HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

Kinect-/Beamer-Autokalibration

Studienarbeit

Abteilung Informatik

Hochschule für Technik Rapperswil

Frühjahrssemester 2013

Autoren: Engelbert Lüchinger und Kevin Vogt
Betreuer: Prof. Oliver Augenstein

Inhaltsverzeichnis

1. Allgemein

1.1. Aufgabenstellung

Studiengang:	Informatik(I)
Semester:	FS 2013 (18.02.2013-15.09.2013)
Durchführung:	Studienarbeit
Fachrichtung:	Software
Institut:	Diverses
Gruppengrösse:	2 Studierende
Status:	zugewiesen
Verantwortlicher:	Oliver Augenstein
Betreuer:	Oliver Augenstein
Ausschreibung:	Fernziel dieser Arbeit ist die Entwicklung eines Computerspiels ("Minigolf im Wohnzimmer") in qualitativ hochwertiger Weise. Das Projekt ist in mehrere Schritte unterteilt, von denen der erste Schritt im Rahmen der Studienarbeit durchgeführt werden soll: In dieser Studienarbeit betrachten wir nur die Initialisierungsphase des Spiels, d.h. den Aufbau einer Hindernisbahn für das Minigolfspiel. Die Initialisierungsphase ist in mehrere Schritte unterteilt. Die funktionalen Anforderungen an die Studienarbeit sind: 1. Automatisches Erkennen der maximalen Spielfeldgrösse. Diese ist durch ein von einem Beamer auf den Boden projiziertes Bild gegeben. In das Spielfeld gelegte 2 dimensionale Objekte sollen von der Kinect erkannt werden. 2. Autokalibrationsphase. Die Koordinatensysteme von Beamer und Kinect sollen so aufeinander abgestimmt werden, dass die Kinect ein vom Beamer projiziertes Objekt an der Stelle "sieht", an die es vom Beamer projiziert wurde. Als Test (ev. lohnt es sich Schritte 3 und 4 vorher abzuschliessen) wird ein Gegenstand in das Spielfeld gelegt und die Kinect soll den Beamer instruieren, dieses Objekt zu zeichnen. Die Autokalibration ist erfolgreich, wenn das vom Beamer gezeichnete Bild, das auf das Spielfeld gelegte Objekt "ausmalt".

	3. Einführung eines universellen 2 dimensionalen, kartesischen Koordinatensystems. Es soll ein Koordinatensystem eingeführt werden, das es erlaubt, Objekte verzerrungsfrei auf den Boden zu projizieren. Z.B. soll ein Quadrat auf dem Boden als Quadrat erscheinen. 4. Von der Kinect gelieferte Koordinaten sollen auf dieses Koordinatensystem umgerechnet werden und in diesen Koordinaten spezifizierte Objekte sollen in Beamerkoordinaten umgerechnet werden können. 5. Eine Demoapplikation des Autokalibrationsprozesses ist zu entwickeln. Die Demoapplikation soll geeignet sein, den Prozess der Autokalibration im Rahmen der Veranstaltung "Informatik zum anfassen" demonstrieren zu können. Ausserdem soll sie zeigen, dass in das Spielfeld gelegte Objekte korrekt erkannt werden und dass das konstruierte Koordinatensystem es erlaubt, Objekte verzerrungsfrei in das Spielfeld zu projizieren. Erweiterte Anforderungen: 6. In das Spielfeld gelegte 3 dimensionale Objekte sollen mit geeigneten Heuristiken sinnvoll erkannt werden. Nicht-funktionale Anforderungen: 1. Alle Schritte sollen fehlertolerant implementiert sein. D.h. der Beamer und die Kinect sollen beliebig im Raum platziert werden können und die Autokalibration soll auch unter ungünstigen Verhältnissen funktionieren. 2. Nach Abschluss der Kalibration, soll das konstruierte Koordinatensystem durch eine einfach benutzbare API zur Verfügung stehen.

1.2. Eigenständigkeitserklärung

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Rapperswil, 31.05.2013

Name, Unterschrift:

Lüchinger Engelbert Vogt Kevin

2. Abstract

Im Frühjahrsemester 2012 haben Philipp Eichmann und Roman Giger in ihrer Arbeit "Minigolf im Wohnzimmer" [BAMiW] den Prototyp eines 2D-Minigolfspiels entwickelt, welches Kinect und Beamer als Input- bzw. Output-Devices verwendet. Ein Spielfeld wird dabei mittels Beamer auf den Boden projiziert. Die Kinect wird sowohl für das Erkennen des Feldes als auch für das Erkennen der Spielergesten genutzt.

In unserer Arbeit behandeln wir die Problematik der Synchronisation zwischen Kinect, Beamer und realer Welt genauer. Die Hauptproblematik war die Definition eines zwei dimensionalen karthesischen Koordinatensystems für Objekte in der realen Welt, sowie dessen Synchronisation mit den Koordinatensystemen von Kinect und Beamer. Dieser Prozess sollte unabhängig von der Positionierung von Kinect und Beamer vollautomatisch ablaufen und für zukünftige Projekte in Form einer API zur Verfügung gestellt werden.

Die Funktionalität der Arbeit umfasst das Erkennen eines Feldes, welches vom Beamer projiziert wird, mit Hilfe der Kinect, das dazugehörige Umrechnen in die verschiedenen Koordinatensysteme und das Erkennen und Vermessen von realen Hindernissen innerhalb des vom Beamer ausgeleuchteten Bereichs.

Zudem liefert unsere Arbeit eine API, welche beispielsweise von Spiele-Entwicklern genutzt werden kann. Diese können damit eigene Spiele basierend auf diesem Kinect & Beamer-Ansatz entwickeln, ohne die Synchronisationsproblematik vollständig verstehen zu müssen.

3. Management Summary

3.1. Ausgangslage

Das Spielkonzept mit Kinect und Beamer wurde bereits von Philipp Eichmann und Roman Giger behandelt. Die Idee ist, dass das vom Beamer projizierte Bild als interaktives Spielfeld genutzt werden kann. Dabei ist die Kinect für das Ausmessen des Spielfeldes sowie das Erkennen der Spielergesten zuständig.

Unsere Arbeit legt den Fokus auf die Thematik und Problematik, welche hinter diesem Spielkonzept steckt. Sie umfasst als Hauptziel die automatische Kalibration der beiden Devices, Kinect und Beamer. Ohne diese Kalibration wäre es unmöglich ein solches Spiel benutzen zu können. Viele Berechnungen werden nämlich verborgen vor dem Anwender getätigt. Die automatische Kalibration soll es für den Anwender so einfach wie möglich machen, die Kinect und den Beamer in Einklang zu bringen.

Des Weiteren ermöglicht unsere Arbeit zukünftigen Software-Entwicklern weitere Spiele basierend auf diesem Kinect/Beamer-Konzept zu implementieren. Die Problematik bleibt nämlich jeweils die gleiche. Die Entwickler bekommen einige Funktionen zur Verfügung gestellt mit deren Hilfe sie gewünschte Anwendungsfälle umsetzen können, ohne sich mit der eigentlichen Problematik beschäftigen zu müssen.

3.2. Vorgehen, Technologien

Beamer und Kinect können fast beliebig im Raum aufgestellt werden. Dabei muss darauf geachtet werden, dass das komplette Beamerbild auf dem Boden zu liegen kommt. Zudem muss die Kinect das ganze Beamerbild erkennen können. Idealerweise werden Beamer und Kinect nebeneinander oder aufeinander platziert. Konkret bedeutet dies, dass Beamer und Kinect in die gleiche Richtung ausgerichtet werden sollten. Mittels einer einfach gehaltenen Benutzeroberfläche (GUI) kann die Kalibration gestartet werden.

Entwickelt wurde die Software in C#. Für die Benutzeroberfläche und das Beamerbild kam WPF (Windows Presenation Foundation) zum Einsatz. Die Software ist für das Abfragen der Kinect-Daten, das Berechnen sowie für das Zeichnen des Feldes auf den Beamer zuständig.

3.3. Ergebnisse

Wir konnten in unsrer Arbeit diverse mathematische Formeln entwickeln, die allesamt ein Teilproblem unserer analysierten Problemstellung lösen. Diese Formeln haben wir in einem selbst entwickelten Framework programmiert. Dabei wurde des Framework so aufgebaut, dass die

Algorithmen beliebig ausgetauscht oder um neue Algorithmen ergänzt werden können. Dem Benutzer stellen wir ein API zur Verfügung um unser Framework zu nutzen.

3.4. Ausblick

Es gibt einige denkbare Weiterführungen unserer Arbeit. Mittels der API könnten Spiele entwickelt werden, welche das Spielprinzip benutzen, wie es bereits mit dem 2D-Minigolfspiel von Philipp Eichmann und Roman Giger veranschaulicht wurde. [BAMiW]

Des Weiteren würden Optimierungen des Kalibrationsprozesses bzw. der Algorithmen in Frage kommen. Dabei bieten vor allem die Performance und die Genauigkeit der Umrechnungen Verbesserungspotential. Einige Optimierungsvorschläge haben wir bereits während dieser Arbeit formuliert.

4. Technischer Bericht

4.1. Analyse

4.1.1. Technologieentscheide

- Visual Studio 2012 (Update 1) & Resharper 7.1.2
- Kinect SDK 1.7
- Kinect Developer Tools & Kinect Studio (Version 1.7)
- Sandcastle
- GitHub

Bei der Auswahl der Entwicklungsumgebung haben wir uns sehr schnell für das Microsoft Visual Studio 2012 entschieden, da die komplette Entwicklung auf C# basieren sollte und dies durch das Visual Studio am besten abgedeckt wird. Resharper erweitert das Visual Studio um einige nützliche Features, welche das Programmieren sehr viel angenehmer und produktiver machen. Zudem liefert es wichtige Refactoring-Tools, welche erlauben, effizienteren und lesbareren Code zu schreiben.

Seit dem 18.März 2013 ist die Kinect SDK 1.7 verfügbar, welche die beiden neuen Features "Kinect Interactions" und "Kinect Fusion" liefert. Ebenso erhielten die Kinect Developer Tools sowie das Kinect Studio ein Update auf Version 1.7. Wir verwenden seither diese Versionen, obwohl die neuen Features für uns vorerst keinen enormen Gewinn mit sich bringen. Die Kompatibilität zu unserem bisherigen Code, welcher noch auf Basis des Kinect SDK 1.6 entwickelt wurde, ist offiziell von Microsoft gewährleistet, d.h. alle bisher implementierten Funktionalitäten laufen weiterhin wie gewünscht.

Die Kinect Developer Tools bieten einige einfache Code-Beispiele und deren Anwendung an, welche vor allem zu Beginn des Projektes hilfreich waren. Damit konnten wir uns recht schnell in die Kinect SDK einarbeiten. Das Kinect Studio wird mit den Developer Tools mitgeliefert und ermöglicht es dem Entwickler die unterschiedlichen Streams der Kinect aufzunehmen und zu einem späteren Zeitpunkt wieder zugeben. Zudem erlaubt das Tool Kinect-Anwendungen zu analysieren, indem es nützliche Informationen zu den einzelnen Streams liefert.

Für die Generierung einer übersichtlichen Code-Dokumentation in HTML-Form haben wir uns auf das frei verfügbare Sandcastle geeinigt.

Als Versionsverwaltungsystem haben wir GitHub gewählt, da wir uns damit bereits aus früheren Projekten bestens auskennen.

4.1.2. Beamer

Ein Beamer wurde uns für unsere Arbeit freundlicherweise zur Verfügung gestellt. Dabei handelt es sich um ein DLP Modell DS+25 der Marke Christie. Als Auflösungen standen zur Auswahl:

- 1024 x 768
- 1400 x 1050
- 1600 x 1200

In unserer Arbeit mussten wir den Beamer in der Auflösung 1600 x 1200 betreiben, das sonst nicht das ganze Bild angezeigt wurde. Die Wahl des Beamers ist für unsere Software nicht relevant. Es lassen sich problemlos auch andere Modelle nutzen, sofern sie vom Betriebssystem unterstützt werden.

4.1.3. Kinect

4.1.3.1. Modelle

Microsoft bietet ihren Bewegungssensor "Kinect" in den zwei folgenden Ausführungen an [WikiKin]:

- Kinect for Xbox360 (Nov '10)
- Kinect for Windows (Feb '12)

Die beiden Modelle bestehen aus den gleichen Hardwarekomponenten. Lediglich die Funktionalitäten wurden für die "Kinect for Windows" ein wenig erweitert. Die wahrscheinlich interessantesten Features sind der sogenannte Near Mode und die verbesserte Personenerkennung. Der Near Mode ermöglicht es dem Tiefensensor, Objekte bereits in einer Entfernung von 0.4m zum Kinect-Sensor zu erkennen. Mit dem "Default Mode" werden Objekte erst ab einer Entfernung von 0.8m sicher erkannt. Die Farbbilder der Kinect sind von diesen Mode-Einstellungen nicht betroffen [MSDNCoSp].

Da wir für unsere Studienarbeit ein möglichst grosses, von einem Beamer projiziertes Bild erkennen wollen, kommt eigentlich nur der "Default Mode" in Frage, was uns die Freiheit lässt, beide Kinect-Modelle verwenden zu können. Bereits in der ersten Projektwoche hatten wir jedoch Probleme mit der "Kinect for Xbox360". Da wir beide ein MacBook Pro besitzen, wollten wir für die Entwicklungsumgebung eine virtuelle Windows 8 Maschine nutzen. Die "Kinect for Xbox360" verweigert es jedoch über eine virtuelle Maschine angesteuert zu werden. Glücklicherweise hat

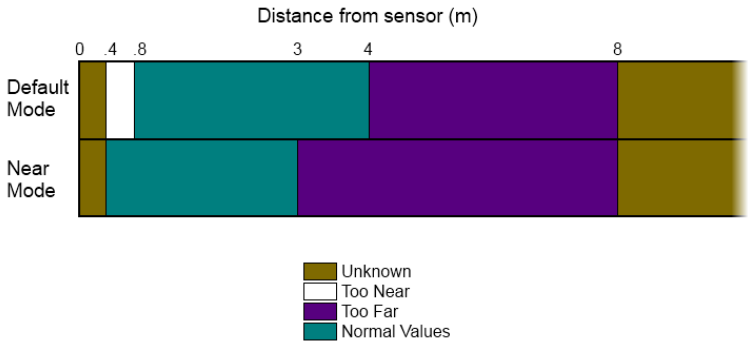


Abbildung 4.1: Near Mode vs. Default Mode

die "Kinect for Windows" diese Einschränkung nicht, was uns dazu veranlasst hat eine solche zu bestellen.

4.1.3.2. Streams

Die Kinect-Hardware umfasst im Wesentlichen drei verschiedene Sensoren:
[BKPwMK] [WikiKin]

- Farb-Kamera (ähnlich wie eine Webcam)
- PrimeSense-Tiefensensor (bestehend aus einem Infrarot-Laser und einem CMOS-Sensor)
- Mikrofon-Array (4 Mikrofone an der Front der Kinect)

Abbildung 4.2: Kinect Hardware

Für unsere Arbeit stehen sowohl Farb-Kamera als auch der Tiefensensor im Mittelpunkt. Das Mikrofon-Array, welches für Sprachsteuerung eingesetzt werden kann, ist für uns momentan noch nicht von Bedeutung.

Nun stellt sich die Frage, wie man denn an die Daten dieser unterschiedlichen Sensoren herankommt. Die Antwort auf diese Frage liefert uns das Stichwort "Streams". Die Kinect bietet

nämlich 3 verschiedene Streams zum Auslesen der Sensor-Daten an. Folgende Erläuterungen sollten Klarheit schaffen.

1. Color Image Stream

Wie es der Name schon verrät, greift dieser Stream auf die Daten der Farb-Kamera zu. Die Kamera funktioniert prinzipiell wie eine gewöhnliche Webcam, d.h. sie liefert einfach ein Farb-Bild der Umgebung. Die Bilddaten können in folgenden unterschiedlichen Formaten und Auflösungen abgefragt werden:

- RgbResolution640x480Fps30 (default)
- RgbResolution1280x960Fps12
- YuvResolution640x480Fps15

2. Depth Image Stream

Hier ist gleich klar, dass es sich um den PrimeSense-Tiefensensor handeln muss. Die Tiefendaten umfassen für jedes gelieferte Pixel zweierlei Informationen, nämlich die dazugehörige Tiefe (in Millimeter) sowie einen sogenannten Player-Index, welcher es ermöglicht die Daten verschiedener Spieler zu unterscheiden. Der Bereich, in welchem korrekte Tiefendaten ermitteln werden können, ist beschränkt. Wie es bereits weiter oben illustriert wurde, kommt es dabei darauf an, welchen Modus (Normal oder Near) man verwendet. Liegen Pixels ausserhalb des erkennbaren Bereiches, so beträgt der Tiefenwert des betroffenen Pixels einen Wert, wie z.B. 0x0000 (zu nahe), 0x0FFF (zu weit weg) oder 0x1FFF (unbekannt).

Auch der PrimeSense-Tiefensensor kann in verschiedenen Auflösungen genutzt werden:

- 640x480 (default)
- 320x240
- 80x60

3. Skeleton Stream

Obwohl wir diesen Stream nicht benutzt haben, möchten wir ihn dennoch der Vollständigkeit halber kurz beschreiben. Die Skeletondaten werden anhand des PrimeSense-Tiefensensors ermittelt. Sie umfassen jegliche Daten zur 3D-Position eines Spielers. Deshalb wird dieser Stream hauptsächlich zur Erkennung von Gesten der Spieler verwendet.

4.1.4. Aufbau Beamer/Kinect

Wie in der Aufgabenstellung beschrieben, ist es möglich die Kinect und den Beamer unabhängig voneinander im Raum zu platzieren. Daraus ergeben sich zwei unterschiedliche Problemstellungen.

Betrachterproblem

Die erste Problematik behandelt die Positon der Kinect im Raum. Dabei spielt es eine Rolle wie diese relativ zum Beamer ausgerichtet wird. Das gleiche Bild wird je nach Position der Kinect unterschiedlich wahrgenommen.

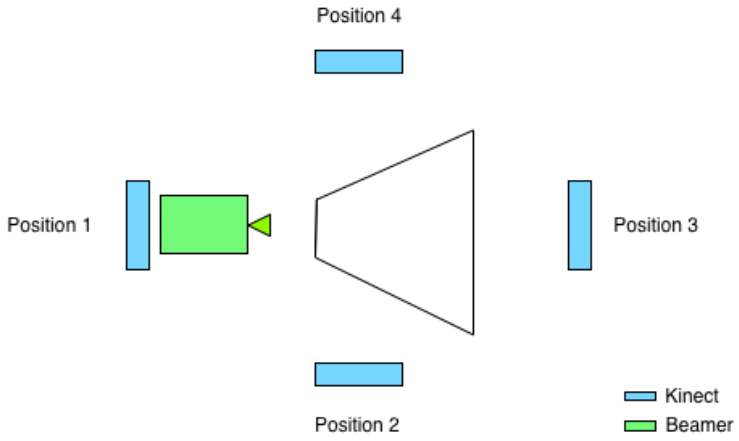
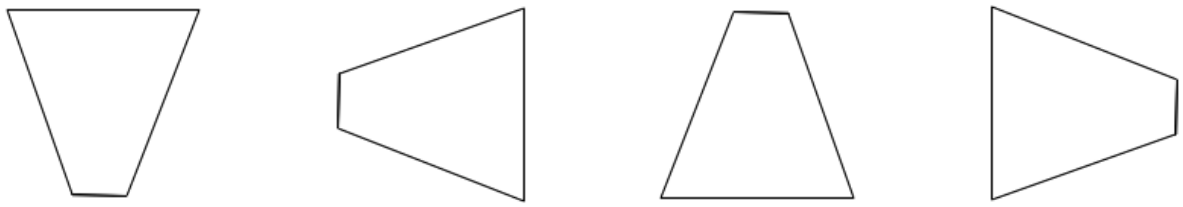


Abbildung 4.3: Betrachterproblematik der Kinect

Die folgenden Bilder illustrieren die Aufnahmen, welche die Farbkamera der Kinect an den jeweiligen Positionen aufnimmt.

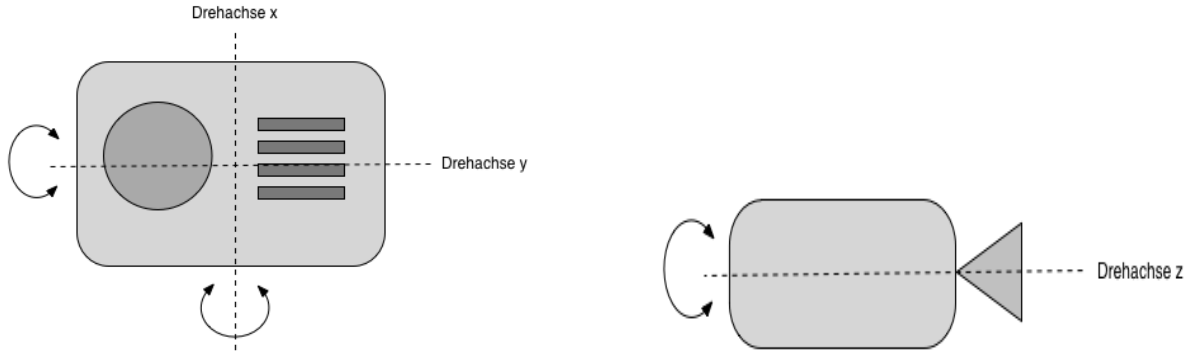


(a) Position 1 (b) Position 2 (c) Position 3 (d) Position 4

Abbildung 4.4: Aufgenommene Bilder aus Sicht der Kinect

Bildverzerrung

Der Beamer lässt sich an insgesamt drei Achsen drehen.



(a) Beameransicht Vorne (b) Beameransicht Seite

Abbildung 4.5: Beameransichten

Die geometrische Form des Beamerbildes ist somit abhängig vom Neigungs-, Dreh- und Kippwinkel des Beamers. Es können die unterschiedlichsten Polygone entstehen. Würde man nun beispielsweise einen Kreis im Beamerbild zeichnen, würde eine ellipsenartige Figur resultieren.

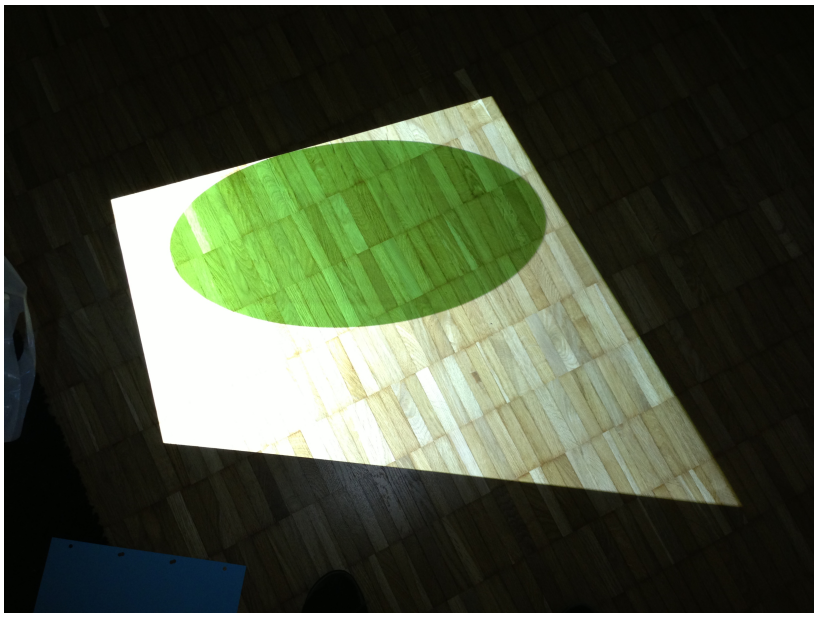


Abbildung 4.6: Verzerrung einer Kreisfläche

4.1.5. Phaseneinteilung

Aufgrund der Aufgabenstellung, haben wir uns entschieden die Probleme in zwei Phasen zu unterteilen, welche zeitlich nacheinander ablaufen.

Kalibrationsphase

Die Kalibrationsphase erfolgt als erstes und kalibriert die Kinect und den Beamer. Die Phase beinhaltet folgende Schritte:

- Kalibrieren des Beamerbildes mit dem Kinectbild
- Synchronisation der verschiedenen Koordinatensysteme
- Erstellen einer abstrakten 2D Fläche, welche es ermöglicht Gegenstände verzerrungsfrei zu projizieren oder auszumessen

Operationsphase

Die Operationsphase erfolgt unmittelbar nach der Kalibrierung und muss folgende Funktionen erfüllen:

- Gegenstände in der projizierten Fläche erkennen
- Schwerpunkt eines Objektes bestimmen und ausmessen
- Einfärben der Pixel, welche ein erkanntes Objekt darstellen

4.1.6. Koordinatensysteme

Folgende Koordinatensystem sind bereits durch den Einsatz von Beamer und Kinect gegeben.

Beamerkoordinatensystem
Der Beamer, genau genommen das Beamerbild, basiert auf einem zwei dimensionalen kartesischen Koordinatensystem, wobei die Einheit Pixel ist. Die für uns interessante Fläche ist abhängig von der Beamerauflösung.

Kinect Bildkoordinatensystem
Das von der Kinect gesehene Bild ist ähnlich dem Beamerbild und basiert ebenfalls auf einem zwei dimensionalen kartesischen Koordinatensystem. Auch hier ist die Einheit in Pixel gegeben. Die Auflösung des Bildes ist von der Farbkamera der Kinect abhängig.

Kinect Tiefenbild Koordinatensystem

Die Kinect kann zu einzelnen Pixeln auch einen Tiefenwert liefern, was das Bildkoordinatensystem um die Dimension der Tiefe erweitert. Dabei muss beachtet werden, dass die Pixel immer noch in Pixeln, die Tiefe aber in Millimetern gemessen wird.

Damit wir unsere Algorithmen einsetzen können, müssen wir die obigen Koordinatensysteme erweitern.

Real World Koordinatensystem
Um die projizierte Fläche messen zu können, braucht es ein kartesisches Koordinatensystem in einem einheitlichen Metermass. Deshalb konstruieren wir uns auf Basis der Kinect Tiefendaten ein kartesisches Koordinatensystem in Milimetern. Diese Real World Koordinaten dienen auch als Grundlage für das Area Koordinatensystem.

Area Koordinatensystem
Um die projizierte Fläche in einem zwei dimensionalen kartesischen Koordinatensystem darstellen zu können, mussten wir ein weiteres virtuelles System schaffen. Diese wird benötigt um ein Bild verzerrungsfrei auf die Fläche projizieren zu können. Wird beispielsweise ein Kreis in das Area Koordinatensystem gezeichnet, sollte auch ein Kreis dargestellt werden.

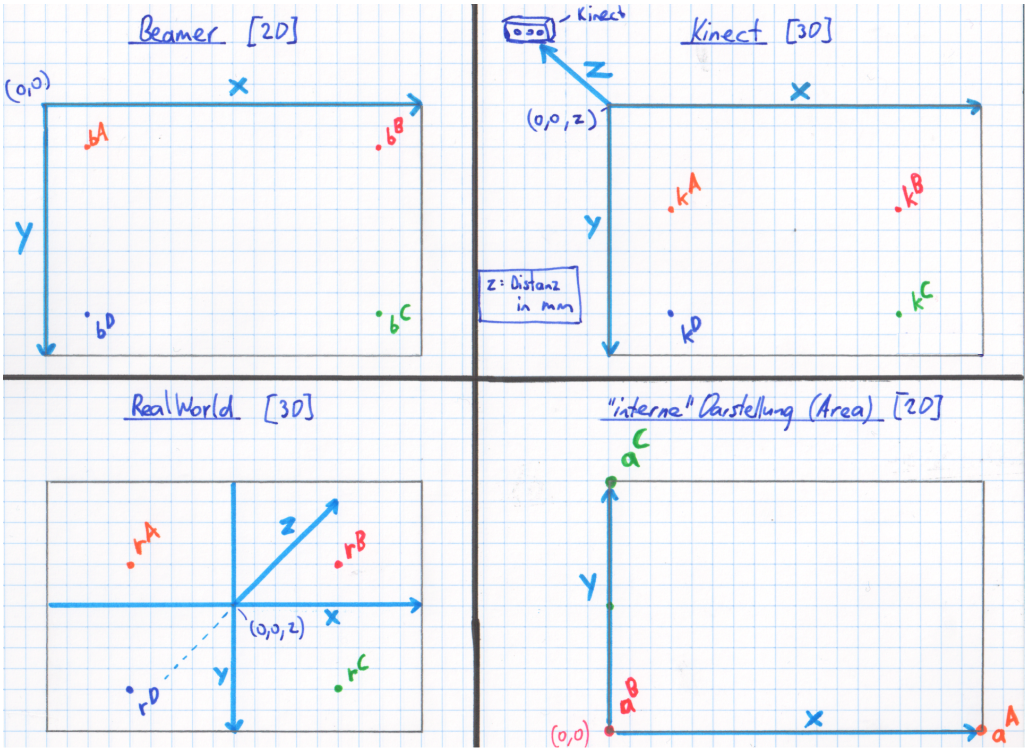


Abbildung 4.7: Vergleich der Koordinatensysteme

4.1.7. Lösungsansatz

Unser Lösungsansatz ist es die verschiedenen Koordinatensysteme miteinander zu synchronisieren. Wird beispielsweise ein Kreis in die Area Fläche gezeichnet, muss die Software in der Lage sein, die entsprechenden Punkten in den Beamer Koordinaten zu finden.

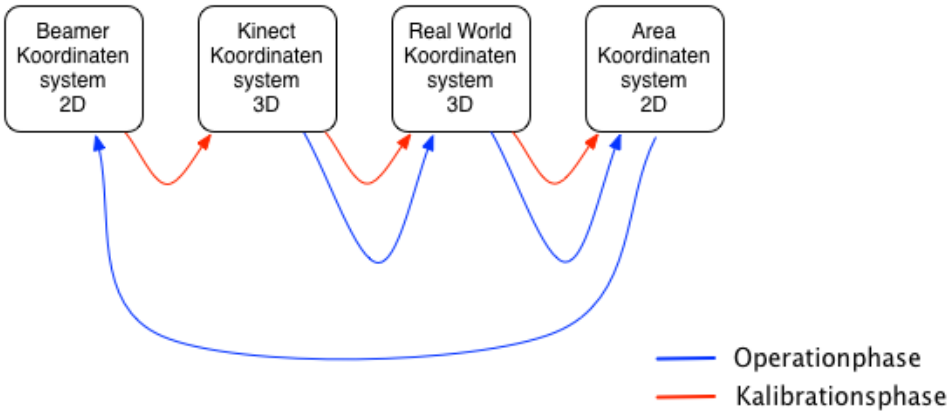


Abbildung 4.8: Koordinatensystem-Synchronisation

In der Kalibrationsphase werden alle Koordinatensysteme systematisch synchronisiert. Dies bildet die Grundlage um in der Operationsphase Punkte in allen Koordinatensystemen bestimmen zu können.

4.1.8. Arbeitsaufteilung

Zu Beginn unserer Arbeit haben wir die Problemstellung in die zwei folgenden Teilbereiche aufgeteilt:

1. Kinect
    - Differenzbilder zur Erkennung der Eckpunkte
    - Erfassen der 3D-Punkte der Kinect
    - Umwandeln der erkannten Punkte in RealWorld- sowie Area-Punkte
2. Beamer
    - Entwickeln eines optimalen Kalibrationsbildes

- Erstellen des zu projizierenden Beamerbildes
- gewünschtes Bild- an Projektor senden

Der Kinect-Teil wurde von Kevin Vogt bearbeitet. Engelbert Lüchinger widmete sich dem Beamer-Teil.

Die verwendeten Algorithmen wurden soweit möglich diesen zwei Teilbereichen zugeteilt. Bei einer Vielzahl der Algorithmen wurde der Entwurf zusammen erarbeitet und anschliessend individuell implementiert.

Generell haben wir uns jeweils von Montag bis Donnerstag genügend Zeit für die Studienarbeit reserviert. An diesen Tagen wurde jeweils in einem kurzen Update-Meeting der aktuelle Stand untereinander besprochen. Jeden Mittwoch fand zusätzlich ein Meeting mit Professor Oliver Augenstein statt.

4.2. Entwurf

4.2.1. Software-Architektur

Um die Software möglichst flexibel gestalten zu können, haben wir sie in Komponenten aufgeteilt. Dabei hat jede Komponente ihren eigenen Zweck zu erfüllen.

Komponente	Beschreibung
Calibration	Enthält alle Algorithmen für die Kalibration. Stellt die Verbindung zu den anderen Komponenten sicher. Ist die einzige Schnittstelle um mit der Software von aussen zu kommunizieren.
Kinect	Eine Abstraktion der Kinect. Stellt alle grundlegenden Funktionen bereit um die Kinect zu steuern.
Beamer	Ähnlich der Kinect, kapselt diese Komponente die Steuerung des Beamers.

Tabelle 4.1: Softwarekomponenten

4.2.2. Designpatterns

Um die Anforderungen an die Softwarequalität zu erfüllen, haben wir einige aus dem Software Engineering bekannten Gang of Four Patterns eingesetzt:

Facade Pattern
Dem Benutzer bieten wir zwei verschiedene Kalibrierungsfacaden an. Mit der Autokalibrierungsfacade wird eine vollumfängliche Kalibrierung der Kinect und des Beamers ausgeführt, bei der die optimalsten von uns implementierten Algorithmen verwendet werden. Als Alternative hat der Benutzer die Möglichkeit auf die Kalibrierung Einfluss zu nehmen, indem er die Kalibrierung Schritt für Schritt durchführt. Um dies erreichen zu können, stellen wir eine eigene Facade Klasse zur Verfügung.

Strategy Pattern
Damit die Algorithmen austauschbar sind, eignet sich das Strategy Pattern bestens. Dies ermöglicht dem Benutzer eigene Algorithmen zu entwickeln und mittels Dependency Injection in unserer Software zu nutzen.

4.2.3. Mannigfaltigkeit

Das Prinzip der Mannigfaltigkeit wurde uns von Professor Oliver Augenstein vorgestellt, um ein Durcheinander aufgrund der verschiedenen Koordinatensystemen zu vermeiden. Eine genauere Definition findet sich unter Wikipedia [WikiManni]. Zusammengefasst besagt das Prinzip, dass ein globaler Raum mit Hilfe von Karten betrachtet werden kann (Analogie: Erdkugel zu

Landkarten). In unserem Fall beschreiben Punkte den besagten globalen Raum. Die diversen Koordinatensysteme stellen dabei die Karten dar, mit denen die Punkte betrachtet werden können. Dies ermöglicht es uns nun immer von einem bestimmten Punkt zu sprechen, ohne dass er die globale Identität verliert. Die Punkte werden wie folgt beschrieben.

Bezeichnung	**Typ**	**Beispiele**
Punkt	globaler Raum	A, B, C, D
Beamer Punkt	Karte	b^A, b^B, b^C, b^D
Kinect Punkt	Karte	k^A, k^B, k^C, k^D
Real World Punkt	Karte	r^A, r^B, r^C, r^D
Area Punkt	Karte	a^A, a^B, a^C, a^D

Tabelle 4.2: Mannigfaltigkeit

Nachfolgend werden wir uns an diese Konvention halten. Somit ist immer klar ersichtlich von welchen Punkten gesprochen wird.

4.2.4. Anordnung der Eckpunkte

Um das Betrachterproblem zu lösen, muss in jedem Koordinatensystem ein Eckpunkt eindeutig identifizierbar sein. Deshalb haben wir und entschieden die Eckpunkte wie folgt anzuordnen.

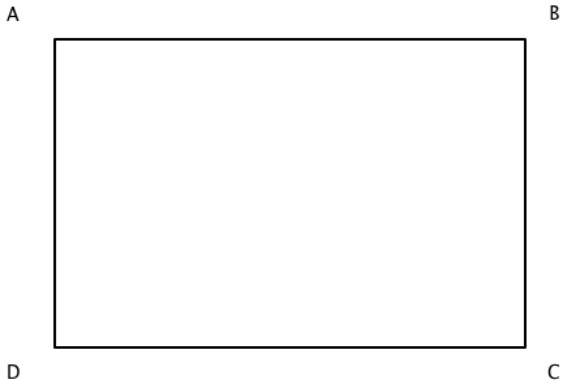


Abbildung 4.9: Anordnung der Eckpunkte

Da das erste Koordinatensystem in der Kalibrierungsphase das Beamer Koordinatensystem ist, kommt dieses Konzept dort schon zum Zug.

4.2.5. Algorithmen

4.2.5.1. Differenz-Bild berechnen

Eine der grundlegendsten Fragen, welche uns zu Beginn unserer Arbeit beschäftigt hat, befasste sich damit, wie wir das projizierte Bild des Beamers erkennen bzw. von der Umgebung unterscheiden können. Uns wurde schnell klar, dass wir dieses Problem mittels Bildbearbeitungstechniken lösen müssen, da wir uns nur auf die Farb-Kamera der Kinect stützen können. Der Tiefensensor hilft uns bei dieser Problematik nämlich nicht weiter, da das projizierte Bild ja dieselbe Distanz wie der Boden hat. Während der Recherche sind wir auf die Technik "Differenzbilder" gestossen. Dabei geht es darum zwei Bilder auf deren Unterschiede hin zu untersuchen. Für uns bedeutet dies, dass wir zwei Bilder benötigen, welche sich nur im projizierten Bild des Beamers unterscheiden. Dies kann auf verschiedene Art und Weise erreicht werden.

Variante	**1. Bild**	**2.Bild**
1	kein Bild	einfarbiges Bild
2	einfarbiges Bild (z.B. schwarz)	Anderes einfarbiges Bild mit genügend Unterschieden im RGB-Wert (z.B. weiss)
3	Schachbrettmuster (z.B. schwarz/weiss)	Invertiertes Schachbrettmuster (z.B. weiss/schwarz)
4	Nur die Ecken detektieren: Kleines Schachbrettmuster in den Ecken des Bildes (z.B. schwarz/weiss)	Nur die Ecken detektieren: Invertiertes Schachbrettmuster in den Ecken (z.B. weiss/schwarz)

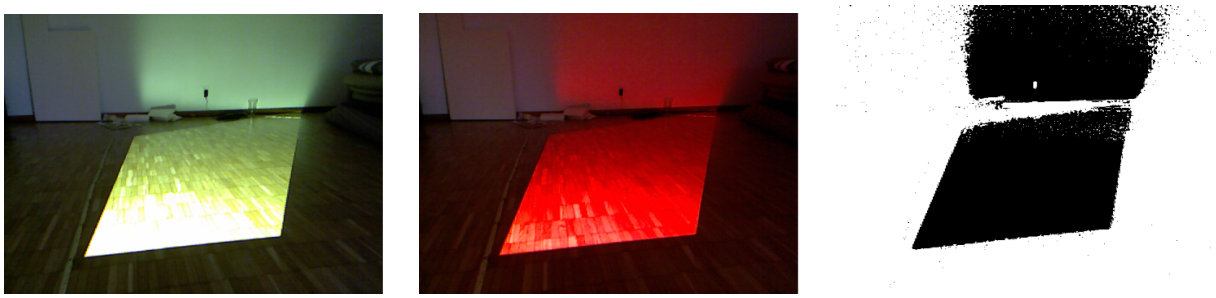
Tabelle 4.3: Differenz-Bilder: Mögliche Bildkombinationen

Die oben aufgelisteten Varianten bringen jeweils Vor- und Nachteile mit sich. Deshalb mussten wir diese mittels Experimenten testen und bewerten. Dabei sind wir zu folgenden Erkenntnissen gekommen:

Variante 1 ist mit grosser Wahrscheinlichkeit die Variante, welche man als Erstes ausprobieren wird. Leider funktioniert diese nicht in allen Fällen, da die Kinect-Sensoren automatisch die Lichteinflüsse, welche auf das Bild wirken, korrigieren. Hier haben wir das Problem, dass die Helligkeit der beiden Bilder nicht übereinstimmt, da der Beamer ja eine weitere Lichtquelle darstellt. Im 1. Bild, wo der Beamer gar nichts projiziert, herrschen daher andere Lichtverhältnisse. Aus Variante 1 konnten wir den Schluss ziehen, dass wir den Beamer idealerweise in beiden Bildern miteinbeziehen sollten.

Bei Variante 2 projizierten wir abwechslungsweise zwei Farben, welche sich in ihren Farbwerten möglichst markant unterscheiden. Die Kombination schwarz/weiss hat den grössten Unterschied, liefert aber in dieser Konstellation dieselben Ergebnisse wie Variante 1, da komplett schwarz in etwa der Situation "Kein Bild" entspricht. Wir versuchten es mit unterschiedlichen Farbkombi-

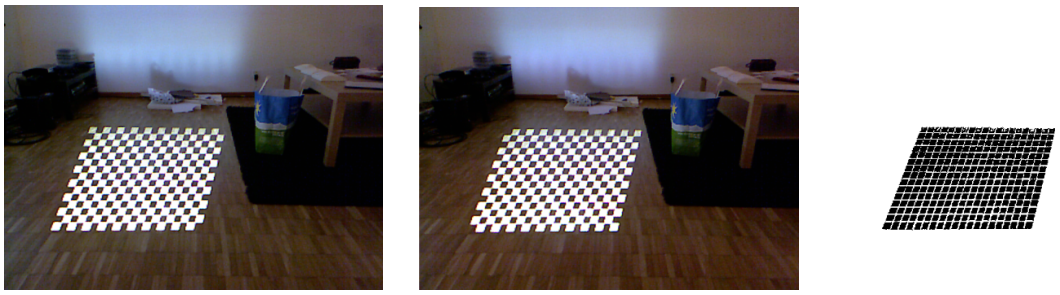
nationen. Einige funktionierten besser als andere, aber dennoch konnten wir keine konstanten Ergebnisse erzielen.



(a) 1. Bild (b) 2. Bild (c) Differenz-Bild

Abbildung 4.10: Differenz-Bilder - Variante 2

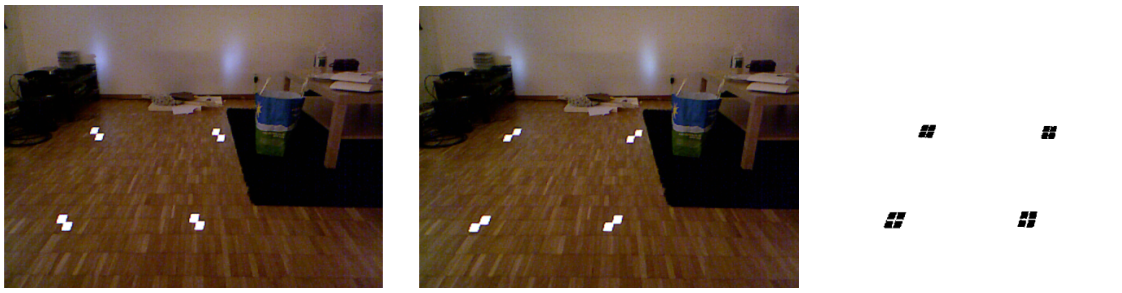
Für Variante 3 haben wir nochmals die Überlegung aufgegriffen, dass schwarz und weiss den grössten farblichen Unterschied in allen RGB-Werten aufweisen. Da aber ein komplettes schwarzes Bild Probleme mit der Autokorrektur verursachte, haben wir es mit verschieden grossen Schachbrettmustern versucht und festgestellt, dass es generell sehr gut funktioniert.



(a) 1. Bild (b) 2. Bild (c) Differenz-Bild

Abbildung 4.11: Differenz-Bilder - Variante 3

Variante 4 ging aus den Ergebnissen aller 3 anderen Varianten hervor. Es gibt nämlich Probleme, wenn man mit dem Beamer komplette Bilder projiziert, einen leicht spiegelnden Untergrund hat und sich in unmittelbarer Nähe eine Wand befindet. Das Beamer-Bild erzeugt dann an der Wand eine Reflektion, welche meistens auch als Differenz erkannt wurde. Da unsere Auto-Kalibration bei den unterschiedlichsten Bodenbelägen und in allen genügend grossen Räumen funktionieren sollte, haben wir uns überlegt, ob es nicht ausreicht, lediglich die Eckpunkte des Vierecks zu detektieren. Dies führte schliesslich dazu, dass wir nur noch kleine Schachbrettmuster in den Ecken des Beamer-Bildes anzeigen lassen.



(a) 1. Bild (b) 2. Bild (c) Differenz-Bild

Abbildung 4.12: Differenz-Bilder - Variante 4

Thresholding:

Nun wissen wir, welche Konstellationen sich für das Berechnen des Differenzbildes eignen. In diesem Abschnitt erläutern wir, mit welcher Technik das Differenzbild generiert wird. Das Stichwort dafür heisst "Thresholding". Der Threshold wird benötigt, um dem Algorithmus zu sagen, ab welchem Farbunterschied sich wirklich etwas im Ursprung-Bild geändert hat. Bei kleinen Abweichungen handelt es sich meist um Ungenauigkeiten des Kinect-Sensors oder generell um ein Rauschen im Bild. Kurz gesagt, ändert sich die Farbe eines Pixels mehr als der spezifizierte Threshold, so muss es sich um eine tatsächliche Änderung handeln.

1. Ansatz: Differenz der RGB-Werte eines Pixels

Unser erste Gedanke war, dass wir einfach Pixel für Pixel der beiden Bilder auf eine genügend grosse Abweichung in allen Farbwerten, also Rot, Grün und Blau vergleichen. Diesen Ansatz haben wir während den Recherchen in diversen Quellen gefunden [HaKin]. Zu Beginn hat dieser auch völlig ausgereicht, da wir jeweils mit schwarz und weiss, welche sich ja bekanntlich in allen RGB-Werten unterscheiden, getestet haben. Zudem haben wir diese Methode meist geprüft, indem wir für das zweite Bild Objekte (z.B. Bücher, Blätter usw.) als "Differenz" verwendet haben. Als wir weitere Tests mit Hilfe des Beamers durchgeführt haben, stellten wir fest, dass dieser Ansatz nicht für alle Farb-Kombinationen (z.B. grün und rot) geeignet ist. Dies veranlasste uns dazu, einen weiteren Ansatz auszuarbeiten.

2. Ansatz: vektorielle Differenz

Der zweite Ansatz basiert auf der Idee, dass jede Farbe des RGB-Farbraumes als 3D-Vektor ausgedrückt werden kann. Dabei stehen die Koordinaten x/y/z für die Zahlenwerte des R/G/B-Wertes. Man kann sich also den RGB-Farbraum als Würfel vorstellen. Nun lässt sich für jede Farbe ein Vektor innerhalb dieses Würfels finden (z.B. Schwarz = (0,0,0) und Weiss = (255,255,255)). Wie bereits weiter oben erläutert bilden die zwei Farben Schwarz und Weiss die grösste Differenz, was in diesem Koordinatensystem der Diagonale des Würfels entspricht. Die Differenz zweier Farben wird mittels einer gewöhnlichen Vektor-Subtraktion berechnet. Danach kann die Länge des Differenz-Vektors berechnet werden und anhand dieser wird entschieden, ob

der angegebene Threshold überstiegen wurde. Trifft dies zu, so hat sich die Farbe des Pixels wirklich geändert und es wird im Differenz-Bild auf schwarz gesetzt. Wenn nicht, wird das Pixel auf weiss gesetzt. Dabei entspricht die Farbe Schwarz einer Änderung.

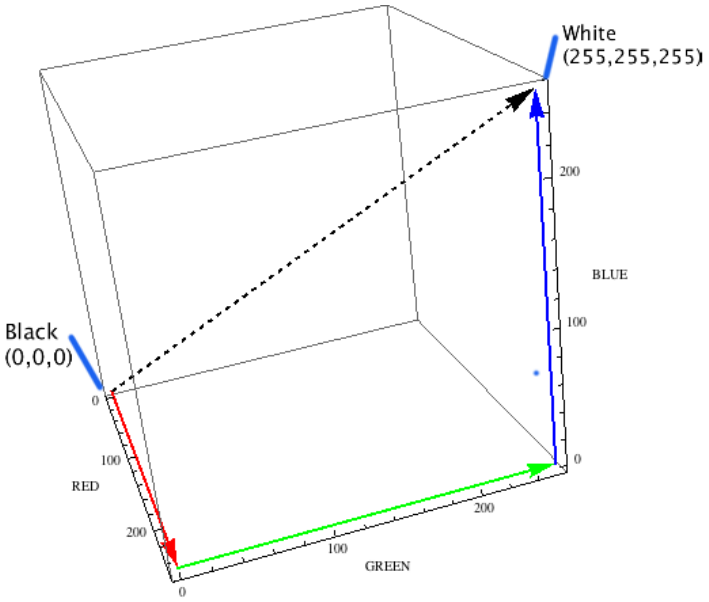
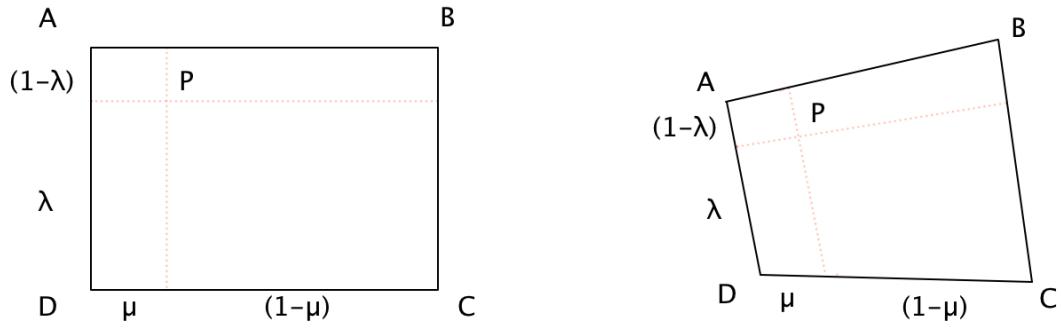


Abbildung 4.13: Prinzip - vektorielle Differenz

4.2.5.2. Baryzentrische Koordinaten

Mehrmals kommt es vor, dass Punkte aus einem kartesischen Koordinatensystem in ein anderes kartesisches Koordinatensystem umgerechnet werden müssen. Um das zu erreichen, stützen wir uns auf die Eigenschaften von baryzentrischen Koordinaten.

Für unseren Ansatz müssen in beiden Koordinatensystemen mindestens vier Punkt gegeben sein.



(a) Beamer Koordinaten

(b) Kinect Koordinaten

Abbildung 4.14: Beamer zu Kinect Koordinaten

Daraus lässt sich folgender Satz herleiten, der für beide Koordinatensysteme gilt.

$$\vec{P} = (1-\lambda)(1-\mu)\vec{D} + (1-\lambda)\mu\vec{C} + \lambda(1-\mu)\vec{A} + \lambda\mu\vec{B} \tag{4.1}$$

Diese Formel ist in jedem Koordinatensystem gültig.

Ist in einem Koordinatensystem ein Punkt geben, kann mit der Determinanten Gleichung μ und λ berechnen:

$$det((1-\mu)\vec{D} + \mu\vec{C} - \vec{P}, (1-\mu)\vec{A} + \mu\vec{B} - \vec{P}) = 0 \tag{4.2}$$

$$det((1-\lambda)\vec{D} + \lambda\vec{A} - \vec{P}, (1-\lambda)\vec{C} + \lambda\vec{B} - \vec{P}) = 0 \tag{4.3}$$

4.2.5.3. k-Means

Um die projizierte Fläche, im Kinectbild, erkennen zu können, haben wir entschieden den k-Means-Algorithmus anzuwenden. Eine genaue Beschreibung des Algorithmus ist in Wikipedia [WikiKMeans] zu finden.

Aus dem Differenzbild, welches wir zuvor erzeugen, ergibt sich nun in den Ecken eine Anhäufung von schwarzen Pixeln, welche auch als Cluster angesehen werden können. Nun kann mit dem k-Means-Algorithmus der Schwerpunkt des Clusters ermittelt werden. Dabei berechnen wir den Schwerpunkt wie folgt:

$$\overrightarrow{k^C} = \frac{\sum_{i=0}^{n} \overrightarrow{k_i^P}}{n} \tag{4.4}$$

Abbildung 4.15: k-Means Algorithmus visualisiert

Der Vorteil dieser Vorgehensweise ist, dass sich der Schwerpunkt, auch bei verstreuten Fehler im Differenzbild, kaum verschiebt.

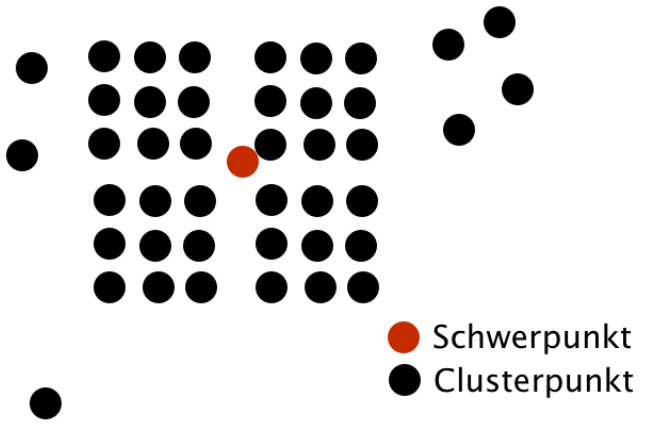


Abbildung 4.16: k-Means Algorithmus mit Fehler

4.2.5.4. Umrechnung der X/Y-Koordinaten von Pixel in Millimeter

Die Kinect bringt ein weiteres Problem mit sich. Der Depth Image Stream liefert die z-Koordinaten in Millimeter, die x-/y-Koordinaten jedoch in Pixel. Für den bevorstehenden Basiswechsel benötigen wir aber zwingenderweise Koordinaten, welche alle die gleiche Dimension aufweisen. Deshalb mussten wir eine Umrechnung von Pixel in Millimeter implementieren.

Nach dem Erstellen des KinectPoint-Arrays haben wir die Situation, dass die Koordinaten eines KinectPoints folgende Form haben:

$$k^P = \begin{pmatrix} k_x^P[Pixel] \\ k_y^P[Pixel] \\ k_z^P[mm] \end{pmatrix}$$

Um dieses Problem lösen zu können, blieb uns keine andere Wahl als zu bestimmen, welche Grösse in Millimeter denn ein Pixel überhaupt hat. Damit wir das gewünschte Ergebnis erhalten, verfolgten wir zwei unterschiedliche Ansätze, nämlich "Bestimmung durch Ausmessen" sowie "Bestimmung mit Hilfe der Mathematik".

Ansatz 1: Bestimmung durch Ausmessen:

Die Idee dieses Ansatzes ist, dass wir den Bereich, welcher der Depth Image Stream der Kinect abtasten kann, ausmessen. Dabei kommt es natürlich darauf an, wie weit die Kinect von der Wand entfernt ist. Da wir wissen, dass der Depth Image Stream stets ein Bild von 640x480 Pixel zurückliefert, wissen wir, dass die Pixel bei zunehmender Distanz immer grösser werden. Pixel, welche eine Tiefe von 0.8 m besitzen, sind in Millimeter ausgedrückt kleiner als Pixel, welche 4m

von der Kinect entfernt sind. Da der Range der Kinect im normalen Modus von 0.8 m bis 4 m reicht, mussten wir bei den Ausmessungen nur Tiefen innerhalb dieses Ranges berücksichtigen.

Folgende Formeln wurden zur Bestimmung der gesuchten Konstanten verwendet.

Verhältnis Breite-Tiefe: $\frac{640 \cdot z}{s}$

Verhältnis Höhe-Tiefe: $\frac{480 \cdot z}{t}$

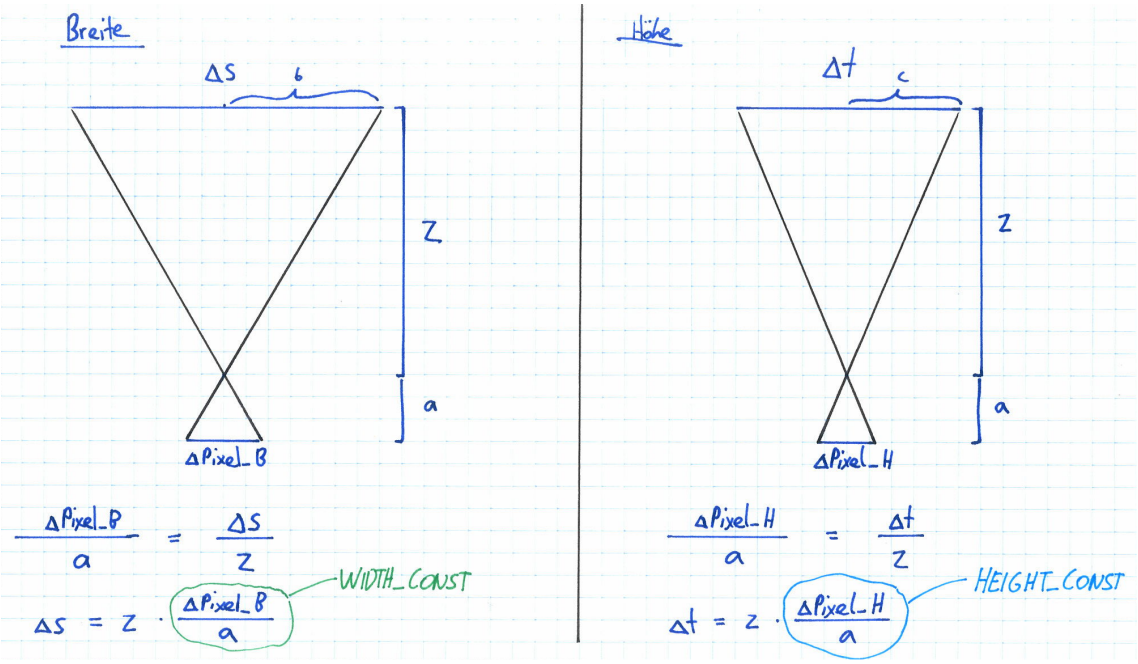


Abbildung 4.17: Pixel in mm - Bestimmung durch Ausmessen

Da die Werte der einzelnen Messungen mit den verschieden gewählten Tiefen schwach abweichende Resultate liefern, nehmen wir deren Mittelwert als Referenz. Dieser Mess-Ansatz liefert uns also einen Referenz-Wert sowohl für die Breite als auch einen für die Höhe:

```
WIDTH_CONST:   544.945
HEIGHT_CONST:  585.258
```

Um diese Konstanten weiter zu verbessern, wäre es denkbar noch weitere Messpunkte zu definieren und in die Messreihe aufzunehmen. Dies würde dann zu optimaleren Ergebnissen in der Pixel-Millimeter-Umrechnung führen. Vorerst reichen uns aber die bisher bestimmten Werte.
Der Versuchsaufbau sowie die Messergebnisse werden im Anhang A erläutert.

Ansatz 2: Bestimmung mit Hilfe der Mathematik:

Der mathematische Ansatz basiert auf einfachsten trigonometrischen Formeln. Da wir von verschiedenen Quellen wissen, dass die Kinect-Kamera horizontal einen Winkel von 57 Grad und vertikal einen Winkel von 43 Grad besitzt, können wir Formeln zur Berechnung des erkennbaren Tiefenbildes aufstellen [BKPwMK]. Dabei verwenden wir die Symbole wie in der Skizze ersichtlich.

Breite: $s = 2 \cdot z \cdot \tan(28.5)$

Höhe: $t = 2 \cdot z \cdot \tan(21.5)$

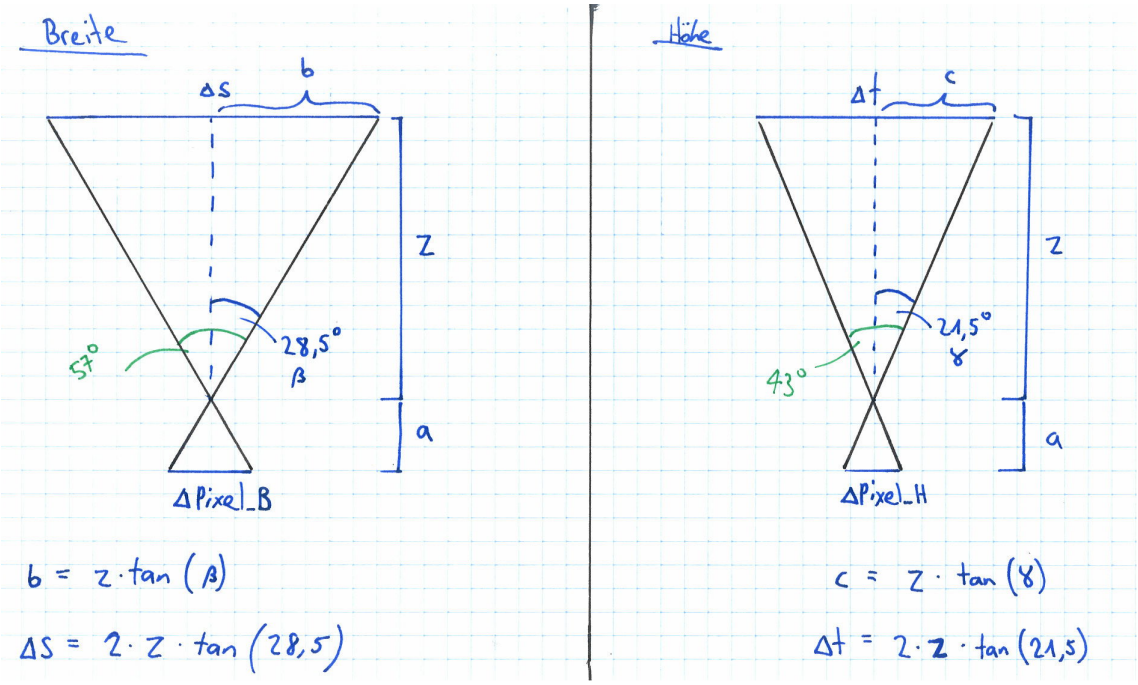


Abbildung 4.18: Pixel in mm - mathematischer Ansatz

Diese Formeln liefern uns die erkennbare Breite und Höhe in Millimetern. Da wir wiederum wissen, dass das erkannte Bild aus 640x480 Pixel besteht, können wir also die Ausmessungen eines einzigen Pixels bestimmen. Damit lässt sich herleiten, wie das Verhältnis der Breite bzw. der Höhe in Bezug auf die Tiefe aussieht.

```
WIDTH_CONST:  589.367
HEIGHT_CONST: 609.275
```

Mögliche Optimierung des mathematischen Ansatzes:

Im weiteren Verlauf unserer Arbeit sind wir auf eine MSDN-Seite [MSDNConst] gestossen, welche System-Konstanten der Kinect beschreibt. Interessanterweise werden dort zwei Konstanten erwähnt.

```
NUI_CAMERA_DEPTH_NOMINAL_HORIZONTAL_FOV: 58.5
NUI_CAMERA_DEPTH_NOMINAL_VERTICAL_FOV:   45.6
```

Dabei steht "FOV" für "Field of View" und entspricht den Winkeln der Kinect-Kamera. Diese Konstanten weisen also eine geringe Abweichung von den sonst überall verwendeten Winkeln 57 und 43 Grad auf. Die oben erwähnten Formeln geben für diese Winkel folgende Werte:

```
WIDTH_CONST:  571.401
HEIGHT_CONST: 570.937
```

Es ist deutlich erkennbar, dass diese Werte weniger von den ausgemessenen Werten abweichen. Möglicherweise könnten diese Winkel-Konstanten einen entscheidenden Hinweis auf die Optimierung unserer Konstanten WIDTH_CONST und HEIGHT_CONST beisteuern.

Fazit und Lösungsansatz:

Es ist sehr gut ersichtlich, dass die Werte des Berechnung-Ansatzes ein wenig vom Verhältnis der ausgemessenen Punkte abweichen. Dies könnte unter anderem daran liegen, dass die spezifizierten Winkel der Kinect-Kamera je nach Modell von den 57 bzw. 43 Grad abweichen. Natürlich sind auch Messfehler im ersten Ansatz nicht auszuschliessen. Allgemein bringt die Thematik das Problem mit sich, dass unterschiedliche Kinect-Modelle auch unterschiedliche Werte liefern. Dieses Problem ignorieren wir zu diesem Zeitpunkt unter der Annahme, dass die Werte sich nicht signifikant unterscheiden und somit doch ein brauchbares Ergebnis erwartet werden kann.

Auf einer Website der Technischen Universität München [TUMCalCam] findet man eine hergeleitete Parameter-Tabelle. An dieser haben wir uns in dem Sinn orientiert, dass unsere gemessenen Werte näher an den von ihnen verwendeten Parameter liegen als unsere berechneten.

Dies war einer der Hauptgründe, weshalb wir vorerst für die Auto-Kalibration die Werte unseres ersten Ansatzes gewählt haben. Sollten diese Werte nicht die gewünschten Ergebnisse mit sich bringen, so sind auf jeden Fall weitere Untersuchungen zu tätigen. Wie bereits im obigen Abschnitt erläutert, wären zusätzliche Messreihen der erste Schritt für die Optimierung. Des Weiteren wäre eine Kombination aus den ausgemessenen und den berechneten Konstanten denkbar.

Die aus diesem Problem resultierenden Konstanten werden nun für das Umrechnen eines KinectPoint-Arrays in ein sogenanntes RealWorld-Array verwendet. "RealWorld" aus dem Grund, dass die Koordinaten der KinectPoints in einer einheitlichen, realitätsnaher Dimension, nämlich Millimeter, vorliegen. Basierend auf diesem neu kreierten Array lässt dann zu einem späteren Zeitpunkt der Basiswechsel vollziehen.

Die eigentliche Umrechnung der Koordinaten geschieht mittels folgenden Formeln. Dabei steht WIDTH und HEIGHT für die Breite bzw. Höhe des Kinect-Bildes.

`WIDTH =`	640 Pixel
`HEIGHT =`	480 Pixel
`WIDTH_CONST =`	544.945
`HEIGHT_CONST =`	585.258

$$r_x^P = \left(k_x^P - \frac{WIDTH}{2}\right) \cdot \frac{k_z^P}{WIDTH\_CONST} \tag{4.5}$$

$$r_y^P = \left(k_y^P - \frac{HEIGHT}{2}\right) \cdot \frac{k_z^P}{HEIGHT\_CONST} \tag{4.6}$$

$$r_z^P = k_z^P \tag{4.7}$$

r_x^P, r_y^P sowie r_z^P haben jeweils die Einheit Millimeter.

Optimierung: Berechnung der Z-Werte

Die oben hergeleitete Formel liefert unter Umständen falsche x/y-Koordinaten basierend auf der z-Koordinate, sobald letztere einen ungültigen Wert hat.

Ein erster Optimierungsansatz, welchen wir implementiert haben, basiert auf dem Wissen, dass die Kinect ein Lockkamera-Modell darstellt. Dies ermöglicht uns mit nur 3 bekannten RealWorld-Punkten eine Ebene aufzuspannen, auf welcher alle weiteren Punkte des KinectPoint-Arrays nach der Transformation in RealWorld-Punkte liegen sollten bzw. müssen. Für uns bedeutet dies, dass wir zuerst die vier Eckpunkte des Beamer-Bildes wie gehabt mit den weiter oben erläuterten Formeln (4.5), (4.6) und (4.7)berechnen. Dabei ist es unerlässlich, dass diese Punkte einen validen z-Wert besitzen. Diese Problematik wird in 4.2.5.6 behandelt.

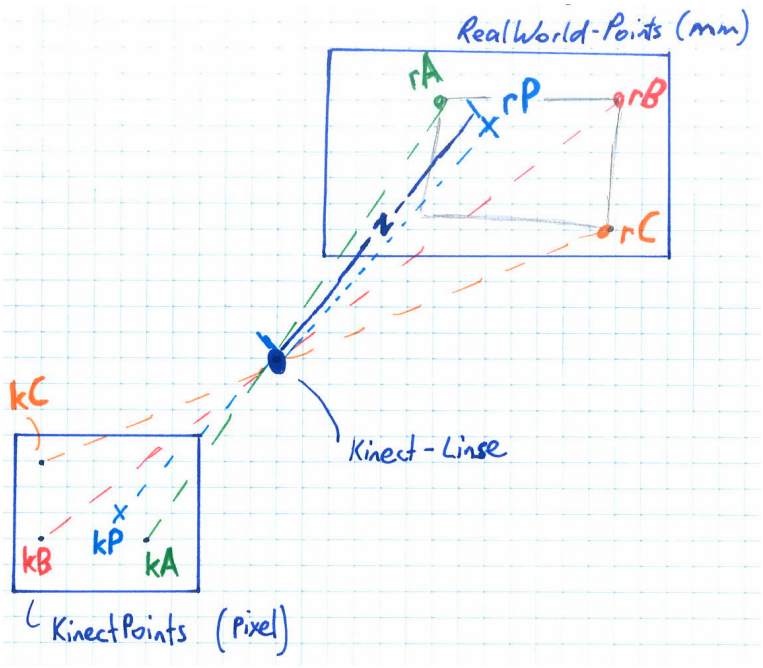


Abbildung 4.19: Lochkamera-Modell als Ansatz für die Berechnung von z

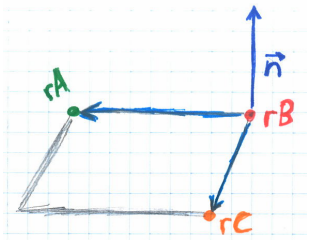


Abbildung 4.20: Z-Berechnung: rA, rB und rC spannen eine Ebene auf

Z-Werte der weiteren Punkte berechnen:
Erst jetzt ändert sich unsere Vorgehensweise. Anstatt weitere Kinect-Punkte mit den gleichen Formeln (4.5), (4.6) sowie (4.7) in RealWorld-Punkte umzurechnen, verwenden wir jetzt neue Formeln basierend auf geometrischen Berechnungen innerhalb des Lochkamera-Modells (Abb. 4.19). Dies heisst konkret, dass die weiteren Punkte aufgrund von drei der vier bereits berechneten Eckpunkte berechnet werden können.

Dabei haben wir folgende Formeln hergeleitet. k^P entspricht dabei dem KinectPoint des Punktes P, welcher in die RealWorld-Koordinaten umgerechnet werden soll.

$$k^P = \begin{pmatrix} k_x^P[Pixel] \\ k_y^P[Pixel] \\ k_z^P[mm] \end{pmatrix}$$

$$\vec{n} = (\overrightarrow{r^A} - \overrightarrow{r^B}) \times (\overrightarrow{r^C} - \overrightarrow{r^B}) \tag{4.8}$$

$$q = \vec{n} \cdot \overrightarrow{r^B} \tag{4.9}$$

$$x = \frac{k_x^P - \frac{WIDTH}{2}}{WIDTH\_CONST} \tag{4.10}$$

$$y = \frac{k_y^P - \frac{HEIGHT}{2}}{HEIGHT\_CONST} \tag{4.11}$$

Aus dem x und dem y ergibt sich ein neuer Vektor: $\overrightarrow{r^K} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

$$r_z^P = \frac{q}{\vec{n} \cdot \overrightarrow{r^K}} \tag{4.12}$$

$$r_x^P = \left(k_x^P - \frac{WIDTH}{2}\right) \cdot \frac{r_z^P}{WIDTH\_CONST} \tag{4.13}$$

$$r_y^P = \left(k_y^P - \frac{HEIGHT}{2}\right) \cdot \frac{r_z^P}{HEIGHT\_CONST} \tag{4.14}$$

Dabei ist zu beachten, dass die Formeln für die x- und die y-Komponente gleich geblieben sind. Lediglich der Z-Wert ist unterschiedlich hergeleitet worden. In Formel (4.5) und (4.6) ist es k_z^P und neuerdings ist es der zuvor berechnete r_z^P.

Weiterführende Optimierung:

Es wäre noch eine weitere Optimierung denkbar. Der Algorithmus, welchen wir momentan verwenden, benötigt lediglich 3 Punkte, um die Ebene aufzuspannen. Dies ist unter Umständen

sehr ungenau. In einem nächsten Schritt könnte man zwischen den z-Werten, welche die Kinect liefert, und den z-Werten, welche wir berechnen, eine Ausgleichsebene legen. Diese würde dann ein genaueres Resultat liefern, da die Fehler der beiden Ansätze durch das Ausgleichen minimiert werden können.

Diese Optimierung wurde bisher nur besprochen, jedoch noch nicht implementiert. Wir konnten aber den Vergleich zwischen den zwei unterschiedlichen Ansätzen, mit welchen wir an die z-Koordinaten gelangen, grafisch darstellen und nachvollziehen. Das Bild 4.21 illustriert diesen Vergleich. Es ist klar zu erkennen, wo das vom Beamer projizierte Bild liegt. Der Farbverlauf, welcher mehrere Graustufen umfasst, kommt wie folgt zu stande:

Als Ausgangsfarbe haben wir die Graustufe, RGB [128,128,128], gewählt, da diese in der Mitte des RGB-Farbraumes liegt und wir zwei Arten von Fehlern darstellen können müssen. Einerseits kann es sein, dass die Differenz zwischen gemessenem z-Wert und berechneter z-Wert positiv ist, andererseits kann diese aber auch negativ sein. Laut unserer Definition wird die Graustufe heller, wenn die Differenz positiv ist. Ist also der gemessene z-Wert grösser als der berechnete z-Wert, so wird das fokussierte Pixel proportional zur maximalen Abweichung aller z-Werte heller als die ursprüngliche Graustufe. Die maximale, positive Abweichung resultiert in der Farbe Weiss. Genau dieselben Regeln gelten für die negative Differenz. Dort wird der Farbverlauf jedoch dunkler und resultiert im Minimum in der Farbe Schwarz. Ist also der von der Kinect gemessene z-Wert kleiner als der berechnete z-Wert, so tritt die zweite Fallunterscheidung ein.

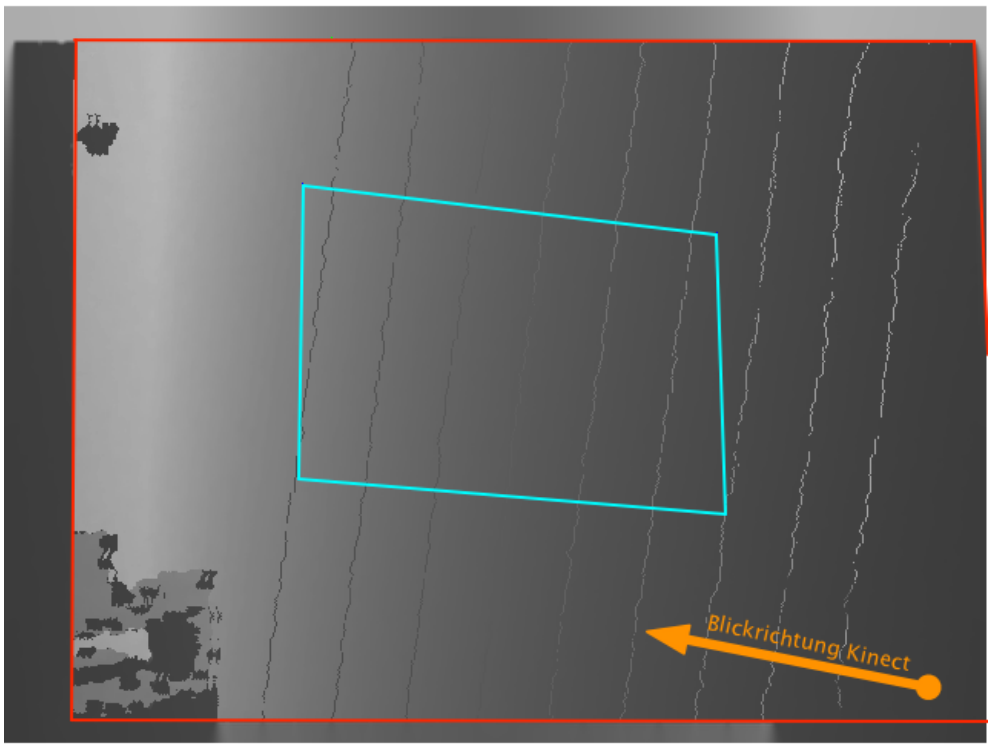


Abbildung 4.21: Analyse: Optimierungsmöglichkeit durch Ausgleichsebene

Anmerkung: Sowohl die Streifen, welche von unten nach oben durch das Bild verlaufen, als auch die Ränder können vernachlässigt werden, da diese Mapping-Fehler der Kinect-Kameras darstellen. Diese Problematik wird zu einem späteren Zeitpunkt noch genauer betrachtet.

Es ist erkennbar, dass in der Nähe der Kinect die Farben dunkler sind. Das heisst, dass dort die berechneten Werte grösser sind als die gemessenen. Da der Farbverlauf konstant von dunkel nach hell verläuft, ist zu vermuten, dass eine Ausgleichsebene tatsächlich eine Verbesserung mit sich bringen würde.

4.2.5.5. Basiswechsel

Der Basiswechsel wird benötigt um die von der Kinect erkannten Punkte in ein neues Koordinatensystem, welches das 2D-Spielfeld (Area) repräsentieren soll, umrechnen zu können. Für das 2D-Spielfeld wird nur das Beamer-Bild benötigt, welches die Kinect mittels der bereits oben erläuterten Thresholding-Technik erkennt. Um die Umrechnung durchführen zu können, benötigen wir mindestens 3 Punkte, welche im neuen Koordinatensystem vorhanden sein müssen. Dafür bieten sich die Eckpunkte des Beamer-Bildes an, da diese zudem die Spielfeldbegrenzung liefern.

Die nachfolgende Skizze zeigt das Zusammenspiel der beiden Koordinatensysteme "RealWorld" und "Area". Oben links befindet sich die Kinect, welche die Koordinaten x, y und z für alle erkannten Punkte liefert. Die z-Achse kommt gerade auf der Richtung des Sensors zu liegen. Die Punkte r^{P_1} bis r^{P_4} sollen ein mögliches Beamer-Bild darstellen. Um nun den eigentlichen Basiswechsel vorzunehmen, nehmen wir die 3 Punkte r^{P_1}, r^{P_2} sowie r^{P_4}. Unsere Überlegung ist, dass der Eckpunkt, welcher die zwei kürzesten Distanzen zu seinen direkten Nachbarpunkten aufweist, der beste Ursprungspunkt für unser neues Koordinatensystem ist. Dabei werden in diesem Fall nur die Distanzen zu den zwei Nachbarpunkten r^{P_2} und r^{P_4} betrachtet, da diese theoretisch einen rechten Winkel mit Ursprung r^{P_1} bilden. Wir sagen bewusst "theoretisch", weil sie aufgrund von Verzerrungen des Beamer-Bildes möglicherweise nicht als rechte Winkel wahrgenommen werden. Das Feld sollte jedoch immer als rechtwinklig gesehen werden. Schliesslich gibt es noch den Punkt X, welcher einen beliebigen von der Kinect gelieferten Punkt darstellt. Solche Punkte müssen von ihren 3D-Koordinaten mit Hilfe der Mathematik in das neue 2D-Koordinatensystem transformiert werden können.

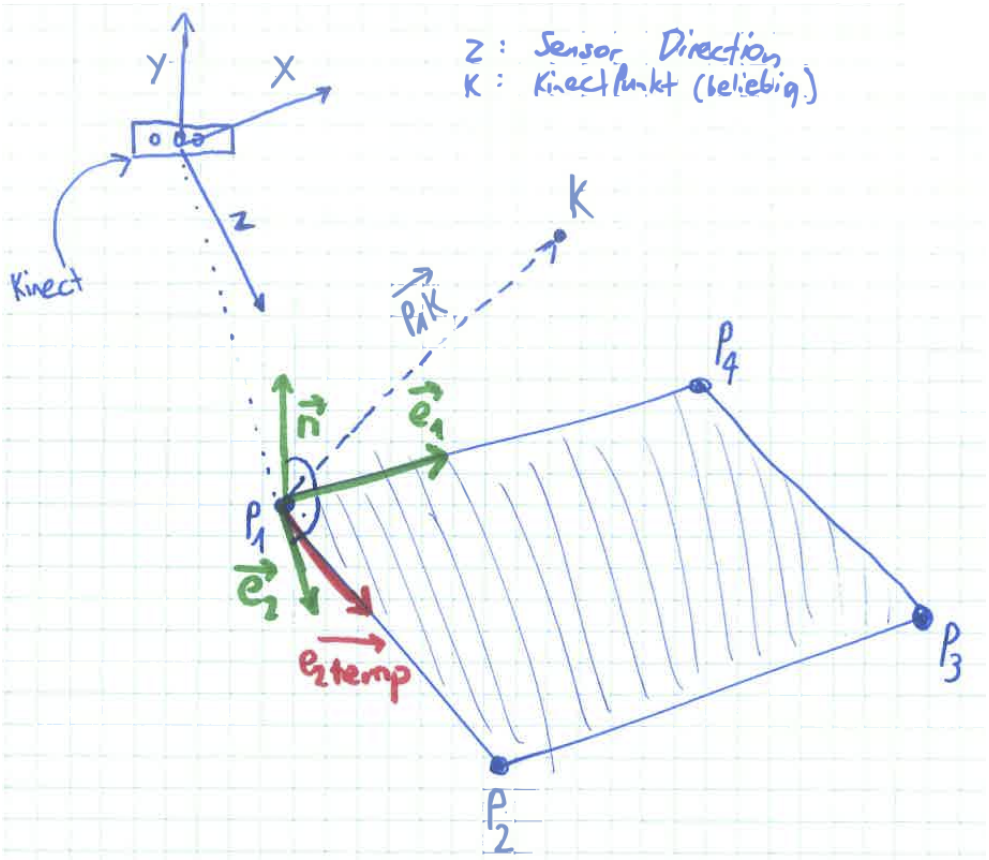


Abbildung 4.22: Skizze - Basiswechsel

Nachdem nun eine grafische Vorstellung der Situation vorhanden sein sollte, erläutern wir die mathematischen Berechnungen, welche zur gewünschten "internen" Darstellung führen. Wir teilen die Problematik in zwei Bereiche auf:

1. Initialisieren des neuen Koordinatensystems mittels Basiswechsel:

Die drei gewählten Punkte bilden eine Ebene. Um an das neue Koordinatensystem zu kommen, müssen wir als Erstes den Normalenvektor $\vec{n}$ zu dieser Ebene finden. Dies erreichen wir wie folgt:

$$\begin{aligned} \vec{e_1} &= r^{P_4} - r^{P_1} \\ \overrightarrow{e_2 temp} &= r^{P_2} - r^{P_1} \\ \vec{n} &= \vec{e_1} \times \overrightarrow{e_2 temp} \end{aligned} \tag{4.15}$$

In der obigen Formeln haben wir einen temporären Vektor $\overrightarrow{e_2 temp}$ verwendet. Der Grund dafür ist, dass es je nach projiziertem Bild sein kann, dass die eigentlichen Vektoren, welche die

erkannte Fläche umschliessen sollten, nicht als rechtwinklig erkannt werden, obwohl sie theoretisch rechtwinklig sind. Wir benötigen also diesen temporären Vektor, um in einem ersten Schritt den Normalenvektor $\vec{n}$ bestimmen zu können. Als Nächstes können wir mit Hilfe der Vektoren $\vec{n}$ und $\vec{e_1}$ den "korrigierten" Vektor $\vec{e_2}$ berechnen. Somit erhalten wir ein 3D-Koordinatensystem, dessen Achsen alle orthogonal zueinander liegen.

$$\vec{e_2} = \vec{e_1} \times \vec{n} \tag{4.16}$$

Um den Basiswechsel abzuschliessen, müssen die erhaltenen Vektoren $\vec{n}$, $\vec{e_1}$ und $\vec{e_2}$ noch normiert werden.

$$\begin{aligned} \vec{n_n} &= \frac{\vec{n}}{\sqrt{{n_x}^2 + {n_y}^2 + {n_z}^2}} \\ \vec{e_{1n}} &= \frac{\vec{e_1}}{\sqrt{{e_{1x}}^2 + {e_{1y}}^2 + {e_{1z}}^2}} \\ \vec{e_{2n}} &= \frac{\vec{e_2}}{\sqrt{{e_{2x}}^2 + {e_{2y}}^2 + {e_{2z}}^2}} \end{aligned} \tag{4.17}$$

Diese drei normierten Vektoren stellen nun das Grundgerüst unseres Koordinatensystems für die "interne" Darstellung dar. Im nächsten Abschnitt beschreiben wir das Vorgehen, mit welchem man RealWorld-Punkte in das neue Koordinatensystem transformieren kann.

2. Umrechnen eines RealWorld-Punktes → Area Punkt:

Gegeben sei ein RealWorld-Punkt r^K in der Form: $\vec{k} = r^K - r^{P_1} = \begin{pmatrix} r_x^K \\ r_y^K \\ r_z^K \end{pmatrix}$

Vektor $\vec{k}$ entspricht dabei dem Vektor, welcher von r^{P_1} zu r^K verläuft (siehe Abbildung 4.8). Die dem Punkt K entsprechenden Koordinaten in der "internen" Darstellung erhalten wir mit Hilfe der folgenden drei Formeln:

$$\begin{aligned} I_x &= \vec{e_{1n}} \cdot \vec{k} \\ I_y &= \vec{e_{2n}} \cdot \vec{k} \\ I_z &= \vec{n_n} \cdot \vec{k} \end{aligned} \tag{4.18}$$

4.2.5.6. Recover Missing Depth Information

Wie bereits bei der Vorstellung der Kinect-Streams erläutert, gibt es beim Depth Image Stream Bereiche (Ranges), welche fehlerhafte Tiefendaten liefern bzw. in welchen die Tiefendaten einfach nicht ermittelt werden können. Dabei wird zwischen drei unterschiedlichen Fehlertypen unterschieden [MSDNCoSp]:

Out-of-Range Typ	**zurückgelieferter Wert**
too near	0x0000 → 0
too far	0x0FFF → 4095
unknown	0x1FFF → -1 in managed code (sonst 8191)

Tabelle 4.4: Depth Image Stream - Out-of-Range Werte

Da wir aber mit den Tiefenwerte der einzelnen Punkten Berechnungen durchführen müssen, wie beispielsweise bei der Berechnung der x/y-Koordinaten in Millimeter, ist es sehr schlecht, wenn diese auf falschen Tiefenwerten basieren. Deshalb mussten wir einen Weg suchen diese Werte korrigieren zu können.

Nebst dem Problem, dass der Tiefensensor falsche Tiefenwerte liefern kann, gibt es noch ein weiteres. Beim Mapping des Farb- und des Tiefenbildes der Kinect gibt es vor allem am Rand des Bildes Punkte, welche nicht korrekt aufeinandergelegt werden können und deshalb entweder nur eine Farb- oder eine Tiefenkomponente aufweisen. Diese Punkte haben deshalb ebenfalls fehlerhafte Tiefenwerte und mussten bei der Erarbeitung des Algorithmus mitberücksichtigt werden.

Für das Entwerfen unseres Algorithmus haben wir die Annahme getroffen, dass benachbarte Punkte höchstwahrscheinlich ähnliche Tiefenwerte haben, d.h. die Abweichung sollte sehr gering sein. Zudem können wir annehmen, dass die Kinect bei der Anwendung unserer Software so verwendet wird, dass sie das Beamerbild zwingend komplett erkennen muss. Dies bedeutet wiederum, dass die Kinect etwa 70% der Punkte in einer Ebene sieht, was unsere Annahme unterstützt, dass benachbarte Punkte ähnliche Tiefendaten liefern. Problematisch sind eher die Konturen allfälliger 3D-Objekte, welche wir während unserer Arbeit nicht betrachten. Dieses Phänomen konnten wir bereits einige Male feststellen, da die von der SDK gelieferte Mapping-Funktion für die unterschiedlichen Bilder der Kinect ebenfalls Probleme bei den Konturen von dreidimensionalen Gegenständen hat.

Eine weitere Problematik liegt vor, wenn alle Nachbarn eines Punktes fehlerhafte Tiefenwerte haben. Dies veranlasste uns dazu einen Korrektur-Radius einzuführen, welcher es uns erlaubt, weitere, indirekte Nachbarn in die Berechnung einzubeziehen, sofern alle direkten Nachbarn keine gültigen Tiefendaten aufweisen.

Algorithmus:

Der erste Teil des Algorithmus sucht mit Hilfe des Korrektur-Radius so lange Nachbarn bis mindestens einer einen korrekten Tiefenwert liefert. Das heisst, dass der Korrektur-Radius beliebig erweitert werden kann, wenn keine gültigen Werte vorliegen. In der Regel findet aber der Algorithmus sehr schnell korrekte Werte. Die heiklen Stellen sind jeweils die Ränder des Kinect-Bildes aufgrund der Fehler der Mapping-Funktion. Innerhalb des Kinect-Bildes treten sehr wenig Fehler auf. Mittels einiger Tests konnten wir feststellen, dass etwa 15% aller Kinect-Punkte nicht gemapped werden können. Praktisch alle dieser Punkte sind Teile des Randes. Natürlich können auch dreidimensionale Gegenstände innerhalb des Bildes vorkommen. Da aber dort nur die Konturen problematisch sind, finden wir sehr schnell wieder valide Werte.

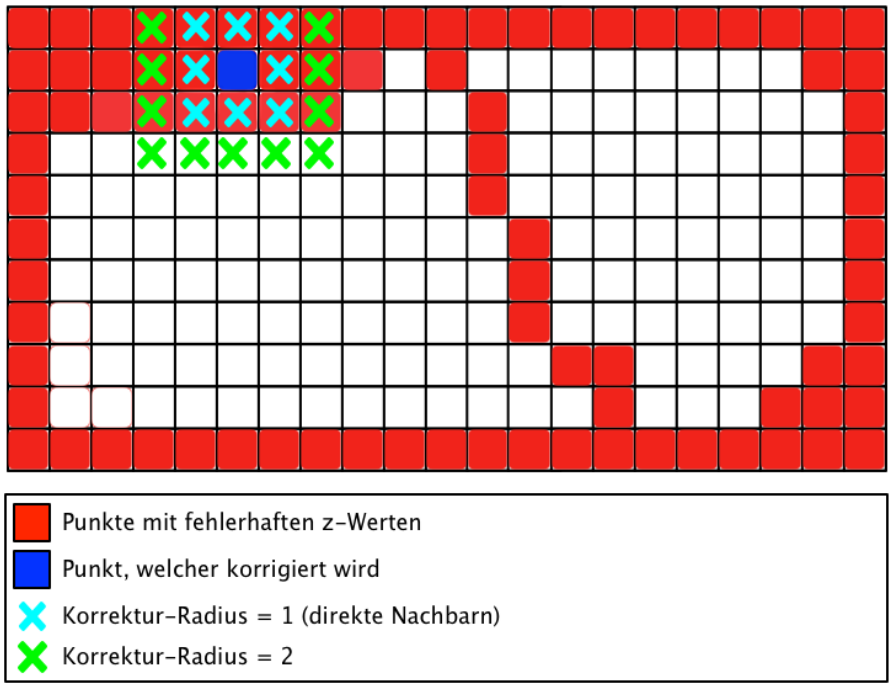


Abbildung 4.23: Skizze - Recover Missing Depth Information

Der zweite Teil des Algorithmus ist für das Berechnen des neuen korrigierten Tiefenwertes für die fehlerhaften Punkte zuständig. Dazu werden alle gültigen Nachbarn-Punkte verwendet, welche mittels dem ersten Teil bestimmt wurden. Nun wird der Mittelwert all dieser Punkte bestimmt und dem zu korrigierenden Punkt als neuer Z-Wert (Tiefenwert) zugewiesen.

Mit Hilfe dieses Algorithmus können wir die Anzahl fehlerhafter Tiefendaten auf 0 senken. Dies gewährleistet, dass alle auf den Z-Werten basierenden Berechnungen von nun an nur noch mit korrekten Werten agieren.

Mögliche Optimierung:

Obwohl der Algorithmus vorerst völlig ausreicht, haben wir uns bereits Gedanken über eine allfällige Verbesserung gemacht. Es wäre denkbar Nachbar-Punkte, welche horizontal oder vertikal auf der selben Linien wie der zu bestimmende Punkt liegen, für die Mittelwert-Berechnung höher zu priorisieren als die diagonal verschobenen Punkte.

4.2.5.7. Find Optimal Field

Es muss möglich sein, dass wir innerhalb des vom Beamer projizierten Bildes ein optimales Feld finden. Das optimale Feld ist definiert, sodass es die grösstmöglichste Rechteck-Fläche im Beamerbild umfasst. Dabei darf aber logischerweise keine Seite und keine Ecke des Rechtecks ausserhalb des Beamerbildes zu liegen kommen.

Diese Anforderung kann nur erfüllt werden, wenn wir das Rechteck so zeichnen, dass die beiden kürzesten Seiten des Beamerbildes die Seiten des Feld-Rechtecks bilden.

Das Finden dieses Rechtecks liefert uns zusätzlich den optimalsten Koordinatenursprung für das "interne" Koordinatensystem (Area). Die beiden kürzesten Seiten des Beamerbildes schliessen den besten Ursprungspunkt ein (wie bereits in 4.2.5.5 erwähnt). In einem nächsten Schritt muss jetzt nur noch festgelegt werden, welche der beiden Seiten die x-Achse bzw. die y-Achse repräsentieren. Laut unserer Definition wird die längere der beiden Seiten als x-Achse und die kürzere als y-Achse genommen. Diese Seitenbestimmung führt uns zudem zu den beiden zusätzlich benötigten Eckpunkten für den Basiswechsel. Deshalb können wir von nun an den Basiswechsel basierend auf den optimalsten Eckpunkten durchführen.

Algorithmus:

Der Algorithmus ist im Wesentlichen sehr einfach. Dabei muss zu Beginn lediglich die Distanz zwischen den einzelnen Eckpunkten von jedem Punkt aus berechnet werden. Die Distanz der diagonalen Verbindungen werden dabei nicht betrachtet. Da die Eckpunkte bereits sowohl in RealWorld-Punkten als auch in RealWorld-Vektoren vorliegen, können wir die Distanz sehr einfach vektoriell bestimmen.

Als Nächstes muss der Punkt gesucht werden, welcher die beiden kürzesten Distanzen zu seinen Nachbarn aufweist. Dieser wird automatisch zum Ursprungspunkt gewählt.

Zu guter Letzt müssen nur noch die Nachbarpunkte so identifiziert werden, damit klar ist, mit welchem Nachbarn der Ursprungspunkt die x-Achse bzw. die y-Achse bildet.

4.3. Realisierung

4.3.1. Kalibrationsphase

Das nachfolgende Flussdiagramm illustriert den programmtechnischen Ablauf einer Kalibration.

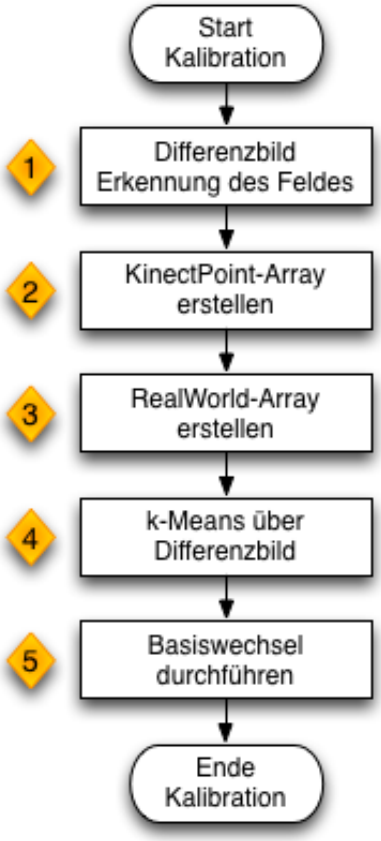


Abbildung 4.24: Realisierung - Kalibrationsablauf

In den nächsten Abschnitten wollen wir uns die einzelnen Schritte aus Sicht der Software genauer betrachten.

4.3.1.1. Differenzbilder ermitteln

Der erste Schritt unserer Kalibration besteht darin, die Eckpunkte unseres Beamerbildes zu finden. Dazu verwenden wir den Algorithmus "Differenz-Bild berechnen" (Abschnitt 4.2.5.1). Dazu müssen die Eckpunkte einzeln angezeigt und erkannt werden. Durch dieses Vorgehen können wir in jedem Fall das Betrachterproblem lösen und damit jeden Eckpunkt eindeutig identifizieren.

4.3.1.2. KinectPoint-Array erstellen

Als Nächstes geht es um das Erstellen eines Arrays, welches aus KinectPoints besteht, wobei ein KinectPoint nebst den x/y/z-Koordinaten noch RGB-Werte und ein Type besitzt. Der Grund für das Kreieren eines eigenen Punktes ist, dass die Kinect keine direkte Möglichkeit bietet, das byte-Array des ColorImageStreams und das short-Array des DepthImageStreams zu kombinieren oder gar korrekt übereinstimmend auszulesen. Man könnte sich jetzt denken, es sei doch sehr einfach die beiden Streams zu kombinieren, wenn man sowieso dieselbe Auflösung verwendet. Dies geht in der Praxis jedoch nicht, da die x/y-Koordinaten der beiden Streams nicht miteinander übereinstimmen. Auch dafür gibt es einen guten Grund, denn die beiden Sensoren sind hardwaremässig leicht verschoben. Glücklicherweise liefert die Kinect SDK eine Methode *(CoordinateMapper.MapDepthFrameToColorFrame())*, welche es ermöglicht die Pixel des DepthStreams-Frames auf die entsprechenden Pixel des ColorImageStreams-Frames zu mappen. Mit Hilfe dieser Methode konnten wir unser eigenes Array mit KinectPoints zusammenstellen, welches jetzt die Informationen der beiden Streams vereinigt.

In Abbildung 4.25 sehen Sie ein Beispiel eines KinectPoint-Arrays, welches in ein Bitmap konvertiert wurde, um eine Ausgabe in einem WPF-Projekt zu ermöglichen. Wie unschwer zu erkennen ist, gibt es einige Pixel, welche rot eingefärbt sind. Auf dieses Phänomen wollen wir genauer eingehen.

Problem 1: Genaues Mapping der beiden Streams
Die rot eingefärbten Pixel stellen all diejenigen Stellen dar, wo die Funktion *CoordinateMapper.MapDepthFrameToColorFrame()* kein passendes ColorStream-Pixel zu einem gegebenen DepthStream-Pixel finden konnte. Auffällig ist, dass bei allen Experimenten der komplette Rand nicht gematched werden kann. Dies hat damit zu tun, dass die Sensoren leicht verschoben sind. Zudem haben wir festgestellt, dass in der Mitte des Bildes teilweise rote Pixel erscheinen. Dabei handelt es sich womöglich um Ungenauigkeiten der Sensoren der Kinect. Uns ist in diesem Zusammenhang aufgefallen, dass diese Stellen meist in der Nähe eines 3D-Objektes (z.b. Tisch, Stuhl, Übergang Boden zu Wand etc. vorkommen). Bei flachen Ebenen wie beispielsweise dem vom Beamer projizierten Bild konnten wir glücklicherweise keine oder sehr wenige derartige Ausreisser feststellen. Deshalb konnten wir davon ausgehen, dass alle Pixel des Bereiches, welcher für uns von Bedeutung ist, richtig erkannt und zusammengefügt werden können.

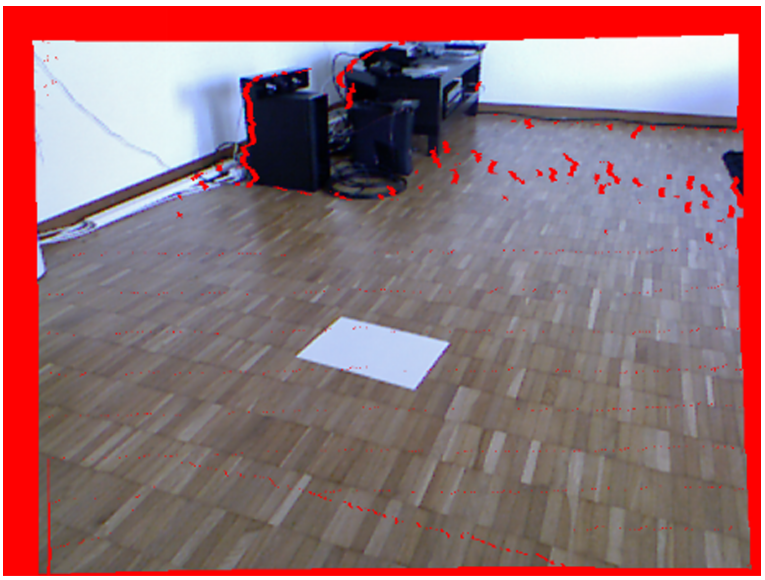


Abbildung 4.25: KinectArray-Image

Diese Erkenntnis gibt uns Hinweise darüber, wie unser Spielfeld in Zukunft projiziert werden sollte. Dabei gelten sicherlich unter anderem die folgenden Regeln für das Projizieren des Beamer-Bildes:

- Das komplette Bild sollte möglichst in der Mitte des von der Kinect abtastbaren Bereiches liegen.
- Das Bild darf nicht durch irgendwelche 3D-Objekte (Tisch, Wand etc.) gestört oder beeinflusst werden.

Mittels dem Algorithmus "Recover Missing Depth Information" (Abschnitt 4.2.5.6) haben wir jedoch die Möglichkeit den Schaden dieses Problems so einzugrenzen, dass uns die z-Werte keine Berechnungen mehr verfälschen. Lediglich die Farbinformationen dieser Punkte können wir nicht wieder herstellen, was aber für uns nicht von grosser Bedeutung ist, da wir uns nur auf das Beamerbild fokussieren müssen.

Problem 2: Die Kinect liefert das Bild gespiegelt an der y-Achse
Während der Implementation haben wir bemerkt, dass die Kinect nicht das gleiche Bild liefert, wie jenes, dass wir mit unseren Augen wahrnehmen. Da die Kinect eine Lochkamera darstellt, ist das Bild an der y-Achse gespiegelt. Dies bedeutet, dass der eigentliche Ursprung des Bildes oben recht liegt. Unser Koordinatensystem der Kinect hingegen haben wir aus Kompatibilitätsgründen zu den anderen Koordinatensystem so ausgelegt, dass sich der Ursprung des Bildes oben links befindet. Für die Berechnungen, welche Punkte aus dem KinectPoint-Array benötigen, stellt dies ein Problem dar. Aufgrund dieser Tatsache ist es nötig, dass KinectPoint-Array an der y-Achse zu drehen.

4.3.1.3. RealWorld-Array erstellen

Wie bereits erläutert, weisen die Koordinaten des KinectPoint-Arrays unterschiedliche Dimensionen auf. Mittels des beschriebenen Algorithmus "Umrechnung der x/y-Koordinaten von Pixel in Millimeter" (Abschnitt 4.2.5.4) können wir dieses Problem elegant beseitigen. Diese Umrechnung führt schliesslich auf ein neues Array, das RealWorld-Array.

Beim Erstellen dieses RealWorld-Arrays werden die x/y-Koordinaten aller Punkte eines KinectPoint-Arrays in Millimeter umgerechnet.

Nach der erfolgreichen Umrechnung der KinectPoints in RealWorld-Points, wird dieses Array als Grundlage für den bevorstehenden Basiswechsel in das "interne" Koordinatensystem verwendet.

4.3.1.4. k-Means über Differenzbild

Bevor wir den bereits angekündigten Basiswechsel durchführen können, benötigen wir erst einmal die Eckpunkte des Beamerbildes bzw. des projizierten Feldes. Hier kommt nun der k-Means Algorithmus zum Einsatz. Dieser liefert uns für die vier Differenzbilder aus Schritt 1 jeweils die KinectPoint-Koordinaten der einzelnen Eckpunkten.

Die ermittelten Koordinaten der Eckpunkte werden des Weiteren noch in RealWorld-Koordinaten umgerechnet.

4.3.1.5. Basiswechsel durchführen

Sobald wir alle Eckpunkte des Feldes in RealWorld-Koordinaten vorliegen haben, können wir mit dem Aufbau unseres "internen" Koordinatensystems beginnen. Verwendet wird dabei der Algorithmus "Basiswechsel" (Abschnitt 4.2.5.5), welcher uns ein initialisiertes Koordinatensystem für die Area liefert.

Von nun an ist es möglich mittels der bereits erläuterten Formeln bestehende RealWorld-Points in Area-Points umzurechnen.

4.3.2. Operationsphase

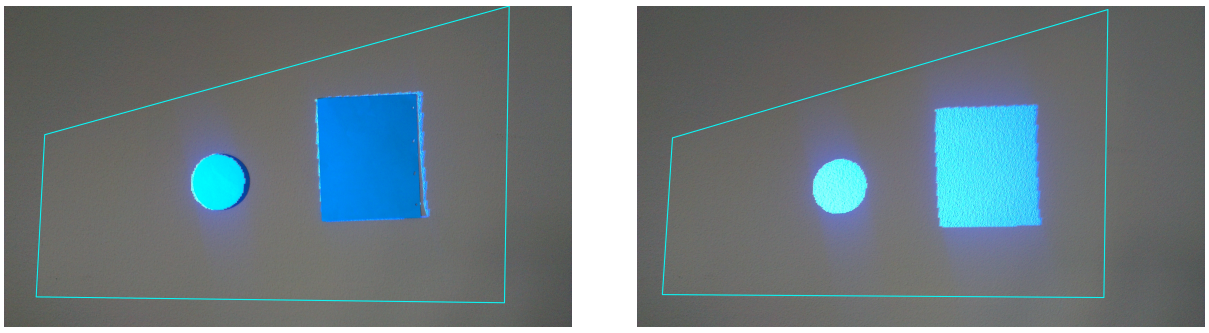
4.3.2.1. Objekterkennung

Als Grundlage der Objekterkennung kommt erneut der Algorithmus "Differenz-Bild berechnen" (Abschnitt 4.2.5.1) zum Einsatz. Die erkannten KinectPoints werden danach in RealWorld-Points umgerechnet. Dabei wird der zweite Ansatz des Algorithmus "Umrechnung der x/y-Koordinaten von Pixel in Millimeter" (Abschnitt 4.2.5.4) verwendet, wobei die bereits berechneten Eckpunkte als Referenzpunkte der Ebene dienen. Zum Schluss müssen die RealWorld-Points mit Hilfe des "Basiswechsel" (Abschnitt 4.2.5.5) in Area-Points umgewandelt werden.

4.3.2.2. Objekte auf Beamer projizieren

Das Einfärben der erkannten Objekte geschieht mittels eines Lookup-Array, welches während der Kalibrationsphase erstellt wird. In einem ersten Versuch haben wir eine Lookup-List verwendet. Dabei ist uns ein Perfomanceproblem aufgefallen, da es $n * O(n^2)$ Vergleichsoperationen benötigt. Insgesamt war die Laufzeit $O(n^2)$. Mit einem Lookup-Array sinkt die Laufzeit auf O(n). Da wir zu diesem Zeitpunkt mittels den Eckpunkten die Fläche des Beamer-Bild im Kinect-Bereiches kennen, könnte man als Optimierung nur diese Punkte umwandeln.

Abbilgung 4.26 illustriert die Projektion der erkannten Objekte, welche nach der Kalibration innerhalb des Beamerbildes platziert wurden.



(a) Vergleich zur Position der realen Objekten

(b) Qualität der Einfärbung

Abbildung 4.26: Eingefärbte Objekte

4.4. API

4.4.1. Wichtige Klassen

Das folgende Klassendiagram beschreibt die verschiedenen Facades, welche die API bilden.

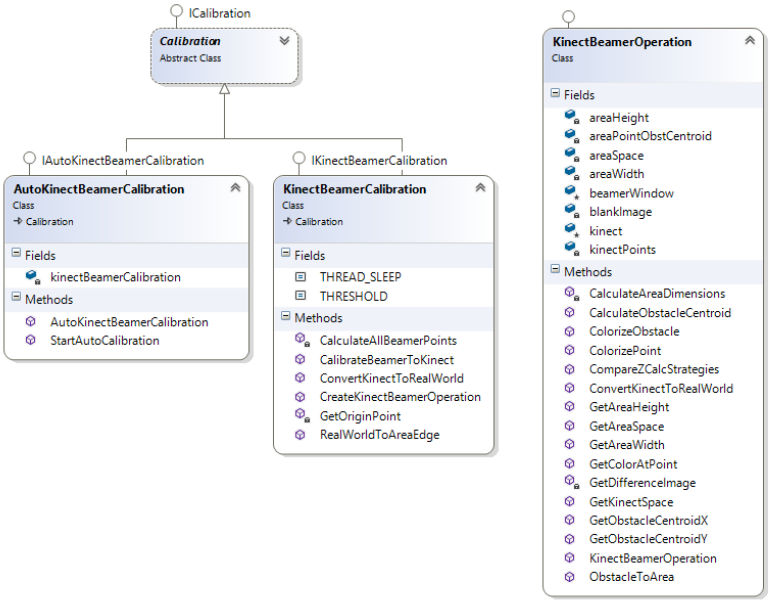


Abbildung 4.27: Klassendiagram

4.4.1.1. AutoKinectBeamerCalibration

Die Klasse `AutoKinectBeamerCalibration` umfasst lediglich die Methode `StartAutoCalibration`. Diese Methode führt die ganze Kalibrationsphase mit den von uns vordefinierten Algorithmen aus.

4.4.1.2. KinectBeamerCalibration

Die Klasse `KinectBeamerCalibration` beinhaltet die einzelnen Schritte und bietet die Möglichkeit Einfluss auf die Kalibrationsphase zu nehmen. Den Methoden kann eine `Strategy` als Parameter übergeben werden.

4.4.1.3. KinectBeamerOperation

Nachdem die Kalibrationsphase beendet wurde, wird ein Objekt der Klasse `KinectBeamerOperation` erstellt. Diese Klasse kapselt den Zugriff auf die Area-Koordinaten.

4.5. Projekt: Informatik zum Anfassen

Ein Projektziel war es den ganzen Kalibrationsprozess im Rahmen eines Ausstellungsprojekt in einer GUI zu visualisieren. Aufgrund fehlender Zeit, mussten wir diesen Punkt auf ein Minimum beschränken.

4.5.1. Ideen

Folgende Ideen wollten wir präsentieren:

- Die einzelnen Kalibrationsschritte sollen wizardmässig aufgebaut werden. Somit kann man den ganzen Prozess Schritt für Schritt erklären.
- Beim Erstellen eines Differenzbildes haben wir zwei unterschiedliche Algorithmen programmiert. Der User soll beide ausführen und die entsprechenden Ergebnisse vergleichen können.
- Ein Objekt kann vom Benutzer in die projizierte Fläche gelegt werden. Dieses wird dann farblich markiert.
- Ein Bild, welches auf dem Computer gezeichnet worden ist, sollte verzerrungsfrei und winkeltreu auf die Fläche projiziert werden.

4.5.2. Dashboard

Um unsere Klassenbibliothek zu testen, haben wir ein WPF-Dashboard entwickelt, welches uns ermöglicht einige Algorithmen zu visualisieren. Das Dashboard deckt einige Ideen, die wir für das Projekt Informatik zum Anfassen haben rudimentär ab.

4.6. Schlussfolgerung

Während unserer Arbeit haben wir in einem ersten Schritt die vorgegebenen Problemstellungen analysiert. Aufgrund der Ergebnisse konnten wir geeignete Lösungsansätze erarbeiten. Dabei haben wir bei einigen Lösungen bereits Optimierungspunkte formuliert. Der Aufbau eines Kalibration-Frameworks mit den implementierten Algorithmen runden die Arbeit ab. Unvorhergesehene Probleme verzögerten den gesamten Projektablauf. Somit mussten einige Projektziele neu priorisiert und der Projektumfang reduziert werden. Dabei litt vor allem das Teilprojekt Informatik zum Anfassen.

Weitere Arbeiten könnten sich mit den Optimierungspunkten und dem Ausbau des Teilprojektes befassen.

Anhang

A. Messergebnisse: Bestimmung der Pixel-Grösse in mm

Der Versuchsaufbau ist sehr einfach. Wir haben auf dem Boden verschiedene Distanzen zur Wand abgemessen und markiert (z.B. 0.5m, 1m etc.). Danach haben wir die Kinect bei den verschiedenen Distanzen platziert und jeweils die Breite und die Höhe des erkennbaren Tiefenbildes gemessen. Die Resultate wurden in einem Excel-File zusammengetragen. Es ist möglich, dass das Tiefenbild Pixel beinhaltet, welche nicht dieselbe Breite und Höhe aufweisen. Aus diesem Grund mussten wir bei all unseren Messungen sowohl Breite als auch Höhe bestimmen. Mit Hilfe des Strahlensatzes konnten wir dann das Verhältnis von Breite bzw. Höhe zur Tiefe berechnen. Dieses liefert uns dann die gewünschten Konstanten, welche für das Berechnen der x/y-Koordinaten notwendig sind.

Abbildung A.1: Bestimmung durch Ausmessen - Versuchsaufbau

Das nachfolgende Bild illustriert den Versuchsaufbau aus Sicht der Kinect-Kamera (mittels Kinect Explorer).

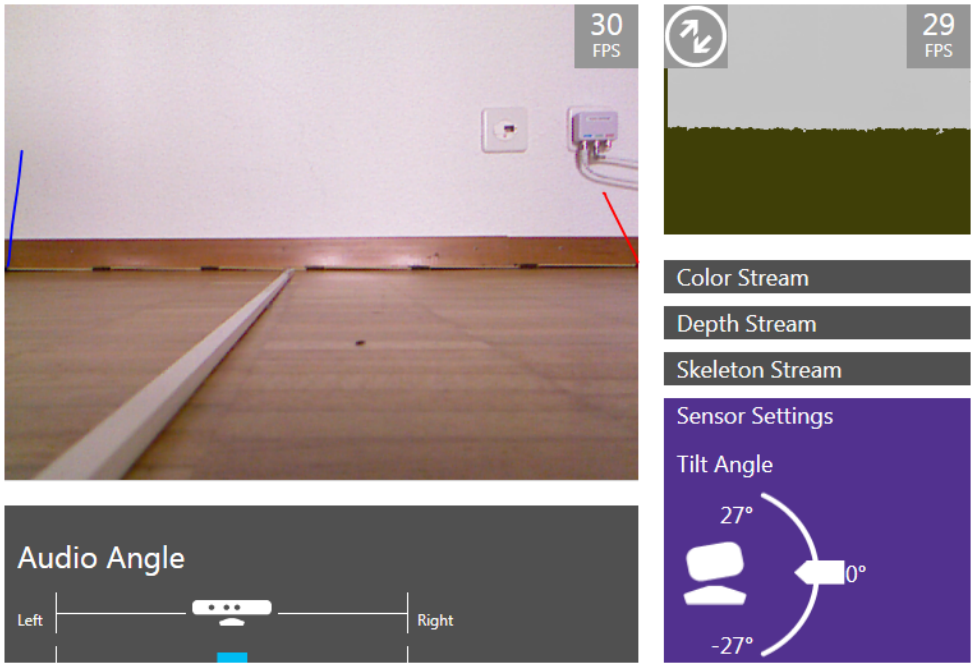


Abbildung A.2: Bestimmung durch Ausmessen - Kinect Explorer

Die nachfolgenden Ausschnitte unseres Excel-Files bieten eine knappe Übersicht über die einzelnen Messreihen sowie die daraus resultierenden Konstanten WIDTH_CONST und HEIGHT_CONST. Zudem ist auch die Abweichung zwischen Berechnung und Ausmessung ersichtlich. Dabei wurden für die Berechnung die Winkel 57 Grad (horizontal) und 43 Grad (vertikal) verwendet.

	Bestimmung von s = 2b						
z (Tiefe) [m]	Berechnung [m]	Ausmessung [m]	Strahlensatz verältnis	Berechnung mit Strahlensatz	Abweichung Strahlensatz mit Messung [mm]	Abweichung [m]	Abweichung [mm]
0.8	0.869			0.940			
0.9	0.977			1.057			
1.0	1.086	**1.170**	**547.009**	1.174	-4.430	0.084	84.089
1.1	1.195			1.292			
1.2	1.303			1.409			
1.3	1.412			1.527			
1.4	1.520			1.644			
1.5	1.629	1.760	**545.455**	1.762	-1.645	0.131	131.133
1.6	1.737			1.879			
1.7	1.846			1.997			
1.8	1.955			2.114			
1.9	2.063			2.231			
2.0	2.172	2.360	**542.373**	2.349	11.141	0.188	188.177
2.1	2.280			2.466			
2.2	2.389			2.584			
2.3	2.498			2.701			
2.4	2.606			2.819			
2.5	2.715			2.936			
2.6	2.823			3.054			
2.7	2.932			3.171			
2.8	3.041			3.288			
2.9	3.149			3.406			
3.0	3.258			3.523			
3.1	3.366			3.641			
3.2	3.475			3.758			
3.3	3.584			3.876			
3.4	3.692			3.993			
3.5	3.801			4.111			
3.6	3.909			4.228			
3.7	4.018			4.345			
3.8	4.126			4.463			
3.9	4.235			4.580			
4.0	4.344			4.698			

WIDTH_CONST **544.945** (Mittelwert)

Abbildung A.3: Ergebnisse der Breiten-Ausmessung

	Bestimmung von s = 2b						
z (Tiefe) [m]	Berechnung [m]	Ausmessung [m]	Strahlensatz verältnis	Berechnung mit Strahlensatz	Abweichung Strahlensatz mit Messung [mm]	Abweichung [m]	Abweichung [mm]
0.8	0.630			0.656			
0.9	0.709			0.738			
1.0	0.788	0.860	558.140	0.820	-39.849	0.072	72.179
1.1	0.867			0.902			
1.2	0.945			0.984			
1.3	1.024			1.066			
1.4	1.103			1.148			
1.5	1.182	1.190	605.042	1.230	40.227	0.008	8.269
1.6	1.261			1.312			
1.7	1.339			1.394			
1.8	1.418			1.476			
1.9	1.497			1.558			
2.0	1.576	1.620	592.593	1.640	20.302	0.044	44.358
2.1	1.654			1.722			
2.2	1.733			1.804			
2.3	1.812			1.886			
2.4	1.891			1.968			
2.5	1.970			2.050			
2.6	2.048			2.132			
2.7	2.127			2.214			
2.8	2.206			2.296			
2.9	2.285			2.378			
3.0	2.363			2.460			
3.1	2.442			2.542			
3.2	2.521			2.624			
3.3	2.600			2.706			
3.4	2.679			2.789			
3.5	2.757			2.871			
3.6	2.836			2.953			
3.7	2.915			3.035			
3.8	2.994			3.117			
3.9	3.073			3.199			
4.0	3.151			3.281			

HEIGHT_CONST **585.258** (Mittelwert)

Abbildung A.4: Ergebnisse der Höhen-Ausmessung

B. Messergebnisse: Objektschwerpunkte in Area

In dieser Messreihe wollten wir die Genauigkeit der von uns konstruierten Area überprüfen. Da dieses Koordinatensystem in Millimetern arbeitet, konnten wir alle Messungen mit Hilfe eines einfachen Meterstabes durchführen.

B.1. Messaufbau

Die Messung haben wir komplett ohne Beamer gemacht und als Alternative ein A0 Poster Papier als Grundlage genommen. In den jeweiligen Ecken, haben wir die Schachbrettmuster gezeichnet. Als Objekt haben wir ein Polygon aus Karton gebastelt und mittels den Diagonalen den Schwerpunkt eingezeichnet. Mit Hilfe eines Geodreiecks und zwei Meterstäben konnten wir somit die x- und y-Achse ausmessen.

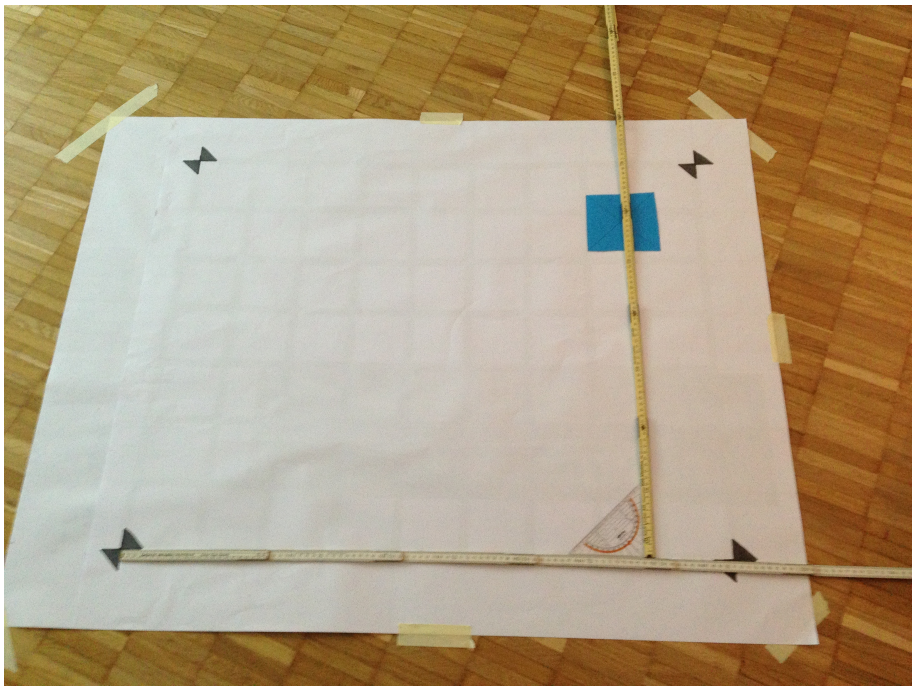


Abbildung B.1: Messaufbau der Objektmessung

Unser Kartonpolygon haben wir an 12 Orten auf der Fläche gemessen.

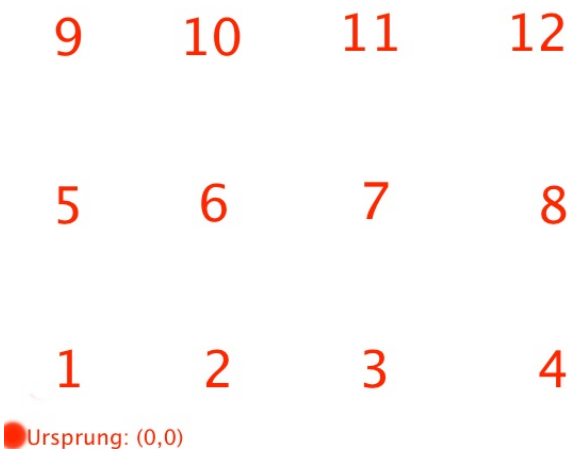


Abbildung B.2: Anordnung der Messung

B.2. Messergebnis

Die Messergebnisse haben wir Tabellarisch festgehalten.

Kinect Objektmessung							
Messpunkt	Area x[mm]	Area y[mm]	Messung x[mm]	Messung y[mm]	Differenz x[mm]	Differenz y[mm]	quadratischer Fehler
1	102.53	113.61	105	114	2.47	0.39	6.253
2	297.69	126.61	307	129	9.31	2.39	92.3882
3	538.58	144.67	539	144	0.42	-0.67	0.6253
4	781.65	150.43	777	141	-4.65	-9.43	110.5474
5	132.95	355.85	130	355	-2.95	-0.85	9.425
6	312.57	350.97	317	350	4.43	-0.97	20.5658
7	567.68	364.86	575	360	7.32	-4.86	77.202
8	795.06	372.05	785	357	-10.06	-15.05	327.7061
9	147.41	592.6	147	580	-0.41	-12.6	158.9281
10	362.94	590.32	370	576	7.06	-14.32	254.906
11	577.05	587.22	580	576	2.95	-11.22	134.5909
12	782.04	605.79	775	580	-7.04	-25.79	714.6857

Abbildung B.3: Messergebnis der Objektausmessung

In der blauen Spalte sieht man das Ergebnis, welches unsere Software liefert. Die braune Spalte enthält die gemessenen Messresultate. Die Abweichung der beiden Verfahren haben wir in der pinken Spalte aufgeführt. Da die Abweichung auch ins Negative fällt, haben wir in der orangen Spalte den quadratischen Fehler berechnet.

B.3. Interpretation

Aus der Messung geht hervor, dass die Punkte um den Ursprung genauer sind als die Punkte die vom Ursprung weiter entfernt sind. Als zweite Erkenntnis geht hervor, dass der Fehler nicht linear ansteigt. Um den Fehler zu minimieren, könnte man mittels dem Least squares Verfahren einen kleineren quadratischen Fehler produzieren.

C. Differenzbild zu Area

Um zu verifizieren, ob die Area korrekt aufgebaut wir, haben wir folgendes Testszenario aufgebaut.

C.1. Messaufbau

Auf einem A0 Poster Papier haben wir ein Grid aus Isolierband platziert, welches genau die Fläche ausfüllt, welche wir bereits während des Tests "Objektschwerpunkte in Area" verwendet haben. Somit konnten wir sicherstellen, dass die Area wieder gleich aufgebaut wird.

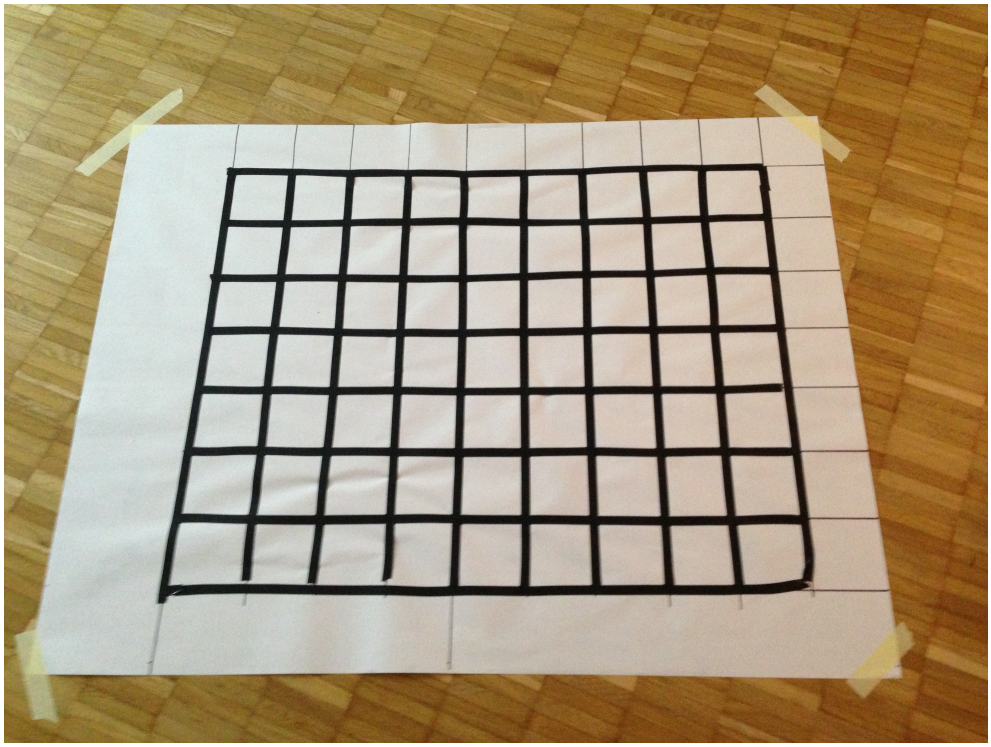


Abbildung C.1: Grid in der Area

Als Nächstes haben wir die bekannten Kalibrierungsalgorithmen angewendet, um die Area im Dashboard anzeigen zu lassen

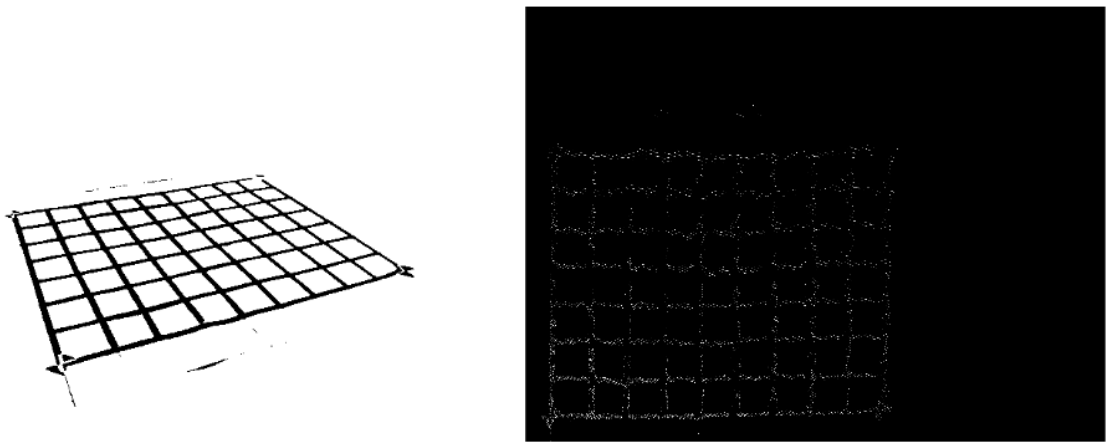


Abbildung C.2: Differenzbild und Area

C.2. Interpretation

Aus dem Area Bild ist ersichtlich, dass die Wirklichkeit schon ziemlich gut abgebildet wird. In diesem Szenario kann es noch vorkommen, dass einige Kinect-Punkte keine Tiefendaten enthalten. Das erklärt auch warum das Grid in der Area nicht sehr dicht ist. Eine zweite Erklärung ist, dass die Area auf ein Bitmap mit fixer Grösse gemappt wurde und somit einige Punkte verloren gehen. Die fehlenden Tiefendaten lassen sich mit dem Nachbarschaftsalgorithmus korrigieren. Um die Area flexibler zu gestalten, müsste das Area Bitmap dynamisch erstellt werden.

D. Benutzerhandbuch

D.1. Klassenbibliothek erstellen

Als Erstes muss aus unserer Visual Studio Solution eine Klassenbibliothek (DLL) Datei erstellt werden. Dies kann auf zwei Arten erstellt werden.

D.1.1. Visual Studio

Um die Klassenbibliothek mit Visual Studio zu kompillieren, muss das Solution File geöffnet werden.

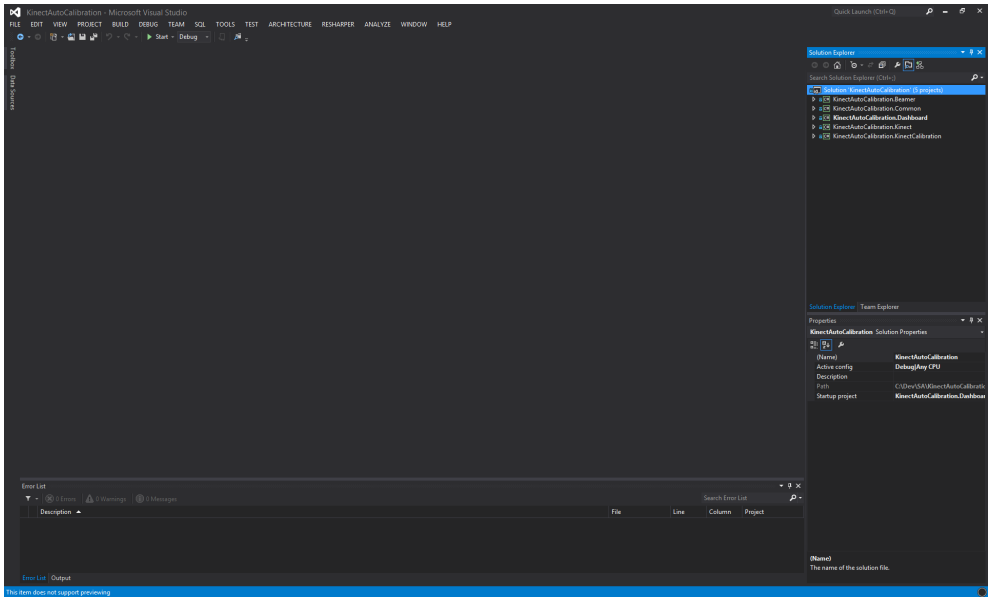


Abbildung D.1: Visual Studio mit geöffneter Solution

Als Nächstes kann über das Build Menü "Build Solution" ausgeführt werden.

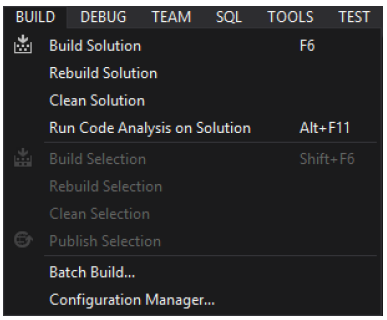


Abbildung D.2: Build Solution Menu

Nun ist im Pfad Projektordner/KinectAutoCalibration.Calibration/bin/Debug die DLL KinectAutoCalibration.Calibration.dll verfügbar.

D.1.2. Konsole

Der ganze Kompilierungsvorgang kann auch mittels der Kommandozeile erfolgen:

$csc/target:library/out:KinectAutoCalibration.Calibration.dll *.cs$

D.2. Einbinden der Klassenbibliothek

Als Nächstes muss die Klassenbibliothek in einem Projekt referenziert werden. Mittels Rechtsklick auf das Projekt, kann eine Klassenbibliothek referenziert werden.

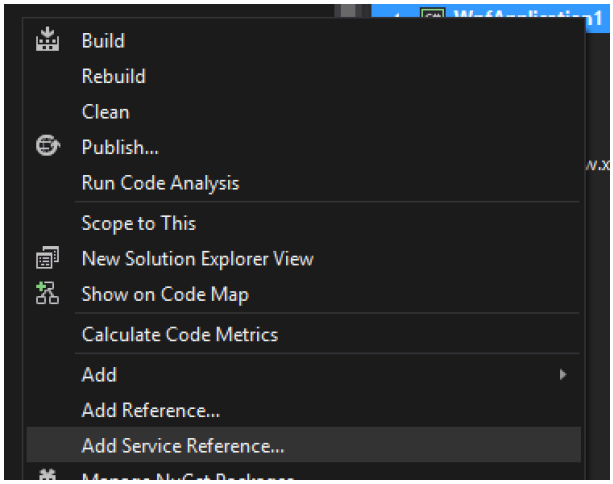


Abbildung D.3: Referenz hinzufügen

Im neuen Dialog kann man nun die DLL ausgewählt werden.

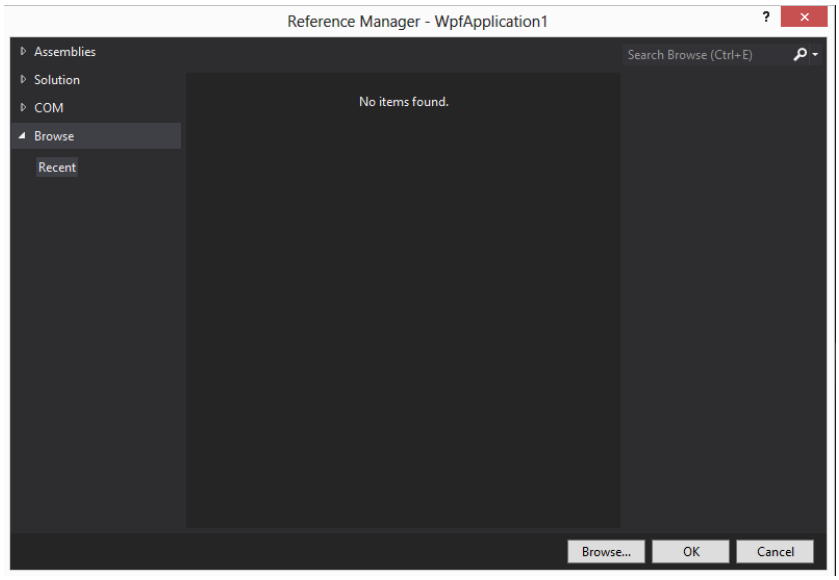


Abbildung D.4: Reference Manager

Mit einem Klick auf OK ist die DLL im Projekt verfügbar.

D.3. Namespace einbinden

Im C#-Code muss nun noch der Namespace eingebunden werden.

```
using KinectAutoCalibration.Calibration;
```

Abbildung D.5: using Statement in C#

Somit sind alle Zugriffe auf die API möglich.

E. Erfahrungsberichte

E.1. Engelbert Lüchinger

Die Zusammenarbeit mit Kevin Vogt stand schon früh fest, da wir uns bereits aus der Lehrzeit kennen und zusammen das Studium begonnen haben. Dazu kommt, dass wir in der gleichen Wohngemeinschaft wohnen, was Teamdiskussionen sehr erleichtert und wir somit flexibel arbeiten konnten.

Als die Themenwahl für unsere Studienarbeit bevorstand, haben wir uns bei Professor Oliver Augenstein informiert. Schnell konnte ich mich von dem Projekt Kinect- / Bamer-Autokalibration überzeugen. Einerseits war die Idee des Projektes sehr innovativ und andererseits hörten sich die Problematiken sehr fordernd an.

Die Problemdomäne unterteiltet sich in zwei Unterprobleme: die Entwicklung von Algorithmen , sowie das Erstellen eines Framework. In beiden Gebieten konnte ich noch keine Erfahrung vorweisen. Dies machte es sehr schwierig das Projekt einzuschätzen und zu planen.

Bei der Ausarbeitung der Algorithmen, war es schwierig zu beurteilen ob ein Algorithmus funktionierte und die Güte für unsere Problemstellung ausreicht. Auch wenn wir Messreihen gemacht haben, war es für mich schwierig die Ergebnisse richtig zu interpretieren. Bei Schwierigkeiten konnten wir uns immer an Professor Oliver Augenstein wenden. Die Wöchentlichen Meetings waren immer sehr aufschlussreich und ich konnte immer meinen Horizont erweitern. Beim programmieren des Framworks fehlte es mir an der Erfahrung um beurteilen zu können ob ein Ansatz gut funktionieren wird oder ob es eine bessere alternative gibt. Dies führte dazu dass zahlreiche Refactoring Schritte gemacht werden musste, was wiederum Zeit beanspruchte.

Rückblickend kann ich sagen, dass ich bei der Studienarbeit nochmals sehr viel gelernt habe. Das Experimentieren mit Formeln und das arbeiten mit Tools, wie beispielsweise Mathematica, zur Lösungsfindung hat mir sehr viel Freude bereitet. Auch habe ich einen Eindruck erhalten, wie man ein mathematisches Softwareprojekt bearbeiten kann.

E.2. Kevin Vogt

Mit Engelbert als Arbeitskollege habe ich sehr früh gerechnet, da wir uns schon seit der Ausbildung zum Informatiker EFZ bestens kannten. Bereits damals haben wir praktisch alle Gruppenarbeiten gemeinsam erfolgreich absolviert. Zudem wohnen wir in der gleichen Wohngemeinschaft. Dies war meiner Meinung nach ein entscheidender Vorteil für unsere Zusammenarbeit, welche stets angenehm und äusserst produktiv war. Vor allem der Fakt, dass wir uns bezüglich unserer Stärken perfekt ergänzen, sehe ich als einer der Erfolgsfaktoren unserer Arbeit.

Bereits vor der Themenauswahl war mir klar, dass ich nicht unbedingt ein klassisches Software-

Engineering Projekt bearbeiten wollte. Wir wollten eine Arbeit, welche sehr abwechslungsreich ist. Da wir wussten, dass Professor Oliver Augenstein schon einige interessante Arbeiten betreute, fragten wir ihn frühzeitig an. Er präsentierte uns einige Ideen und schliesslich einigten wir uns darauf die Kinect-/Beamer-Autokalibration umzusetzen. Während sich Engelbert mit den Problematiken des Beamers auseinandersetzte, war ich für den Kinect-Teil verantwortlich.

Die Arbeit bot meiner Meinung nach sehr viele interessante Aspekte. Nebst der Entwicklung einer Software, konnte man mathematischen Lösungen für eigene Algorithmen ausarbeiten. Vor allem das Erarbeiten der Konzepte für unsere Algorithmen und deren Implementierung bereiteten mir sehr viel Freude, obwohl es nicht immer leicht war. Eine der grössten Herausforderungen fand ich die Korrektheit unserer Algorithmen zu überprüfen und deren Ergebnisse korrekt zu interpretieren. Bei solch mathematischen Problemen stand uns glücklicherweise Oliver Augenstein stets zur Seite. Er erklärte uns mit viel Enthusiasmus mathematische Problemstellungen und Lösungen. Generell beurteile ich die Zusammenarbeit mit Oliver Augenstein als ausgezeichnet und sehr angenehm.

Diese Studienarbeit war für mich sehr lehrreich. Ich habe gelernt, wie man mit Problemen, welche man zu Beginn nicht vermutet, umgeht. Vor allem die Kinect brachte einige Tücken mit sich, welche zu lösen waren. Am meisten Mühe hatte ich dabei mit dem Zeitmanagement, welches oft durch solche unerwarteten Probleme beeinflusst wurde. Dies wirkte sich leider darauf aus, dass wir nicht alle geplanten Features umsetzen konnten.

Für die Bachelor-Arbeit möchte ich das Zeitmanagement besser kontrollieren können. Deshalb haben wir uns bereits jetzt vorgenommen ein Projektmanagement-Tool, wie Redmine, einzusetzen.

Generell denke ich aber, dass uns unsere Studienarbeit sehr gelungen ist.

Literaturverzeichnis

[HaKin] Nicolas Burrurs, Florian Echtler, Daniel Herrera C., and Matt Parker *Hacking the Kinect* Apress, 2012

[BKPwMK] Jarrett Webb and James Ashley *Beginning Kinect Programming with the Microsoft Kinect SDK* Apress, 2012

[BAMiW] Philipp Eichmann und Roman Giger *Minigolf im Wohnzimmer* HSR, 2012

[TUMCalCam] Technische Universität München *Calibration of the Camera* http://vision.in.tum.de/data/datasets/rgbd-dataset/file_formats#calibration_of_the_color_camera, 13.04.2013

[WikiKin] Wikipedia *Kinect* http://en.wikipedia.org/wiki/Kinect

[WikiKMeans] Wikipedia *K-Means* http://de.wikipedia.org/wiki/K-Means-Algorithmus

[WikiManni] Wikipedia *Mannigfaltigkeit* http://de.wikipedia.org/wiki/Mannigfaltigkeit

[MSDNCoSp] MSDN *Coordinate Spaces* http://msdn.microsoft.com/en-us/library/hh973078.aspx

[MSDNConst] MSDN *Constants* http://msdn.microsoft.com/en-us/library/hh855368

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

API	Application Programming Interface
CMOS	Complementary Metal Oxide Semiconductor
DLL	Dynamic Link Library
DLP	Digital Light Processing
FPS	Frames per Seconds
GUI	Graphical User Interface
HTML	Hypertext Markup Language
MSDN	Microsoft Developer Network
SDK	Software Development Kit
WPF	Windows Presentation Foundation
XML	Extensible Markup Language