Fachbereich 4: Informatik

Erweiterung eines Tools zur Ermittlung der Auswirkung von Kameraparametern auf den Messfehler bei der Vermessung von Einknickwinkeln

Bachelorarbeit
zur Erlangung des Grades
BACHELOR OF SCIENCE
im Studiengang Informatik

vorgelegt von

Manuel Spies

Betreuer: Dipl.-Inform.Simon Eggert, Institut für Softwaretechnik, Fachbereich Informatik, Universität Koblenz-Landau
Betreuer: Dipl.-Inform.Christian Fuchs, Institut für Softwaretechnik, Fachbereich Informatik, Universität Koblenz-Landau
Erstgutachter: Prof. Dr. Dieter Zöbel, Institut für Softwaretechnik, Fachbereich Informatik, Universität Koblenz-Landau

Koblenz, im Januar 2014

Kurzfassung

Diese Bachelorarbeit behandelt die Zusammenführung der bereits vorliegenden Winkelrekonstruktions- und Simulationskomponente und erweitert diese mit Funktionen, um die Durchführung von systematischen Tests zu ermöglichen. Hierzu wird die Übergabe von Bildern aus der Simulationskomponente an die Winkelrekonstruktionskomponente ermöglicht. Des weiteren wird eine GUI zur Testlaufsteuerung und Parameterübergabe sowie eine Datenbankanbindung zur Speicherung der verwendeten Einstellungen und erzeugter Daten angebunden. Durch die Analyse der erzeugten Daten zeigt sich eine ausreichende durchschnittliche Präzision von 0.15° und eine maximale Abweichung der einzelnen Winkel von 0.6°. Der größte Gesamtfehler beläuft sich in den Testläufen auf 0.8°. Der Einfluss von fehlerhaften Parametern hat von variable zu Variable unterschiedliche Auswirkungen. So verstärkt ein Fehler in Höhe den Messfehler um ein vielfaches mehr, als ein Fehler in der Länge der Deichsel.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.
Die Vereinbarung der Arbeitsgruppe für Studien- und Abschlussarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	ja ☐	nein ☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	ja ☐	nein ☐

Koblenz, den 22. Januar 2014

Inhaltsverzeichnis

1 Einleitung **13**
1.1 Motivation 13
1.2 Ziel der Arbeit 14
1.3 Aufbau der Arbeit 14

2 Grundlagen **17**
2.1 Winkelrekonstruktionsalgorithmus 17
2.1.1 Einführung 18
2.1.2 Winkelberechnung 21
2.2 Simulationswerkzeug 23
2.3 Fehlerkonditionierung 24

3 Eigener Ansatz **27**
3.1 Softwareaufbau 27
3.1.1 Aufbau 27
3.2 Simmulation 29
3.2.1 Änderungen 29
3.3 Datenbankaufbau 32
3.3.1 Datenbankarchitektur 32
3.3.2 Datenbankzugriff 34
3.4 GUI 40

4 Experimente **47**
4.1 Welche Fehler sollen gefunden werden? 47
4.2 Welche Parameter beeinflussen das System? 48
4.3 Testläufe 49
4.4 Auswertungsmethoden 53
4.4.1 Genauigkeit 53
4.4.2 Stabilität 54

5 Auswertung **55**
5.1 Präsentation der Ergebnisse 55
5.2 Bewertung der Ergebnisse 63

6 Fazit **65**

Tabellenverzeichnis

2.1 Definition der verwendeten Bezeichnungen 18

3.1 Flags zu Öffnung einer Datenbankverbindung unter SQLite 39

4.1 Beispielwerte: CameraParameter 49
4.2 Beispielwerte: ModelParameter 50
4.3 Beispielwerte: MotionParameter 50

5.1 Aufgetretene Typ-Drei Fehlerfälle 57
5.4 Eckdaten der einzelnen mit Parameterfehler belasteten Testläufe . . 58
5.2 Auszug der Winkeldaten aus einem Testlauf bei Gamma = 0° Teil 1 59
5.3 Auszug der Winkeldaten aus einem Testlauf bei Gamma = 0° Teil 2 60
5.5 Grenzfall von Gamme-Fehler 63

Abbildungsverzeichnis

1.1 Schematischer Aufbau eines Fahrzeuggespanns mit Bezeichnungen . 14

2.1 Schematischer Aufbau eines Fahrzeuggespanns mit Bezeichnungen . 18
2.2 Fall 2: Zwei Marker liegen auf einer Sichtlinie 19
2.3 Fall 3: Ein Marker wird durch eine Dreiecksseite erkannt 20
2.4 Fall 1: Alle drei Marker werden richtig erkannt 20

3.1 Datenbankarchitektur 33
3.2 GUI des Programms 41

4.1 Vereinfachte Darstellung Parameterübergabe 48
4.2 Fall 1: Es wurden alle Marker erkannt 51
4.3 Fall 3: Es können alle Marker erkannt werden, jedoch wurde Marker R durch den Schenkel SV erkannt. 52
4.4 Fall 2: Durch eine Überlappung der Marker können nur zwei der Marker erfasst werden 53

5.1 Erster Teil: Dreidimensionale Darstellung der errechneten Kappa-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerlosen Parametern 55
5.2 Zweiter Teil: Dreidimensionale Darstellung der errechneten Kappa-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerlosen Parametern 56
5.3 Erster Teil: Dreidimensionale Darstellung der errechneten Gamma-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerlosen Parametern 56
5.4 Zweiter Teil: Dreidimensionale Darstellung der errechneten Gamma-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerlosen Parametern 57
5.5 Erster Teil: Dreidimensionale Darstellung der errechneten Kappa-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerhaftem Parameter: Höhe - 1 61

5.6 Zweiter Teil: Dreidimensionale Darstellung der errechneten Kappa-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerhaftem Parameter: Höhe - 1 61
5.7 Erster Teil: Dreidimensionale Darstellung der errechneten Gamma-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerhaftem Parameter: Höhe - 1 62
5.8 Zweiter Teil: Dreidimensionale Darstellung der errechneten Gamma-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerhaftem Parameter: Höhe - 1 62

Kapitel 1

Einleitung

1.1 Motivation

Die bisherigen Tests des Winkelrekonstruktionsalgorithmus wurden anhand eines physikalischen Modells wie in Abbildung 1.1 durchgeführt. Das Modell stellt eine feste Konstellation von Zugfahrzeug und Anhänger dar und lässt sich in den Maßen nicht bzw. nur sehr umständlich verändern.
Ein solcher Versuchsaufbau wirft bei einer ausführlichen Auswertung auf Genauigkeit und Stabilität des Winkelrekonstruktionsalgorithmus einige Probleme auf.
So kann ein Versuchsdurchlauf sowohl durch vorhandene Umgebungseinflüsse wie Licht und Wärme, als auch durch Eigenschaften des Modells selbst wie zB. Kamera-Eigenschaften, Messgenauigkeit beeinflusst werden.
Um die Stabilität des Algorithmus zu bewerten, müssen diese Eingabewerte angepasst werden können. Mit dem Modell lässt sich dies nur mit großem Aufwand bewerkstelligen.
Des weiteren ist, durch die Verwendung des Modells, das Erzeugen von ausreichend großen Datensätzen zur Durchführung einer qualitativ/quantitativen Analyse in der praxis nicht durchführbar. Dies würde das manuelle Einstellen jedes möglichen Winkels erfordern und ist schon aufgrund des reinen Zeitaufwands nicht durchführbar.
Zuletzt stellt auch die Aufzeichnung der Werte, wie sie den aktuellen Ist-Zustand des Modells dokumentieren, einen nicht zu bewältigenden Mehraufwand dar. Nach jeder neuen Einstellung des Trucks müssten die anliegenden Winkel manuell nachgemessen werden.
Erste Schritte hin zu einem automatisierten Testwerkzeug wurden bereits mit der Erstellung einer rudimentären Simulationskomponente gemacht. Die Simulation erstellt eine virtuelle Darstellung eines Gliederfahrzeugs und berechnet ein Bild, wie es auch in der Realität entstehen würde.

Abbildung 1.1: Schematischer Aufbau eines Fahrzeuggespanns mit Bezeichnungen

1.2 Ziel der Arbeit

Diese Bachelorarbeit integriert die Simulationskomponente in die bestehende Winkelrekonstrunktionssoftware und erweitert diese, um systematische Test des Algorithmus zu ermöglichen. Neben der Einbindung der Simulationskomponente wird die Software um eine Datenbankkomponente erweitert, welche sowohl die Parameter des Modells und des Testlaufs als auch die erhaltenen Testergebnisse dokumentieren soll.
In der Auswertung der erzeugten Daten soll eine erste Aussage über die Genauigkeit des verwendeten Algorithmus erarbeitet und der Einfluss von Abweichungen in den Modellparametern analysiert und bewertet werden.

1.3 Aufbau der Arbeit

Nach dieser Einleitung werden zunächst die Grundlagen der Arbeit vorgestellt. Diese beinhalten das von der Arbeitsgruppe ‘Echtzeitsysteme’ erstellte Simulationswerkzeug und den Algorithmus zur Winkelbestimmung, wie er von Frau Dipl. math. Elisabeth Balcerak entwickelt wurde. Des weiteren werden die Grundlagen zum verwendeten Verfahren der Fehlerkoditionierung vorgestellt, welches in einem späteren Kapitel anhand der vorliegenden Daten weiter vertieft werden wird.
Im Anschluss wird im Kapitel 3 das erstellte Testwerkzeug vorgestellt. Zunächst werden die Anforderungen an das System definiert und den Funktionen der Software zugeordnet. Der Abschnitt ‘Softwareaufbau’ beschäftigt sich mit der Architektur

der Kernsoftware und beschreibt die Änderungen, die an der Winkelrekonstruktionskomponente sowie an der Simulation vorgenommen wurden. Dies soll einen Überblick über den Aufbau der Software bieten. Auch wird die auf den Parameterbedarf der einzelnen Komponenten sowie deren Interpretation eingegangen. Daraufhin erklärt der Abschnitt 'Datenbankaufbau' den Aufbau sowie die Verwendung der SQLite-Datenbank zur Verwaltung von Testdaten und Testparametern. Der letzte Abschnit des 3.Kapitels beschäftigt sich mit dem GUI-Aufbau und geht auf die Funktionsweise der Testsoftware sowie dessen Verwendung ein.
Das 4. Kapitel wendet sich der Durchführung der Tests zu und legt zunächst dar, welche Fehler erwartet werden und welche Parameter Einfluss auf den Algorithmus haben. Anschließend wird der Aufbau eines Testlaufs dargelegt und an einem Beispiel erläutert. Im Abschluss dieses Kapitels werden die angewendeten Methoden zur Auswertung der erzeugten Werte beschrieben.
Kapitel 5 widmet sich der Präsentation der erhaltenen Werte in graphischer und tabellarischer Form und stellt wichtige Grenz- und Fehlerfälle dar. Diese werden anschließend analysiert und in Hinsicht auf Stabilität und Präzision bewertet.
Das Fazit stellt das abschließende Urteil über die Software und die erhaltenen Daten vor. Abschließend werden im Ausblick mögliche Ansätze für weitere Arbeiten um die Software vorgeschlagen.

Kapitel 2

Grundlagen

2.1 Winkelrekonstruktionsalgorithmus

Der Algorithmus basiert auf dem mathematischen Modell, das von Frau Dipl. math. Elisabeth Balcerak [Bal05] entwickelt wurde. Dieses System berechnet die beiden Gierwinkel θ_{12}und θ_{23}der Verbindungsstange (siehe Abbildung 2.1) anhand der Koordinaten von Markern in einer Ebene, die orthogonal zur Drehachse liegt. Um die Eigenschaften des zu testenden Systems besser zu verstehen, wird im Anschluss der für die Praxis relevanten Fall durch gerechnet.

2.1.1 Einführung

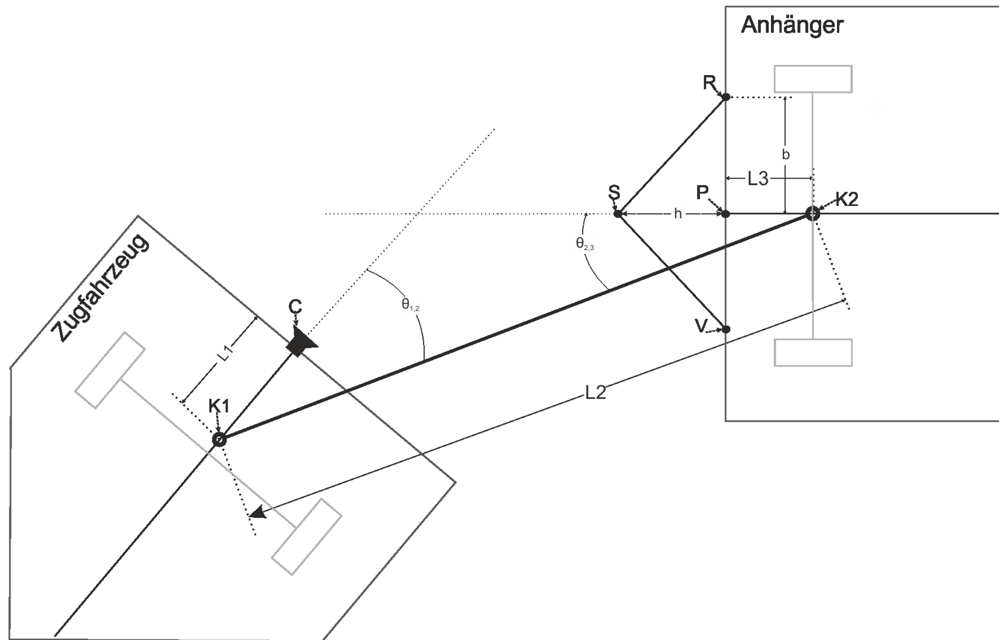

Abbildung 2.1: Schematischer Aufbau eines Fahrzeuggespanns mit Bezeichnungen

Beschreibung	Bezeichnung
Kameraposition	C
Aufhängepunkt am Zugfahrzeug	K1
Aufhängepunkt am Anhänger	K2
Winkel an K1	θ_{12} oder γ
Winkel an K2	θ_{23} oder κ
Mittelpunkt der Pyramidengrundfläche	P
Abstand Kamera zum Drepunkt K1	L_1
Abstand Drehpunkt K1 und K2	L_2
Abstand Drehpunkt K2 und P	L_3
Abstand Punkt P zu Punkt V oder R	b
Abstand Punkt P zu Punkt S	h
Schenkel des Dreiecks	m

Tabelle 2.1: Definition der verwendeten Bezeichnungen

Anhand der Positionen der Marker *R, S, V* werden die Aufnahmewinkel berechnet, wie sie relativ zum Kameralot anliegen. Diese werden in der nachfolgenden Berech-

nung als C_R, C_S, C_V bezeichnet. In der Tabelle 2.1 werden die für die Berechnung der Winkel benötigten Symbole den Entsprechungen im Model zugeordnet. In einem theoretischen Modell kann es zu insgesamt drei möglichen Kombinationen der Markerreihenfolge kommen.
Diese Fälle sind:

1. $C_R < C_S < C_V$

2. $C_R = C_S < C_V$ oder $C_R < C_S = C_V$

3. $C_S < C_R < C_V$ oder $C_R < C_V < C_S$

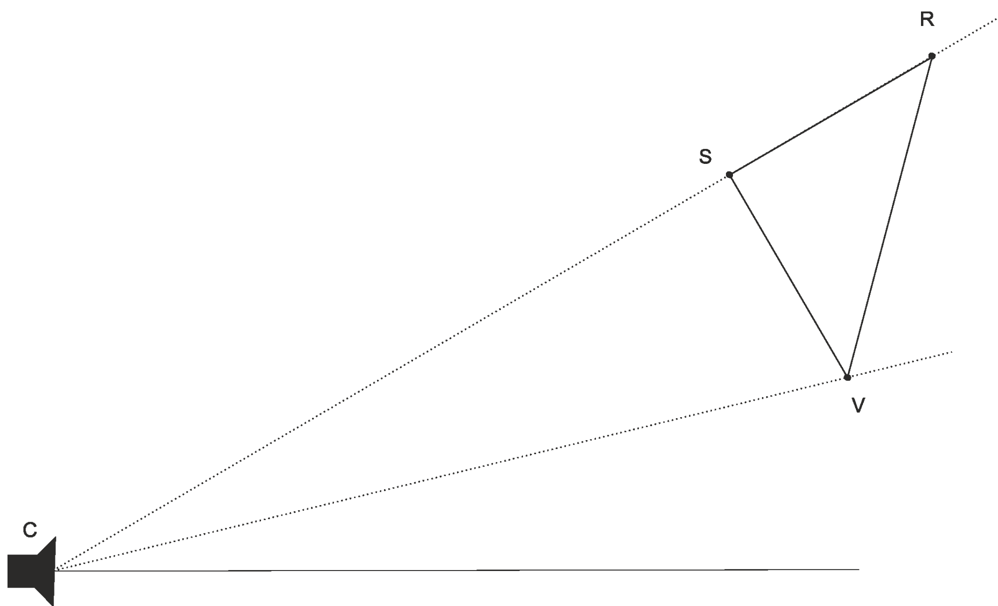

Abbildung 2.2: Fall 2: Zwei Marker liegen auf einer Sichtlinie

Der Fall 2 stellt die Überlappung zweier Marker aus der Sicht der Kamera dar. In diesem Fall ist die Markererkennung nicht in der Lage, alle drei Positionen zu bestimmen. Da der Algorithmus keine Aussage machen kann, ob eine Überlappung vorliegt oder ein Marker nicht im Bild liegt, kann keine Berechnung des Winkels vorgenommen werden.

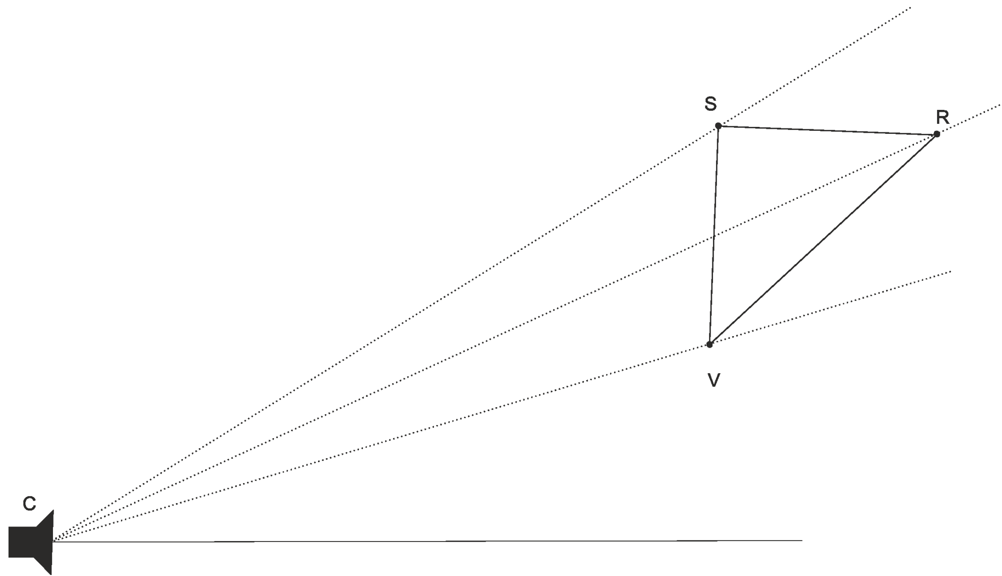

Abbildung 2.3: Fall 3: Ein Marker wird durch eine Dreiecksseite erkannt

Der Fall 3 bezeichnet die Situation, in der einer der Marker R oder V durch den gegenüberliegenden Schenkel des Dreiecks erkannt wird. Dieser Fall tritt jedoch bei einem physikalischen Modell nicht auf. Hier werden meist undurchsichtige Materialien zur Platzierung der Marker verwendet und verhindern somit das Auftreten dieses Fehlerfalls. Dadurch werden nur zwei Marker erkannt werden.

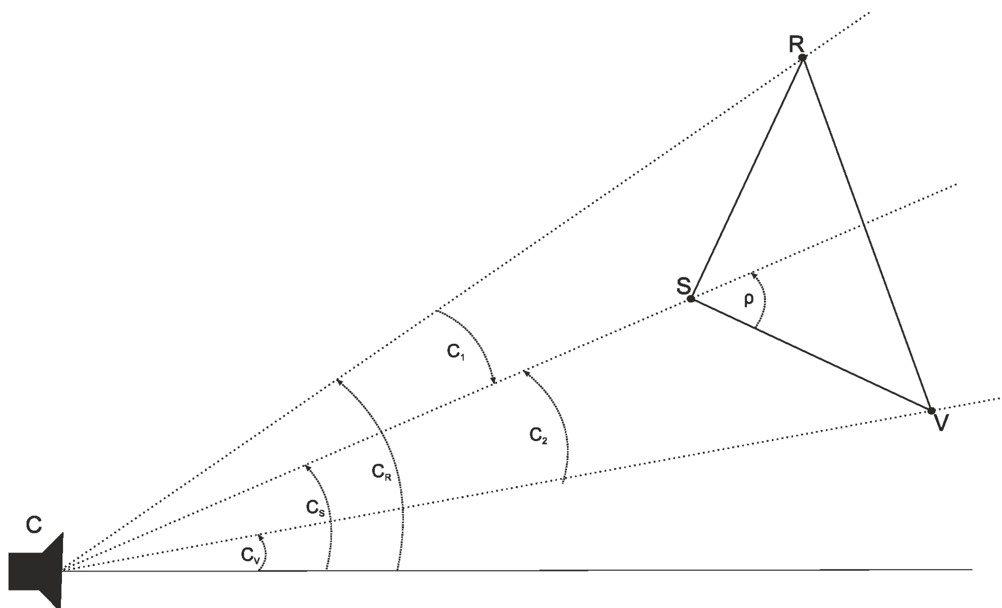

Abbildung 2.4: Fall 1: Alle drei Marker werden richtig erkannt

Der Fall 1 stellt den im Algorithmus dargestellten Regelfall da. Die Berechnung der Gierwinkel wird im folgenden Abschnitt vorgestellt.

2.1.2 Winkelberechnung

Als erstes werden die Winkel für die Dreiecke SCR für C_1 und SCV für C_2 berechnet (siehe Abbildung 2.4).

$$C_1 = C_R - C_S \tag{2.1}$$

$$C_2 = C_S - C_V \tag{2.2}$$

Die Gerade $\overline{CS}$ teilt den innere Winkel des Dreiecks am Punkt S in zwei Winkel, S_1 und S_2. Diese Winkel betragen in der Summe $2 * \phi$.

$$S_1 + S_2 = 2 * \phi \tag{2.3}$$

Des weiteren lassen sich die Winkel CRS (R_0) und CVS (V_0) wie folgt ausdrücken.

$$R_0 = S_1 - C_1 \tag{2.4}$$

$$V_0 = S_2 - C_2 \tag{2.5}$$

$$V_0 = 2 * \phi - C_1 - C_2 - R_0 \tag{2.6}$$

Für weitere Verwendung sei D definiert als:

$$D = 2 * \phi - (C_1 + C_2) \tag{2.7}$$

d.h.

$$V_0 = D - R_0 \tag{2.8}$$

Mit Hilfe der Winkel C_1 und C_2, den Winkeln R_0 und V_0, dem jeweiligen Schenkel des Dreiecks m und der unbekannten Strecke $\overline{CS}$ gilt:

$$\frac{m}{\sin C_1} = \frac{\overline{CS}}{\sin R_0} \tag{2.9}$$

und

$$\frac{m}{\sin C_2} = \frac{\overline{CS}}{\sin V_0} \tag{2.10}$$

Daraus lässt sich folgendes herleiten:

$$\overline{CS} = m * \frac{\sin R_0}{\sin C_1} = m * \frac{\sin V_0}{\sin C_2} \tag{2.11}$$

Mit der Gleichung 2.6 ergeben sich daraus:

$$\sin R_0 = \frac{\sin C_1}{\sin C_2} * \sin(D - R_0) \tag{2.12}$$

$$\sin V_0 = \frac{\sin C_2}{\sin C_1} * \sin(D - V_0) \tag{2.13}$$

Durch die Division durch $\cos R_0$ und der Umformung von $\sin(D - R_0)$ erhält man:

$$\tan R_0 = \frac{\sin C_1 * \sin D}{\sin C_2 + \sin C_1 * \cos D} \tag{2.14}$$

oder entsprechend für V_0

$$\tan V_0 = \frac{\sin C_2 * \sin D}{\sin C_1 + \sin C_2 * \cos D} \tag{2.15}$$

Daraus ergeben sich die benötigten Winkel zur Bestimmung der Strecke $\overline{CS}$:

$$\overline{CS} = m * \frac{\sin R_0}{\sin C_1} \tag{2.16}$$

$$\overline{CS} = m * \frac{\sin V_0}{\sin C_2} \tag{2.17}$$

Mit $\overrightarrow{p_{C_S}}$, dem durch C_S bestimmten, normierten Richtungsvektor, berechnet sich die Punkte R,S und V durch:

$$S = C + \overline{CS} * \overrightarrow{p_{C_S}} \tag{2.18}$$

Der Hilfspunkt R_S ist so definiert, dass die Strecke $\overline{SR_S}$ die Länge m hat und parallel zu $\overrightarrow{p_{C_S}}$ liegt.

$$R_S = S + m * \overrightarrow{p_{C_S}} \tag{2.19}$$

Nun können durch eine Drehung des Punktes R_S die Punkte R und V definiert werden.
Notation: D(A, α, B) bezeichnet eine Drehung von Punkt A um den Winkel α um den Punkt B.

$$R = D(R_S, S_1, S) \tag{2.20}$$

$$V = D(R, -2 * \rho, S) \tag{2.21}$$

oder auch:

$$V = D(R_S, -S_2, S) \tag{2.22}$$

$$R = D(V, 2 * \rho, S) \tag{2.23}$$

Damit sind die Punkte R, S und V bekannt. Um nun den Punkt K_2 zu berechnen, dreht man den normierten Vektor $\overrightarrow{SR}$ um den Winkel $-\rho$

$$-\rho = (-\arctan(\frac{b}{h})) \tag{2.24}$$

mit dem Punkt S als Drehzentrum. Anschließend multipliziert man diesen Vektor mit dem Wert von $h + \overline{PK_2}$.

$$R_S = D(R, -\arctan(\frac{b}{h}, S)K2 = S + \overline{SR_S} * \frac{h + \overline{PK_2}}{m} \tag{2.25}$$

Ebenso kann der Punkt P berechnet werden:

$$P = S + \overline{SR_S} * \frac{h}{m} \tag{2.26}$$

Aus den bis her berechneten Größen lassen sich nun die Winkel θ_{12} und θ_{23}
Notation: (x_A, y_A) bezeichnen die Koordinaten von Punkt A.

$$\theta_{12} = \arctan(\frac{y_{K_2}}{x_{K_2}}) \tag{2.27}$$

$$\theta_{23} = \arctan(\frac{(0 - x_{K_2}) * (y_P - y_{K_2}) - (x_P - x_{K_2}) * (0 - y_{K_2})}{(0 - x_{K_2}) * (x_P - x_{K_2}) - (y_P - y_{K_2}) * (0 - y_{K_2})}) \tag{2.28}$$

Somit sind die zwei Gierwinkel bekannt.

2.2 Simulationswerkzeug

Das bestehende Simulationswerkzeug, welches bereits von der Arbeitsgruppe 'Echtzeitsysteme' erstellt wurde[EFBZ13], stellt einen RayTracer da, der eine vereinfachte Repräsentation des Fahrzeuggespanns modelliert und aus der Kameraposition heraus ein Bild auf die Markerpyramide erzeugt. Der RayTracer erzeugt ein Gitter mit ebenso vielen Maschen wie das zu erhaltene Bild an Pixeln besitzt. Dieses Gitter wird zwischen die Kameraposition und die Szene des Modells positioniert, wobei der Abstand zur Kamera durch den gewünschten Aufnahmewinkel bestimmt wird. Die Erzeugung des Bildes erfolgt, indem durch jede Masche des Gitters ein virtueller Strahl geschickt wird. Trifft dieser Strahl ein Objekt des Modells, so wird die Farbe dieses Objektes der zum Strahl gehörenden Masche zugewiesen. Die Verwendung des Simulationswerkzeugs geschieht über vier Datenstrukturen: *ModelParameter, CameraParameter, MotionParameter* und *AngleParameter.*
ModelParameter beinhaltet die Längenangaben des Modells wie L_1, L_2 und L_3 sowie Breite und Höhe des Dreiecks. Auch die Größe und Form der verwendeten Marker werden definiert.Für die Markerform kann aus drei verschiedenen Modellen gewählt werden:

1. Weißer Kreis
2. Weißes Viereck
3. Viereck mit schwarz-weiß Matrix

CameraParameter bestimmt die Eigenschaften der Kamera, wie Auflösung des Bildes und Aufnahmewinkel. Zudem kann bei der Erzeugung des Bildes ein Rauschen zugeschaltet werden.
MotionParameter legt den Start- und End-Winkel eines Simulationsdurchlaufs sowie die Anzahl der Schritte fest. Diese Werte werden nur benötigt, wenn die Erzeugung der Bilder nicht auf Anfrage von Außen geschieht, sondern vom RayTracer am Stück durchgeführt werden soll.
AngleParameter hat die Funktion das Modell in einen bestimmten Kappa- und Gammawinkel zu versetzen (siehe Tabelle 2.1). Diese Methode kann sowohl intern, als auch extern verwendet werden und ermöglicht die gezielte Erzeugung eines Bildes mit spezifischen Winkelmaßen.

Um zufällige Fehler und Ungenauigkeiten des Modells erzeugen zu können, kann ein Wobble-Generator zugeschaltet werden. Dieser verfälscht mit Hilfe eines Gauss-'schen Zufallszahlengenerators die Eingabeparameter der Strukturen *Model* und *Camera*, kann jedoch auch dem Bild ein weißes Rauschen hinzufügen.
Zur Verarbeitung der übergebenen Parameter im RayTracer müssen diese in eine zweite Datenstruktur überführt. Dabei werden einige Werte umgerechnet, wie zB. aus der Breite b wird eine Breite für $\overline{PR}$ und $\overline{PV}$. Zudem können eventuell gewünschte Fehler aufaddiert werden. Diese neue Struktur wird als *RenderConfiguration* bezeichnet und beinhaltet sowol die original Strukturen *Camera* und *Model* als auch die veränderten *RenderCamera* und *RenderModel*.
Als Resultat eines Simulationsdurchlaufs wird ein OpenCV-Bild im IplImage-Format zur weiteren Verwendung übergeben.

2.3 Fehlerkonditionierung

Zur Konditionierung der Ungenauigkeiten, die bei der Verwendung von fehlerhaften Eingabeparameter in der Winkelrekonstrution entstehen, wird das aus der Numerik stammende Verfahren zur Bestimmung der absoluten Kondition [Fab05] verwendet. Diese wird κ_{abs} bezeichnet und definiert als:

$$\kappa_{abs} = \limsup_{x \to x_0} \frac{||f(x) - f(x_0)||}{||x - x_0||} \tag{2.29}$$

Hierbei bezeichnet x die fehlerfreien Parameter und x_0 die fehlerbehafteten Parameter. Die Funktion $f(x)$ bezieht sich auf den Algorithmus und nimmt als Argument

ein Parameterset x entgegen. x stellt hierbei einen Vektor da, der alle verwendeten Parameter des Modells beinhaltet. Auch das Ergebnis der Funktion $f(x)$ stellt einen Vektor der 2^n-Dimension dar mit den Werten für die Winkel κ und γ für jedes Bild des Testlaufs.
Zur Auswertung der Funktion müssen die verwendeten Vektoren nach der Bildung der Differenzen durch eine Norm in einen eindimensionalen Wert überführt werden.

Kapitel 3

Eigener Ansatz

Zur Erfüllung der Anforderungen an die Software werden folgende Veränderungen und Zusätze an der bestehenden Winkelrekonstruktionskomponente durchgeführt:

Zur Durchführung von Tests und Testreihen ist es möglich, Parameter in einer GUI an die Software zu übergeben. Des weiteren können mehrere Testlaufkonfigurationen in eine Schlange eingereiht und nacheinander ausgeführt werden. Um eine Auswertung der Ergebnisse zu ermöglichen werden sowohl die Testparameter, als auch die während eines Testdurchlaufs ermittelten Positions- und Winkeldaten in eine Datenbank gespeichert. Durch eine Speicherung der Testparameter ist es möglich, unterschiedliche Versionen des Algorithmus unter gleichen Bedingungen zu testen.

3.1 Softwareaufbau

In diesem Abschnitt wird der Aufbau der Software behandelt, wie er nach dem Zusammenfügen der von der Arbeitsgruppe “Echtzeitsysteme” bereitgestellten Winkelrekonstrutions- und Simulations-Komponente entstanden ist. Zunächst werden die einzelnen Klassen und deren wichtigsten Funktionen beschrieben. Im Abschnitt “Änderungen” werden die im Zuge dieser Arbeit durchgeführten Erweiterungen an diesen Klassen vorgestellt.

3.1.1 Aufbau

Winkelrekonstruktionskomponente

Den Kern der Software stellt der *CommunicationManager* da. Er startet die Kette der Bilderzeugung, Bildverarbeitung und Winkelrekonstruktion und beendet diese mit der Verbreitung der rekonstruierten Winkel an Konsole, Datenbank oder

andere Abnehmer. Es kann in einem Programm immer nur eine Instanz des *CommunicationManager* geben, da die Klasse als Singelton programmiert ist. Diese Art der Programmierung gewährt auch den Zugriff auf diese Instanz von jedem Punkt des Programms.

```
public:
   void Test();

   void TriggerAngleBroadcast(System::AngleFrame* poses);
```

Die *Test()*-Funktion startet die Auswertungskette mit dem Aufruf des *ImageAcquisitionGuard*. Wurde ein Durchlauf eines Bildes bis zur Winkelrekonstruktion durchlaufen, so wird der Callback *TriggerAngleBroadcast()* aufgerufen. Dieser schreibt die errechneten Winkel in die Konsole und in die angebundene Datenbank.
Es gibt drei "Guard"-Klassen, welche die eigentlichen Worker verwalten und die Ergebnisse an ihren Nachfolger weitergeben. Diese heißen *ImageAcquisitionGuard*, *ImageProcessingGuard* und *AngleReconstructionGuard*. Jeder Guard hat zur Laufzeit einen einzigen Worker, wobei es eine beliebig große Auswahl an möglichen Workern geben kann. In der aktuellen Implementation der Software existieren für den ImageAcquisitionGuard die Worker *CameraGrabber* und der neu hinzugekommene *SimulationGrabber*. Der *ImageProcessing* verwaltet den *StandartImageProcessing* und der *AngleReconstructionGuard* hat den *PatentAngleReconstruction*. Jeder Guard besitzt in ihrem öffentlichen Interface drei Funktion. Einen Setter zum Setzten des Workers, eine Funktion zum Starten des Workers und eine *getInstance()*-Funktion, die nur die Erstellung von einer Instanz nach dem Singelton-Pattern ermöglicht. Zu dem hat jeder Guard die Aufgabe, das Ergebnis des Workers an die nachfolgende Instanz weiterzugeben. Diese Übergabe der Daten erfolgt durch Frames, die sich von der Klasse *Frame* ableiten. In der Software werden drei dieser Guards verwendet: *ImageFrame*, *Pose3DFrame* und *AngleFrame*.

```
class Frame
{
   private:
     long m_timeStamp;
   public:
     void setTimeStamp(long data)
         {
           m_timeStamp = data;
         }
     long getTimeStamp() const { return m_timeStamp;
         }
};
```

Die Klasse Frame vererbt an jede abgeleitete Klasse die Möglichkeit der Zuweisung eines Werts an die Variable *m_timeStamp*. In der Software wird in dieser die *FrameId* gespeichert und nach jeder Verarbeitung der überbrachten Daten an den ausgehenden Frame übergeben.
Besonders zu beachten ist, dass sich die *Test()* Funktion und das anschließende Rendern eines Bildes in einem eigenen Thread abläuft. Nach jeder Fertigstellung eines Bildes wird durch den *ImageAcquisitionGuard* die nachfolgende Markererkennung ebenfalls in einem neuen Thread ausgelagert. Der finale Schritt in der Berechnung, die Winkelrekonstruktion, arbeitet ebenfalls in einem vom *ImageProcessingGuard* gestarteten Thread. Diese Thread-Aufrufe sind asynchron und können daher auch parallel ablaufen.

3.2 Simmulation

Die Simmulationskomponente wird durch die Klasse *ModelRenderer* verwaltet. Sie ermöglicht die Übergabe der Parameter, die zum erstellen des Modells benötigt werden und ermöglicht das Starten eines Renderprozesses. Für diesen Zweck werden drei unterschiedliche Funktionen zur Verfügung gestellt:

```
void RenderMotion(Parameters::MotionParameter& motion);
void RenderFrame(Parameters::AngleParameter& angles);
IplImage* RenderFrameSync(
                Parameters::AngleParameter& angles,
                Parameters::RenderConfiguration& config);
```

RenderMotion ermöglicht die Erzeugung der Bilder eines Testlaufs in einer vom Renderer gesteuerten Schleife. Die Übergabe des Wertes muss vom Renderer vollzogen werden. Die zweite Funktion rendert die übergebene Winkelkonstellation und sendet diese wie *RenderMotion* an den Empfänger. Funktion 3 *RenderFrameSync* ermöglicht ebenfalls ein Rendern eines Bildes, liefert dieses jedoch nach Abschluss des Vorgangs in Form eines Rückgabewertes an die Aufrufende Instanz zurück.
In der Software wird die dritte Vorgehensweise gewählt, aufgrund der Ähnlichkeit zur Vorgehensweise des *CameraGrabber*s.

3.2.1 Änderungen

Zentrale Änderungen an dem Winkelrekonstruktionsalgorithmus beziehen sich auf die Implementierung des *SimulationGrabber*. Hier wurden zunächst Setter zur Übergabe der *Motion*-Parameter angelegt.

```
void SimulationGrabber::setMotionParameter(
                Parameters::MotionParameter& _motion)
```

```
{
    m_Motion = _motion;
    m_FrameCount = 1;

    m_AngleStepKappa = (m_Motion.KappaEndAngle -
            m_Motion.KappaStartAngle) / m_Motion.FrameCount;
    m_AngleStepGamma = (m_Motion.GammaEndAngle -
            m_Motion.GammaStartAngle) / m_Motion.FrameCount;

    m_Angles.Kappa = m_Motion.KappaStartAngle;
    m_Angles.Gamma = m_Motion.GammaStartAngle;

    System::CommunicationManager::getInstance().setSteps(
            m_AngleStepKappa,
            m_AngleStepGamma,
            m_Motion.FrameCount);
    System::CommunicationManager::getInstance().setStart(
            m_Motion.KappaStartAngle,
            m_Motion.GammaStartAngle);
}
```

Diese Funktion ermöglicht die Übergabe des *Motion*-Parameter und errechnet die Größe der Winkeländerungen von κ und γ zwischen jedem erzeugten Bild. Des weiteren wird die aktuell zu rendernde Winkelkonstelation auf den Startwert gesetzt. Zu Beginn eines Testlaufs wird die FrameId, die nach der Berechnung des Bildes dem *ImageFrame* als Timestamp übergeben wird, auf 1 initialisiert. Um den zu einer Id gehörenden Winkel auch außerhalb des Grabbers rekonstruieren zu können, werden die berechneten Parameter an den *CommunicationManager* übergeben. Nun kann das erste Bild gerendert werden. Dazu wird vom zugehörigen *ImageAcquisitionGuard* die *AcquireImage()*-Funktion aufgerufen.

```
System::ImageFrame* SimulationGrabber::AcquireImage()
{
    IplImage* frame = 0;
    Parameters::RenderConfiguration config;

    Model::ModelRenderer& mr =
                    Model::ModelRenderer::Instance();
    frame = mr.RenderFrameSync(m_Angles, config);

    m_Frame = cvCloneImage(frame);
    cvReleaseImage(&frame);
```

Der erste Schritt setzt den Pointer, in dem im Anschluss das Bild gespeichert werden soll, auf 0 initialisiert. Anschließend erstellen wir eine dynamisches *Render-Config*-Objekt in dem wir vom RayTracer die verwendete Konfiguration erhalten. Nun wird die Instanz des, als Singelton programmierten *ModelRenderes* angefordert. Mit dem Aufruf der Funktion *mr.RenderFrameSync(m_Angle, config)* wird ein Bild mit dem eingeschlagenen Winkel aus *m_Angle* gerendert. Der Pointer *frame* erhält als Rückgabewert des *RennderFrameSync(..)*-Befehls die Adresse des erstellten Bildes. Damit durch die Löschung des Bildes bei einen neuen Durchlauf des *ModelRenderers* nicht die Verarbeitung des Bildes verhindert wird, muss das Bild durch die Funktion *cvCloneImage(frame)* gesichert werden.

```
    m_Angles.Kappa += m_AngleStepKappa;
    if(m_Angles.Kappa > m_Motion.KappaEndAngle)
    {
        m_Angles.Kappa = m_Motion.KappaStartAngle;
        m_Angles.Gamma += m_AngleStepGamma;
        if(m_AngleStepGamma == 0 || m_Angles.Gamma >
            m_Motion.GammaEndAngle)
        {
            System::CommunicationManager::getInstance()
                                            .m_Stop = true;
            m_FrameCount = 0;
        }
    }
    System::ImageFrame * m_if = new System::ImageFrame(
                                    m_Frame,
                                    m_FrameCount);
    m_FrameCount++;
    return m_if;
}
```

Nach der Erzeugung des Bildes wird die nächste Winkelposition eingestellt. Durch die Forderung, das ein Wert echt größer als der Endwinkel sein muss, um die Bedingung zu erfüllen, erhält man bei einem Winkelbereich von 90° bzw. -45° bis +45° mit einer Geforderten Anzahl von 90 Frames eine 1°- Schrittgröße. Anschließend wird ein statisches *ImageFrame* erzeugt und mit dem aktuellen Bild so wie der aktuellen FrameId ausgestattet. Nachdem die *FrameId* für den nächsten Durchlauf erhöht wurde, wird das *ImageFrame* als Rückgabewert an den *ImageAcquisitionguard* zurückgegeben.
Der *CommunicationManager* wurde, um den gerenderten Winkel dem durch die Winkelrekonstruktion berechneten Winkel zuzuordnen, durch die Variablen *m_-*

KappaStep, *m_ GammaStep*, *m_ StartKappa*, *m_ StartGamma* und *m_ Steps* erweitert. In der Funktion *TriggerAngleBroadcast(...)* wird der rekonstruierte Winkel zum einen in den zugehörigen Datenbank-Table geschrieben, zum anderen wird er im Comma-Separated-Value-Format in die Konsole ausgegeben. Dies kann durch die Ausgabe in einem .txt-Dokument neben der Datenbank zur weiteren Auswertung der erhaltenen Daten herangezogen werden. Auch ist es möglich, durch das Setzen des *m_ Stop*-Flags die Ausführung der *Test()*-Schleife zu unterbrechen und den aktuellen Testrun zu beenden.
Eine letzte kleine Änderung war an der *ModelRender*-Klasse nötig. Hier wurden Setter und Getter zum Auslesen derRenderConfiguration hinzugefügt. Mit *SetWobbleStatus* und *SetWobbleParameters* wird es ermöglicht die WobbleGenerator-Konfiguration von außen zu setzen und den WobbleGenerator ein bzw. aus zu schalten.

3.3 Datenbankaufbau

Zur Dokumentierung der erhaltenen Daten wird eine SQLLite-Datenbank verwendet. Diese stellt eine einfache und zuverlässige Datenbank dar, die zudem die erzeugte Datenbank in einer einzigen Datei speichert. Dies Erleichtert bei der Durchführung mehrerer Testreihen mit unterschiedlichen Versionen die Ordnung der Daten.

3.3.1 Datenbankarchitektur

Die Datenbank besteht aus 8 Tables in sie in der Abbildung 3.1 dargestellt ist.
Der Table *Testrun* enthält die ID des Testruns in Form eines Integers, eine Beschreibung mittels eines Strings und eines Datums, das wiederum als Integer gespeichert wird. Die ID fungiert zum einen als PRIMARY KEY des Tables und zum anderen verweist er als FOREIGN KEY auf die ihm zugeordneten Tables-Einträge vom *Motion-*, *Model-* und *Camera*-Table. Die einzelnen Parameter eines Testlaufs werden in separaten Tables gespeichert. Gemeinsam haben sie einen PRIMARY KEY, welcher gleichzeitig die Referenz des FOREIGN KEYS aus dem Table *Testrun* darstellt. Neben den normalen *Camera-* und *Modell*parametern werden in den Tables *RenderedModel* und *RenderedCamera* die speziell für den Modellaufbau im RayTracer bestimmten Parametersets gespeichert. Die in einem Testdurchlauf generierten Daten werden in den Tables *Coordinats* und *Angle* gespeichert. Der Table *Coordinats* besteht aus der *testrunId*, die als FOREIGN KEY auf die ID im Table *Testrun* verweist und der *frameId*, die zusammen mit der *testrunId* den PRIMARY KEY bildet. Die weiteren Spalten des Tables erhalten jeweils die X- und Y-Koordinaten der Marker *R,S* und *V*. Auch der *Angle*-Table benutzt die Spalten *testrunId* und

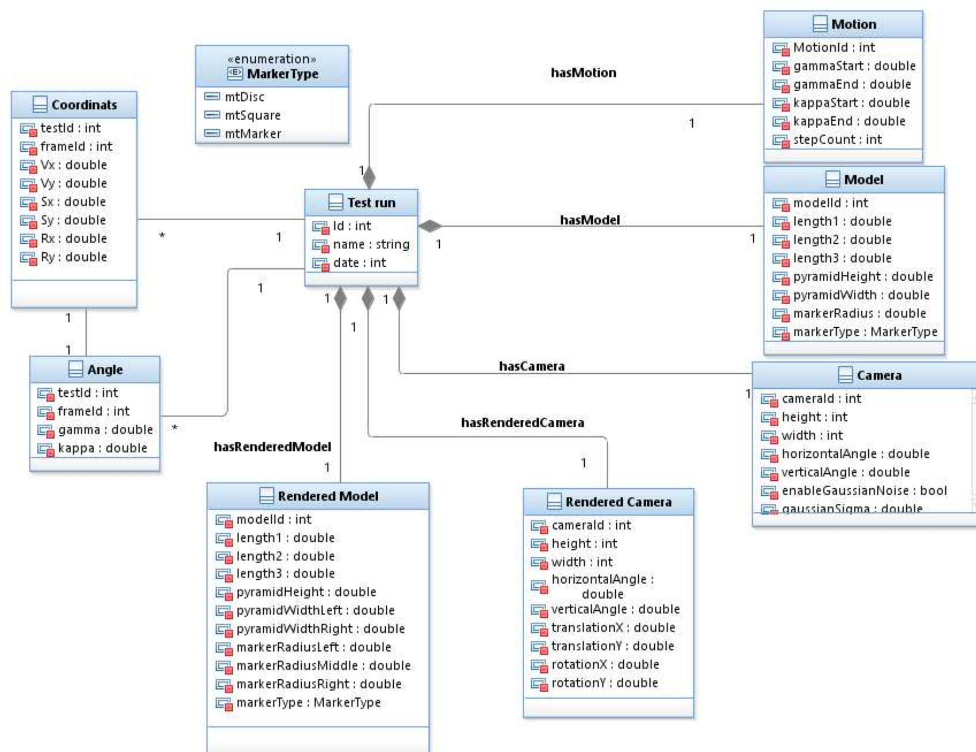

Abbildung 3.1: Datenbankarchitektur

frameId als PRIMARY KEY, zu dem jedoch verweist dieser auch als FOREIGN KEY auf den dieser *frameId* entsprechenden *Coordinats*-Eintrag. Die Restlichen zwei Spalten beinhalten die Winkel-Werte für κund γ.

3.3.2 Datenbankzugriff

Der Datenbankzugriff aus dem Programm heraus erfolgt über das SQLLite-C/C++ Interface [SQL13]. Dieses ist ein auf C basierende API, mit deren Hilfe eine Kommunikation zwichen Programm und SQLLite-Datenbank möglich ist. Die Kommunikation mit der Datenbank erfolgt im *DatabaseHandler*. Er verwaltet den Zugriff auf die Datenbank und erlaubt nur das Ausführen von Schreib- und Leseoperationen über von ihm bereitgestellte Funktionen. Diese Funktionen sind öffentlich (public) zugänglich:

```
static DatabaseHandler& getInstance();
~DatabaseHandler();
```

Mit der Funktion *getInstance()* wird eine Instanz der Klasse angelegt und die Datenbankverbindung initialisiert. Es kann durch die Verwendung des Singelton-Pattern zur Laufzeit des Programms nur eine Instanz der Klasse existieren. Dadurch kann nur eine offene Verbindung zu einer Datenbank erfolgen. Der fehlerfreie Zugriff von mehreren Threads gleichzeitig wird durch SQLite verwaltet und muss nicht zusätzlich abgesichert werden. Mit dem Destructor wird die Verbindung zu Datenbank beendet und das *DatabaseHandler*-Objekt zerstört.

```
int writeTest(int id, char* name, int date,
        Parameters::MotionParameter& motion,
        Parameters::ModelParameter& model,
        Parameters::CameraParameter& camera);
int readTest(int id, Parameters::MotionParameter& motion,
        Parameters::ModelParameter& model,
        Parameters::CameraParameter& camera);

int writeRenderParameter(int id, char* name, int date,
        Parameters::MotionParameter& motion,
        Parameters::RenderConfiguration& config);
int readRenderParameter(int id,
        Parameters::RenderConfiguration& config);
```

Diese Funktionen ermöglichen es, Testparameter auf der Datenbank zu speichern oder von ihr zu lesen.Die Funktionen *writeTest* und *readTest* ermöglichen den Zugriff auf die Parameter der Winkelrekonstruktion und der allgemeinen Testparameter. Auf die *RenderConfiguration* kann durch zwei eigene Funktionen zugegriffen werden. Diese *RendereConfiguration* enthält neben den für die Simulation

angepassten Parametern auch die ursprünglichen Parameter *Camera* und *Model*. Im Zuge eines Aufrufs von *writeRenderParameter()* geschieht auch ein Aufruf von *writeTest()*. Das Ausführen von *readRenderParameter* fügt dem übergebenen *RenderParameter* die *RenderModel* und *RenderCamera* hinzu.

```
int writeMarkerPosition(System::Poses3DFrame* poses);
int writeReconstructedAngles(System::AngleFrame* angles);
```

Diese Funktionen erlauben das Schreiben der aktuell berechneten Marker-Koordinaten und rekonstruierten Winkel.

```
std::vector<TestrunEntry> m_ids;
int createTestIdVector();

int m_TestrunId;
```

Des weiteren gibt es noch den Vektor m_ids, in dem mit der Funktion *createTestIdVector()* die Ids aller zu dem Zeitpunkt in der Datenbank vorhandenen *Testrun*-Einträge gespeichert werden. Die globale Variable *m_ TestrunId* beinhaltet die aktuelle Id des laufenden Testruns. Sie wird für die Speicherung der Koordinaten- und Winkel-Daten benötigt.
Zur Verwendung der Datenbank werden noch zusätzliche Variablen und eine Funktion benötigt. Diese haben die Sichtbarkeit ‘private’.

```
sqlite3 *db;
char *zErrMsg;
int rc;
char sql[400];
```

Der Pointer *db* zeigt auf die aktuelle Datenbankverbindung. Sie wird hauptsächlich als Argument für SQL-Anfragen benötigt. *rc* beinhaltet den Fehlercode. Ist er nach der Ausführung eines Befehls nicht 0, so ist ein Fehler aufgetreten und die dazugehörige Fehlerbeschreibung wird in *zErrMsg* gespeichert. Das Array *sql* dient als Speicher für den als nächstes zu übermittelnden SQL-Befehl. Dieses wird benötigt, da in einem char* durch Funktionen wie *sprintf(...)* nicht ohne weiteres Werte eingesetzt werden können.
Die eigentlichen Befehle zum Schreiben und Lesen von Parametern sind mit der Sichtbarkeit ‘protected’ ausgezeichnet. Dies ist bedingt durch die benutzte C-API von SQLite, in der die callback-Funktionen zur Verarbeitung der Ergebnis-Tables einer SQL-Anfrage wie SELECT verwendet wird. Die callback-Funktion ist jedoch nicht mit der Schreibweise von C++ kompatibel. Die Funktionen für den einzelnen Zugriff auf die Tables der Datenbank sind:

```
int readMotion(int id, Parameters::MotionParameter& motion);
int readModel(int id, Parameters::ModelParameter& model);
```

```
int readCamera(int id, Parameters::CameraParameter& camera);
int readCameraRendered(int id,
        Parameters::RenderedCameraParameter& camera);
int readModelRendered(int id,
       Parameters::RenderedModelParameter& model);

int writeMotion(int id, Parameters::MotionParameter& motion);
int writeModel(int id, Parameters::ModelParameter& model);
int writeCamera(int id, Parameters::CameraParameter& camera);
int writeCameraRendered(int id,
        Parameters::RenderedCameraParameter& camera);
int writeModelRendered(int id,
        Parameters::RenderedModelParameter& model);
```

Der schreibende und lesende Zugriff auf die Datenbank wird anhand des Befehls *writeModel(...)* und *readModel(...)* repräsentativ für die anderen beschrieben.

```
int DatabaseHandler::readModel(int id,
       Parameters::ModelParameter& model)
   {
      sprintf(sql, "select * from model where id=%i ;", id);
      rc = sqlite3_exec(db, sql,callbackModel,
                (void*) &model,&zErrMsg);
      if(rc != 0)
      {
         //std::cout << "Fehler: "<< zErrMsg << std::endl;
         return -1;
      }
      return 0;
}
```

Die Funktion beginnt mit dem *sprintf(...)*-Befehl. In diesem wird dem char[] *sql* ein String übergeben, der die Anfrage an die Datenbank enthält. Das letzte Argument wird in dem Prozess an Stelle des Zeichens *%i* gesetzt. Diese Anfrage ist: 'SELECT * FROM model WHERE id = (%i wird ersetzt durch id);'. Dieser Befehl wählt jede Zeile des Tables 'Model' aus, in der die Spalte mit dem Namen id den Wert der Variable id hat. Im nächsten Schritt wird mit der Funktion *sqlite3_exec(...)* an die Datenbank aus dem Pointer *db* die Nachricht in *sql* versendet. Im Zuge der Auswertung des Befehls wird für jede gefundene Zeile der Callback *callbackModel(...)* mit *model* als Parameter aufgerufen. Tritt ein Fehler auf, wird die Nachricht in der Variablen *zErrMsg* übergeben. Ist während der Ausführung des SQL-Befehls ein Fehler seitens SQL aufgetreten, so ist *rc* ungleich 0 und der Fehlertext wird in den Standart-cout Channal geschrieben. Die aufrufende Funkti-

on hat im Anschluss das Resultat in dem von ihr übergebenen Objekt, in diesem Fall *model*.
Die *callbackModel(..)* Funktion sieht wie folgt aus:

```
int callbackModel(void *model, int argc, char **argv,
                  char **azColName)
{
        Parameters::ModelParameter* _model =
        static_cast<Parameters::ModelParameter*>(model);
        _model->Length1 = strtod(argv[1], 0);
        _model->Length2 = strtod(argv[2], 0);
        _model->Length3 = strtod(argv[3], 0);
        _model->PyramidHeight = strtod(argv[4], 0);
        _model->PyramidWidth = strtod(argv[5], 0);
        _model->MarkerRadius = strtod(argv[6],0);
        int _markertype = atoi(argv[7]);
        if (_markertype == 0)
                    _model->Marker = _model->mtSquare;
        else if(_markertype == 1)
                   _model->Marker = _model->mtSquare;
        else _model->Marker = _model->mtMarker;

        return 0;
}
```

Die Funktion nimmt als ersten Parameter den in der *sqlite3_exec(...)* übergebenen *model*-Void-Pointer. Dazu werden von SQLlite weitere Parameter übergeben: *argc* gibt die Anzahl der Table-Spalten an, *argv* beinhaltet Pointer zu den Werten der jeweiligen Zeile und *azColName* enthält die Namen der Spalten. Zunächst wird der im Funktionsaufruf zu (Void*) umgewandelte *model*-Parameter zu seinem ursprünglichen Typ zurückgesetzt. Nun werden die in *argv* als char* gespeicherten Zelleninhalte mittels *strtod(...)* in die benötigten numerischen Werte zurückgeführt. Für den Fall des MarkerTyps wird mittels Fallunterscheidung der richtige Enum-Wert übergeben. Ist jede Anweisung ausgeführt worden, so wird der Callback beendet und im Zuge der *sqlite3_exec(...)*–Funktion wird entweder der nächste Callback aufgerufen oder die Ausführung abgeschlossen.
Die Ausführung eines schreibenden Zugriffs ist deutlich einfacher.

```
int DatabaseHandler::writeModel(int id,
           Parameters::ModelParameter& model)
{
    int _markertype = 2;
    if(model.Marker == model.mtDisc) _markertype = 0;
```

```
    if(model.Marker == model.mtSquare) _markertype = 1;
    sprintf(sql, "insert into
           model(id, length1, length2, length3, PYRAMIDHEIGHT,
           PYRAMIDWIDTH, MARKERRADIUS ,MARKERTYPE)
           values(%i, %f, %f, %f, %f, %f, %f, %i);"
           , id, model.Length1, model.Length2, model.Length3,
           model.PyramidHeight, model.PyramidWidth,
           model.MarkerRadius, _markertype);
    rc = sqlite3_exec(db, sql,NULL,0,&zErrMsg);
    if(rc != 0)
    {
          //std::cout << "Fehler: "<< zErrMsg << std::endl;
          return -1;
    }
    return 0;
}
```

Hier wird zunächst der MarkerTyp durch Fallunterscheidung in einen numerischen Wert umgewandelt. Im Anschluss wird das auszuführende Char-Array erneut mit *sprintf(...)* erzeugt. Diesmal lautet der Befehl 'INSERT INTO model(id, length1, length2, length3, PYRAMIDHEIGHT, PYRAMIDWIDTH ,MARKERRADIUS ,MARKERTYPE) VALUES(%i, %f, %f, %f, %f, %f, %f, %i);'. In diesem Befehl werden jetzt die nachfolgenden Parameter nacheinander in den jeweils vordersten Platzhalter vom Typ %i für Integer oder %f für Floatzahlen eingesetzt. Nun wird erneut mit dem Befehl *sqlite3_ exec* das *sql*-Array an die *db*-Datenbank gesendet. Diesmal werden Parameter 3 und 4 nicht benötigt. Erneut wird im Fehlerfall die entsprechende Nachricht in *zErrMsg* gespeichert. Nach der Überprüfung von *rc* und gegebenenfalls der Ausgabe des Fehlerbericht wird die Funktion beendet. Abschließend bleibt noch die letzte Funktion der Klasse:

```
private:
    int erstelleDatenbank();
```

Diese Funktion wird bei der Erstellung der Instanz vom Konstruktor ausgeführt und erstellt eine Verbindung zu einer vorhandenen Datenbank oder erstellt bei dessen Fehlen eine neue.

```
int DatabaseHandler::erstelleDatenbank()
{
    char* _sql;
    rc = sqlite3_open_v2("testdatenbank.sqlite", &db,
                              SQLITE_OPEN_READWRITE,NULL);
    if (rc != 0)
```

```
    {
        rc = sqlite3_open_v2("testdatenbank.sqlite", &db,
            SQLITE_OPEN_READWRITE|SQLITE_OPEN_CREATE,NULL);
        if(rc != 0)
        {
            return -1;
        }
        ...
```

Zunächst wird eine lokale Variable für spätere SQL-Befehle erstellt. Das öffnen einer Verbindung zu einer Datenbank geschieht über den Befehl *sqlite3_open_v2(...)*. Dieser Befehl nimmt den Namen der Datenbank, den Pointer auf diese sowie mögliche Flags. Der letzte Parameter bezeichnet den Namen eines VFS-Moduls, wenn es verwendet wird. Da dies nicht der Fall ist, wird er mit NULL besetzt. Das Flag ist hier mit SQLITE_OPEN_READWRITE belegt, das eine Verbindung zu einer bestehenden Datenbank mit schreibenden und lesenden Zugriff erstellt. Mögliche Flags sind:

Flag	Beschreibung
SQLITE_OPEN_READONLY	Nur schreibenden Zugriff
SQLITE_OPEN_READWRITE	Lesender und Schreibender Zugriff
SQLITE_OPEN_CREATE	Kann Datenbank erstellen
	(nur mit _READWRITE-Flag)

Tabelle 3.1: Flags zu Öffnung einer Datenbankverbindung unter SQLite

Wird also keine Datenbank gefunden, entsteht ein Fehler der in *rc* gespeichert wird. In der Folgenden Fehlerbehandlung wird nun eine neue Datenbank im Schreib, und Lese-Modus erstellt. Tritt nun erneut ein Fehler auf, wird dieser Ausgegeben und die Funktion wird beendet. Das Erstellen eines Tables in der Datenbank erfolgt wie an diesem Beispiel:

```
_sql = "CREATE TABLE
        testrun(
                ID int PRIMARY KEY NOT NULL,
                name text,
                date int,
        FOREIGN KEY (id) REFERENCES model(id),
        FOREIGN KEY (id) REFERENCES motion(id),
        FOREIGN KEY (id) REFERENCES camera(id));";
rc = sqlite3_exec(db,_sql,NULL,0, &zErrMsg);
```

Hier wird erneut ein String in _*sql* gespeichert. Die SQL-Anweisung erstellt einen Table mit dem Namen 'Testrun', dieser Table erhält eine Spalte mit Namen "ID" und vom Typ "int". "ID" wird als PRIMARY KEY (Primärschlüssel) des Tables gesetzt und darf nicht leer sein. Hinzu kommen die Spalte "name", die vom Typ "text" ist und eine weitere Spalte vom Typ "int" mit dem Namen 'date'. Der Spalte ID wird eine weitere Funktion zugewiesen. Sie fungiert als FOREIGN KEY und referenziert diese Spalte ID mit den Spalte ID vom Table *Model*, *Camera* und *Motion*.
Bemerkt werden sollte noch, dass mit der Verwendung von SQLite ein bereits erstellter Table nicht mehr verändert werden kann. Somit ist die Reihenfolge der Erstellung wichtig. Tables mit FOREIGN KEY dürfen nicht vor dem referenzierten Table erstellt werden. An Stelle der Bezeichnung "int" kann auch der Begriff "Integer" verwendet werden. Wird jedoch der Primärschlüssel vom Typ "Integer" definiert, so wird diese Spalte mit der Zeilennummerierung gleich gesetzt.

3.4 GUI

GUI-Elemente

Die GUI wurde mit QT-Version 4.8.5 erstellt und besteht aus drei Bereichen:

1. Parametereingabe

2. Datenbankzugriff

3. Testserienverwaltung

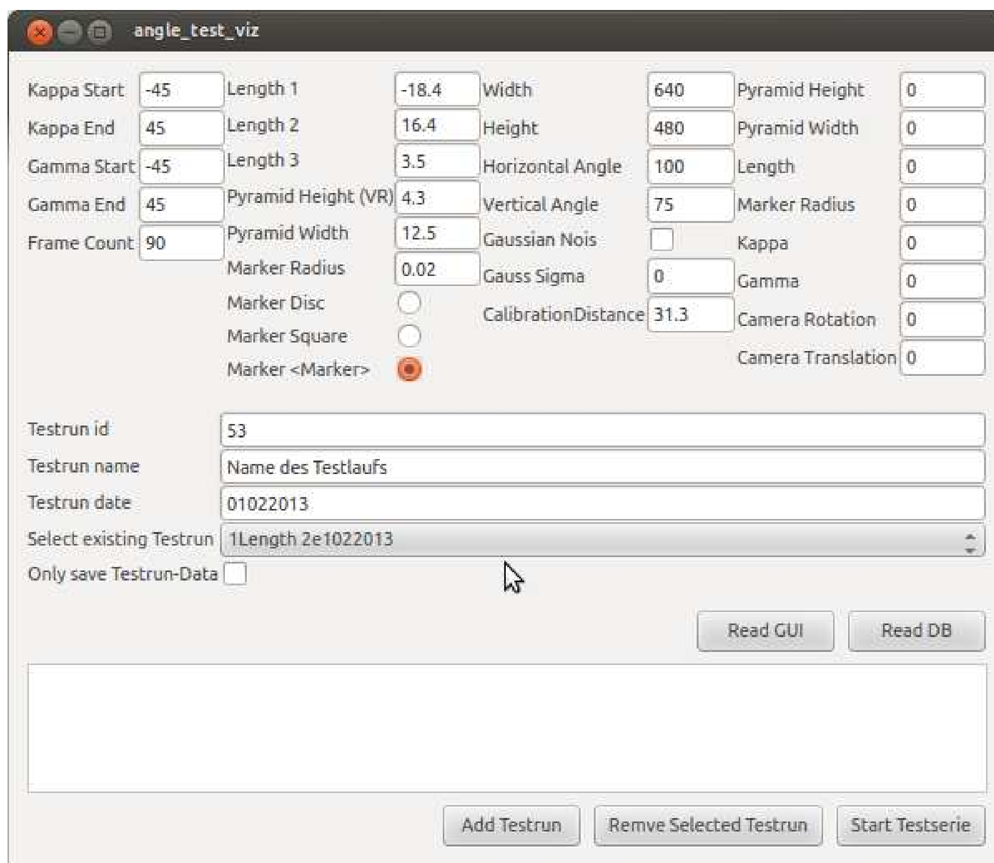

Abbildung 3.2: GUI des Programms

Im ersten Abschnitt ist die Eingabe der Parameter möglich mit Ausnahme der vom Renderer selbst erstellten RenderConfig. Die erste Spalte an Eingabefeldern ermöglicht die Erstellung der *Motion*-Parameter. In der zweiten Spalte lassen sich die Parameter des *Model* übergeben und die dritte Spalte deckt die Parameter der *Camera* ab. Hierbei ist jedoch die CalibrationDistance nicht mehr aktiv, da sie aus den anderen Parametern hergeleitet werden kann. In der rechten Spalte wird die Möglichkeit geboten, die Sigma-Werte des WobbleGenerators zu verändern.
Unter den Eingabefeldern lassen sich die Daten des Testrun-Tables eingeben. Dies beinhaltet ein Feld zur Eingabe der ID, eins zum Eintragen einer Beschreibung und ein Feld für ein Datum bzw. Versionsnummer. Die TestrunId nimmt zum Programmstart den höchsten in der Datenbank vorhandenen Wert an und erhöht diesen um eins. Die ID wird nach jedem Start eines Testlaufs mit Werten aus der GUI erhöht. Darunter wird einem in einem Dropdown-Menue die Möglichkeit geboten, eine alte Konfiguration der Parameter aus der Datenbank zu laden. Wird ein Eintrag aus dem Dropdown-Menue ausgewählt, so werden die jeweiligen Daten in der Parametereingabe gesetzt. Zum Einfügen von Testdaten in die Datenbank ohne

zuvor einen Testrun zu starten, wird durch die Checkbox unter dem Dropdown-Menue ermöglicht. Dadurch ist eine manuelle Veränderung aller Daten über den externen Zugriff auf die Datenbank möglich. Unter dem Datenbanksekment befinden sich die ersten Steuer-Buttons. Hier ist es möglich, eine Simulation mit den Werten aus den Eingabefeldern der GUI zu starten oder den aktuell ausgewählten Testrun von der Datenbank zu starten. Wurde die Checkbox 'Only save Testrun-Data' gesetzt, werden bei einem Start aus der GUI nur die Parameter gespeichert. Das unterste Segment zeigt eine Liste, in der alle für eine Testserie ausgewählten Testläufe angezeigt werden. Mit Hilfe der Buttons am unteren Fensterrand lassen sich Testläufe hinzufügen oder bestehende entfernen. Schlussendlich kann die angelegte Testserie über den Button 'Start Testserie' gestartet werden.

GUI-Funktionen

Die GUI wird in der Klasse *DataInputFrame* erstellt und verwaltet. Das Auslesen der jeweiligen Parameterquellen (GUI und Datenbank) wird in den folgenden zwei Funktionen durchgeführt:

```
void readValuesDb(int id, Parameters::MotionParameter& mop,
    Parameters::RenderConfiguration& config);
void readValuesGui(Parameters::MotionParameter &mop,
    Parameters::CameraParameter& cam,
    Parameters::ModelParameter& mod,
    Model::WobbleGenerator::WobbleParameters& wob);
```

readValuesDb(...) liest, wie im oberen Kapitel beschrieben, die Parameter *Motion* und die RenderConfiguration bestehend aus den Strukturen *Camera* und *Model* sowie deren gerenderten Gegenstücke *RenderCamera* und *RenderModel* aus. *readValuesGui(...)* wiederum liest die Parameterdaten aus den QEdit-Feldern der GUI aus. Die eigentliche Parameterübergabe erfolgt in der folgenden Funktion:

```
void DataInputFrame::initConfig(
                    Parameters::ModelParameter& model,
                    Parameters::CameraParameter& camera)
{
    std::string path;
    System::GetCurrentWorkingDirectory(path);

    Config::ConfigProvider& inst =
                    Config::ConfigProvider::getInstance();

    std::string p = "default.xml";
    inst.LoadFrom(p);
```

```
    inst.setProfile("kleinerLkw");

    inst.set<float>("L3", model.Length3);
    inst.set<float>("L2", model.Length2);
    inst.set<float>("L1", model.Length1);

    inst.set<float>("W", model.PyramidWidth );
    inst.set<float>("H", model.PyramidHeight);
```

In dieser Funktion werden die *Camera*- und *Model*-Parameter an die Winkelrekonstrukion übergeben. Um möglichst wenig am bestehenden Code zu verändern, werden zunächst die alten Daten aus der früher verwendeten Config.xml-Datei geladen, und diese anschließend im *ConfigProvider* verändert. Durch Aufrufe wie *inst.set<float>('L3', model.Length3);* werden zunächst die Werte für $[L_1, L_2, L_3$ und die Dreiecksparameter W und H übergeben.

```
    float calibDist = model.Length2
                    - model.Length1
                    - model.Length3;
    inst.set<float>("CalibrationDistance", calibDist

    float imageWidthIn = (tan((camera.HorizontalAngle/2)
                       * M_PI / 180.0))
                       * calibDist * 2;
    inst.set<float>("ImageWidthInCm", imageWidthIn);
    return;
}
```

Anschließend werden die Variablen *calibDist* und *imageWidth* berechnet. *calibDist* stellt die Distanz zwischen Kamera (Punkt C) und dem Mittelpunkt der Dreiecksbasis (Punkt P) bei einem Winkel von $\kappa = \gamma = 0°$. Damit ist $calibDist = L_2 - L_1 - L_3$. Die zweite Variable *imageWidthInCm* beschreibt die Breite des Bildes auf Höhe der *calibDist*. Diese wird wie folgt berechnet: $imageWidthInCm = \tan(\frac{Kameraaufnamewinkel_{horizontal})}{2}$

$$imageWidthInCm = \tan(\frac{Kameraaufnamewinkel_{horizontal})}{2})*calibDist*2 \quad (3.1)$$

In C++ ist der Wert der trigonometrischen Funktionen in der Einheit Radiant und muss durch $frac\pi 180$ in die Maßeinheit Grad umgerechnet werden. Um einen Testlauf zu starten wird die Funktion *startSimulation(...)* aufgerufen.

```
int DataInputFrame::startSimulation(
```

```
                    Parameters::MotionParameter& motion)
{
    Config::ConfigProvider& inst =
                    Config::ConfigProvider::getInstance();

    ImageAcquisition::SimulationGrabber* sim =
                     new ImageAcquisition::SimulationGrabber();
    sim->setMotionParameter(motion);
    ImageAcquisition::ImageAcquisitionGuard::getInstance()
                    .SetAquisition(sim);
```

Als erster Schritt wird dem *SimulationGrabber* die *Motion*-Parameter übergeben und dieser als Bildprovider im *ImageAcquisitionGuard* festgelegt.

```
    ImageProcessing::StandardImageProcessing * sip =
          new ImageProcessing::StandardImageProcessing();
    ImageProcessing::ImageProcessingGuard::getInstance()
          .SetProcessor(sip);

    AngleReconstruction::PatentAngleReconstruction* ar
          = new AngleReconstruction::
          PatentAngleReconstruction();
    AngleReconstruction::AngleReconstructionGuard
          ::getInstance()
          .SetAngleReconstruction(ar);
```

Als nächstes werden die Worker der zwei anderen Guards (ImageProcessingGuard und AngleReconstructionGuard) zugewiesen.

```
    if(fillDB->isChecked())
          System::CommunicationManager::getInstance()
                    .m_Stop = true;

    std::thread t1(std::bind(
          &System::CommunicationManager::Test,
          &(System::CommunicationManager::getInstance())));
```

Ist die CheckBox *'Only save Testrun-Data'* oder auch *fillDB* markiert, wird die *Test()*-Funktion sofort nach dem Betreten wieder verlassen. Nun, da alle Parameter gesetzt sind, wird ein neuer Thread erstellt, der seine Berechnung in der *Test()*-Funktion des *CommunicationManager* startet und die Simulation beginnt.

```
    t1.join();
```

```
    delete ar;
    ar = 0;

    delete sip;
    sip = 0;

    delete sim;
    sim = 0;

    return 0;
}
```

Ab diesem Punkt wartet der Hauptthread in *t1.join()* auf das Beenden der Simulation. Ist die Simulation abgeschlossen werden die einzelnen Worker gelöscht und deren Pointer auf 0 gesetzt, um bei mehrfachen Tests ohne Neustart der Software keinen Speicherüberlauf zu erzeugen.

Kapitel 4

Experimente

4.1 Welche Fehler sollen gefunden werden?

In den durchgeführten Testreihen soll das System zur Einknickswinkelberechnung unabhängig von externen Einflüssen bewertet werden. Durch die Auswertung der Testdaten werden hauptsächlich zwei Fehlerarten erwartet:

1. Ungenauigkeit des Algorithmus
2. Ungenauigkeit durch Parameterfehler

Die Ungenauigkeit des Algorithmus stellt den Fehler dar, der durch die Auswertung der aufgenommenen Bilder sowie beim Berechnen der daraus resultierenden Winkel entsteht. Hierbei handelt es sich vorwiegend um Fehler, die bei der Diskretisierung des Modells entstehen. Eine Quelle dieses Fehlers ist die begrenzte Auflösung des Kamerabildes und somit die kleinste mögliche Winkeldifferenz zwischen zwei Pixeln. Die von der Markererkennung gelieferten Koordinaten liegen in *float*-Zahlen vor und die dazugehörigen Winkel müssen aus der vorliegenden Winkeltabelle interpoliert werden. Die angewandte Interpolationsmethode bestimmt die Genauigkeit, mit der die der gewünschte Wert errechnet werden kann. Eine Auswertung dieses Fehlers ermöglicht es, die Genauigkeit des verwendeten Algorithmus zu beschreiben und mit anderen Versionen des Algorithmus zu vergleichen. Der zweite erwartete Fehler entsteht, wenn es zu Abweichungen zwischen den Parametern in der Software und den Entsprechungen in der physikalischen Welt kommt. Dies kann zB. durch Materialverformung, Mess- oder Fabrikationsfehler entstehen. Da es jedoch durch die Menge der in die Berechnung einfließenden Parameter zu einer sehr großen Anzahl an möglichen Fehlerkombinationen kommt, wird die hier durchgeführte Auswertung auf jeweils einen verfälschten Parameter beschränkt. Mit diesen Werten lässt sich eine erste Aussage über die Stabilität des Algorithmus

machen. Jedoch ist zu erwarten, das durch das Einbeziehen mehrerer fehlerhafter Parameter eine nicht lineare Verstärkung des berechneten Winkels bewirken kann. Somit ist zunächst nur eine Aussage über die bestmögliche Stabilität möglich.

4.2 Welche Parameter beeinflussen das System?

Die Beeinflussung des Systems geschieht hauptsächlich direkt über die an die Winkelrekonstruktion übergebenen Parameter.

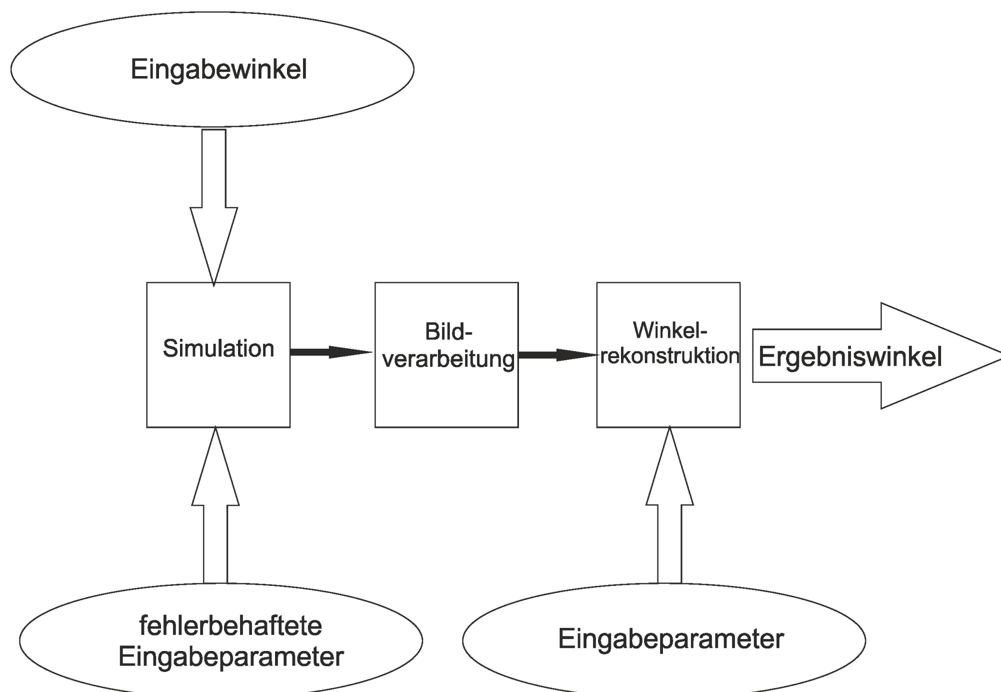

Abbildung 4.1: Vereinfachte Darstellung Parameterübergabe

Diese sind zunächst nur die Werte L_1, L_2, L_3 sowie die Dreiecksmaße H und W. Für die Berechnung des Winkels wird jedoch die Kameraposition als Nullpunkt definiert, der beim $0°/0°$-Winkel genau gegenüber des Markerdreiecks liegt. Somit ist die Position der Kamera auch eine beeinflussende Variable. Über die Berechnung der Bildbreite läuft auch der horizontale Aufnahmewinkel in den Algorithmus ein. Zusammen mit der Auflösung, die zur Bestimmung des Verhältnisses von einem Pixel im Bild zur entsprechenden Länge im Modell benötigt wird, stellen Aufnahmewinkel und Auflösung nicht-fehlerbehaftete Parameter dar. Diese fehlerfreien

Parameter beeinflussen nur die Qualität des Algorithmus. So werden bei hohen Aufnahmewinkeln und kleinen Markern diese oft nicht erkannt, während bei zu geringen Aufnahmewinkeln und größeren Gierwinkeln ein oder mehrere Marker außerhalb des Bildbereichs verschwinden können. Auch die Markermaße beeinflusst in sofern das System, wie von ihr die Qualität der erhaltenen Koordinaten der Markererkennung abhängig ist. Sie haben jedoch keinen direkte Bedeutung in der Funktion des Algorithmus.

4.3 Testläufe

Aufbau

Ein Testlauf besteht aus den zugehörigen Parametersets *Camera* und *Model* und den Werten aus *Motion*, die die Anzahl der erzeugten Aufnahmen und den Winkelbereich der erzeugten Modellszenen festlegen. Bei einem einfachen Test auf die Genauigkeit des Systems ist es nur nötig, die grundlegenden Parameter anzugeben. Diese sind *Model*, *Camera* und *Motion*. Sollte jedoch die Stabilität, also die Anfälligkeit gegenüber Ungenauigkeit bei den Eingabeparametern, festgestellt werden, so werden neben den RenderParameter *RenderModel* und *RenderCamera* auch ein unverfälschter Referenzdurchlauf durchgeführt werden. Die Veränderung der RenderParameter kann über den in der Simulation integrierten Wobble-Generator erfolgen. Für genaue Untersuchungen ist dies aber nicht praktikabel, da der gauss'schen Zuffalszahlengenerator für die Erzeugung der erwünschten Fehler eine zu hohe Anzahl an Durchläufen benötigt. Ein wie unten dargestellter Testlauf von 90 Bildern muss für den Abschluss des Testlaufs $(90 + 1) * (90 + 1)$ Bilder rendern. Die $+1$ resultiert aus der Berechnung der Schrittgröße und der Formulierung der Abbruchbedingungen im *SimulationGrabber*.Für die 8281 Bilder werden bei einer Rate von 3 Bildern pro Minute ca. 46 min für einen Durchlauf benötigt.

Beispiel

Die in den durchgeführten Tests verwendeten Parameter wurden aus dem Modellbau-Gespann aus Abbildung 1.1 hergeleitet. Der horizontale Aufnahmewinkel der Kamera wurde für diese Versuchsreihen auf 70° festgelegt. Bei höheren Werten wie 90° oder 100° kann die Markererkennung besonders im 0° Bereich die Marker nicht mehr durchgehend erkennen.

Bildbreite	Bildhöhe	horiz. Winkel	vert. Winkel	Gauss Rauschen	Sigma
640 pixel	480 pixel	70	52.5°	nein	0

Tabelle 4.1: Beispielwerte: CameraParameter

Die Modellparameter sind weitestgehend in cm angegeben, lediglich die Breite bzw. Radius des Markers wurde in Meter gemessen. Als Markertyp ist ein Viereck mit aufgedruckter Matrix, auch Marker genannt, gewählt.

Length 1	Length 2	Length 3	Dreiecks-Hhöhe	Dreicks-Breite	Marker-Radius	Markertyp
-18.4 cm	16.4 cm	3.5 cm	4.3 cm	12.5 cm	0.02 m	Marker

Tabelle 4.2: Beispielwerte: ModelParameter

Damit ist der Modellaufbau erschöpfend beschrieben. Ein Testlauf wird zwischen den Winkeln -45° und 45° durchgeführt, da es in den durchgeführten Testläufen jenseits dieser Grenzen zu keiner vollständigen Markererkennung kam. Diese Werte gelten sowohl für κ als auch γ. Die verwendeten 90 Frames führen zu einer Schrittgröße von 1° zwischen jedem gerenderten Bild.

Start-κ	End-κ 2	Start-γ	End-γ	Frames
-45°	45°	-45°	45°	90

Tabelle 4.3: Beispielwerte: MotionParameter

Die in einem Simulationslauf erzeugten Bilder lassen sich in die möglichen Positionen des Dreiecks einordnen.

Abbildung 4.2: Fall 1: Es wurden alle Marker erkannt

Dieser Fall bildet den Normalfall für eine erfolgreiche Winkelrekonstruktion. Es werden Alle drei Marker erkannt und deren Position an den Algorithmus weiter gegeben. Dieser errechnet daraus die zwei Winkel für κ und γ. Die erhaltenen Werte stimmen mit einer geringen Fehlertoleranz mit den original Winkelwerten überein. Fällt dieser Unterschied unerwartet groß aus, besteht die Möglichkeit, das der folgende Fall eingetreten ist:

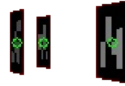

Abbildung 4.3: Fall 3: Es können alle Marker erkannt werden, jedoch wurde Marker R durch den Schenkel SV erkannt.

Dieser Fehler kann sich in einen Datensatz eventuell unbemerkt einschleiche da hier alle drei Marker regulär erkannt wurden. Die Markererkennung und der Algorithmus sind zurzeit noch nicht in der Lage, die Anordnung der erkannten Marker zu überprüfen.Das Auftreten dieses Fehlers kann an den erhaltenen Werten genau festgestellt werden. Dieser Fall hat jedoch in der Praxis nahezu keine Relevanz. Im Gegensatz zum folgenden Fall:

Abbildung 4.4: Fall 2: Durch eine Überlappung der Marker können nur zwei der Marker erfasst werden

Hier überlappen sich zwei Marker und machen es der Markererkennung unmöglich, alle drei Marker zu erkennen. Dies führt zu einem Abbruch der Berechnung des aktuellen Bildes. Dadurch wird dieser Fall genau so behandelt, wie ein aus dem Sichtfeld geratener Marker. Im Gegensatz zu dem letzten Fehler kann dieser nicht von den anderen Fehlerquellen ohne Werte unterschieden werden.

4.4 Auswertungsmethoden

4.4.1 Genauigkeit

Die Auswertung des Algorithmus auf seine Genauigkeit geschieht durch den Vergleich des in der Simulation gerenderten Winkels und des durch die Winkelrekonstruktion errechneten Winkel. Hierbei erhalten wir jeweils einen Vektor der Form:

$$\begin{pmatrix} \kappa \\ \gamma \end{pmatrix} \tag{4.1}$$

Die Differenz der beiden Vektoren ergibt einen weiteren Vektor $\begin{pmatrix}\Delta\kappa \\ \Delta\gamma\end{pmatrix}$ der die Unterschiede zwischen den Winkeln enthält. Nun kann durch die Berechnung von

$$\Delta diff_{gesamt} = |\Delta\kappa| + |\Delta\gamma| \tag{4.2}$$

die gesamte Abweichung von den Eingabewinkeln berechnet werden. Für die Bewertung ist vor allem die maximale Abweichung relevant und können somit die Genauigkeit des Algorithmus beschreiben.

4.4.2 Stabilität

Zur Auswertung der Auswirkungen von fehlerbehafteten Eingabeparameter wird die in den Grundlagen erwähnte absolute Kondition verwendet. Hierzu werden zwei Testläufe miteinander verglichen, ein Testlauf mit Parameterfehler und ein Testlauf ohne Fehler.
Für die Funktion $f(x_0)$ werden die berechneten Winkel eines Parametersets x_0 herangezogen, das keine Fehler zwischen *Parametern* und *RenderParametern* aufweist. Die Funktion $f(x)$ jedoch besitzt in den Eingabeparametern in einer Stelle eine Differenz zwischen *Parametern* und *RenderParametern*. Nun werden diese Werte in die Formel eingesetzt. Da es sich bei den Ein- und Ausgabeparametern der Funktion um einen Vektor handelt, muss eine Norm angewandt werden. Der erhaltene Vektor aus der Differenz $f(x) - f(x_0$ ergibt:

$$f(x) - f(x_0 = \begin{pmatrix}\kappa \\ \gamma\end{pmatrix} - \begin{pmatrix}\kappa_0 \\ \gamma_0\end{pmatrix} = \begin{pmatrix}\Delta\kappa \\ \Delta\gamma\end{pmatrix} \tag{4.3}$$

Die Auswertung dieses Vektors ist mit Hilfe der Maximal-Norm . Diese ist für einen n-dimensionalen Vektor definiert durch:

$$||x - x_0|| = \max_{i=0}^{n}(|x_i|) \tag{4.4}$$

Diese Norm kann auch für die Berechnung des Vektors $||x - x0||$ verwendet werden. Für einen berechneten Winkel ergibt sich daraus:

$$\frac{\max \Delta Winkel}{\max \Delta Parameter} \tag{4.5}$$

Aufgrund der Einschränkungen in der Software kann nur eine Beschränkte Anzahl an Testdaten generiert werden. Die Aussage einer wie hier beschriebenen Auswertung gibt daher nur eine grobe Orientierung, wie sich die Kondition des Algorithmus verhält.

Kapitel 5

Auswertung

5.1 Präsentation der Ergebnisse

Genauigkeit

Aus einem großen Testdurchlauf von ca. 25000 Bildern auf Genauigkeit des Algorithmus wurden folgende Graphen erstellt:

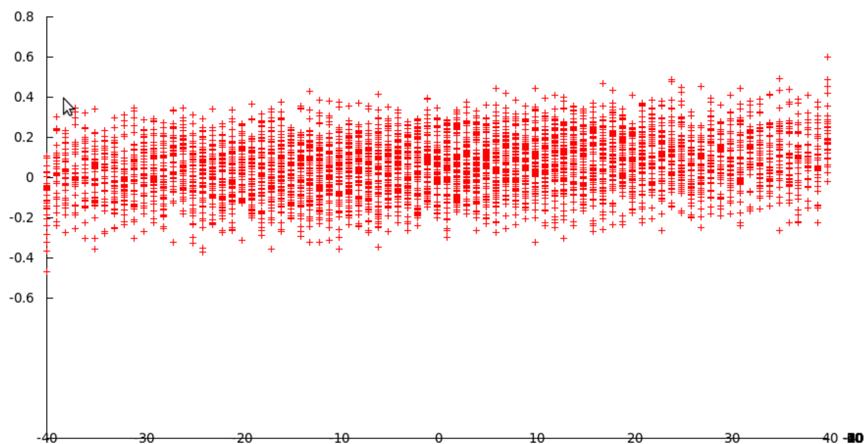

Abbildung 5.1: Erster Teil: Dreidimensionale Darstellung der errechneten Kappa-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerlosen Parametern

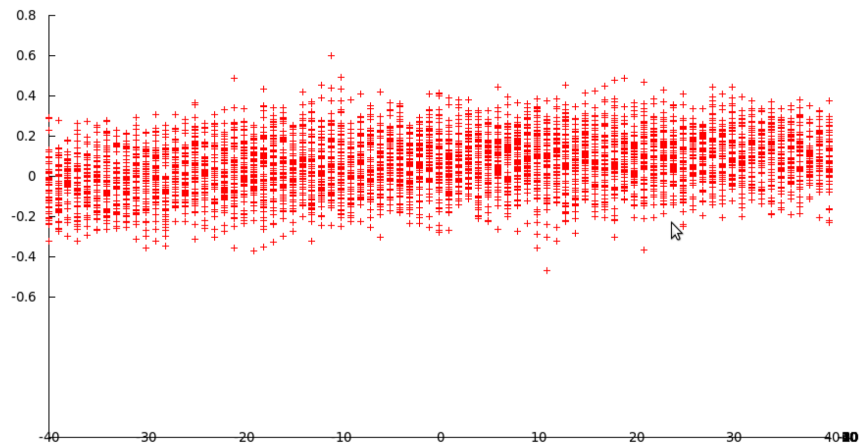

Abbildung 5.2: Zweiter Teil: Dreidimensionale Darstellung der errechneten Kappa-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerlosen Parametern

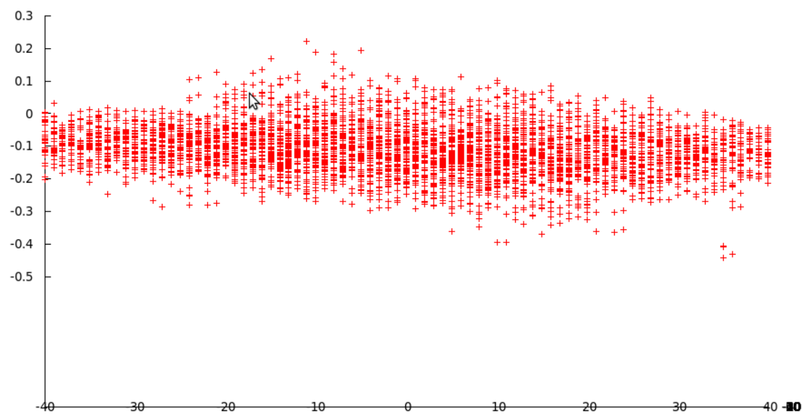

Abbildung 5.3: Erster Teil: Dreidimensionale Darstellung der errechneten Gamma-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerlosen Parametern

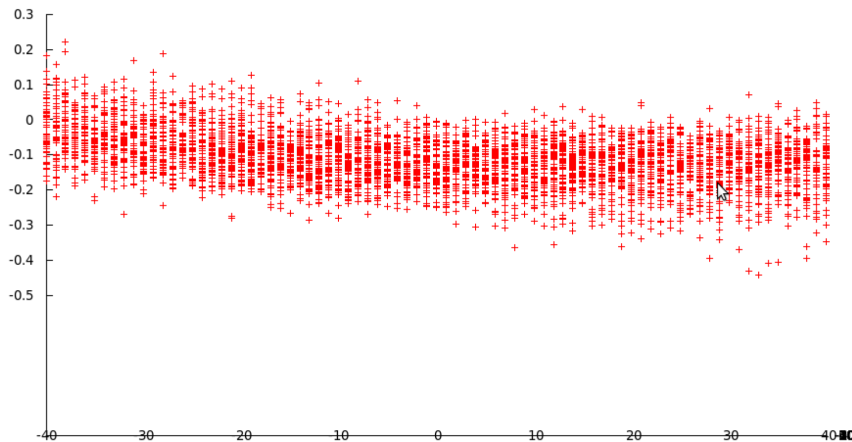

Abbildung 5.4: Zweiter Teil: Dreidimensionale Darstellung der errechneten Gamma-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerlosen Parametern

Hier halten sich die Abweichungen der Winkel in den meisten Fällen unter 0.8°. Zu beachten ist, das bei großen Gamma- und Kappa-Werten der dritte Fehlerfall, wie in Abbildung 2.3 dargestellt, häufig aufgetreten ist und dadurch Fehler über 5° aufgetreten sind. Aufgrund der Differenz zwischen regulären Werten und der Fehlerfälle wurden diese aus der Graphik her ausgefiltert. Ein kleiner Auszug aus den Testdaten liefert die folgende Tabelle:

FrameId	Soll-γ	Soll-κ	Ist-γ	Ist-κ	$\Delta\kappa$	$\Delta\kappa$
9	-40	-36	-12.0287	-64.0106	-27.9713	28.0106
10	-40	-35.5	–	–	–	–
11	-40	-35	-12.0339	-63.0942	-27.9661	28.0942
12	-40	-34.5	–	–	–	–
13	-40	-34	-12.0331	-62.0596	-27.9669	28.0596
14	-40	-33.5	–	–	–	–
15	-40	-33	-12.0132	-60.7531	-27.9868	27.7531
16	-40	-32.5	-12.0326	-60.4925	-27.9674	27.9925
17	-40	-32	-12.0092	-59.7253	-27.9908	27.7253
18	-40	-31.5	-12.0363	-59.4421	-27.9637	27.9421
19	-40	-31	-12.0383	-58.9264	-27.9617	27.9264

Tabelle 5.1: Aufgetretene Typ-Drei Fehlerfälle

Hier sind Abweichungen bis zu +/-27° erfasst worden. Betrachtet man jedoch die Summe der beiden Abweichungen, so bewegen sich diese jedoch um 0°. Wurde in einem Bild nicht alle 3 Markerpositionen erfasst, so wurde in den Daten ein Ersatzwert eingetragen. Hier "–". In den Daten wurden auch die durch Aufnahmen wie Abbildung 4.3 entstehenden Fehler heraus gefiltert, da es sich dabei um einen Fehler handelt, der so regulär der Natur nicht vorkommen wird.

Die Auswertung dieses Testlaufs ergibt eine maximalen Gamma-Abweichung von 0.46° und eine maximale Kappa-Abweichung von 0.60°. Diese Maximalwerte werden jedoch meist nur bei hohen Winkelwerten erreicht. Der Mittelwert beträgt ca 0.11°für Gamma und 0.12° für Kappa. Die durchschnittliche Abweichung am Winkel κ bleibt im Vergleich zwischen $|\gamma| > 22°$ zum Winkelbereich $|\gamma| < 22°$konstant, während sich der Durchschnitt der Winkelfehler von γ um den Winkel Gamma = 0° etwas verstärkt.

Stabilität

Für den Test auf Stabilität wird zunächst der Testrun von Tabelle 5.2 als Referenzwerte der Funktion $f(x_0$ verwendet. Für eine erste Aussage werden die Parameter zunächst um eine Einheit verfälscht. Die Eckdaten der einzelnen Testläufe werden in der Tabelle 5.3 dargelegt.

verfälschter Wert	Größe des Fehlers	Max($\Delta\kappa$)	Max($\Delta\kappa$)	Konditionszahl
Kein Fehler	0 cm	0.60	0.46	—
Höhe	1cm	6.46°	4.93°	6.32
Breite	2cm	5.49°	8.49°	6.54
L1	1 cm	2,71°	2.72°	5.05
L2	1 cm	0.55°	0.55°	1.14
L3	1 cm	2.63°	2,66°	4.78

Tabelle 5.4: Eckdaten der einzelnen mit Parameterfehler belasteten Testläufe

FrameId	Soll-γ	Soll-κ	Ist-γ	Ist-κ	$\Delta\kappa$	$\Delta\kappa$
3246	0	-35	–	–	–	–
3247	0	-34	0.123586	-33.7236	-0.123586	-0.276394
3248	0	-33	0.173632	-32.9297	-0.173632	-0.0703229
3249	0	-32	0.127392	-31.8391	-0.127392	-0.160932
3250	0	-31	0.048208	-31.0788	-0.048208	0.078845
3251	0	-30	0.131952	-29.7333	-0.131952	-0.266652
3252	0	-29	0.126134	-29.0001	-0.126134	0.000141674
3253	0	-28	0.105454	-27.8431	-0.105454	-0.156869
3254	0	-27	0.0983526	-27.109	-0.0983526	0.108957
3255	0	-26	0.0723447	-25.9687	-0.0723447	-0.031309
3256	0	-25	0.104517	-25.0067	-0.104517	0.00673656
3257	0	-24	0.0605487	-24.0376	-0.0605487	0.0376022
3258	0	-23	0.0582806	-23.1627	-0.0582806	0.162742
3259	0	-22	0.00746727	-22.2106	-0.00746727	0.210621
3260	0	-21	0.0817007	-20.9984	-0.0817007	-0.00158587
3261	0	-20	0.196187	-19.883	-0.196187	-0.117012
3262	0	-19	0.0614267	-18.9655	-0.0614267	-0.0344672
3263	0	-18	0.081764	-18.0114	-0.081764	0.0114067
3264	0	-17	0.100612	-17.0584	-0.100612	0.05835
3265	0	-16	0.118204	-16.1068	-0.118204	0.10682
3266	0	-15	0.134407	-15.1578	-0.134407	0.157756
3267	0	-14	0.149362	-14.2118	-0.149362	0.21184
3268	0	-13	0.248838	-12.754	-0.248838	-0.246029
3269	0	-12	0.18118	-11.8656	-0.18118	-0.134444
3270	0	-11	0.102483	-11.0649	-0.102483	0.0648995
3271	0	-10	0.234932	-9.82227	-0.234932	-0.177727
3272	0	-9	0.0672906	-9.14636	-0.0672906	0.146356
3273	0	-8	0.0751428	-8.2236	-0.0751428	0.223603
3274	0	-7	0.123942	-7.04797	-0.123942	0.0479658
3275	0	-6	0.166015	-5.92276	-0.166015	-0.0772368
3276	0	-5	0.170575	-5.02105	-0.170575	0.0210456
3277	0	-4	0.0435863	-4.07945	-0.0435863	0.0794543
3278	0	-3	0.12874	-3.10997	-0.12874	0.109973
3279	0	-2	0.169907	-1.98204	-0.169907	-0.0179551
3280	0	-1	0.126383	-0.947656	-0.126383	-0.0523442
3281	0	0	0.126622	-0.0780658	-0.126622	0.0780658

Tabelle 5.2: Auszug der Winkeldaten aus einem Testlauf bei Gamma = 0° Teil 1

FrameId	Soll-γ	Soll-κ	Ist-γ	Ist-κ	$\Delta\kappa$	$\Delta\kappa$
3282	0	1	0.125619	0.792885	-0.125619	0.207115
3283	0	2	0.082873	1.82649	-0.082873	0.173514
3284	0	3	0.124322	2.95422	-0.124322	0.0457842
3285	0	4	0.209194	3.92415	-0.209194	0.0758534
3286	0	5	0.0825942	4.86545	-0.0825942	0.134546
3287	0	6	0.0863538	5.76823	-0.0863538	0.231772
3288	0	7	0.129162	6.89291	-0.129162	0.107089
3289	0	8	0.17758	8.06931	-0.17758	-0.0693091
3290	0	9	0.184937	8.9929	-0.184937	0.00709958
3291	0	10	0.0178351	9.66841	-0.0178351	0.331592
3292	0	11	0.151008	10.9108	-0.151008	0.0892339
3293	0	12	0.0725204	11.7115	-0.0725204	0.288506
3294	0	13	0.00392141	12.6013	-0.00392141	0.398712
3295	0	14	0.104124	14.0591	-0.104124	-0.0591019
3296	0	15	0.119158	15.0054	-0.119158	-0.00541945
3297	0	16	0.135527	15.9548	-0.135527	0.045193
3298	0	17	0.153394	16.9066	-0.153394	0.0934138
3299	0	18	0.172613	17.8598	-0.172613	0.140198
3300	0	19	0.193426	18.814	-0.193426	0.185987
3301	0	20	0.0575412	19.7332	-0.0575412	0.266844
3302	0	21	0.172628	20.8492	-0.172628	0.150789
3303	0	22	0.248115	22.0605	-0.248115	-0.0604541
3304	0	23	0.197364	23.0131	-0.197364	-0.0131479
3305	0	24	0.194645	23.8898	-0.194645	0.110169
3306	0	25	0.150651	24.8589	-0.150651	0.141086
3307	0	26	0.184053	25.8203	-0.184053	0.179713
3308	0	27	0.157806	26.9625	-0.157806	0.0375451
3309	0	28	0.152354	27.6983	-0.152354	0.301738
3310	0	29	0.131693	28.857	-0.131693	0.143008
3311	0	30	0.12638	29.5894	-0.12638	0.41063
3312	0	31	0.209082	30.9341	-0.209082	0.0658599
3313	0	32	0.129659	31.6943	-0.129659	0.305724
3314	0	33	0.0854088	32.7892	-0.0854088	0.210829
3315	0	34	0.138091	33.5855	-0.138091	0.414453
3316	0	35	–	–	–	–

Tabelle 5.3: Auszug der Winkeldaten aus einem Testlauf bei Gamma = 0° Teil 2

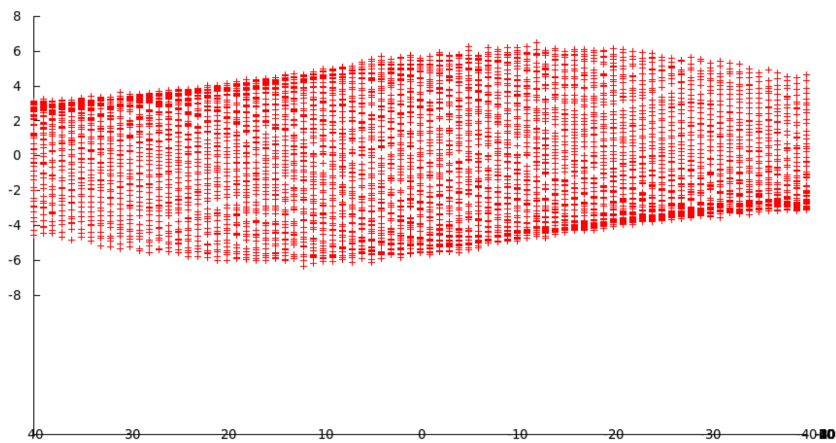

Abbildung 5.5: Erster Teil: Dreidimensionale Darstellung der errechneten Kappa-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerhaftem Parameter: Höhe - 1

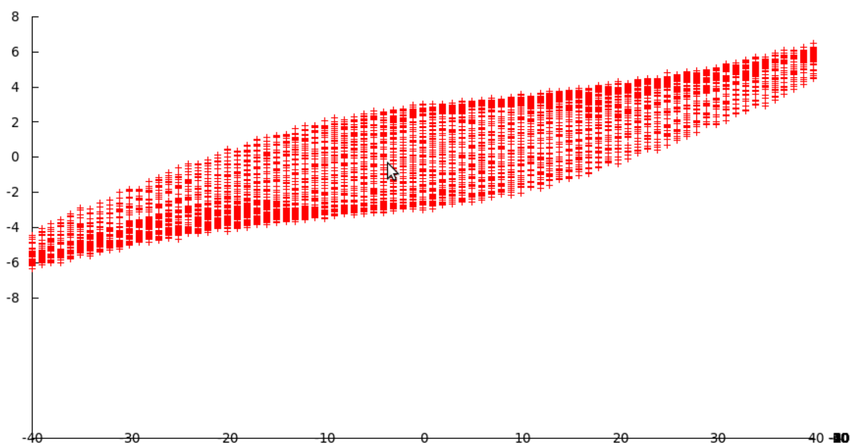

Abbildung 5.6: Zweiter Teil: Dreidimensionale Darstellung der errechneten Kappa-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerhaftem Parameter: Höhe - 1

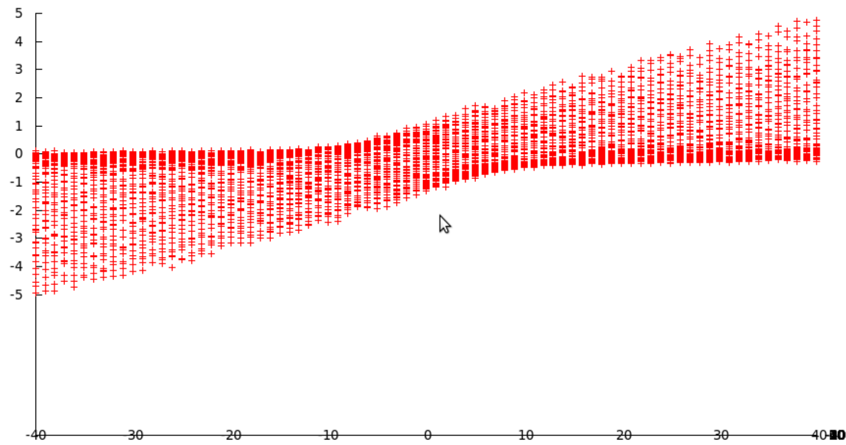

Abbildung 5.7: Erster Teil: Dreidimensionale Darstellung der errechneten Gamma-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerhaftem Parameter: Höhe - 1

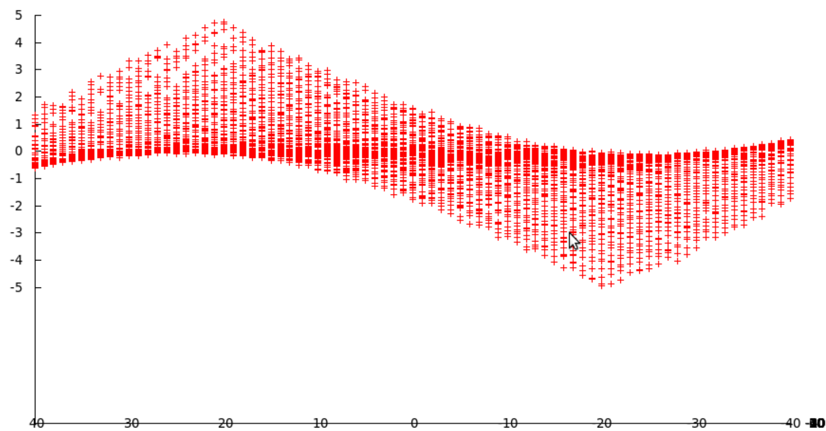

Abbildung 5.8: Zweiter Teil: Dreidimensionale Darstellung der errechneten Gamma-Abweichungen zwischen Renderwinkel und rekonstruierter Winkel mit fehlerhaftem Parameter: Höhe - 1

5.2 Bewertung der Ergebnisse

Die Genauigkeit der einzelnen Winkel können mit einem Maximalen auschlag von 0.6° und 0.46° als hinreichen Genau angesehen werden. Da dies jedoch die größte Abweichung darstellt und die durchschnittliche Abweichung mit 0.14° deutlich niedriger ist, kann man dem Algorithmus als genau einstufen. Die Ungenauigkeit steigt zudem meist nur bei großen Kappa-Winkeln. Auch verhält sich der ermittelte Gamma-Fehler weitestgehend auf gleichem Niveau. Lediglich bei hohen Einknickwinkeln ($\kappa + \gamma$ können durch Probleme bei der Markererkennung Messfehler auftreten. Diese sind meist einzeln stehende Mess-Erfolge, während im nahen Umfeld wenige erfolgreiche Bildauswertungen möglich waren.

FrameId	Soll-γ	Soll-κ	Ist-γ	Ist-κ	$\Delta\kappa$	$\Delta\kappa$
5986	33	32	111	111	111	111
5987	33	33	111	111	111 111	
5988	33	34	111	111	111	111
5989	33	35	33,442	34,8085	-0,442013	0,191497
5990	33	36	33,2684	35,7723	-0,268362	0,227663
5991	33	37	111	111	111	111
5992	33	38	111	111	111	111
5993	33	39	111	111	111	111

Tabelle 5.5: Grenzfall von Gamme-Fehler

Betrachtet man den maximalen gemeinsamen Fehler, so ergibt sich eine Abweichung zu den eingeschlagenen Winkeln von bis zu 0.8°. Dies würde bei einem 10m langen LKW-Anhänger zu einer Abweichung von 14 cm führen. Die Steigerung gegenüber der Einzelabweichung fällt mit 4 cm relativ gering aus. Kann nach Betrachtung der erzeugten Daten eine recht hohe Präzision nachgewiesen werden, so ist der Durchschnitt des Messfehler für beide Winkel kleiner als 0.15°. Zudem muss mit den
Diese Beurteilung bezieht sich jedoch nur auf das ungestörte System. In einem fehlerbehafteten System können bereits geringe Ungenauigkeiten großen Einfluss auf das System haben. Als Beispiel wird ein Fehler auf der Strecke von L1 von 1 cm beobachtet. Die original Strecke beträgt 18.4 cm, jedoch wirkt diese Störung als Verfünffachung des maximalen Winkelfehlers. Betrachtet man den Durchschnitt der Abweichungen so liegt die Differenz zwischen 0.14° im optimalen System und 1.7° im gestörten System. Damit hat sich bei einem Fehler von 5% die durchschnittliche Messungenauigkeit um $\tilde{1}$200% gesteigert. Auch Änderungen an Breite und Höhe haben ähnlich starke Auswirkungen auf die Genauigkeit des Systems. Bei einer Änderung der Höhe um 23% Wächst die maximale Ungenauigkeit um $\tilde{1}$000%

während eine Veränderung der Breite des Dreiecks um 16% eine Verschlechterung des Kappa-Fehlers um 915% und beim Gamma-Fehler sogar um 1845%.
Einzig die Auswirkungen des Parameters L2 fällt aus dem Rahmen. Hier verändert sich bei einer Verkürzung der Strecke von 16.4 auf 15.4 lediglich der Gamma-Winkel und nähert sich dem Kappa-Winkel auf $0.55°$ an.
Soll eine tiefer gehende Analyse erfolgen, so müssen über den erwarteten Messfehler-Raum eine größere Menge an Testläufen durchgeführt werden.

Kapitel 6

Fazit

Ziel der Bachelorarbeit war die Erstellung eines Testwerkzeug, mit Hilfe dessen ein umfassendes Testen des Winkelrekonstruktionsalgorithmus möglich ist. Die erarbeitete Testsoftware ist in der Lage, strukturiert Testläufe durchführen zu können und diese zu speichern. Testläufe können zwecks Vergleichbarkeit von unterschiedlichen Versionen des Algorithmus unter gleichen Bedingungen wiederholt werden. Dadurch ist es möglich, nach Veränderungen an der Winkelrekonstruktion mittels Regressionstests die Funktion des Algorithmus zu bewerten. Aufgrund des Zeitaufwands, der bei einer Durchführung von mehreren Testläufen entstehen kann, können Serien von Tests erstellt werden. Erzeugte Daten können aus der Datenbank ausgelesen und mit externen Programmen analysiert werden.
Aus diesen Daten konnte dann eine erste Aussage über die Präzision der Winkelberechnung gemacht werden. Diese schreibt dem Algorithmus ein recht hohes Maas an Genauigkeit an. Bei der Bewertung der Auswirkungen von Parameterfehlern stellte sich heraus, das der Parameter L2 einen sehr geringen Einfluss auf die Winkelberechnung hat, wohingegen L1 und die Höhe "h" eine deutliche Verschlechterung der Winkel bewirkt haben. Interessant ist der Einfluss der Breit "w", da hier eine doppelt so große Steigerung des Gamma-Winkels gegenüber dem Kappa-Winkel beobachtet werden konnte.
Abschließend lässt sich das System als präzise bewerten. Es werden jedoch noch intensivere Tests benötigt um eine Umfassende Aussage machen zu können, vor mit Kombinationen von mehreren Parameterfehlern.

Literaturverzeichnis

[Bal05] BALCERAK, E: *Mathematisches Modell der Einknickwinkelbemessung mit Hilfe der Kamera für LKW mit Anhünger*. Universität Koblenz-Landau, 2005

[EFBZ13] EGGERT, Simon ; FUCHS, Simon ; BOHDANOWICZ, Frank ; ZÖBEL, Dieter: *Reaktive optische Einknickwinkelvermessung bei Gliederfahrzeugen*. 2013

[Fab05] FABENDER, Prof. Dr. H.: *Skript zur Einführung in die Numerik*. Institut Computational Mathematics TU Braaunschweig, 2004/05

[SQL13] SQLITE: *C-Bibliotk sqlite*. `http://www.sqlite.org/`, 2013