen Punkten kleiner als die angegebene Schwelle wird. In unserem Fall, kennt der Benutzer die Wändenanzahl und kann dem entsprechend auch die Zahlen der Wiederholungen selbst angeben.

3.4 Arbeit mit Ebenen

In 3-D Rekonstruktion arbeitet man sehr viel mit den Ebenen, welche die Hauptdatenstruktur meiner Arbeit darstellen. Diese werden aus der Punktwolke extrahiert und als Grundgerüst für das 3-D Netz verwendet.

Eine Ebene kann man durch einen Punkt und einen Normalvektor bestimmen.

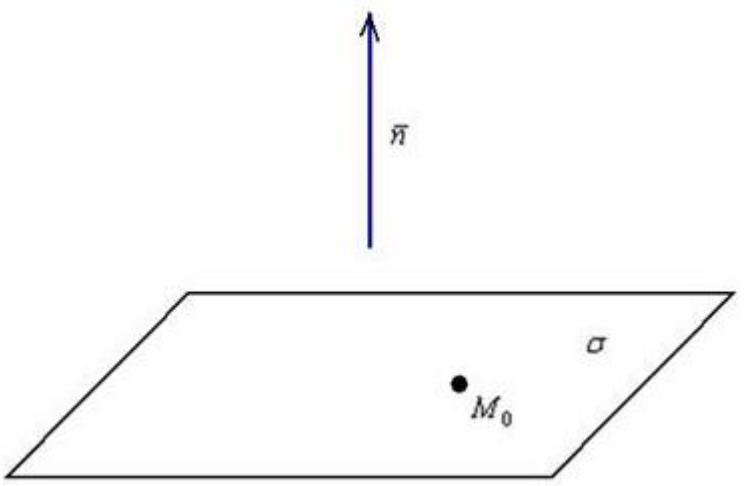

Abbildung 3.6: Die durch den Punkt und den Normalvektor bestimmte Ebene

Wie ich schon erwähnt habe, arbeitet der RANSAC-Algorithmus viel mit Ebenen. Erstmal wird die Gleichung der Ebene aus drei Punkten gebildet. Dafür kann man eine folgende Formel benutzen:

$$\begin{vmatrix} x-x_0 & x_1-x_0 & x_2-x_0 \\ y-y_0 & y_1-y_0 & y_2-y_0 \\ z-z_0 & z_1-z_0 & z_2-z_0 \end{vmatrix} = 0$$

Wenn die drei Punkte, die nicht auf einer Linie liegen, bekannt sind, kann man zwei nicht-kollineare Vektoren finden, die parallel zu dieser Ebene sind.

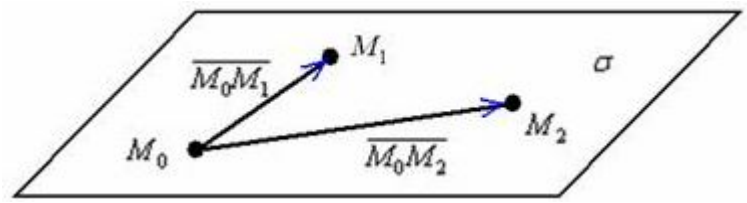

Abbildung 3.7: Zwei durch drei Punkten bestimmte nicht-kollineare Vektoren der Ebene

So sind die zwei Vektoren:

$$\overline{M_0M_1}(x_1 - x_0; y_1 - y_0; z_1 - z_0)$$
$$\overline{M_0M_2}(x_2 - x_0; y_2 - y_0; z_2 - z_0)$$

Setzt man die Vektoren in die oben stehende Gleichung und löst sie, kommt man zu folgendem Ergebnis:

$$A^*x + B^*y + C^*z + D = 0$$

wo Vektor (A,B,C) der Normalvektor (nicht normalisiert) der Ebene ist und D den Abstand bis zum Ursprung bestimmt.

Damit Berechnungen mit Ebenen richtig funktionieren, muss der Normalvektor erstmal normalisiert werden, das heißt zu der Form des Einheitsvektors gebracht werden. Das kriegt man hin indem man jede Koordinate des Vektors durch den Betrag dividiert.

Nachdem alle Ebenen gefunden sind, müssen diese bearbeitet werden. Wir haben schon alle Punkte bestimmt, die auf dieser Ebene liegen. Der nächste Schritt wäre all diese Punkte zu strukturieren, und zwar so, dass diese einen Rechteck bilden würden, um eine Wand eines Gebäudes zu repräsentieren.

Die Ebene ist eine 2-D Struktur, deswegen ist es nicht leicht mit dieser in einem 3-D Raum zu arbeiten. Um das zu erleichtern, transformieren wir die Ebene zuerst in 2-D Koordinatensystem. Dafür muss jeder Punkt der Ebene auf x-y-Ebene projiziert werden.

Mit einer Hilfsmatrix *R* wird ein lokales Koordinatensystem der Ebene bestimmt. Diese Matrix wandelt die Ebene in eine x-y-Ebene um. Demnächst müssen alle Punkte, die zu der Ebene gehören, in das Ebenen-Koordinatensystem transformiert werden.

Für die Projektion von Punkten verwendet man den folgenden Formel:

wo Xwelt der zu transformierende Punkt ist und p der die Ebene bestimmende Punkt ist.

$$T_{Welt->Ebene} = R^{-1} * (x_{Welt} - p)$$

Jetzt kann man ruhig mit x-y-Koordinaten arbeiten. Für die Rücktransformation tut man folgendes:

$$T_{Ebene->Welt} = R * x_{Ebene} + p$$

Nachdem man alle Punkte einer Ebene erfolgreich transponiert hat, ist es jetzt die Aufgabe, einen Rechteck zu finden. Den Rechteck kann man durch vier Eckpunkte definieren. Die Idee dabei ist es, in einer Menge nach Punkten zu suchen, die zu einer Ebene gehören und diese als Eckpunkte fest zu halten.

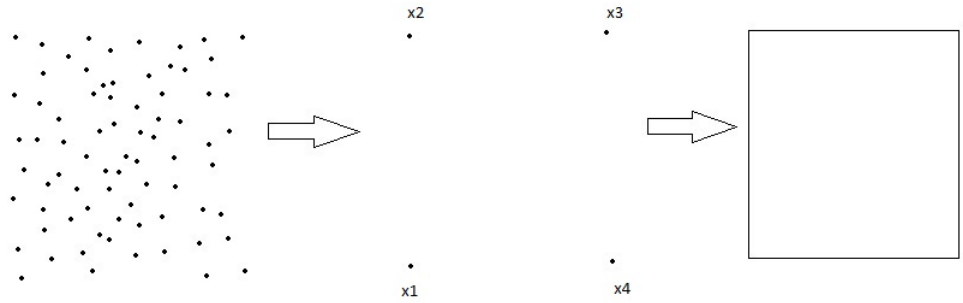

Abbildung 3.8: Eckpunkte finden, um Rechteck zu bestimmen

Es gibt vier Punkten, die ich finden soll:
- Der Punkt links unten - x1 - hat die kleinsten x- und y-Werte
- Der Punkt linkst oben - x2 - hat den kleinsten x-Wert und den größten y-Wert
- Der Punkt rechts oben - x3 - hat die größten x- und y-Werte
- Der Punkt rechts unten - x4 - hat den größten x-Wert und den kleinsten y-Wert

Weiter soll ich nur noch die passenden Punkte aus der Menge raus filtern (siehe Kapitel «Implementierung»).

3.5 3-D Modellierung

Wie ich bereits erwähnt habe, besteht ein 3-D Modell aus einem Dreiecksnetz. Nachdem wir für jede Ebene, die vier Eckpunkte bestimmt haben, bleibt es uns nur noch diese, miteinander in ein Netz zu verbinden. So haben wir für jede Ebene ein Polygon.

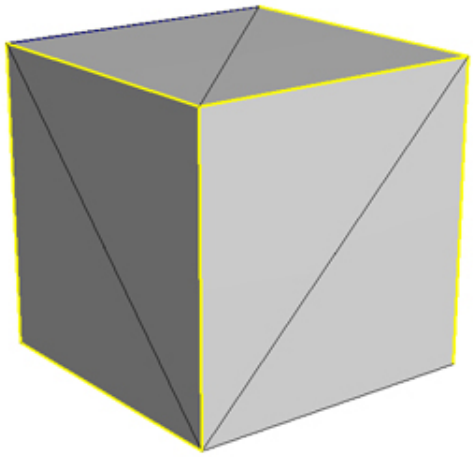

Abbildung 3.9: Ein 3-D Netz eines Würfels

Nachdem wir für alle Ebenen die Eckpunkte gefunden haben, die zurücktransformiert haben und daraus Die Polygone gebildet haben, könnte es passieren, dass die Kanten des Würfels nicht gut aneinander liegen und die Ebenen sich dadurch schneiden. Um diesen Effekt zu beseitigen, muss noch *snapping* gemacht werden. Die Idee ist, alle drei Punkte einer Würfelecke auf einen Punkt zu projizieren(Mehr dazu in Kapitel «Implementierung»).

4 Implementierung

In diesem Kapitel erläutere ich die Hauptpunkte der Implementierung meines Prototyps. Hier kriegt man Einblick auf eine Klassenübersicht und manche Algorithmen mit Codebeispielen.

4.1 Entwicklungsumgebung und benutztes Software

Die bereits erwähnte Software, welche ich für die Vorbereitung der Punktwolke verwendet habe, heißt osm bundler. Diese verwende ich aber nicht direkt in meinem Prototypen. Dies ist nur ein Vorschritt, welches für meine Software die Eingangsdatei zur Verfügung stellt. Mann kann auch ein anderes Softwareprogramm benutzen, welches eine Punktwolke generiert.

Mein Prototyp wird mit Java 7 entworfen. Als Entwicklungsumgebung verwende ich Eclipse Kepler. Ich benutze keine externen Bibliotheken, die zu erst installiert werden müssen. Ich arbeite nur mit einem Projekt namens «Computer Graphics Lab».

4.1.1 Computer Graphics Lab

Die Computer Graphics Lab (CGL) ist ein von Professor Philipp Jenke entwickeltes Projekt, welches die Entwicklung von Grafik-basierten Anwendungen erleichtert. CGL beinhaltet eine Menge von Klassen, welche die wichtigsten geometrischen Datenstrukturen und mathematische Operationen, zusammenfasst. Die Klassen, die ich in meinem Projetk benutze, sind zum Beispiel *Plane, PointCloud, Vector3, IMatrix3 u.a.*.

Die Anzahl von Packages, die CGL zur Verfügung stellt, ist sehr umfangreich. Man kann die Auflistung in der Abbildung 4.1 sehen.

Außerdem verwendet und erweitert CGL die Bibliothek JOGL. Die 3-D Darstellung wird mit Java realisiert. Dieses Projekt fasst alle CG-Objekte zusammen, indem es für jedes Objekt ein Node-Element erstellt. Ein Netz aus Nodes wird definiert und auf dem Bildschirm gerendert. Jedes Node kann man bearbeiten und/oder entfernen.

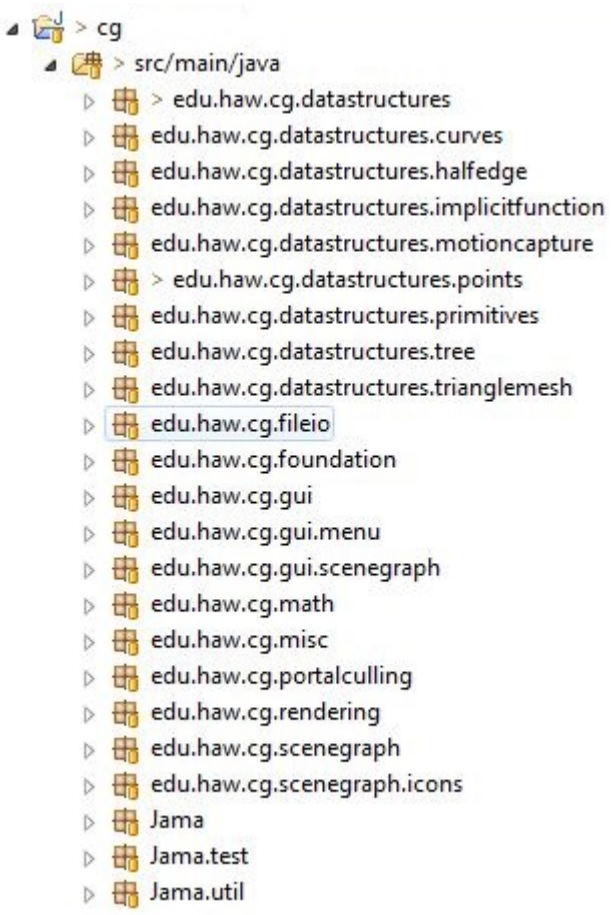

Abbildung 4.1: Übersicht von Computer Graphics Lab Packages

4.2 Übersicht von Klassen

Die Abbildung 4.2 stellt ein Klassendiagramm dar. Die Klasse *UrbanReconstruction* dient als Eingangspunkt in das Programm und macht nichts anderes als ein Exemplar von *UrbanGUI*. GUI wird als JOGL-Element registriert und angezeigt.

4.2.1 PlyConverter

Die Klasse *PlyConverter* dient dazu, um die externe .ply-Datei in eine interne Datenstruktur *PointCloud* zu konvertieren. Dieses Verfahren habe ich im Kapitel «Design» erläutert. Jetzt noch ein Paar Worte zu Implementierung.

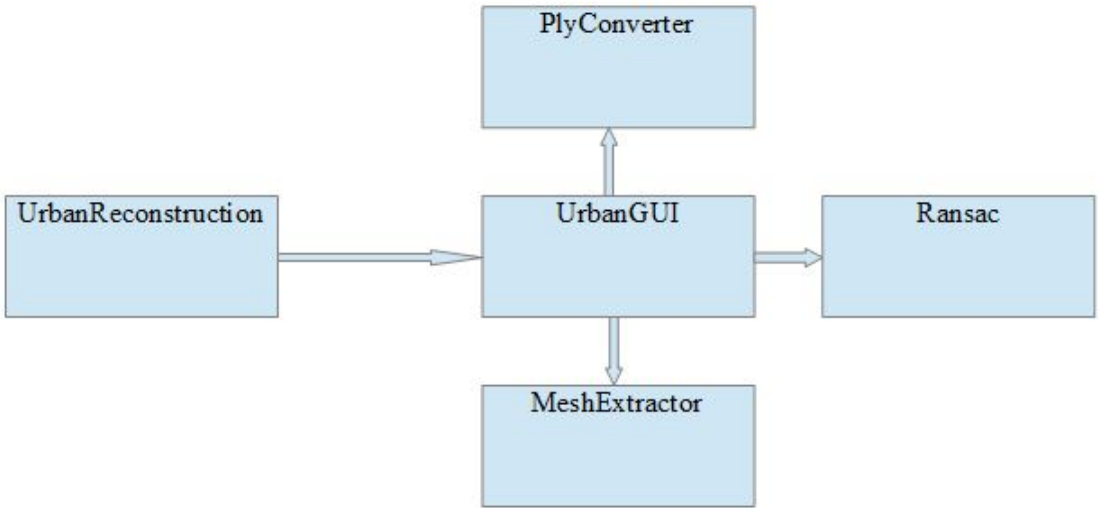

Abbildung 4.2: Klassendiagramm

Zuerst wird jede Zeile mit Buffered File Reader gelesen und in einer Liste gespeichert(Listing 4.1).

```
String line = "";
List<String> points = new LinkedList<String>();
BufferedReader br = new BufferedReader(new FileReader(ply));
do      {
        line = br.readLine();
        } while (!line.equals("end_header"));
while (line != null) {
        line = br.readLine();
        if (line != null)
        points.add(line);
}
```

Listing 4.1: PlyConverter: Data einlesen

Danach werden aus jeder Zeile die Zahlen extrahiert und zu einem *Point* erfasst(Listing 4.2). Die Funktion *divideString(double[] point, String s)* speichert neun Zahlenwerte aus der Zeile *s* in einem Array *point*.

```
double[] point = new double[9];
int NUMBER_OF_POINTS = points.size();
IPointCloud pointCloud = new PointCloud();
```

```
for (int i = 0; i < NUMBER_OF_POINTS; i++) {
        divideString(point, points.get(i));
        IVector3 position = VectorMatrixFactory.newIVector3(point[0],
        point[1], point[2]);
        IVector3 color = VectorMatrixFactory.newIVector3(point[6]/255
        , point[7]/255, point[8]/255);
        IVector3 normal = VectorMatrixFactory.newIVector3(point[3],
        point[4], point[5]);
        pointCloud.addPoint(new Point(position, color, normal));
}
```

Listing 4.2: PlyConverter: Punkte extrahieren

Damit die Farbe auch richtig angezeigt wird, muss ein Wert noch durch 255 geteilt werden.

4.2.2 Ransac

Die Klasse *Ransac* implementiert den in dem Kapitel «Design» beschriebenen RANSAC-Algorithmus. Als Input kriegt der Algorithmus zwei Parameter:

- Die Toleranzschwelle t hängt von der Punktestreuung und Dichte ab und experimentell wird bestimmt. In meinem Experiment wurde als passendes Wert, 0.02 angegeben/ausgerechnet.

- Eine wahrscheinliche Anzahl, der zu einer Ebene gehörenden Punkten heißt *forseeable_support*. Ist die Anzahl von den Punkten der Punktwolke und die Anzahl von den Ebenen(Wände des Gebäudes) bekannt, so könnte man aus diesen, den Schätzwert berechnen.

Zuerst werden alle für die Berechnung benötigte Werte initialisiert bzw. berechnet. Zum Beispiel sollen die akzeptable Fehler und die Anzahl von Wiederholungen wie folgt berechnet werden:

```
double e = 1 - (double) forseeable_support / (double) numberOfPoints;
double N = Math.round(Math.log(1 - alpha)
                      / Math.log(1 - Math.pow((1 - e), 3)));
```

Listing 4.3: Anzahl von Wiederholungen und die akzeptable Fehler

Bei jedem Durchlauf wählt RANSAC zufällig drei Punkte aus, um eine Ebene zu bilden. Bei einem neuen Durchlauf sollen diese Punkte nicht mehr vorkommen. Dafür verwende ich den folgenden Code(Listing 4.4):

```
Point[] sample = new Point[3];
boolean[] checked = new boolean[numberOfPoints];
int r1, r2, r3;
do      {
```

```
        r1 = (int) (Math.random() * numberOfPoints);
        r2 = (int) (Math.random() * numberOfPoints);
        r3 = (int) (Math.random() * numberOfPoints);
        } while (r1 == r2 || r1 == r3 || r2 == r3 || checked[r1]
                || checked[r2] || checked[r3]);
sample[0] = points.get(r1);
sample[1] = points.get(r2);
sample[2] = points.get(r3);
checked[r1] = true;
checked[r2] = true;
checked[r3] = true;
```

Listing 4.4: Geprüfte Punkte entfernen

Im nächsten Schritt werden die für jeden Punkt, Abstände bis zur Ebene berechnet. Die Funktion *checkPoint(int pos, List<Point> points)* gibt diesen Abstand für den Punkt am Platz *pos* der Liste *points* zurück. Es wird hier die Funktion *computeDistance()* der Klasse *Plane* aus CGL benutzt. Alle Abstände, die kleiner oder gleich als die Toleranzschwelle *t* sind, werden in der Liste *distances* gespeichert.

Zuletzt wird geprüft, ob die neue Ebene mehr Punkte, als die vorher gefundene Ebene besitzt. Wenn das der Fall ist, Wird die alte Ebene durch neue ersetzt(Listing 4.5).

```
if (distances.size() > bestSupport) {
        bestSupport = distances.size();
        bestPlane = this.plane;
}
```

Listing 4.5: Eine bessere Ebene speichern

4.2.3 MeshExtractor

Die Klasse *MeshExtractor* sucht einen Rechteck in einer Menge von Punkten einer Ebene. Das Grundprinzip war im Kapitel «Design» erläutert. Die Klasse hat nur drei Methoden:

- *convert()* - die Methode transformiert alle Punkte der Ebene in das Ebenen-Koordinatensystem(Listing 4.6).

```
R = null;
RT = null;
ebene[i] = new PointCloud();
Plane pl = UrbanGUI.planes.get(i);
R = VectorMatrixFactory.createCoordinateFrameZ(pl.getNormal());
```

```
RT = R.getTransposed();
IPointCloud temp = UrbanGUI.points.get(i);
for (int j = 0; j < temp.getNumberOfPoints(); j++) {
        IVector3 sub = temp.getPoint(j).getPosition()
                .subtract(pl.getPoint());
        IVector3 pos = RT.multiply(sub);
        Point point = new Point(pos, // Position
                temp.getPoint(j).getColor(), // Color
                temp.getPoint(j).getNormal() // Norm);
        ebene[i].addPoint(point);
}
```

Listing 4.6: Punkte in das Ebenen-Koordinatensystem transformieren

Die Methode *VectorMatrixFactory.createCoordinateFrameZ()* aus CGL-Projekt erstellt das lokale x-y-Koordinatensystem für die Ebene dar. Die Punkte werden transformiert nach dem Verfahren, das im Kapitel «Design» beschrieben ist.

- *findMeshPoint()* - die Methode durchsucht die Menge, von zu einer Ebene gehörenden Punkten nach vier Eckpunkte ab, die einen Rechteck bilden. Wie diese Punkte definiert sind, habe ich im Kapitel «Design» beschrieben. Um diese vier Punkte zu definieren, brauche ich vier Zahlen: den kleinsten und den größten x- und y-Wert. Ich definiere diese wie folgt:

```
double x1 = Double.MAX_VALUE;
double x2 = Double.MIN_VALUE;
double y1 = Double.MAX_VALUE;
double y2 = Double.MIN_VALUE;
```

Listing 4.7: x und y mit größtem und kleinstem double-Wert definieren

Danach vergleiche ich jeden Punkt mit diesen Werten. Falls der x-Wert kleiner, als der vorherige kleinste x-Wert ist, ersetze ich diesen. Dasselbe mach ich mit dem größten Wert und mit dem y-Wert (Listing 4.8).

```
for (int i = 0; i < pc.getNumberOfPoints(); i++) {
        if (pc.getPoint(i).getPosition().get(0) < x1)
                x1 = pc.getPoint(i).getPosition().get(0);
        if (pc.getPoint(i).getPosition().get(0) > x2)
                x2 = pc.getPoint(i).getPosition().get(0);
        if (pc.getPoint(i).getPosition().get(1) < y1)
                y1 = pc.getPoint(i).getPosition().get(1);
        if (pc.getPoint(i).getPosition().get(1) > y2)
                y2 = pc.getPoint(i).getPosition().get(1);
```

```
}
```

Listing 4.8: Die kleinste und größte Werte der Menge finden

Jetzt bleibt es nur die passenden Werte in die passenden Punkte einzusetzen.

- *convertBack()* - Die Methode transformiert die gefundenen Eckpunkte zurück, in das Welt-Koordinatensystem(Listing 4.9).

```
R = null;
Plane pl = UrbanGUI.planes.get(i);
R = VectorMatrixFactory.createCoordinateFrameZ(pl.getNormal());
IPointCloud pc = new PointCloud();
IPointCloud temp = findMeshPoint(ebene[i]);
for (int j = 0; j < temp.getNumberOfPoints(); j++) {
IVector3 pos = R.multiply(temp.getPoint(j).getPosition());
pos = pos.add(pl.getPoint());
Point point = new Point(pos, // Position
                temp.getPoint(j).getColor(), // Color
                temp.getPoint(j).getNormal() // Norm
);
pc.addPoint(point);
}
```

Listing 4.9: Punkte in das Welt-Koordinatensystem transformieren

Hier läuft der Prozess rückgängig, welches im Kapitel «Design» bereits beschrieben wurde. Die vier Eckpunkte werden in das Welt-Koordinatensystem transformiert und an die Klasse *UrbanGUI* weitergeleitet, damit daraus eine Mesh erzeugt werden kann.

4.2.4 UrbanGUI

Die Klasse *UrbanGUI* hat zwei Funktionalitäten. Erstens: sie realisiert die GUI, mit verschiedenen Eingabefelder und Tasten. Zweitens: sie sammelt die Funktionalität von allen anderen Klassen zusammen.

Für die Realisierung von grafischen Oberflächen benutze ich die Java-Swing Komponente. Die Abbildung 4.3 zeigt die grafische Oberfläche mit einer geladenen .ply-Datei.

GUI verfügt über die folgenden Eingabefelder:

- *Times to repeat* - hier kann man angeben, wie viele Ebenen der Algorithmus finden soll
- *Forseeable support* - vermutliche Anzahl von Punkten, die zu einer Ebene gehören
- *Threshold t* - Abstand zwischen der gewählten Ebene und anderen Punkten.

Die folgende Tasten sind vorhanden:

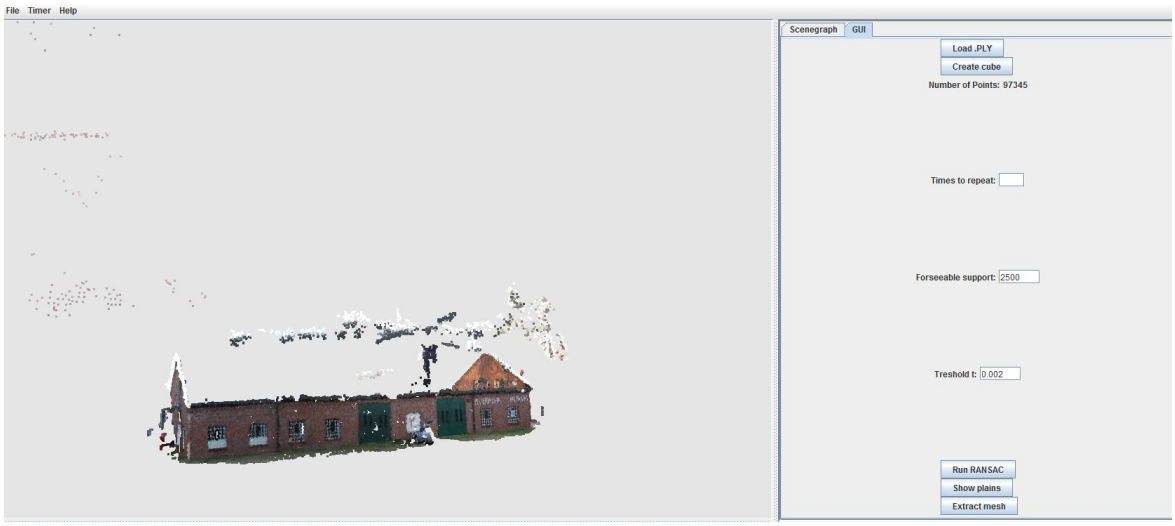

Abbildung 4.3: GUI

- *Load .PLY* - man wählt hier die .ply-Datei zu laden

- *Create Cube* - erstellt ein Testwürfel und konvertiert den in eine Punktwolke

- *Run RANSAC* - startet den RANSAC-Algorithmus mit gewählten Parametern

- *Show plains* - zeigt die gefundene Ebenen an

- *Extract mesh* - konvertiert die gefundene Ebenen in ein 3-D Netz

Die Klasse verfügt über die folgenden internen Methoden:

- *doMesh(IPointcloud pc)* - erstellt einen Rechteck (Dreiecksnetz) aus der Punktwolke *pc*. Die Punktwolke besitzt vier Eckpunkte.

```
ITriangleMesh tm = new TriangleMesh();
int a = tm.addVertex(new Vertex(pc.getPoint(0).getPosition()));
int b = tm.addVertex(new Vertex(pc.getPoint(1).getPosition()));
int c = tm.addVertex(new Vertex(pc.getPoint(2).getPosition()));
int d = tm.addVertex(new Vertex(pc.getPoint(3).getPosition()));
tm.addTriangle(new Triangle(a, b, c));
tm.addTriangle(new Triangle(b, c, d));
tm.computeTriangleNormals();
tm.computeVertexNormals();
meshes.add(tm);
```

Listing 4.10: Rechteck aus vier Punkten machen

- *snap()* - die Methode macht *snapping*. Sie durchsucht die Punkte von einem 3-D Netz und gruppiert diese. Jede Gruppe besitzt die Punkte der Ebenen, die zusammen eine Ecke bilden. Da

diese Punkte nach aneinander liegen, haben sie den kleinsten Abstand zwischen den Punkten. Um solche Gruppen zu identifizieren, sortiere ich die Liste mit Punkten nach dem Abstand, und nehme die ersten Punkte in der Liste, die zu einer Ecke gehören.

```
for (Point p : toSnap) {
        if (!check[toSnap.indexOf(p)]) {
                Map<Double, Integer> d = new HashMap<Double,
                Integer>();
                List<Double> d2 = new ArrayList<Double>();
                for (int i = 0; i < toSnap.size(); i++) {
                        double l = dist(p, toSnap.get(i));
                        d.put(l, i);
                        d2.add(l);
                }
                java.util.Collections.sort(d2);
                List<Integer> join = new ArrayList<Integer>();
                join.add(d.get(d2.get(0)));
                join.add(d.get(d2.get(1)));
                join.add(d.get(d2.get(2)));
                toJoin.add(join);
                check[d.get(d2.get(0))] = true;
                check[d.get(d2.get(1))] = true;
                check[d.get(d2.get(2))] = true;
        }
}
```

Listing 4.11: Zu einer Ecke gehörende Punkte finden

Danach projiziere ich alle zu einer Ecke gehörende Punkte in einen einzigen Punkt.

4.3 Problematiken

Der ganze Aufwand stellt für mich das Hauptproblem meiner Realisierung dar. Mit einer Punktwolke, die aus kleiner Anzahl von Punkten besteht, arbeitet die Software ziemlich schnell. Die Bearbeitung einer .ply-Datei, die aus ca. 5000 Punkten besteht, dauert ungefähr 2 bis 3 Minuten. Das ist die Zeit, in der der RANSAC-Algorithmus die sechs Ebenen findet.

Wenn man ein Gebäude fotografiert und in Punktwolke konvertiert, wandelt diese es in eine .ply-Dateien mit verschiedenen Größen um. Der Algorithmus selbst schafft es, zehn bis zwanzig Tausend Punkte zu bearbeiten. Er wird aber immer langsamer, wenn es um eine größere Menge von Punkten geht.

Zum Testen habe ich ein Feuerwehrmuseum fotografiert. Nach der Konvertierung besteht die Punktwolke-Datei aus fast 100 000 Punkten. Die Bearbeitung so einer Menge von Datensätzen, dauert mehrere Stunden, bis man eine Ebene gefunden hat.

Die zweite Problemstelle bei der Bearbeitung, ist der Snapping-Algorithmus. Bis zu diesem Zeitpunkt habe ich es geschafft, nur Gebäude mit flachen Dächern zu reproduzieren, denn diese haben die Form eines Würfels. In den Ecken des Würfels treffen sich die drei Punkte. Hat das Gebäude zum Beispiel das Schrägdach, muss der Snapping-Algorithmus so angepasst werden, dass er auch die Ecken mit mehreren Punkten erkennt.

Der letzte Punkt stellt eigentlich kein Problem dar, sondern er gehört zu einer noch nicht realisierten Eigenschaft. Das reproduzierte 3-D Modell sieht zur Zeit noch ziemlich “nackt“ aus. Die Idee ist die Aufnahme von Gebäuden, als Textur für das 3-D Modell zu verwendet. Die Texturierung ist aber nicht die trivialste Aufgabe und noch nicht im Prototyp implementiert.

5 Evaluation

In diesem Kapitel möchte ich mein Projekt testen. Ich nehme die einfachsten Testszenario und prüfe, ob meine Software es erfolgreich schafft.

5.1 Ergebnisse und Diskussion

5.1.1 Kubus

Wie ich schon erwähnte, kann man das einfachste Model eines Gebäudes sich als Kubus vorstellen. Mein erster Test ist also zu prüfen, ob meine Software einen idealen Kubus erkennen und reproduzieren kann.

Mit Hilfe von CGL erstelle ich zuerst einen Kubus als 3-D Model. Danach konvertiere ich dieses 3-D Objekt in eine Punktwolke, die aus 5000 Punkte besteht. Endlich verrausche ich die hinkriegte Punktwolke. Das Ergebnis kann man in der Abbildung 5.1 sehen.

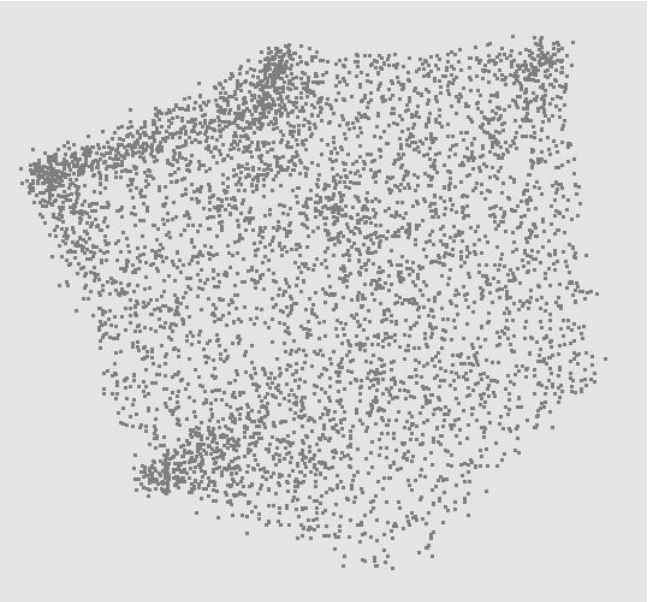

Abbildung 5.1: Kubus: Punktwolke

Die Aufgabe meiner Algorithmen ist es jetzt, den Kubus wieder als 3-D Model darzustellen. Ich starte das Programm mit folgenden Einstellungen: *times to repeat - 6*, da der Kubus sechs Seiten hat, *forseeable support - 800*, 5000 Punkten insgesamt durch sechs Seiten beträgt 800 Punkte pro Seite, *threshold - 0.02*, da ich die Punktwolke mit diesem Wert verrauscht habe.

Der Algorithmus soll die sechs Ebenen finden und aus denen ein 3-D Netz vom Kubus zusammenstellen. Genau das ist hier passiert. Die Abbildung 5.2 zeigt das Ergebnis an.

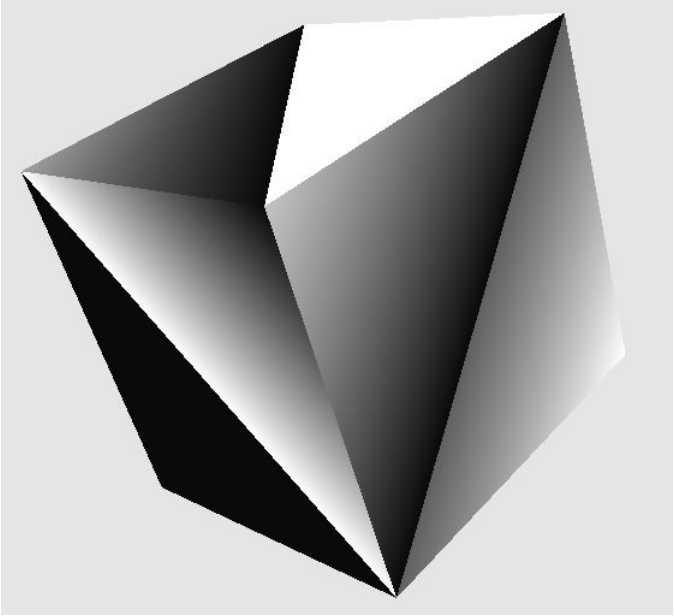

Abbildung 5.2: Kubus: 3-D Netz

5.1.2 Selbst fotografierter Kubus

Das zweite Testobjekt ist ein Kubus, den ich selbst fotografiert und danach mit Hilfe von osm bundler in eine Punktwolke verwandelt habe. Leider habe ich es nicht geschafft, eine Punktwolke-Datei mit einem vollständigen Kubus zu erschaffen. Deswegen habe ich mein Programm mit dem Datensatz eingesetzt, welches sehr präzise war. Diese Punktwolke kann man in der Abbildung 5.3 sehen.

Aus der Abbildung kann man erkennen, dass in diesem Testfall nur maximal 3 sinnvolle Ebenen gefunden werden können, also *times to repeat* wird auf drei gesetzt. Die Werte für zwei andere Parameter habe ich testweise hingekriegt: *forseeable support - 2500*, *threshold - 0.02*.

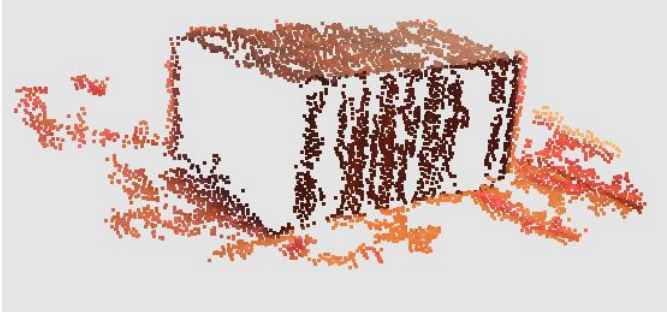

Abbildung 5.3: Fotografierter Kubus: Punktwolke

Da ich keinen vollständigen Kubus hatte, war das Ergebnis auch nicht korrekt. Das Problem lag in meinem Snapping-Algorithmus, welches nur mit Kubus gearbeitet hat. Wenn ich das Snapping abgeschaltet habe, sah das Ergebnis schon besser aus(Abbildungen 5.4 und 5.5).

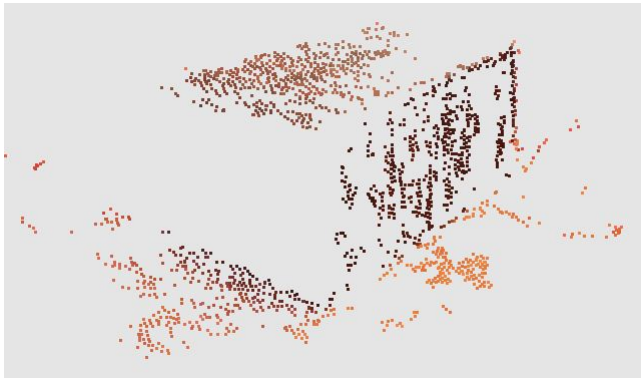

Abbildung 5.4: Fotografierter Kubus: gefundene Ebenen

Die Ebenen wurden von meinem Programm schon richtig erkannt. Dazu kamen aber ach ein Paar Punkte, die eigentlich nicht zur Wand des Kubus gehörten. Die Aufgabe des «future work», ist diesen Fehler zu beseitigen.

Die Dreiecksnetze hat das Programm ohne das Snapping auch richtig extrahiert. Das einzige Problem lag an gedrehten Seite. Wahrscheinlich liegt das an der Transformation von Punkten zwischen der Welt- und Ebenen-Koordinatensystem.

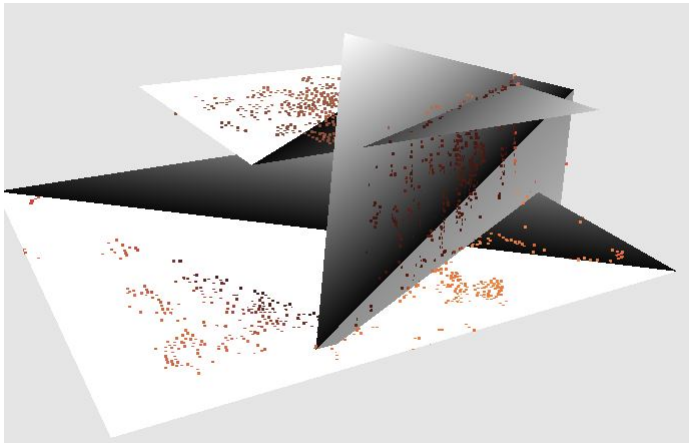

Abbildung 5.5: Fotografierter Kubus: 3-D Darstellung

5.1.3 Gebäude

Den letzten Test habe ich mit selbst fotografiertem Gebäude durchgeführt. Leider hatte ich hier auch den Fall mit der unvollständiger Punktwolke. Deswegen musste ich, genau wie bei letztem Testfall, auch mit Teilobjekt arbeiten und das Snapping abschalten.

Abbildung 5.6: Gebäude: Punktwolke

In der Abbildung 5.6 kann man die Ursprüngliche Punktwolke sehen. Hier wollte ich vier Ebenen erkennen. Die Parameter sind also: *times to repeat - 4*, *forseeable support - 2000*, *threshold - 0.02*.

Auch hier hat der Algorithmus richtig die Ebenen gefunden. Die bestimmte Ebenen kann man in der Abbildung 5.7 sehen.

Abbildung 5.7: Gebäude: gefundene Ebenen

Die Abbildung 5.8 zeigt die extrahierten Dreiecksnetze. Die sind zwar richtig extrahiert, haben aber die falsche Skalierung. Ein passender Snapping-Algorithmus könnte wahrscheinlich dieses Problem lösen.

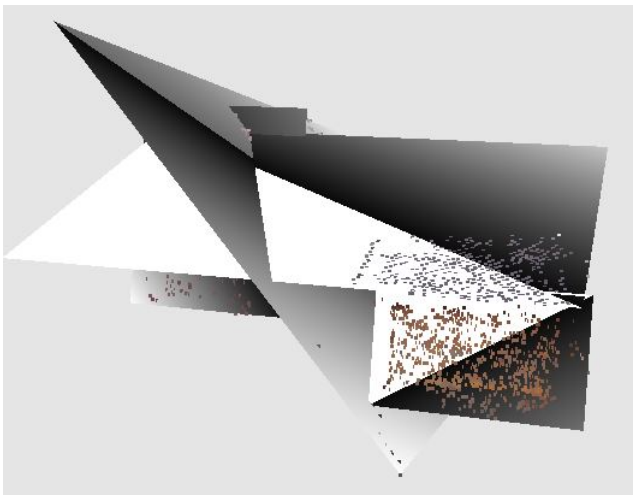

Abbildung 5.8: Gebäude: 3-D Darstellung

Ich habe noch ein Test durchgeführt, mit einer Punktwolke, die fast ein Hundert Punkten hatte. Leider war mein Algorithmus sehr langsam und war auch nach mehreren Stunden nicht fertig.

6 Schluss

In diesem Kapitel fasse ich alles, was ich bis jetzt erreicht habe, zusammen und sage ein Paar Worte dazu, wie man dieses Projekt weiterentwickeln kann.

6.1 Zusammenfassung

Die Idee meiner Arbeit war es einen Prototyp für die 3-D Rekonstruktion von Gebäuden aus Bildren zu entwerfen. Ich habe ein Konzept entwickelt und ein Software geschrieben, um das Ziel zu erfüllen.

Das Konzept betrachtet die Hauptpunkte, die man bei der Realisierung so einer Idee beachten muss. Das ist RANSAC-Algorithmus, der in einer Menge von Punkten ein bestimmtes Muster sucht (in unserem Fall eine Ebene); Snapping-Algorithmus fasst die Punkte eines 3-D Objektes zusammen, die aus irgendwelchem Grund total verstreut sind. Die Arbeit mit den Ebenen ist ein sehr wichtiger Punkt, weil dieser die Wände eines Gebäudes in 3-D Modell, als Ebenen darstellt.

Außerdem habe ich den Begriff Punktwolke erläutert. Da mein Prototyp die Punktwolken(.ply-Dateien) als Input benutzt, ist es sehr wichtig zu wissen, wie diese Datenstruktur aussieht und wie man mit dieser arbeiten kann.

Der von mir entwickelter Prototyp funktioniert sehr gut mit einfachen 3-D Objekten, wie zum Beispiel dem Kubus. Der wird richtig erkannt, und aus einer Punktwolke in ein 3-D Netz rekonstruiert. Es gibt allerdings Komplikationen bei Aufnahme von Gebäuden, die nicht vollständig sind. Der RANSAC-Algorithmus funktioniert sehr gut und die richtige Ebenen werden immer gefunden.

Zusammenfassend würde ich sagen, dass es noch vieles gäbe, was man verbessern könnte. Doch ein Anfang ist jetzt gelegt und der Prototyp wurde als Hauptziel, entworfen.

6.2 Future work

Das Kapitel «Evaluation» hat gezeigt, dass der Snapping-Algorithmus nicht für alle Fälle verwendbar ist. Die Verbesserung diesen Algorithmus ist zunächst das Hauptziel. Zur Zeit

arbeitet er nur mit Ecken eines Objektes, wo sich die drei Punkte treffen. Es muss so geändert werden, dass der Algorithmus die Anzahl von Punkten, für das Schnappen, dynamisch erkennen kann.

RANSAC-Algorithmus kann verbessert werden. Nachdem alle Punkte, die zu einer Ebene gehören, gefunden werden, soll man nach Punkten suchen, die zu einem Rechteck (Wand des Gebäudes) gehören und die übriggebliebenen sollen entfernt werden.

Außerdem kann man mehr Einstellungen für den RANSAC-Algorithmus implementieren. Zurzeit werden alle Ebenen mit dem selben *forseeable support* bearbeitet. Mit einer feineren Einstellung würde man viel bessere Ergebnisse erzielen, und die Leistung somit verbessern.

Die Texturierung von 3-D Modell ist noch nicht implementiert. Das ist ein guter Bereich für die «future work». Man kann vier Aufnahmen von einem Gebäude festhalten und sie als Textur für das 3-D Netz benutzen. Dieser Prozess könnte auch automatisiert werden.

Es kann eine bessere grafische Oberfläche programmiert werden, die es erlaubt, die nicht richtigen Ebenen aus dem Programm zu löschen und sich neue zu suchen. Zurzeit ist es leider nicht möglich.

Literaturverzeichnis

[Baillard und Zisserman 2000] Baillard, C. ; Zisserman, A.: A PLANE-SWEEP STRATEGY FOR THE 3D RECONSTRUCTION OF BUILDINGS FROM MULTIPLE IMAGES. In: *http://www.robots.ox.ac.uk/ vgg/publications-new/Public/2000/Baillard00/baillard00.pdf* (2000)

[osm bundler] bundler osm: https://code.google.com/p/osm-bundler/people/list.

[Furukawa] Furukawa, Yasutaka: Clustering Views for Multi-view Stereo. In: *http://www.di.ens.fr/cmvs/*

[grapp.visigrapp.org/] grapp.visigrapp.org/:

[http://conf.racurs.ru/conf2014/eng/] http://conf.racurs.ru/conf2014/eng/:

[http://www.isvc.net/] http://www.isvc.net/:

[Larsen 2010] Larsen, Christian L.: 3D Reconstruction of Buildings From Images with Automatic Facade Refinement. (2010)

[Otepka u. a. 2013] Otepka, J. ; Ghuffar, S. ; Waldhauser, C. ; Hochreiter, R. ; Pfeifer, N.: Georeferenced Point Clouds: A Survey of Features and Point Cloud Management. In: *ISPRS Int. J. Geo-Inf.l* (2013)

[Suveg und Vosselman 2003] Suveg, Ildiko ; Vosselman, George: Reconstruction of 3D building models from aerial images and maps. In: *Photogrammetry and Remote Sensingl* (2003)

[Szeliski 2010] Szeliski, Richard: Computer Vision: Algorithms and Applications. In: *Springer* (2010)

[Tarsha-Kurdi u. a. 2008] Tarsha-Kurdi, F. ; Landes, T. ; Grussenmeyer, P.: Extended RANSAC algorithm for automatic detection of building roof planes from LIDAR data. In: *The Photogrammetric Journal of Finland, Vol. 21, No. 1* (2008)

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28. Juli 2014 Vitalij Kagadij