Universität Siegen

Naturwissenschaftlich-Technische Fakultät

Department Elektrotechnik und Informatik

Institut für Echtzeitlernsysteme

Masterarbeit

Name: Yuwei Guo

Matr.-Nr.: 961369

Thema: Structure from Motion

Betreuer: Prof. Dr.-Ing. Klaus-Dieter Kuhnert
M.Sc. Simon Hardt

Siegen, 06. Januar 2017

Abstrakt

Structure from Motion (SfM) ist ein Analyseprozess für zweidimensionale Bilder, durch den ein Roboter oder ein System seine lokale Position in einer statischen Szene finden kann, und somit ein Wiederaufbau der Szene gemacht werden kann. Die vorliegende Masterarbeit befasst sich mit den folgenden drei Hauptaufgaben:

1. Basierend auf einer Simulationsdatei (Rosbag) des Roboters *Armor* wird mittels SfM die lokale Position ebendieses Roboters berechnet.

2. Berechne eine dreidimensionale Punktwolke der Szene, in der sich der Roboter befindet.

3. Vergleiche die mittels SfM rekonstruierte Szene mit den Aufnahmen eines Laserscanners.

Die Aufgaben teilen sich wie folgt auf die einzelnen Kapitel auf: Im ersten Kapitel werden die theoretischen Grundlagen von SfM vorgestellt, insbesondere Epipolargeometrie, Korrespondenzpunkte und Triangulation. Im zweiten Kapitel wird die zum Einsatz gekommen Software kurz erläutert. Kapitel drei stellt die Pipeline der *SfMLib* dar, die im Rahmen der Arbeit entwickelt wurde. Dann wird deren Einsatz am Beispiel des *ROS Node SfM Manager* und *SfM Viewer* in Kapitel vier gezeigt. Kapitel fünf vergleicht und bewertet die Ergebnisse von SfM mit denen eines Laserscanners. Zum Schluss wird in Kapitel sechs ein Ausblick gegeben und mögliche Optimierungen für das SfM-Programm werden diskutiert.

Schlüsselwörter : Structure from Motion, SfM, Epipolar Geometry, Robot, ROS, OpenCV, PCL

Inhaltsverzeichnis

1 Theoretische Grundlagen von SfM

In diesem Kapital diskutieren wir die theoretischen Grundlagen für SfM. Normalerweise, wenn die Position der Kamera des Roboters nicht bekannt ist, muss diese als erstes bestimmt werden. Dafür nutzt man korrespondierende Punkte aus zwei oder mehr Bildern, welche im zweiten Unterabschnitt genauer behandelt werden. Am Ende des Kapitels triangulieren wir zwei Sehstrahlen, um einen realen 3D-Punkt zu finden.

1.1 Epipolargeometrie für zwei Bilder

Durch Erfahrung und Vermutungen über die relativen Positionen von Objekten in einem zweidimensionalen Bild kann man relative Tiefeninformationen bezüglich der einzelnen Objekte wahrnehmen. Eine absolute Tiefeninformation aus einem einzelnen Bild auszulesen ist hingegen eine nahezu unlösbare Aufgabe. Stehen jedoch zwei oder mehr Bilder zur Verfügung, können die Schnittpunkte der Sehstrahlen berechnet werden um diese Aufgabe zu lösen. Aus diesem Grund besitzen die meisten Tiere auch zwei (oder mehr) Augen.

In dem vorliegenden Problemfall nutzen wir nicht nur mehrere Bilder, sondern auch die Theorie von Louguet Higgins [1], in der eine räumliche Relation zwischen Pixeln aus zwei relevanten Bildern vorgestellt wird. Wie in Abbildung 1.1 gezeigt, stehen zwei Lochkameras an den Punkten O_L und O_R, welche die beiden Bildebenen I_L und I_R erzeugen. Angenommen die zum realen Objektpunkt X korrespondierenden Punkte X_L und X_R auf den Bildebenen seien bekannt, dann gilt folgende Gleichung, da jeder Punkt durch eine Rotation (R) und eine Translation (T) in einen anderen Punkt überführt werden kann, solange sie sich im gleichen Koordinatensystem befinden.

$$X_R = R * (X_L + T) \tag{1.1}$$

R ist dabei eine Rotationsmatrix, T ist die Translationsmatrix. Jetzt können wir die Gleichung

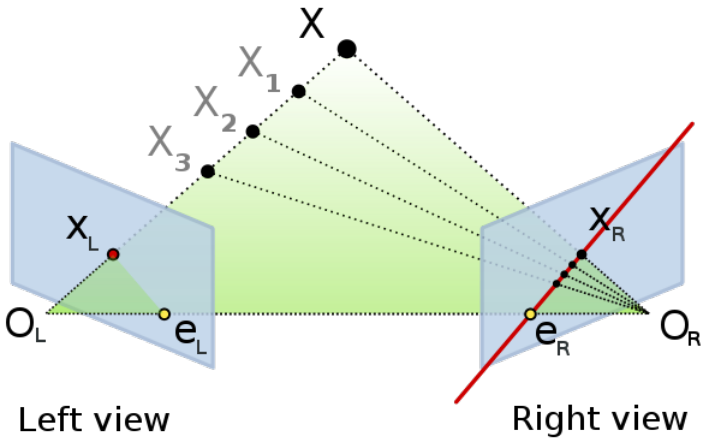

Abbildung 1.1: Epipolargeometrie: zwei Lochkameras stehen an O_L und O_R. Die beiden Sehstrahlen von O_L und O_R durch die Korrespondenzpunkte X_L und X_R treffen sich im realen 3D-Punkt X. Die Schnittpunkte e_L und e_R von der Linie $O_L O_R$ und den Bildebenen bilden die *Epipole*. Die Linien $X_L e_L$ und $X_R e_R$ sind die *Epipolarlinien.[6]*

mit $R * [T]_X * X_L$ multiplizieren und erhalten

$$X_R^T * R * [T]_X * X_L = (R * (X_L + T))^T * R * [T]_X * X_L = 0 = X_R^T * E * X_L. \tag{1.2}$$

Dabei ist $[T]_X$ die Kreuzproduktmatrix mit der Eigenschaft, dass die Kreuzproduktmatrix die Formel gleich null setzt. $E = R * [T]_X$ bildet die sogenannte *essentielle Matrix*. Die essentielle Matrix hat folgende Eigenschaften:

1. Die essentielle Matrix ist eine 3x3-Matrix.
2. $det(E) = 0$. Diese Eigenschaft hilft uns häufig, um die Korrektheit von E zu bestimmen.
3. E ist eine schiefsymmetrische Matrix, die zwei identische singuläre Werte und einen weiteren singulären Wert, der Null beträgt, besitzt.

Wenn die Kameras O_L und O_R schon kalibriert sind (siehe Gleichung 1.3: hier ist C die Kameramatrix, U ist das Bildpixel in homogener Darstellung), können wir noch die folgenden Gleichungen durch die Kalibrationsmatrix erhalten (Gleichungen 1.4, 1.5 und 1.6).

$$U = C * X \tag{1.3}$$

$$X_R^T * E * X_L = U_R^T * C_R^{-1} * E * C_L^{-1} * U_L = U_R * F * U_L = 0 \tag{1.4}$$

$$F = C_R^{-1} * E * C_L^{-1} \tag{1.5}$$

$$E = C_R * F * C_L \tag{1.6}$$

F bildet hier die *Fundamentalmatrix*, welche die Relationsinformation der Korrespondenzpunkte U_L und U_R in beiden Bildern beinhaltet.

Mittels Singulärwertzerlegung (SVD) (Gleichung 1.7) der essentiellen Matrix (in der Annahme, dass diese bereits bekannt ist) können wir die Position der Kamera, wie in [2] beschrieben, bestimmen.

$$E = U * \Sigma * V^T \tag{1.7}$$

$$R = U * W * V^T \quad \text{oder} \quad R = U * W^T * V^T \quad \text{mit} \quad W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.8}$$

$$[T]_X = U * Z * U^T \quad \text{oder} \quad [T]_X = U * Z^T * U^T \quad \text{mit} \quad Z = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{1.9}$$

Da für R und T jeweils zwei Möglichkeiten existieren (Gleichungen 1.8 und 1.9), müssen wir eine den vier möglichen Kombinationen (vgl. Abbildung 1.2) auswählen. Diese Auswahl knüpfen wir an die Bedingung, dass der Großteil der projizierten 3D-Punkte X_i vor den beiden Kameras liegen soll. Sei also

$$n = B(R, T) \tag{1.10}$$

die Funktion, die für eine Kombination aus R und T die Anzahl der Punkte bestimmt, die vor beiden Kameras liegen. Dann suchen wir die Kombination aus R und T, die diese Formel maximiert. Es soll also $max(B(R_i, T_i))$ gelten.

Nun gilt es lediglich noch die essentielle Matrix zu bestimmen. Um diese berechnen zu können, können wir als einzige Ressource auf die Bilder der Kameras zugreifen. Angenommen es seien die korrespondierenden Punkte zweier Bilder bekannt.

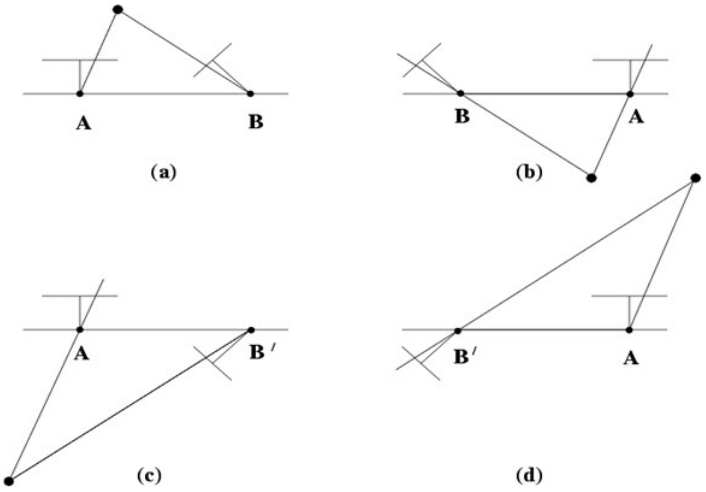

Abbildung 1.2: Vier mögliche Kombinationen von R und T [2].

Dann kann analog zur Gleichung 1.4 mit Hilfe von acht Korrespondenzpunkten die Fundamentalmatrix durch ein lineares Gleichungssystem bestimmt werden:

$$A * x = b. \tag{1.11}$$

Im folgenden Abschnitt soll nun erörtert werden, wie Korrespondenzpunkte gefunden werden können.

1.2 Korrespondenzpunkte

Für den Menschen ist es vergleichsweise einfach korrespondierende Punkte oder Objekten in zwei oder mehr Bildern zu finden. Für den Computer ist dies hingegen eine schwierige Aufgabe ohne triviale Lösung. Zur Findung von Korrespondenzpunkten nutzen wir eine Methode, welche auf der bereits vorgestellten Epipolargeometrie basiert. In Abbildung 1.1 sind die beiden Epipolarpunkte e_L und e_R zu sehen, welche die Schnittpunkte der Linie $O_L O_R$ und den beiden Bildebenen sind. Mit Hilfe dieser Punkte können wir nun korrespondierende Punktepaare finden.

Zuerst wählen wir einen beliebigen Ausgangspunkt X_L auf der Bildebene I_L. Wir wissen, dass der reale Punkte X von dem Strahl, der von O_L durch X_L verläuft, getroffen wird. Es kommen also sämtliche Punkte auf dem Strahl, wie z.B. X_1, X_2, X_3 aus Abbildung 1.1, in Frage, der reale Punkt X zu sein.

Nun betrachten wir die Situation aus Sicht der der Kamera R. Wir gehen davon aus, dass auch hier ein Strahl von O_R durch X existiert, der die Bildebene I_R schneidet. Da die Position von

X jedoch unbekannt ist, können wir X zunächst als Punkt betrachten, der sich auf dem Strahl $O_L X_L$ bewegt. Entsprechend ändert sich der Strahl $O_R X$, wodurch sich auch der Schnittpunkt auf der Bildebene I_R bewegt. Diese Bewegung findet auf der Linie $e_R X_R$ in I_R statt. Das bedeutet, dass der Punkt X_L einen Korrespondenzpunkt X_R auf der Linie $e_R X_R$ hat [1].

Mittels der Fundamentalmatrix (vgl. Gleichung 1.5) können wir die entsprechende Linie $e_R X_R$ einfach bestimmen. Da eine zweidimensionale Linie a*x+b*y+c = 0 durch ein Vektorsystem so wie in Gleichung 1.12 dargestellt werden kann, und die Bildpunkte (Pixeln) durch eine ähnliche Formel (Gleichung 1.4) dargestellt werden können, kann die Linie mit $L = F * U_L$ wie folgt dargestellt werden:

$$[x, y, 1]^T * [a, b, c] = 0 \tag{1.12}$$

$$U_R^T * (F * U_L) = U_R^T * L = 0. \tag{1.13}$$

Die Linie $e_R X_R$ bildet nun eine Menge möglicher Korrespondenzpunkte zu X_L. Nun ist es eine vergleichsweise einfache Aufgabe, einen geeigneten Punkt X_R in dieser Punktmenge zu finden, der zu X_L einen minimale Matching-Fehler hat. In diesem Fall kann man eine numerische Beschreibung der Bildpunktes bilden, z.B. die Umgebung des Punktes, die Luminanz, die Farbe etc. Nehmen wir an, dass eine solche Darstellungsfunktion $b = D(X)$ existiert, eine Beschreibungsfunktion, dann gilt es

$$min(|D(X_L) - D(X_{Ri})|) \tag{1.14}$$

zu bestimmen.

Daraus ergibt sich nun leider ein Henne-Ei-Problem. Um die Fundamentalmatrix zu bestimmen benötigen wir die Korrespondenzpunkte. Gleichzeitig brauchen wir für die Korrespondenzpunkte die Fundamentalmatrix. Epipolargeometrie bieten uns nur eine Methode die möglichen Punktepaare zu finden, um passende Punkte über den relativen Fehler zu bestimmen. Es existieren jedoch noch alternative Lösungen wie z.B. der Harris-Corner-Detector oder anderen Feature-Detektoren. Mit den so gefundenen Punkten kann die Fundamentalmatrix bestimmt werden um dann mittels Epipolargeometrie weitere Korrespondenzpunkte zu bestimmen.

[1] Vorausgesetzt der Punkt X wird von der Kamera R erfasst.

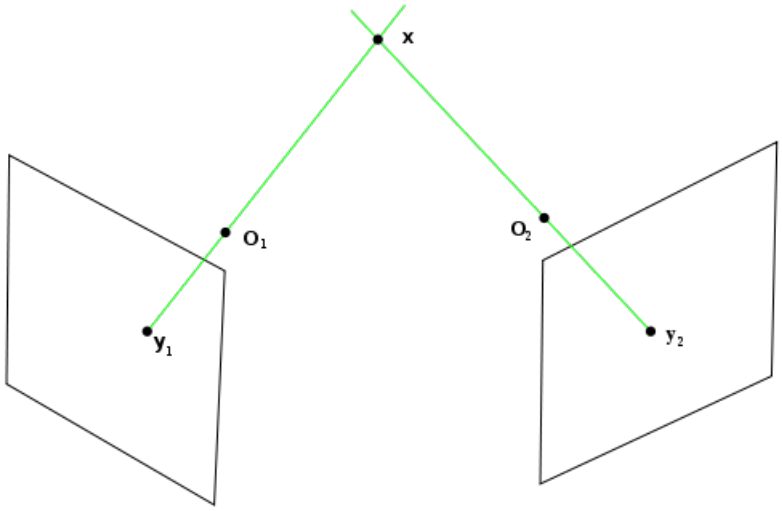

Abbildung 1.3: Ideale Triangulationssituation. Zwei Sehstrahlen treffen sich in der dreidimensionalen Szene an einem Punkt[7].

1.3 Triangulation

Wenn die Position der Kamera und die Korrespondenzpunkte bekannt sind, gilt es nun, daraus eine Punktwolke aus 3D-Punkten zu erstellen, um die Szene zu rekonstruieren. Um diese Problematik zu lösen, wird die Triangulation genutzt. Die Triangulation ist eine uralte Methode, mit der Schifffahrer bereits im 18. Jahrhundert den Abstand zwischen Küste und Schiff messen konnten.

In unserem Fall soll eine Triangulation zwischen den Sehstrahlen $O_L X_L$ und $O_R X_R$ gemacht werden, um den Schnittpunkt X zu finden. Die Abbildung 1.3 zeigt die ideale Triangulation von zwei Sehstrahlen bei Lochkameras in einer dreidimensionalen Szene. Die beiden Sehstrahlen treffen genau auf dem Punkt x aufeinander. In realen Szenarien können die Strahlen jedoch durch Messungenauigkeiten und geometrische Verzerrung windschief sein (siehe Abbildung 1.4).

Um dieses Problem zu lösen verwenden wir den Midpoint-Algorithmus. Dafür wird zunächst der Euklidische Abstand der beiden Strahlen im dreidimensionalen Raum und somit das Segment mit dem geringsten Abstand zwischen den Strahlen berechnet. Der Mittelpunkt des so berechneten Segments wird dann als realer Punkt X angenommen(Gleichungen 1.15, 1.16 und 1.17). Die Linie wird hier durch einen Startpunkt und einen Richtungsvektor dargestellt. Die Parameter a

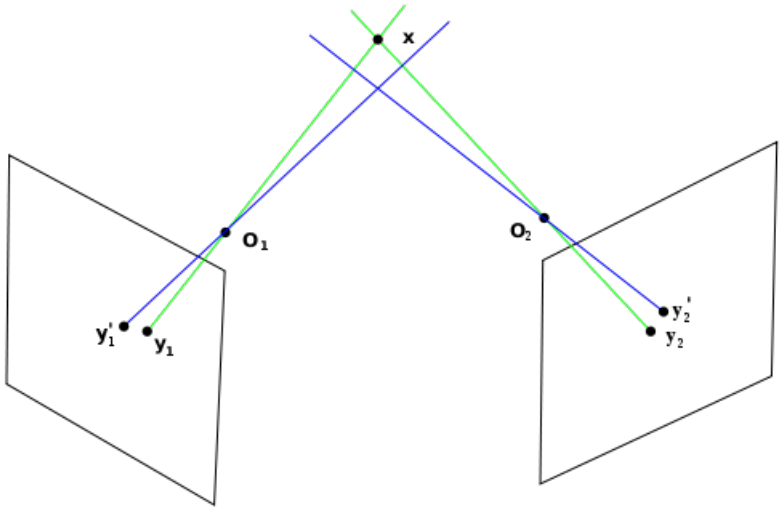

Abbildung 1.4: Reale Triangulationssituation. Die beiden Sehstrahlen (blau) sind windschief[7].

und b entsprechen der relative Länge der Vektoreinheit.

$$D(L_1, L_2)^2 = (L_1 - L_2)^2 \quad \text{mit} \quad L_1(a) = O_1 + a * v_1 \quad , \quad L_2(b) = O_2 + b * v_2 \tag{1.15}$$

$$\begin{cases} \frac{\partial D(L_1,L_2)}{\partial a} = 0 \\ \frac{\partial D(L_1,L_2)}{\partial b} = 0 \end{cases} \tag{1.16}$$

$$X = \frac{L_1(a) + L_2(b)}{2} \tag{1.17}$$

Durch die Nutzung des Startpunkts mit einem Richtungsvektor erhalten wir gleichzeitig eine einfache Lösung für die richtige Kombination von R und T, weil für die korrekten Punkte X_i die Bedingung a, b > 0 erfüllen sollen, also in Blickrichtung der Kameras liegen sollen. Eine Kombination ist also dann geeignet, wenn für den Großteil der X_i die zugehörigen Parameter a, b > 0 gelten.

Eine alternative Methode zur Triangulation bilden die homogenen Koordinaten. Wir können mittels Korrespondenzpunkten U , der Kameramatrix C und der Pose P der Kamera ein lineares Gleichungssystem analog zu Gleichung 1.16 bilden:

$$\begin{cases} U_R * C_R * P_R = X \\ U_L * C_L * P_L = X \end{cases} . \tag{1.18}$$

2 Systemkonfiguration

Das in dieser Arbeit vorgestellte SfM-System ist für Linux entwickelt worden. Hierbei wurde Ubuntu 14.04 als Betriebssystem und C++ als Programmiersprache verwendet, als Entwicklungsumgebung kam der QT-Creator zum Einsatz. Um das Projekt auch auf anderen Plattformen kompilieren zu können wurde CMake als Build-Tool genutzt. In diesem Kapitel werden die zusätzllich zum Einsatz gekommenen Bibliotheken *ROS*, *OpenCV* und *PCL* genauer vorgestellt.

2.1 ROS

ROS (Robot Operating System) ist ein Software-Framework für Roboter, welches auf dem Betriebssystem Linux aufsetzt. ROS wird unter der BSD-Lizenz veröffentlicht und ist somit im Lizenzbereich der Open-Source-Lizenzen einzuordnen. Die Hauptbestandteile von ROS sind Hardwareabstraktion, Gerätetreiber und oft wiederverwendete Algorithmen. ROS fokussiert sich primär auf den Nachrichtenaustausch zwischen Programmen bzw. Programmteilen, Paketverwaltung, Verwaltung von Programmbibliotheken und Betreiben der Software auf mehreren Computern.

Die genutzte ROS Version für das SfM System ist ROS Jade Turtle, welche im Mai 2015 veröffentlicht wurde. Bei der Entwicklung wurden folgende Pakete verwendet:

1. **roscpp**: eine C++ Implementierung für die C++ Programmierung.
2. **std_msg**: Standard ROS Nachrichten Pacet für die Standardnachrichtenstruktur.
3. **sensor_msg**: Dieses Paket definiert Nachrichten für häufig genutzte Sensoren, z. B. Kamera und Laserscanner. Wir nutzen insbesondere Nachrichten für Bilder und Punktwolken.
4. **geometry_msg**: Dieses Paket definiert geometrische Grundelemente in der Robotik wie etwa die Pose im 2D und 3D Raum.
5. **message_filters**: Dieses Paket hilft uns beim synchronen Empfangen von Nachrichten.

2.2 OpenCV

OpenCV ist eine freie Programmierbibliothek mit Algorithmen für Bildverarbeitung und maschinelles Sehen. Sie ist für die Programmiersprachen C, C++ und auch Python entwickelt worden und steht als freie Software unter den Bedingungen der BSD-Lizenz zur Verfügung. Das Wort CV im Namen steht im Englischen für Computer Vision. Die Entwicklung der Bibliothek wurde von Intel initiiert und wird heute hauptsächlich von Willow Garage gepflegt. Die Bibliothek umfasst unter anderem Algorithmen für Gesichtsdetektion, 3D-Funktionalität, Haar-Klassifikatoren, verschiedene sehr schnelle Filter(z. B. Sobel, Canny, Gauß) und Funktionen für die Kamerakalibrierung.

Im Rahmen dieser Masterarbeit ist OpenCV massiv zum Einsatz gekommen um die SfM-Bibliothek zu entwickeln. Die genutzte Version von OpenCV ist 2.4.13. OpenCV hilft uns in dieser Arbeit hauptsächlich um die folgenden Aufgaben zu lösen:

1. Mathematische Objekte und Definition und deren Operationen wie zum Beispiel Bilder, Punkte, Vektoren und Matrizen u.s.w.
2. Feature-Detektor und Methoden zum Vergleich.
3. Relevante Funktionen bei der Kalibrierung von Kameras und zur 3D-Rekonstruktion, z.B. berechnen der Fundamentalmatrix.
4. Debugging, zum Beispiel Darstellung der verarbeiteten Bilder.

2.3 PCL

Die PointCloud Library (PCL) ist eine freie, modulare C++ Bibliothek zur Verarbeitung von Punktwolken. Die Bibliothek beinhaltet zahlreiche Algorithmen die auf n-dimensionalen Punktwolken arbeiten, wie beispielsweise Filterung, Clustering, Segmentierung oder Registrierung. Das Open-Source Projekt wurde erstmals 2010 veröffentlicht und ist unter der BSD-Lizenz lizenziert. Ursprünglich wurde PCL vom selben Entwickler wie ROS ins Leben gerufen, Willow Garage. Mittlerweile wird das Projekt von zahlreichen Universitäten, Forschungseinrichtungen und größeren Firmen vorangetrieben wie dem Fraunhofer Institut, der Uni München, Nvidia und Intel.

Für die Masterarbeit wird PCL zum Anzeigen und Verarbeiten der rekonstruierten Punktwolken genutzt. Für die Darstellung wird vor allem das Paket **pcl_visualizer** genutzt.

3 SfMLib

SfMLib ist das Hauptmodul für das entwickelte SfM System. Die Bibliothek implementiert die Hauptkomponenten des Systeme, unter anderem den *FeatureMatcher*, *CameraPoseComputer*, *Triangulator*, *BundleAdjustor* und das *SfMTool*. Die Struktur der Klassen der SfMLib ist in Abbildung 3.1 dargestellt.

Die Klassen sind stufenweise organisiert. Die oberste Stufe beinhaltet lediglich die Klasse *SfMLibObject*, die Basis Klasse für alle Objekte der Bibliothek.

In der zweiten Stufe definiert die Klasse *HelpFunc* viele Helferfunktionen, wie zum Beispiel die Zerlegung einer Matrix und die Transformation von *cv::Point* zu einer Punktwolke. Die Klasse SfMTool bietet uns diverse komplette SfM-Lösungen für zwei Bilder. Der Benennung der Klassen spiegelt deren Aufgaben wider, zum Beispiel sind in der Klasse *VirtualFeatureMatcher* virtuelle Funktionen zum Vergleich von Merkmalen vorhandenen. Das heißt, dass alle erbenden Klassen die entsprechenden Methoden implementieren müssen.

In der dritten Stufe werden die Ausgangsparameter und Datentypen der Methoden definiert, da wir in jedem Schritt des SfM-Prozesses ein Ergebnis erhalten möchten und die Klassen, die von dieser Stufe geerbt haben ein entsprechendes Ergebnis zurückgeben müssen. Zum Beispiel werden im *SfMLibTriangulator* die Ausgaben als PointCloud und die zugehörigen Operatoren definiert. In dieser Stufe wird die Funktionalität jedoch noch nicht implementiert. Beispielsweise ist die Funktion *DoTriangulation()* im *SfMLibTriangulator* nur als virtuelle Funktion definiert. Erst in der untersten Stufe ist die eigentliche Funktionalität implementiert.

So existieren für den *SfMLibCameraPoseComputer* zwei unterschiedliche Implementierungen, die jeweils auf einem anderen Algorithmus aufbauen. Somit kann ohne großen Aufwand die Implementierung genutzt werden, die im eingesetzten Szenario am sinnvollsten erscheint.

In den folgenden Abschnitten werden die wichtistgen Komponenten der SfMLib (*FeatureMatcher*, *CameraPoseComputer*, *Triangulator*, *BundleAdjustor* und *SfMTool*) im Detail vorgestellt.

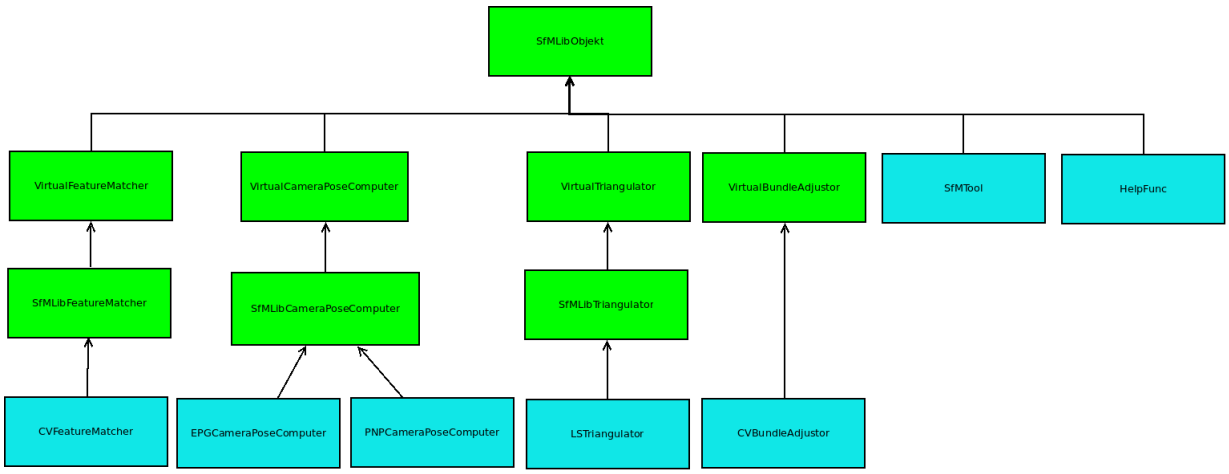

Abbildung 3.1: Klassendiagram von SfMLib

3.1 FeatureMatcher

Bei den theoretischen Grundlagen haben wir festgestellt, dass wir zunächst die Korrespondenzpunkte finden müssen. Die Findung dieser Punkte ist die Aufgabe des FeatureMatchers. Zu dieser Komponente gehören drei wichtige Klassen: *VirtualFeatureMatcher*, *SfMLibFeatureMatcher* und *CVFeatureMatcher*. Die Beziehungen der Klassen zueinander werden im Folgenden erläutert. Als Basisklasse dieser Komponente definiert der VirtualFeatureMatcher die Hauptmethode *DoFeatureMatcher()*. Der SfMLibFeatureMatcher, der von VirtualFeatureMatcher erbt, definiert die zugehörigen Ausgaben. Anschließend erbt die Klasse CVFeatureMatcher von SfMLibFeatureMatcher und implementiert die virtuelle Funktion DoFeatureMatcher().

Zum Finden und Vergleichen von Korrespondenzpunkten kommt die OpenCV-Bibliothek zum Einsatz. Die Pipeline (Abbildung 3.2) des CVFeatureMatcher ist im Folgenden erläutert:

1. Initialisierung des *cv::FeatureDetector*s und *cv::DescriptorExtractor*s. In diesem Schritt können verschiedene Detektoren und Extraktoren definiert werden, indem je nach Bedarf unterschiedliche Parameter an die Konstruktoren übergeben werden können. Zum Beispiel können wir Detektor und Extraktor so konfigurieren, dass sie *Scale-invariant feature transform* (SIFT) einsetzen.
2. Führe eine Merkmalserkennung und Merkmalsextraktion für zwei Bilder mit dem zuvor definierten Detektor-Extraktor-Paar durch.
3. Führen den Merkmalsvergleich mittels *cv::BFMatcher* (Brute-Force-Matcher) durch. Der Brute-Force-Matcher nutzt ein simples Verfahren, das den kleinsten Abstand von einem Featurepunkt zu den Punkten der Menge mit allen anderen Featurepunkten sucht, in dem

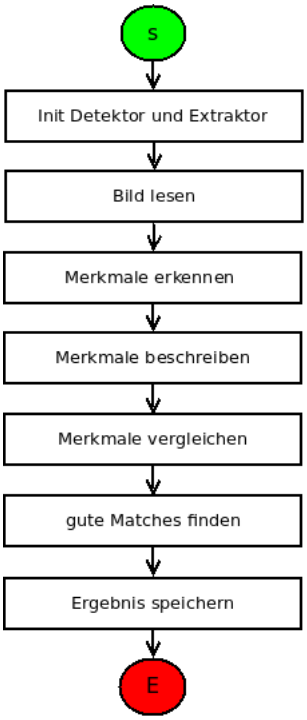

Abbildung 3.2: Die Pipeline im CVFeatureMatcher

er alle Abstände berechnet.

4. Finden der „guten“ Matches. Es wird davon ausgegangen, dass die Abstände der Matches in einer Gauß-Verteilung auftreten. Gute Matches sind dann dadurch definiert, dass diese unterhalb der häufigsten Verteilung aller Abstände liegen (Abbildung 3.3).
5. Speichern der Ergebnisse. Die Ausgabe beinhaltet die Featurepunkte der zwei Bildern und zugehörige gute Matches.

3.2 CameraPoseComputer

Diese Komponente des SfM-Systems bestimmt die Kameraposition. Sie ist in vier Klassen unterteilt: die Basisklasse *VirtualCameraPoseComputer*, die Klasse *SfMLibCameraComputer*, welche die Ausgaben (berechnete Kameraposition und neue Korrespondenzpunkte) definiert und die zwei Implementierungen *EPGCameraPoseComputer* und PNPCameraPoseComputer.

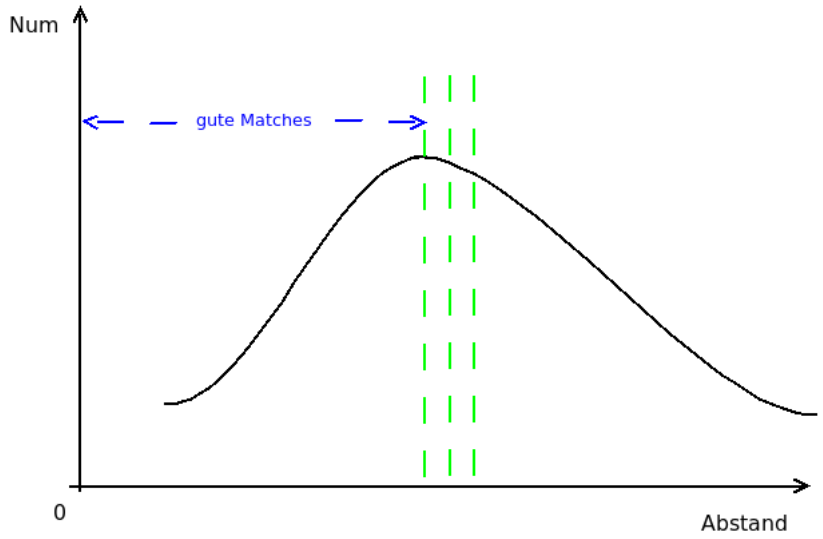

Abbildung 3.3: Die Gaußverteilung der Matches und der Abstand von guten Matches

3.2.1 EPGCameraPoseComputer

Anhand der Theorie der Epipolargeometrie (siehe Abschnitt 1.1) kann der EPGCameraPoseComputer die Position einer zweiten Kamera relativ zur ersten Kamera bestimmen. Die erste Kamera liegt also immer an der Position $[0, 0, 0]$. Der dazugehörige Ablauf ist in Abbildung 3.4 dargestellt und beinhaltet die folgenden Phasen:

1. Lesen der Korrespondenzpunkte aus dem FeatureMatcher.
2. Die Fundamentalmatrix berechnen. Die Funktion *cv::findFundamentalMat* implementiert diesen Vorgang auf Grundlage der Epipolargeometrie.
3. Speichern der neuen Korrespondenzpunkte. Die Funktion *cv::findFundamentalMat* liefert uns unter anderem den Parameter *mask* zurück, welcher die Beziehung zwischen der Fundamentalmatrix und der Punkte beinhaltet. Eine Eintrag 0 in dem Array *mask[i]* bedeutet, dass der Punkt i exakt zur Fundamentalmatrix passt. Wir speichern die am besten passenden Punkte als neuen Korrespondenzpunkte.
4. Die essentielle Matrix nach der Gleichung 1.6 berechnen.
5. Herleitung der SVD-Zerlegung für die essentielle Matrix um die vier Kombinationen von R und T zu bestimmen.
6. Berechnen der möglichen Kombinationen von R_i und T_j.
7. Bestimmen der besten Kombination. Dies geschieht so lange, bis eine passende Kombination gefunden wurde. Hier wird die Klasse *SfMObjectTriangulator* verwendet, um die dreidimensionalen Punkte zu berechnen. Anschließend werden mittels der Funktion

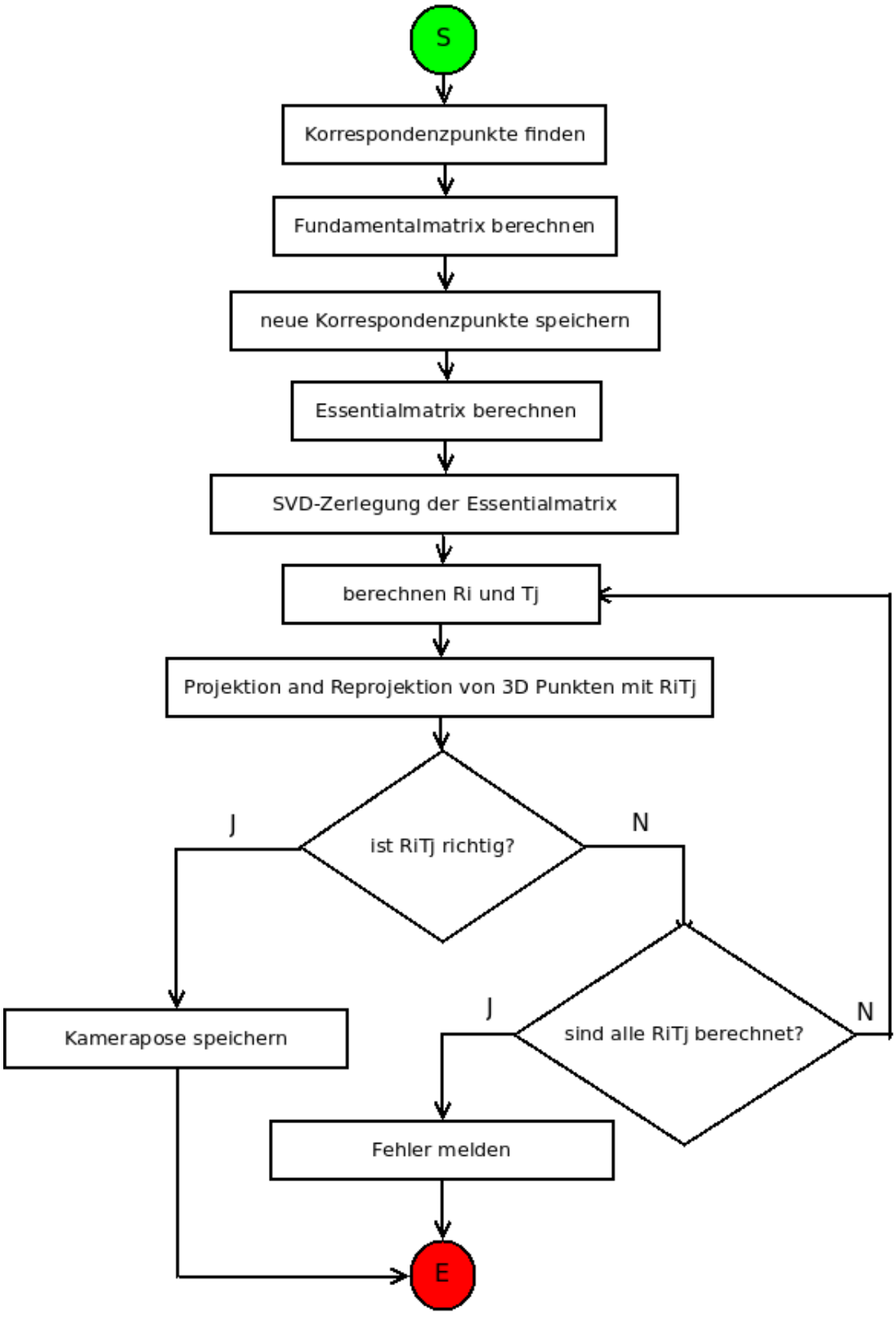

Abbildung 3.4: *Pipeline vom EPGCameraPoseComputer*

cv::perspectiveTransform die 3D-Punkte projiziert. Wenn diese Punkte negative Werte in der Z-Achse haben, sind dies „schlechte“ Punkte. Die Anzahl der schlechten Punkte wird dann addiert und gibt Auskunft über die Güte der Kombination. Außerdem wird die Korrektheit auch noch durch die relativen Fehler der Triangulation bestimmt.

8. Rückgabe des Ergebnisses:

   - Wenn eine gute Kombination gefunden wurde, wird die zugehörige Pose gespeichert.
   - Sonst konnte die Pose nicht korrekt bestimmt werden. Fehlerbehandlung.

3.2.2 PNPCameraPoseComputer

Der *PNPCameraPoseComputer* nutzt die Funktion *cv::solvePnPRansac* um die Position der zweiten Kamera relativ zur ersten Kamera zu bestimmen.

Bei unserem SfM-System wird diesen Methoden immer für den Schritt, der nach dem ersten Rechenschritt erfolgt, genutzt. Das heißt, dass wir den EPGCameraPoseComputer für den ersten Schritt verwenden und ab dann den PNPCameraPoseComputer für alle folgenden Schritte. Der Grund ist, dass der Abstand von zwei Kameras, welcher durch EPGCameraPoseComputer ausgerechnet wird, immer 1 beträgt. Dies ist es für den ersten Schritt egal, da wir den ersten Schritt als eine Einheit definieren können. Aber für jeden weiteren Schritt kann die Entfernung nicht einheitlich definiert werden, deswegen brauchen wir einen anderen Lösungsweg.

Mit den realen dreidimensionalen Punkten und den entsprechenden 2D-Punkten kann die Funktion *cv::solvePnPRansac* die Pose der Kamera mit minimalem Fehler bestimmen (Abbildung 3.5). Für die PNP-Methode werden die gleichen 2D-Korrespondenzpunkte für $Bild[n, n+1]$ und $Bild[n-1, n]$ und die entsprechenden 3D Punkte als $Punktwolke[n]$ benötigt.

Der in Abbildung 3.6 gezeigte Ablauf der PNP-Methode besteht aus den folgenden Schritten:

1. Einlesen der Punktwolke und der Korrespondenzpunkte.
2. Finden der korrespondierenden 2D- und 3D-Punkte. Diese Aufgabe wird durch die Funktion *PNPCameraPoseComputer::FindSame2D3DPoints* umgesetzt.
3. Die Funktion *cv::solvePnPRansac* ausführen. Hiermit wird eine neue Kameraposition berechnet.
4. Beurteilung der Korrektheit der neu berechneten Kameraposition. Die Funktion *cv::projectPoints* kann die 3D Punkte mit der Pose-Matrix der Kamera zurück auf die Bildebene projizieren. Ein relativer Fehler zwischen den ursprünglichen 2D-Punkten und den projizierten

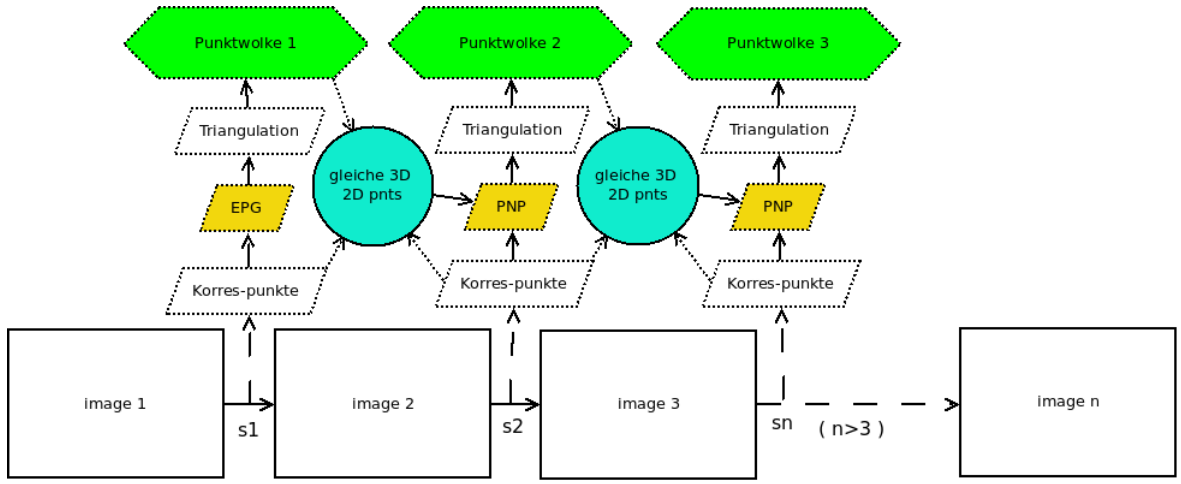

Abbildung 3.5: Beziehung zwischen EPG-und PNPCameraPoseComputer

Punkten wird berechnet. Die Kameraposition mit dem geringsten Fehler wird als korrekte Lösung gespeichert, es sei denn, die Fehler sind zu groß.

3.3 Triangulator

Diese Klasse berechnet aus der Kamerapose und den Korrespondenzpunkten eine dreidimensionale Punktwolke, wobei wir Gleichung 1.16 nutzen.

Der in Abbildung 3.7 gezeigte Ablauf gestaltet sich dabei folgendermaßen:

1. Einlesen der Korrespondenzpunkte und Kamerapose.
2. Berechnen der Triangulation für jeden Punkt.
3. Projizieren des 3D-Punktes zurück auf das Bild, anschließend Berechnung des relativen Fehlers.
4. Wenn der Fehler nicht akzeptiert werden kann, wird der Punkt als „schlechter“ Punkt verworfen.
5. Wenn die Triangulation für alle Punkte berechnet wurde, wird der Durchschnittsfehler ermittelt.

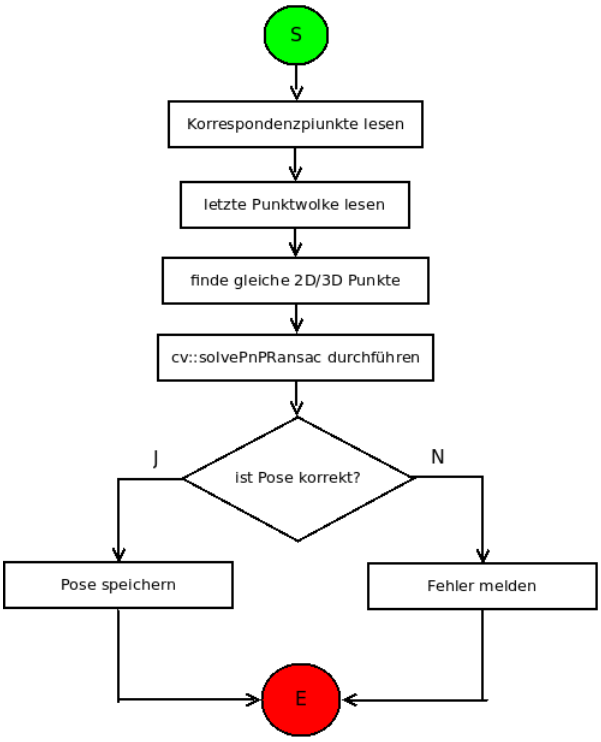

Abbildung 3.6: Die Pipeline von PNPCameraPoseComputer

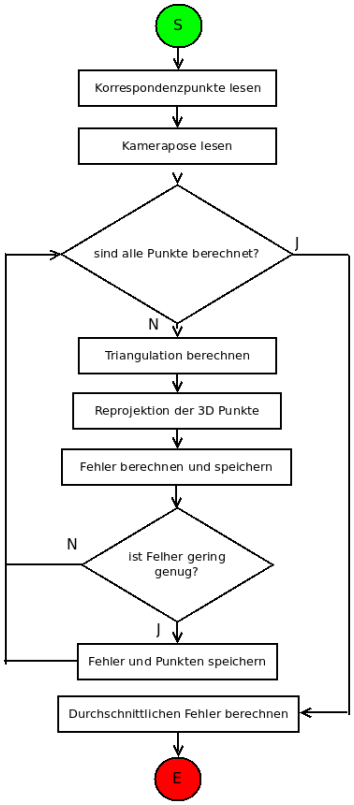

Abbildung 3.7: Pipeline der Triangulation

```
struct CloudPoint{
   cv::Point3d object_pnt; // pnt position
   cv::Vec3b pnt_color;    // pnt color
   cv::Point2d image_pnt_pre, image_pnt_nxt;   // corresponding points in 2 images
   double error_pre, error_nxt;    // reprojection error in 2 images
};
```

Abbildung 3.8: Struct CloudPoint

3.4 BundleAdjustor

Um eine optimale Lösung für SfM zu finden, spielt das Bundle-Adjustment eine wichtige Rolle. Das Bundle-Adjustment bezieht sich auf das Optimieren der „Sehstrahlenbündel“ einer 3D-Szene, die von mehreren Kameras bzw. von einer Kamera aus mehreren Perspektiven aufgenommen wurde. Beim Bundle-Adjustment können gleichzeitig die Positionen der Punkte im 3D-Raum, die Positionen und Orientierungen der beobachtenden Kameras sowie deren internen Kalibrierungsparameter derart an die Messbilder angepasst werden, dass verbleibende Fehler (z. B. durch Bildverzerrungen, Messfehler der Auswertung) möglichst optimal auf alle Beobachtungen verteilt werden.

Die Klasse BundleAdjustor erfüllt genau die Aufgabe des Bundle-Adjustment. Hierfür wird die Funktion *cv::LevMarqSparse::bundleAdjust* genutzt. Dafür müssen zunächst die Eingabeparameter entsprechend konvertiert werden. Bei uns werden die Punktwolken für jeden Schritt in unterschiedlichen Arrays gespeichert. Zudem haben wir keine Informationen über die Sichtbarkeit der 3D-Punkte bezüglich der Kameras. Diese Beziehung können wir jedoch aus unserer Punktestruktur ableiten, da wir die Beziehung zwischen 3D-Punkten und den zugehörigen Korrespondenzpunkten in den beiden Bildern kennen (Abbildung 3.8).

In der Funktion *CVBundleAdjustor::GetImagepntsVSandPnt3D* werden die wichtigsten Parameter für *cv::LevMarqSparse::bundleAdjust* berechnet. Dies beinhaltet das 3D-Punkte-Array, die zugehörige 2D-Punkte der Bilder und das Sichtbarkeitsarray. Im Folgenden sind die dafür nötigen Schritte erklärt (Abbildung 3.9):

1. Initialisierung eines 3D-Punkte-Arrays *cv3d_pnts* um alle Eingabe- und Ausgabe-3D-Punkte zu speichern.
2. Initialisierung eines 2D-Punkte-Arrays *image_pnts* um alle Eingabe- und Ausgabe-2D-Punkte der Bilder zu speichern.
3. Initialisierung eines Sichtbarkeits-Arrays *view* um die Ausgabesichtbarkeit der Bilder zu

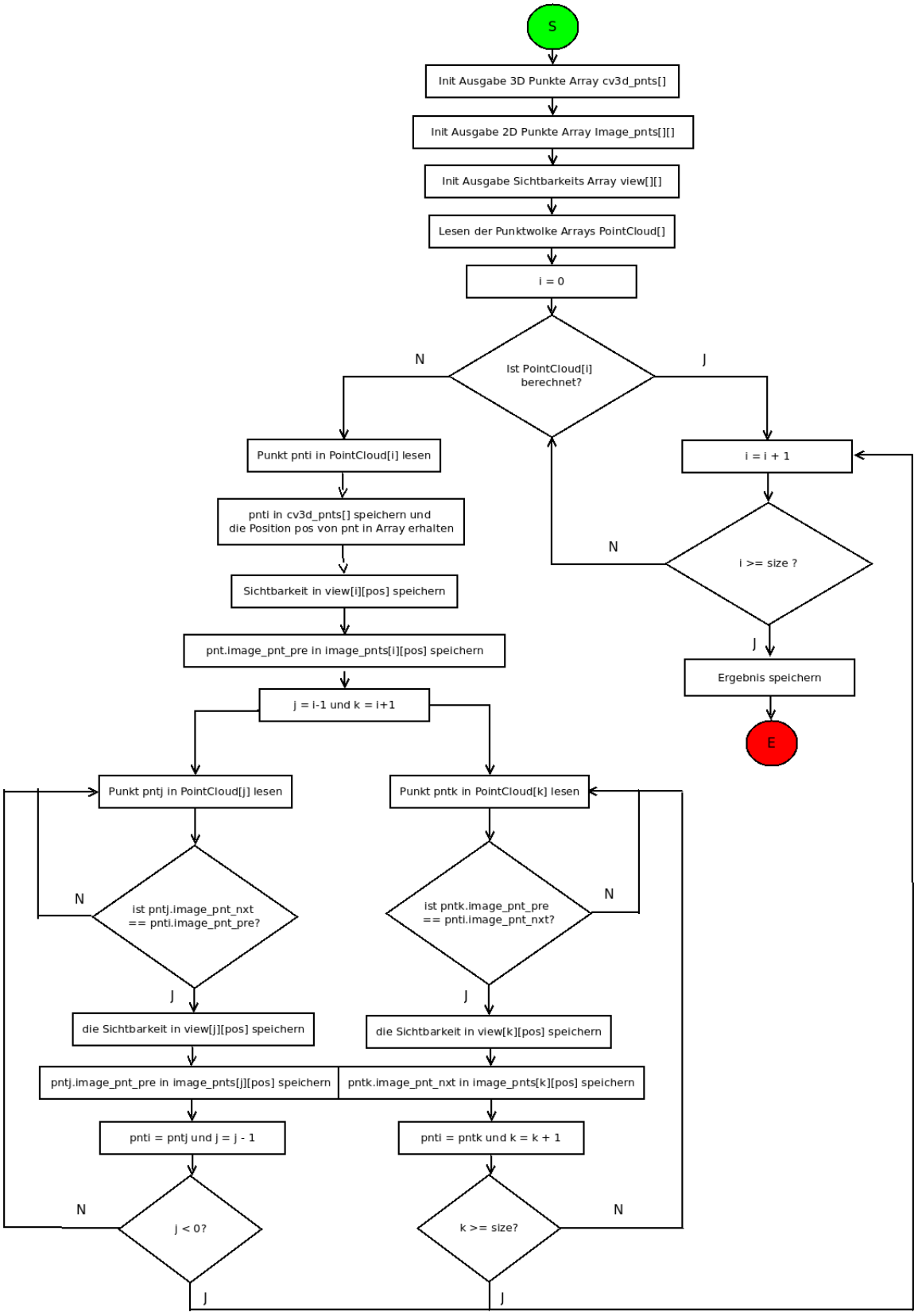

Abbildung 3.9: Pipeline BundleAdjustor

speichern.

4. Einlesen der Punktwolken. Da die Punktwolke in jedem Schritt erzeugt wird, wird diese auch in ein *PointClouds*-Array gespeichert.
5. Initialisierung eines Zeigers für das *PointCloud*-Array, hier i (vgl. Abbildung 3.9).
6. Holen eines Punktes pnti aus dem Array PointClouds[i]. Dieser Punkt wird in *cv3d_pnts* gespeichert, seine Position im Array ist *pos*.
7. Offensichtlich kann der Punkt vom *Bild[i]* gesehen werden(siehe Abbildung 3.5), deshalb wird die Sichtbarkeit *view[i][pos]* mit 1 markiert. Zusätzlich muss der entsprechende Bildpunkt *pnti.image_pnt_pre* des Bildes in *image_pnts[i][pos]* gespeichert werden.
8. Initialisierung von zwei Zeigern $j = i - 1$ und $k = i + 1$ um in der *PointCloud* vor- und zurückgehen zu können.
9. Wir holen uns zwei Punkte *pntj* von *PointCloud[j]* und *PointCloud[k]*.
10. Betrachte die Sichtbarkeit von *pnti* in anderen Bildern: bei dem vorherigen Bild, wenn der Punkt *pnti* sichtbar ist und $pnti.image\_pnt\_pre = pntj.image\_pnt\_nxt$ gilt, dann wird die Position in *View[j][pos]* gespeichert. Gleichermaßen wird bei dem nachfolgenden Bild geprüft, ob $pnti.image\_pnt\_nxt = pntk.image\_pnt\_pre$ gilt uns die Position somit in *View[k][pos]* gespeichert wird.
11. Wenn *pnti* von *View[j]* (*View[k]*) gesehen werden kann, wird der Suchvorgang weiter mit *View[j-1]*(*View[k+1]*) durchgeführt und es wird bei Schritt (9) weiter gemacht. Wenn nicht, kann der Suchvorgang gestoppt werden und es wird zurück zu Schritt (6) gewechselt.
12. Falls alle Punkte in allen *PointClouds[]* fertig bearbeitet wurden, ist die Suche beendet.

Wenn die Ein und Ausgabeparameters für *cv::LevMarqSparse::bundleAdjust* berechnet werden, müssen sie lediglich an diese übergeben und die Funktion muss ausgeführt werden.

3.5 SfMTool

Die Klasse SfMTool bestimmt eine Lösungsmenge der SfM für zwei Bilder. Sie bietet uns drei verschiedene Funktionen namens *SfMTool::SfM42ImagesWithEPG*, *SfMTool::SfM42ImagesWithPNP* und *SfMTool::SfM42ImagesWithPoses*.

Die Pipeline der SfMTool Funktionen ist in Abbildung 3.10 gezeigt und besteht aus den folgenden Schritten:

1. Führe Feature-Matching durch.

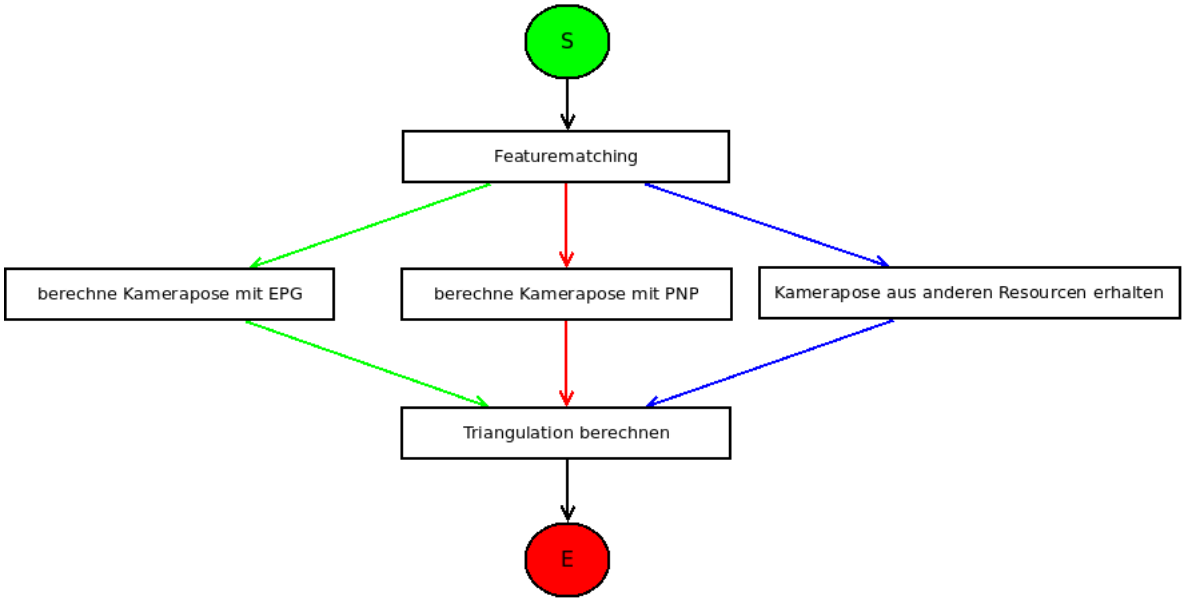

Abbildung 3.10: Pipeline der Klasse SfMTool

2. Je nach gewählter Methode:

    2.1. Kamerapose mittels EPG-Methode berechnen.

    2.2. Kamerapose mittels PNP-Methode berechnen.

    2.3. Kamerapose aus anderen Ressourcen berechnen. Wir erhalten zusätzliche Positionsinformationen via ROS.

3. Durchführen der Triangulation.

Zusätzlich zur Kamerapose und der zugehörigen Punktwolke liefern die Methoden auch einen Triangulationsfehler, mit dem wir die Qualität von SfM beurteilen können. Im nächstem Kapitel wird nun das SfM-System vorgestellt. Das System bietet eine mögliche Lösung des SfM für mehrere Bilder.

4 Das SfM-System

Das SfM-System basiert konzeptuell auf ROS. Es besteht aus den drei ROS-Nodes *SfMManager*, *SfMBA* und *SfMViewer*. Die Hauptaufgabe des Systems besteht darin durch mehrere zweidimensionale Bilder eine dreidimensionale Punktwolke der Szene zu erzeugen und diese gleichzeitig zu optimieren. Dabei liefert der Node *SfMManger* einige Management Werkzeuge bei mehreren Bildern. Der Node *SfMBA* kann Bundle-Adjustment für die durch den *SfMManager* erzeugten Punktwolken durchführen. Der Node *SfMViewer* bietet Werkzeuge zur Anzeige und Vergleichen von verschiedenen Punktwolken. Die Beziehungen der Nodes untereinander ist in Abbildung 4.1 dargestellt.

Der SfMManager empfängt Bilder und Posen des Roboters aus anderen ROS Nodes. Nach dem SfM-Prozess (SfMLib) schickt es die errechnete Punktwolke und die ausgerechneten Posen für jeden Schritt an das ROS-System. Der Node SfMBA sammelt die Punktwolken aus dem SfMManager und führt ein Bundle-Adjustment durch. Dann liefert es eine überarbeitete Punktwolke. Der Node SfMViewer kann diese Punktwolken darstellen, sowohl die Punktwolken aus SfMManager als auch die aus dem SfMBA. Zudem ist er in der Lage, Punktwolken, die durch einen Laserscanner erzeugt wurden, anzuzeigen. Ebenso kann dieser Node die unterschiedlichen Punktwolken miteinander vergleichen. In den folgenden drei Kapiteln sollen die Nodes detaillierter vorgestellt werden.

4.1 SfMManager

Die Aufgabe des SfMManager liegt darin, die Bilder eines Roboters in eine Punktwolke und eine Pose umzuwandeln. Deswegen nutzt der SfMManager die im Rahmen der Arbeit entwickelte Bibliothek SfMLib. Zudem beinhaltet dieser Node zusätzliche Logikelemente um Fehler kontrollieren zu können. Der genaue Ablauf ist in Abbildung 4.2 zu sehen und wird nun genauer erläutert.

1. Zunächst müssen die Bilder des Roboters aus ROS-Nachrichten extrahiert und in Arrays

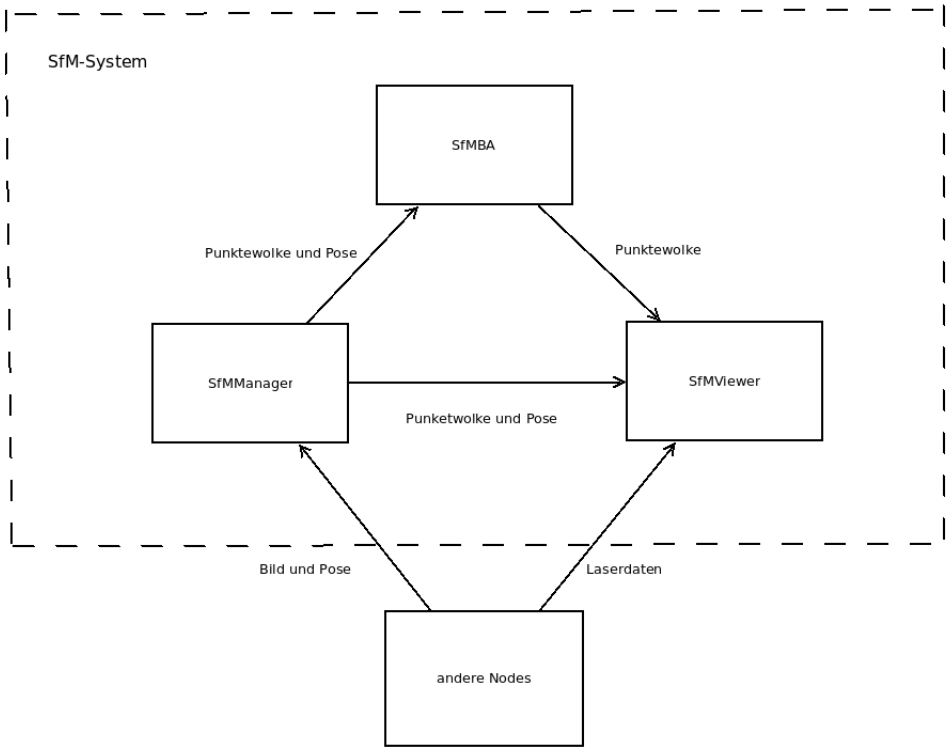

Abbildung 4.1: Die Beziehung von SfM-System Nodes

abgelegt werden. Üblicherweise kann ein einfachert ROS-Subscriber automatisch aktuelle Nachrichten aus einem bestimmten Topic auslesen. Wenn allerdings mehrere Topics synchron empfangen werden sollen, gibt es Probleme der Synchronisierung, da in unserem Fall das Bild und die Pose zur gleichen Zeit empfangen werden sollen. Das ROS-Paket *message_filters::Synchronizer* bietet hierfür eine einfache Lösung, wodurch wir Bild und Pose eines Messzeitpunkts synchronisiert empfangen können. Da wir nur auf begrenzten Speicher zur Zwischenspeicherung von Eingaben und Zwischenergebnissen beschränken können, beachten wir maximal die Aufnahmen, die in einer Sekunde entstanden sind, gleichzeitig. Das entspricht etwa 60 Bildern.

2. Wenn der SfM-Prozess gestartet wird, müssen wir einen Durchlaufzähler nutzen, da es vom Durchlauf abhängig ist, welche SfM-Handlung durchgeführt wird (siehe Abbildung 3.5).
3. Bild und Pose aus dem Array auslesen
4. Durchlaufzähler inkrementieren: $steps = steps + 1$.
5. Hier wird die Unterscheidung gemacht. Falls wir uns im ersten Durchlauf befinden ($steps = 1$), wird SfM-EPG ausgeführt (5.1). Andernfalls wird SfM-PNP genutzt (5.2).

   5.1. Wir führen die SfM-EPG Funktion aus. Die korrekte Ausführung dieses Schrittes wird durch zwei Bedingungen bestimmt: Der Fehler des SfM-Prozesses und die Si-

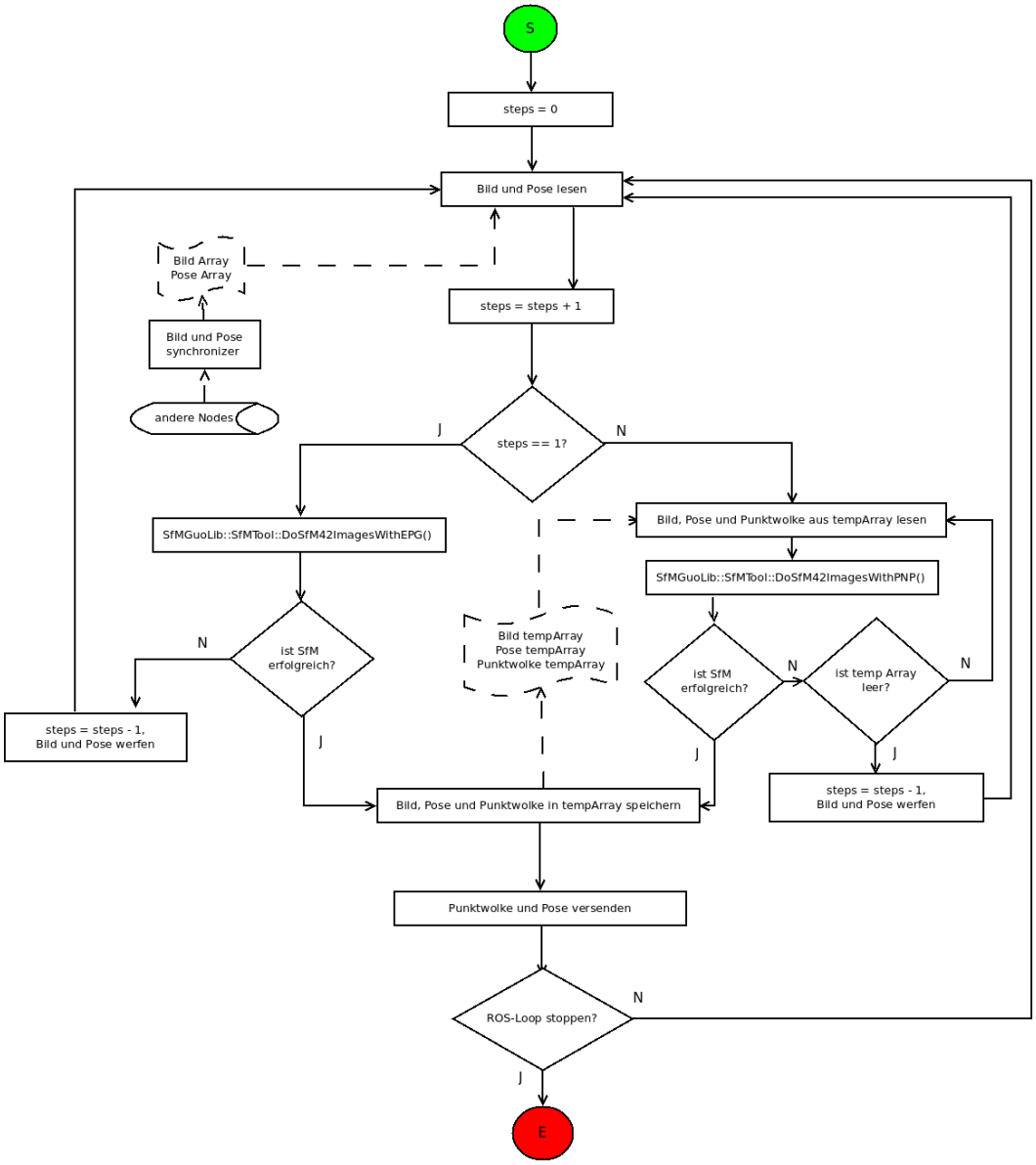

Abbildung 4.2: Pipeline des SfMManagers

tuation der bisherigen SfM-Funktion. Falls dieser Schritt nicht erfolgreich ausgeführt wird, wird der Durchlaufzähler wider dekrementiert und die aktuelle Eingabe wird als ungültig gekennzeichnet.

5.2. Für jeden erfolgten SfM-Durchlauf soll das Bild, die Pose und die Punktwolke in ein temporäres Array gespeichert werden. In diesem Array werden zudem die Ergebnisse der letzten vier bis fünf Durchläufe gespeichert. Sobald das Array voll ist, wird das frühste Ergebnis verworfen. Dieses Array wird als Eingabe für die SfM-PNP-Methode (siehe Abbildung 3.5) genutzt. Wenn der Fehler des aktuelle SfM-Prozesses zu groß wird, können wir keinen SfM-Prozess zwischen dem aktuellen Bild und vorherigem Bild machen. Anschließend wird ein gutes Ergebnis mit einem geringen Fehler ausgewählt. Die Bedingung der Korrektheit ist identische zur SfM-EPG-Methode. Um das System weiter laufen zu lassen, können wir das beste Ergebnis des aktuellen Schritts auswählen.

6. Durch die EPG- oder PNP-Method haben wir die Punktwolke berechnet. Jetzt speichern wir alle benötigten Daten in temporäre Datenstrukturen zur weiteren Nutzung.
7. Das Ergebnis wird nun mittels ROS an andere Nodes gesendet.

4.2 SfMBA

Der ROS-Node SfMBA macht ein Bundle-Adjustment für die empfangenen Punktwolken und Posen aus dem SfMManager-Node. Da sich die Punktmenge mit steigender Programmlaufzeit immer weiter erhöht, wird das Bundle-Adjustment nur bei Bedarf ausgeführt, um den Programmablauf nicht unnötig zu blockieren. Der Ablauf von SfMBA gestaltet sich analog zur Abbildung 4.3 wie folgt:

1. Speichern der Punktwolken und Posen in Arrays.
2. Warten auf ankommendes *DoBA* Signal, welches signalisiert, dass ein Bundle-Adjustment ausgeführt werden soll.
3. Lesen der Punktwolke und der Pose aus den Arrays.
4. Berechnen des Bundle-Adjustment mittels *SfMLib::BundleAdjustor*.
5. Versenden der modifizierten Punktwolke über ROS.
6. Prüfen des aktuellen Status der ROS Schleife. Wenn die Schleife ausfällt, endet das Programm. Wenn nicht, geht es zurück zu (1).

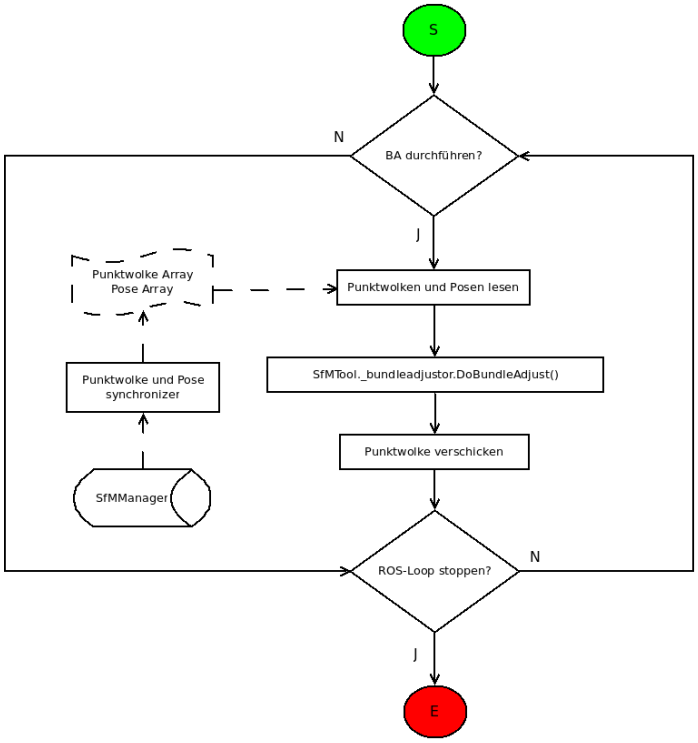

Abbildung 4.3: Pipeline des SfMBA

4.3 SfMViewer

Der ROS-Node SfMViewer ist ein Anzeigetool für Punktwolken. Außerdem ist es in der Lage, die mittels SfM erzeugten Punktwolken mit denen des Laserscanners zu vergleichen.

4.3.1 Anzeigewerkzeug von SfMViewer

Die Anzeige des SfMViewers basiert auf der PCL-Bibliothek [5], genauer auf dem Packet *pcl::visualizer*. Der SfMViewer empfängt zunächst die Punktwolken und Posen aus dem SfMManager, dem SfMBA-Node und anderen ROS-Nodes (z.B. dem Laserscanner). Anschließend werden die Daten in eine *pcl::PointCloud* konvertiert und mittels *pcl::Visualizer* angezeigt.

Der Ablauf des Anzeigetools ist wie in Abbildung 4.4 dargestellt:

1. Zuerst werden synchronisiert die Punktwolke und die Pose aus den Topics von SfMManager oder SfMBA eingelesen.
2. Vor der Weiterverarbeitung der Daten wird ein Boundingbox-Filter ausgeführt. Der fasst Punkte, die innerhalb einer Sphäre liegen, zu einem einzelnen Punkt zusammen. Die rest-

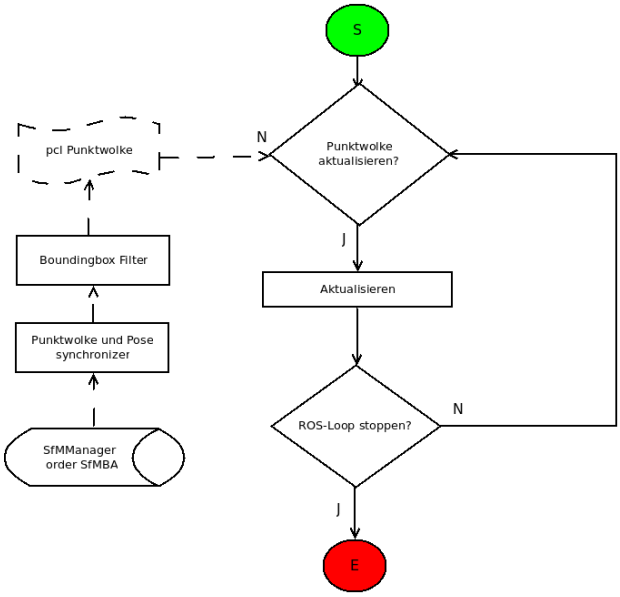

Abbildung 4.4: Pipeline des SfMViewer

lichen Punkte werden verworfen.

3. Die Punkte werden in eine *pcl::PointCloud* konvertiert. Gleichzeitig wird ein Signal an den Visualizer geschickt, der ein Update der Anzeige veranlasst.
4. Die Anzeige reagiert auf das Update-Signal und zeichnet die neue Szene, sobald aktualisierte Punkte zur Verfügung stehen.

Der Node läuft solange, bis er vom Anwender beendet wird.

Der SfMViewer bietet zudem noch viele Möglichkeiten der Interaktion. Beispielsweise kann die Art der Anzeige von Punktwolken variiert oder die Punktwolken durch eine affine Transformation verändert werden.

4.3.2 Punktwolkenvergleich

Wir können im SfMViewer einen Vergleich zwischen unseren SfM-Punktwolken und den Laserscanner-Punktwolken machen, was eine der wichtigsten Funktionalitäten des SfMViewers darstellt. Der Vergleich ist notwendig, da er eine wichtige Methode ist, um die Qualität des SfM-Prozesses zu beurteilen. Die Punktwolke vom Laserscanner wird als eine präzise Referenzdarstellung der Szene genutzt.

Zum Vergleich wird innerhalb einer vergleichsweise kleinen Sphäre geprüft, wie viele Punkte

der SfM-Punktwolke mit einem Punkt der Laserscanner-Punktwolke übereinstimmen. Dieses Verfahren wird auch als Boundingbox-Methode bezeichnet. Die Treffergenauigkeit kann wie in Gleichung 4.1 bestimmt werden. Sie bestimmt, ob ein Punkt P einen Punkt P_n in einer anderen Punktwolke trifft, wobei C ein konstanter Schwellenwert ist und die Funktion norm() den Betrag liefert. Das Trefferverhältnis zwischen getroffenen und nicht getroffenen Punkten bestimmt dann die Genauigkeit der SfM Punktwolke.

$$f(P) = \begin{cases} 1, norm(P - P_n) < C \\ 0, norm(P - P_n) > C \end{cases} \quad n = 1, 2, 3... \tag{4.1}$$

Bevor zwei Punktwolken miteinander verglichen werden können, müssen sie jedoch zunächst in ein gemeinsames Koordinatensystem überführt werden. Zudem muss ein effizienter Algorithmus für den eigentlichen Vergleich gefunden werden.

4.3.2.1 Transformation der SfM-Punktwolke

Im Folgenden stellen wir drei Methoden vor um zwei Punktwolken in ein gemeinsames Koordinatensystem zu transformieren.

Für die erste Methode müssen wir zunächst erneut die Berechnung der Kameraposition in Kapitel 3 betrachten. Dort finden wir den Grund dafür, dass die beiden Punktwolken nicht auf die gleiche Größe skaliert sind (siehe Abbildung 4.5) . Im ersten Schritt des SfM-Prozesses verwenden wir die EPG-Methode, um die Pose der Kamera zu bestimmen. Dabei nutzen wir lediglich eine Pseudoeinheit als Referenz. Diese Einheit korreliert zu keiner Einheit aus der realen Umgebung, weswegen die weiteren Schritte mit der falschen Einheit rechnen. Die erste Lösung besteht also darin, dass wir einen echten Bewegungsabstand innerhalb der EPG-Methode angeben (Gleichung 4.2). Dieser Abstand kann durch die synchronisierten Positionen des Roboters berechnet werden. In unserer Implementierung kann man die Parameter *epg_firstlength* in der Funktion *SfMTool::DoSfM42ImagesWithEPG()* nutzen.

$$[T] = d * [T]_X \tag{4.2}$$

Eine alternative Lösung bietet eine mathematische Methode. Wir gehen davon aus, dass zwischen beiden Punktwolken eine Beziehung besteht, da diese ein Abbild der gleichen Szene sind. Mit den Korrespondenzposen von Laserscanner und SfM können wir ein lineares Gleichungssystem bilden. Durch die Abbildungsmatrix kann ein Punkt in der SfM-Punktwolke in eine Koordinate des Laserscanners abgebildet werden (siehe Gleichungen 4.3 und 4.4). In unserem

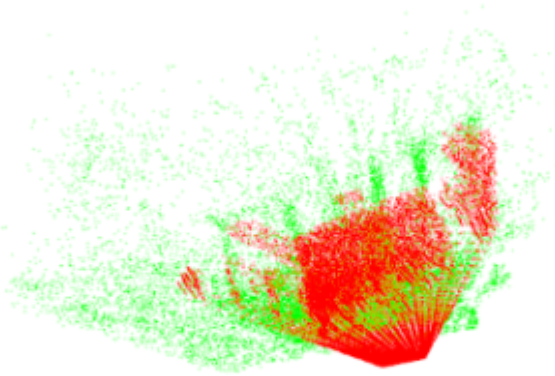

Abbildung 4.5: Die Größen der Punktwolken durch SfM (grün) und den Laserscanner(rot) sind nicht gleich

Fall ist diese Methode jedoch nicht sehr effektiv, da die berechneten Positionen der Kameras zu ungenau sind. Zu Testzwecken und für andere Szenarien kann diese Methode aber mittels *VIEWER::ComputeCompareMatx()* angewendet werden.

$$M * P_{sfm} = P_{laser} \tag{4.3}$$

$$M = [P_{laser}, ...] * [P_{sfm}, ...]^{-1} \tag{4.4}$$

Die dritte Lösung für diese Problematik ist eine Kombination der manuellen und automatischen Methoden. Dabei können wir eine affine Transformation für die SfM-Punktwolke manuell angeben. Diese beinhaltet Verschiebung, Skalierung und Rotation. Dann können automatisch kleinere Anpassungen durchgeführt werden, um die Transformation zu optimieren. Diese Kombination hat sich als effektivste Varianten erwiesen, da bereits nach zwei bis drei manuellen und automatischen Anpassungen eine stabile Transformation mit guter Treffsicherheit erreicht wurde. Das heißt, dass diese Kombination im Normalfall konvergiert. Die Funktionen dafür heißen *VIEWER::AutomaticParameterBasedCompare()* und *VIEWER::AutomaticParameterBasedAngleCompare()*.

4.3.2.2 Effizienter Vergleich

Nehmen wir an, dass es M Punkte in der SfM-Punktwolke und N Punkte in der Laserscanner-Punktwolke gibt. Um einen Treffer für einen Punkt P aus der SfM-Punktwolke zu bestimmen,

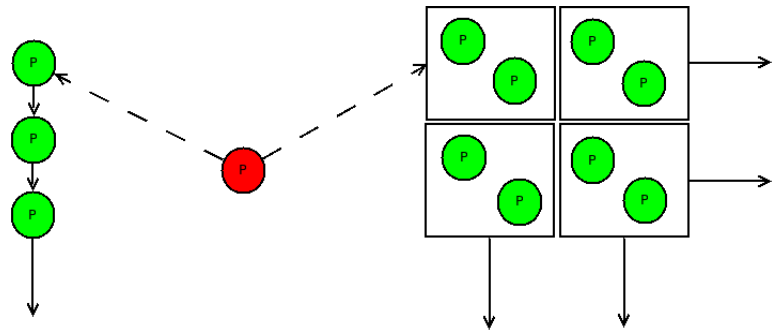

Abbildung 4.6: ***Vom linearen Vergleichen zu blockweisen Vergleichen***

müssen wir $N/2$ Vergleiche machen. Für M Punkte ergibt sich somit ein Rechenaufwand von $\mathcal{O} = 0.5 * (M * N)$. Für große Punktwolken ist der Aufwand damit zu hoch.

Um das Verfahren zu beschleunigen, zerlegen wir die Szene zunächst in mehrere dreidimensionale Blöcke.

Nehmen wir an, dass die Punkte in der Laserscanner-Punktwolke im Zahlenbereich $X * Y * Z$ liegen. Dann legen wir $X * Y * Z$ Blöcke an. In jedem Block werden die Referenzen auf alle darin befindlichen Punkte gespeichert. Und gibt es U Anteile den Blöcken im Benutzten, da das Scenario immer nicht allen Blöcken besitzen kann. Angenommen die Dichte der Punkte ist in alle Blöcken gleich, dann reduzieren wir die Vergleichskosten bestimmen mit $\frac{1}{X*Y*Z*U} * 0.5 * (M * N)$. Die Größe der Blöcke muss dabei abhängig vom Szenario gewählt werden, um einen Kompromiss zwischen dem Overhead für die zusätzliche Speicherung und der Beschleunigung der Vergleiche zu finden.

In unserem Szenario lagen die Punkte der Messung durch den Laserscanner im Bereich x : $(-14.7689, 56.7836)$, y : $(-11.7142, 1.49871)$ und z : $(3.9303, 69.4108)$. Für eine effiziente Berechnung der Zugehörigkeit eines Punktes zu einem Block haben wir jeweils nur den Ganzzahl-Anteil der Koordinaten betrachtet und diesen ins Positive verschoben. In der Praxis können hier auch andere Funktionen für die Umrechnung genutzt werden. Auf diese Weise erhalten wir $71 * 14 * 73 = 72562$ Blöcke für die Referenzen. Und die Nutzrate von Blöcken ist ungefähr 9%.

-	SfM-Punktwolke Num	Laser-Punktwolke Num	Laufzeit
ohne Blöcken	3.519	355.827	$95.4039s$
mit Blöcken	3.455	307.082	$0.01975s$

Tabelle 4.1: Laufzeit-Vergleichen zwischen mit und ohne Blöcken

Durch die Aufteilung in Blöcke erreichten wir in der Praxis eine Beschleunigung um den Faktor 4800 (siehe Tabelle 4.1).

Der zusätzliche Speicheraufwand für die Blöcke beträgt dabei lediglich 3MByte, da wir zu jedem der 710000 Punkte des Laserscanners lediglich eine Referenz von vier Byte gespeichert haben.

5 Evaluation des SfM-System

In diesem Kapitel soll der komplette SfM-Prozess anhand einer Beispielszene evaluiert werden. Dafür wird zunächst die Laufzeitumgebung vorgestellt. Der genutzte Computer basiert auf einer Intel Core i7-3630QM CPU mit 4 Kernen und 8 Threads, einer NVIDIA QUADRO K3000M Grafikkarte und 8 GB Arbeitsspeicher. Auf dem System läuft auf ein 64-bit Ubuntu Linux, Version 14.04. Zum Einsatz kommen zudem OpenCV in Version 2.4.13, PCL in Version 1.3 und ROS Jade Turtle.

Während des Testlaufs hat der SfM-Prozess insgesamt 47 Durchläufe gemacht. Wegen des zu großen Fehlers ist das System zunächst immer abgebrochen. Dies wurde dadurch verursacht, dass der Roboter während der Aufnahme der Testdaten rotiert wurde. Dadurch entsteht ein relativ großer Fehler. Deswegen wurden die Parameter für die Beendigung des Programms für den Testlauf angepasst um auf diese Rotation reagieren zu können.

Im vorliegenden Fall hat das System ungefähr 68.887 3D-Punkte für die SfM Punktwolke berechnet und gleichzeitig wurden ca. 7.203.095 Punkte für die Laserscanner-Punktwolke gesammelt.

In Abbildung 5.1 zeigt der erste Schritt des SfM-Prozesses. Nur wenige Punkte können gesehen werden. In Abbildung 5.2 können wir die Detailweise Information für diesen Schritt finden, die echte Länge des EPG-Schrittes ist 0.102332 und der durchschnitte Fehler von den ausgerechneten Punkten ist 0.564957, da wir schon den Fehler beschrankt, das heißt, dass die schlechtere Ergebnisse ignoriert werden.

Nach fünfen Schritten in Abbildung 5.3 können wir schon eine grobe Struktur der Szene erzeugen. Und in Abbildung 5.4 zeigt das Vergleichen zwischen die SfM-Punktwolke und das Video.

Der SfM-Prozess läuft bis zu einer Ecke und stoppt dort(siehe Abbildung 5.6). Für diesen Schritt sind die Fehler von neunfachen SfM-Prozessen in Abbildung 5.7 gezeigt. Der minimale Fehler beträgt 11.2297, was zu ungenau für den weiteren Schritten. Deshalb stoppt das Programm sofort. Während des SfM-Prozess hat der SfMViewer schon das ganzen Szenario aus Laserscanner gespeichert(Abbildung 5.5).

Auto-Pass-Num	R x	R y	R z	TR
0	-	-	-	0.05889
1	0.01745	0	0	0.11940
2	0.01745	-0.01745	0.01745	0.13433
3	-0.01745	-0.01745	-0.01745	0.13930
4	0	0	0	0.13930

Tabelle 5.1: Vierfache Rotation automatische Anpassung. R: Rotation. TR: Treffenrate. x,y und z sind die Drehwellen

Auto-Pass-Num	PV x	PV y	PV z	GÄ x	GÄ y	GÄ z	TR
0	-	-	-	-	-	-	0.13930
1	0	0	-0.05	0.02	-0.02	-0.02	0.18906
2	-0.05	0	-0.05	0	0.02	0	0.18906
3	-0.05	0	0.05	0	0	0	0.19901
4	0	0	0	0	0	0	0.19901

Tabelle 5.2: Neunfache automatische Anpassung. PV: Positionsverschiebung. GÄ: Größenänderung (Skalierung). TR: Treffenrate. x,y und z sind die Koordinatenrichtungen

Nun vergleichen wir zunächst die manuelle und automatische Anpassungen der SfM- und Laser-Punktwolken. Die SfM-und Laser-Punktwolke ohne Anpassung in X und Z Richtung kann man in Abbildung 5.8 und Abbildung 5.9 finden. Die Treffenrate von nicht anpassenden Punktwolken ist auch nicht befriedigend 5.889%. In dem Szenario reichte eine vierfache Rotation automatische Anpassung aus um eine stabile Trefferrate zu erhalten (siehe Tabelle 5.1). Dabei wurde zusätzlich vier mal die automatisch angepasst(Tabelle 5.2). Durch die automatische Anpassung ist die Trefferrate insgesamt von 5.889% auf über 19.901% gestiegen(siehe Abbildung 5.10 und Abbildung 5.11).

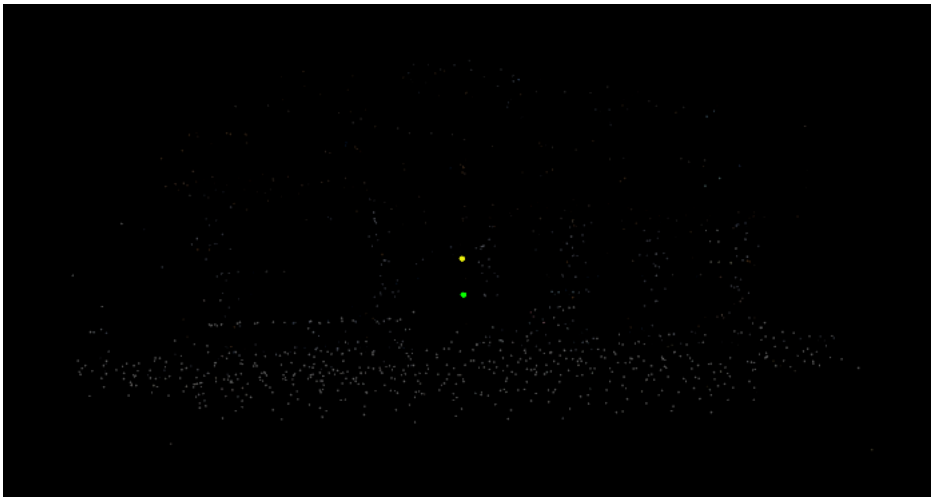

Abbildung 5.1: SfM-Punktwolke für den ersten Prozessdurchlauf (EPG) im SfM-System. Der grüne Punkt entspricht der berechneten Kameraposition, der gelbe Punkt entspricht der Position der Kamera, wie sie vom Roboter angegeben wird

```
testing p_nxt :
[0.9999996286226248, -0.0007451493706885142, -0.0004330208174493019, -0.05579982169482051;
  0.0007460244906575672, 0.9999976727649592, 0.002024330043519373, -0.11485722035963;
  0.0004315113814501309, -0.002024652335864421, 0.9999978572881277, -0.9918135907669775]
98.496659% are in front of camera
98.496659% are in front of camera
Done. (0.518344s)
Triangulator Init seccsessfully
Triangulating...
Done. (3592 points, 0.064233s, mean reproj err for image_nxt = 0.564957
triangulation error is 0.564957
sfm for 2 images is done!(3.285354s)
first step is seccesfull with em_length  : 0.102332
step 1 is completed !
```

Abbildung 5.2: Ergebnis des ersten SfM-Prozessdurchlaufs (EPG) im SfM System mit $epg\_length = 0.102332$. Es entsteht ein relativer Fehler von 0.564957 bei der Reprojektion mittels Triangulation

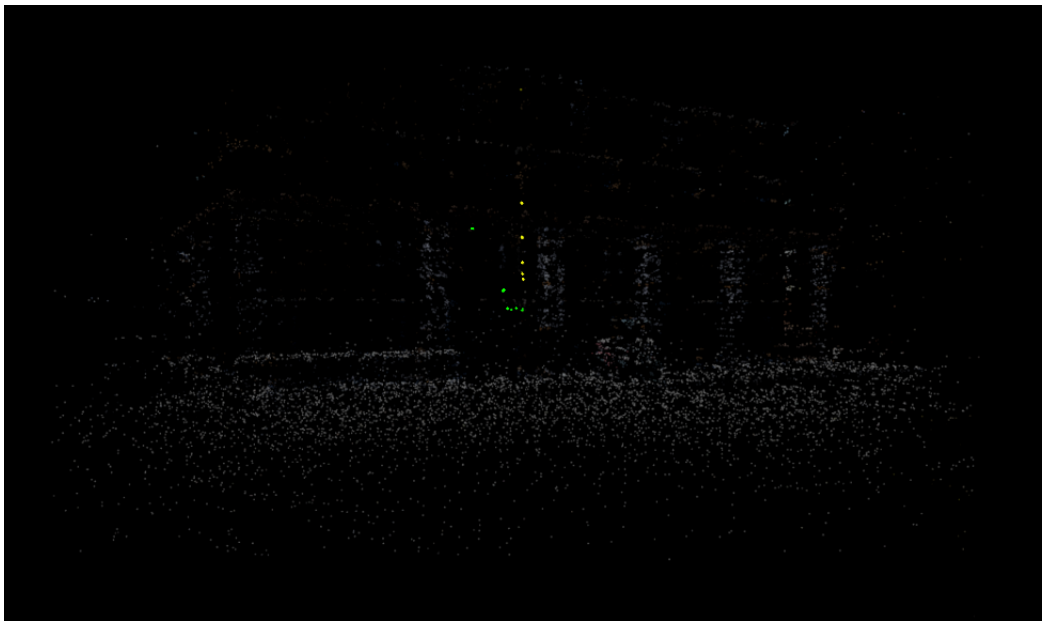

Abbildung 5.3: SfM-Punktwolke nach mehreren Durchläufen. Die Punktwolke ist um einiges dichter als nach im ersten Durchlauf

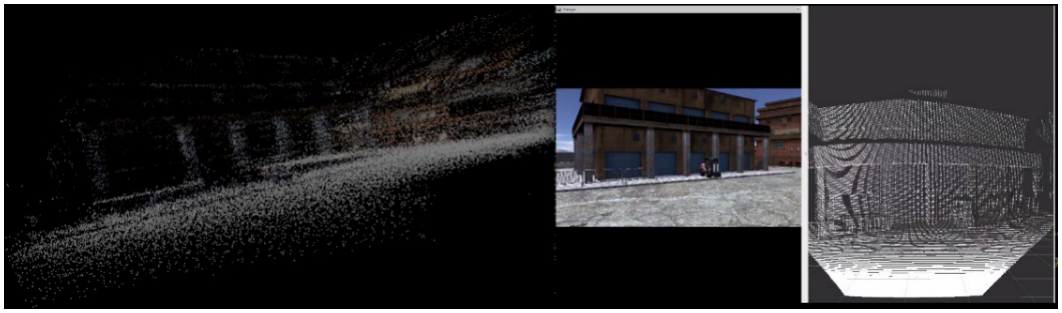

Abbildung 5.4: Vergleich zwischen einer SfM-Punktwolke und einem Video der zugehörigen Szenerie

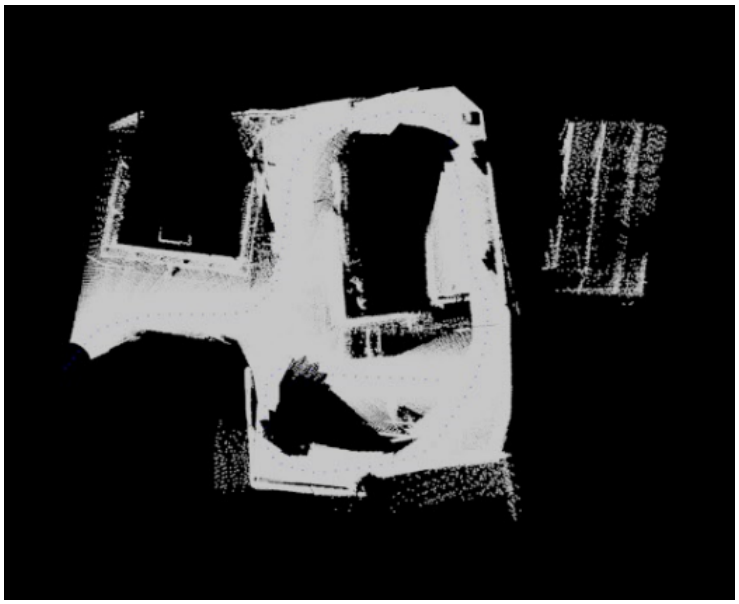

Abbildung 5.5: Vollständige Laser-Punktwolke

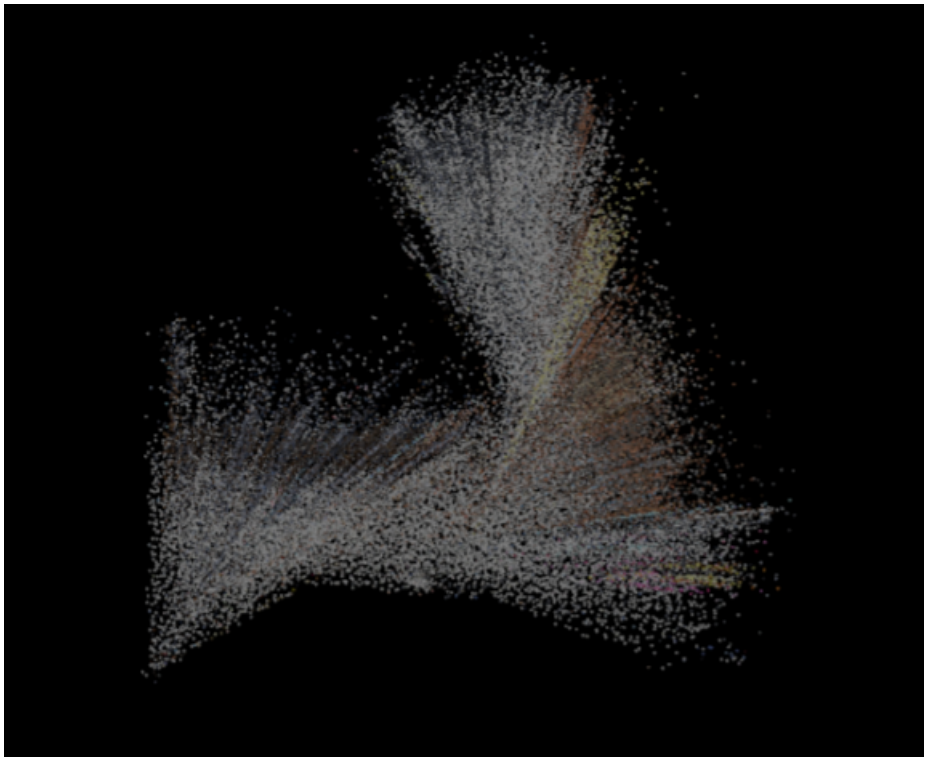

Abbildung 5.6: Der Prozess stoppt an einer Ecke

```
++++++++++++++++++++++++++++++++++++++++++++++++++
 error is happend in SfM PnP step, all results are terriable , they are :
--------------------------------------------------------
| NUM | image | pnt_cloud(size) | tri_error |
| 0 | 3 | 1123 | 67.5594 |
| 1 | 2 | 2221 | 52.5557 |
| 2 | 2 | 3300 | 56.1027 |
| 3 | 1 | 4590 | 29.6638 |
| 4 | 1 | 6165 | 36.2121 |
| 5 | 1 | 7315 | 56.6908 |
| 6 | 0 | 9278 | 11.2297 |
| 7 | 0 | 11395 | 15.5878 |
| 8 | 0 | 13009 | 26.9226 |
yuwei@yuwei-CELSIUS-H920:~/Masterarbeit/rosworkspace_ma$
```

Abbildung 5.7: SfM Prozess stoppt mit zu großem Fehler. Der minimale Fehler beträgt 11.2297

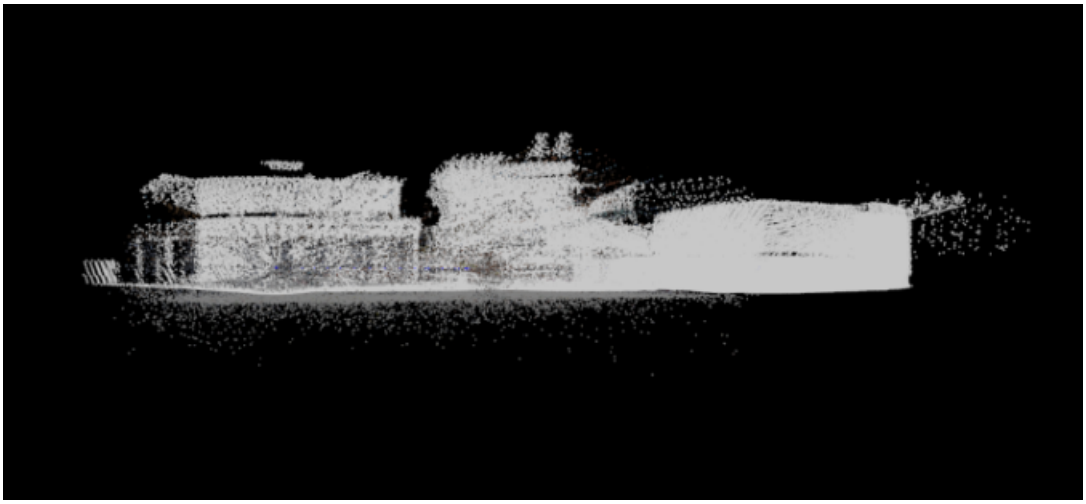

Abbildung 5.8: SfM- und Laser-Punktwolke ohne Anpassung (Z-Richtung)

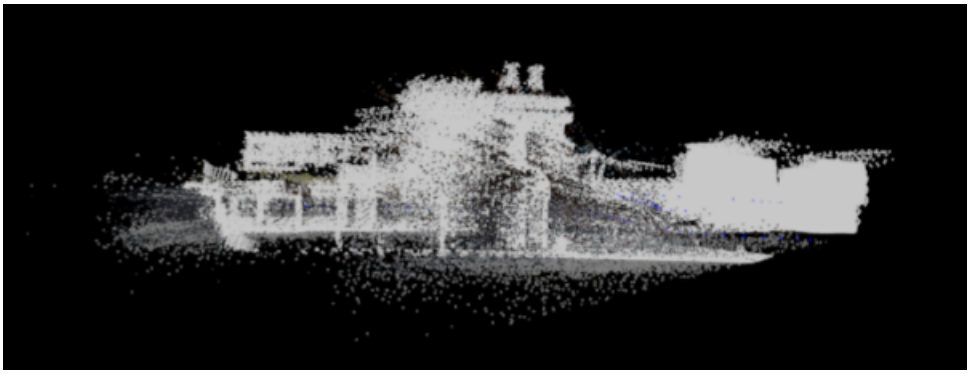

Abbildung 5.9: SfM- und Laser-Punktwolke ohne Anpassung (X-Richtung)

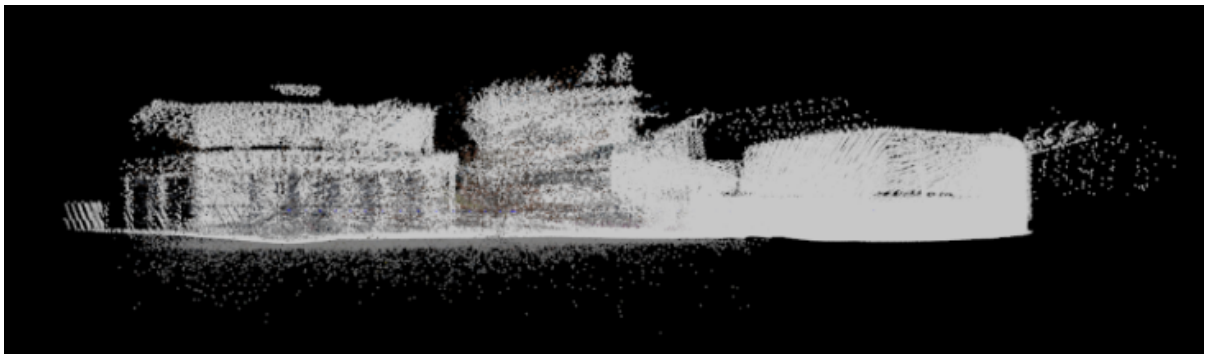

Abbildung 5.10: SfM- und Laser-Punktwolke mit Anpassung (Z-Richtung).

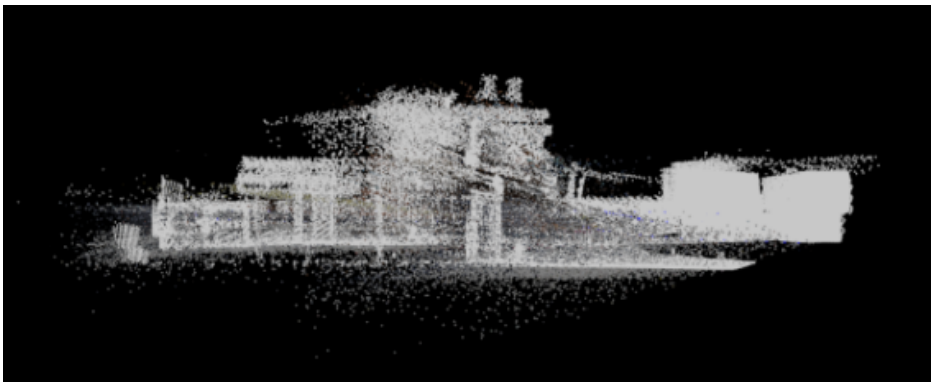

Abbildung 5.11: SfM- und Laser-Punktwolke mit Anpassung (X-Richtung).

6 Fazit

Im Folgenden soll ein kurzes Fazit zu der vorliegenden Masterarbeit gezogen und ein Ausblick auf mögliche Erweiterungen gegeben werden.

6.1 Zusammenfassung

Durch viele Tests und Konfigurationen konnte eine durchschnittliche Trefferrate von mehr als 16% erreicht werden. Das heißt, dass außer Ausreißern, zum Beispiel zu weit entfernten Messpunkten (die nicht selten sind) noch vielen entscheidenden Punkte mit den Messpunkten aus dem Laserscanner nicht übereinstimmen. Das Szenario aus SfM-Prozess ist zurzeit gut aussieht aber nicht genug genau. Aus diesem Grund bin ich nicht zufrieden zu der Trefferquote. Wir müssen noch weiter darauf arbeiten.

Jedoch funktioniert die Bestimmung der Kameraposition und -ausrrichtung noch nicht optimal. Zu Beginn des Prozesses kann die Kameraposition mit einer relativ hohen Genauigkeit bestimmt werden (siehe Abbildung 6.1). Entlang des Weges vergrößert sich durch die Addition von Fehlern auch die Abweichung Fehler der Kameraposition und entfernt sich zu weit von der realen Position(siehe Abbildung 6.2). Bislang konnte ich keine Lösung für dieses Problem finden.

Die im Rahmen dieser Arbeit erstellte Bibliothek und das zugehörige ROS-Programm bieten eine gute Grundlage für mögliche Erweiterungen in der Zukunft. Mit diesem Hintergedanken wurde das Programm auch möglichst modular entwickelt, sodass zukünftige Ergänzungen und Modifikationen kein großes Hindernis darstellen sollten.

6.2 Erweiterung des SfM Systems

In diesen Abschnitt sollen mögliche Erweiterungen für unsere SfM-System diskutiert werden, die ich für sinnvoll erachte und die die Basis einer nachfolgenden Arbeit geben können.

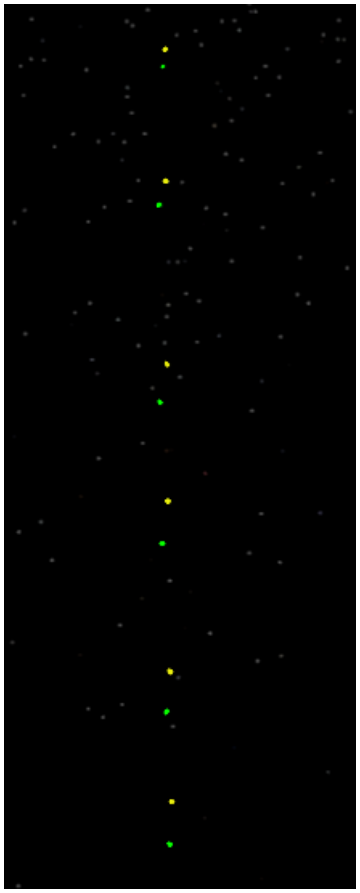

Abbildung 6.1: Geringe Abweichung der Kamerapositionen von den realen Positionen in den ersten sechs Durchläufen.

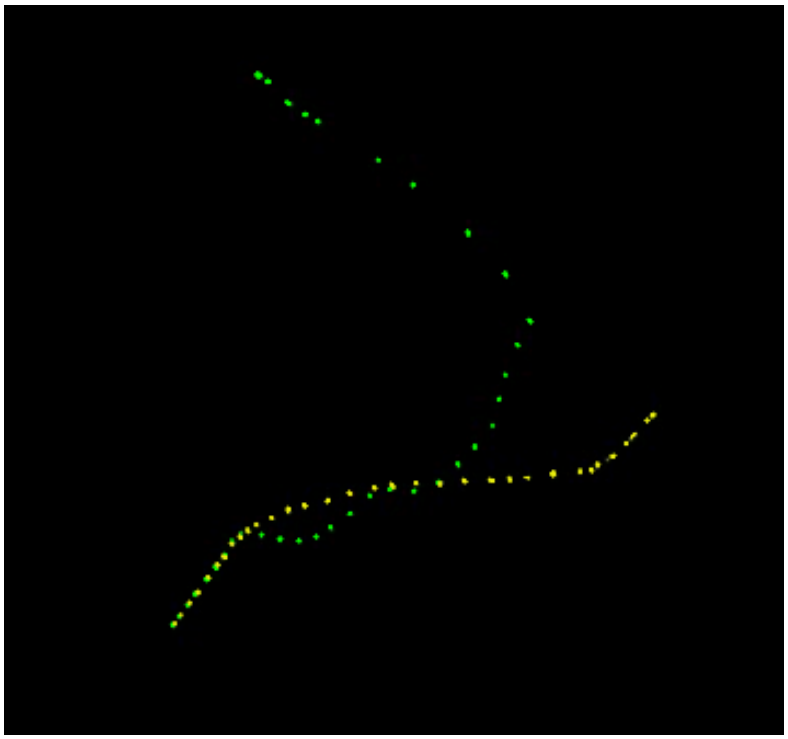

Abbildung 6.2: Die Abweichung der Kameraposition von der realen Position verschlechtert sich mit der steigenden Anzahl der Durchläufe

Der Fehler des SfM-Systems muss weiter reduziert werden. In unserem SfM-Prozess werden allen Punkten auf die berechneten Punkte abgebildet. Das heißt, dass die Fehler eines Durchlaufs einen großen Einfluss auf die kommenden Durchläufe haben kann. Dadurch werden sich die Fehler immer weiter aufaddieren. Das ist der Grund, warum der Prozess aktuell nach einer gewissen Zeitspanne abgebrochen werden muss. In einem echtzeitfähigen System muss der Prozess immer weiter laufen können. Ein Rückblick auf vergangene Daten kostet in einem Echtzeitsystem zu viel Zeit, was auch ein Grund ist, wieso sich die Fehler nur schwer vermeiden ließen.

Eine mögliche Lösung für das Problem setzt voraus, dass die Kameraposition mit einer hohen Genauigkeit bestimmt werden kann. Sobald eine Abbruchbedingung erfüllt ist, der Fehler beispielsweise zu groß ist, kann der gesamte SfM-Prozess neu gestartet werden. Zuvor muss die bisher berechnete Umgebung gespeichert werden, sodass wir mittels einer Abbildungsmatrix die neuen Punktwolken mit den alten fusionieren können.

Auch das Bundle-Adjustment bietet viel Potential für zukünftige Optimierungen. Auf Grund der beschränkten Bearbeitungszeit für die Problemstellung wurde hier lediglich der Bundle-Adjustment-Algorithmus aus OpenCV genutzt. Dieser ist für eine Vielzahl von Szenarien ausgelegt, verbraucht dadurch aber auch unnötig viele Ressourcen und bremst das Gesamtsystem aus.Da die Zeit beschrankt ist, habe ich keine Zeit mehr für das Bundle Adjustment.

Ich bin zuversichtlich, dass eine Folgearbeit große Teile meines SfM-Systems verwenden kann, da es ein effektives Werkzeug ist, mit dem die Fehler bei der Berechnung aller 3D-Punkte und Kamerapositionen reduziert werden kann.

Literaturverzeichnis

[1] A computer algorithm for reconstructing a scene from two prejections, H.C. Longuet-Higgins, UK, 1981

[2] Multiple View Geometry in Computer Vision Second Edition, Richard Hartley, Canberra Australia, 2004

[3] Computer Vision A MODERN APPROACH, David A. Forsyth und Jean Ponce, America, 2003

[4] Stereoanalyse und Bildsynthese, O.Schreer, Berlin, 2005

[5] Computer Vision:Algorithms and Applications, Richard Szeliski, August 2010

[6] Mastering OpenCV with Practical Computer Vision Projects, Daniel Lélis Baggio, Shervin Emami, David Millán Escrivá, Khvedchenia Ievgen, Naureen Mahmood, Jason Saragih, Roy Shilkrot, 2012

[7] Multi-View 3D Reconstruction for Dummies [pdf], Jianxiong Xiao

[8] http://pointclouds.org/documentation/tutorials/pcl_visualizer.php

[9] https://en.wikipedia.org/wiki/Epipolar_geometry

[10] https://en.wikipedia.org/wiki/Triangulation_(computer_vision)

[11] http://wiki.ros.org/ROS/Tutorials

[12] http://pointclouds.org/documentation/tutorials/

[13] http://docs.opencv.org/2.4

[14] https://github.com/royshil/SfM-Toy-Library