EOL: Erweiterung der Teststation mittels 3D Kamera

Studienarbeit T3200

Studiengang Elektrotechnik

Studienrichtung Automation

Duale Hochschule Baden-Württemberg Ravensburg, Campus Friedrichshafen

von

Veaceslav Gorban, Zilong Zhao

Abgabedatum:	15.07.2024
Bearbeitungszeitraum:	08.04.2024 - 15.07.2024
Matrikelnummer:	3703041, 1005147
Kurs:	TEA21
Gutachter der Dualen Hochschule:	Peter Buckel

Erklärung

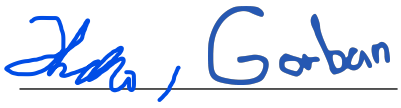
gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2017 in der Fassung vom 25.07.2018.

Ich versichere hiermit, dass ich meine Bachelorarbeit (bzw. Projektarbeit oder Studienarbeit bzw. Hausarbeit) mit dem Thema:

EOL: Erweiterung der Teststation mittels 3D Kamera -

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Friedrichshafen, 15.07.2024



Veaceslav Gorban, Zilong Zhao

Sperrvermerk

gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2017 in der Fassung vom 25.07.2018:

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anders lautende Genehmigung vom Dualen Partner vorliegt.

Friedrichshafen, 15.07.2024



Veaceslav Gorban, Zilong Zhao

Kurzfassung

Die vorliegende Studienarbeit T3200 baut auf den Ergebnissen der vorherigen Arbeit T3100 auf, die sich mit der Erweiterung und Optimierung der End-of-Line Teststation mittels 3D-Kameratechnik beschäftigt. Das Ziel der aktuellen Arbeit besteht darin, die Implementierung und Integration der Kameras zu optimieren und die aufgenommenen Punktwolken für die Fehleridentifikation zu vergleichen.

In der Studienarbeit T3100 wurde der Fokus auf die Fusionierung der Tiefenkamera mit der Farbkamera mittels Kalibrierung und Orientierung gelegt.Durch die Fusionierung der beiden Kamerasysteme können wir der Tiefenkamera Farbinformation hinzufügen. Es entstehen farbliche Punktwolke, welche den Prozess des Punktwolkenvergleiches vereinfachen. Die dadurch gewonnenen Erkenntnisse und Daten setzen unsere aktuelle Arbeit fort.

Die Studienarbeit T3200 zielt darauf ab, die Darstellung der Punktwolken durch physische und softwaretechnische Anpassungen weiter zu verbessern, um schärfere und klarere Punktwolken der Bauteile zu erhalten.Ein weiterer großer Abschnitt umfasst die Entwicklung eines Programms zur Fehlererkennung zweier Bauteile. Das Programm soll zwei Punktwolken vergleichen und die Unterschiede farblich markieren. Anschließend soll es in die Anlage implementiert werden, um bei der Erkennung eines Fehlers das betreffende Bauteil herauszufiltern. Durch weitere umfangreiche Tests und Experimente soll die Leistungsfähigkeit der EoL-Testing Station weiter erhöht und somit die Qualität der Endprodukte nachhaltig verbessert werden.

Inhaltsverzeichnis

Abkürzungsverzeichnis **1**

1 Einleitung **2**
1.1 Problemstellung 2
1.2 Aufgabenstellung 2
1.3 Zielsetzung 3

2 Aktueller Stand der Station **4**
2.1 Abgeschlossene Arbeiten 5
2.2 Aufgetretene Probleme 6

3 Umsetzung **9**
3.1 Distanzoptimierung 9
3.2 Tageslichtoptimierung 11
3.3 Software-Optimierung 14

4 Punktwolkenvergleich **17**
4.1 Methoden 17
4.2 Umsetzung 18
4.2.1 JupyterLab 18
4.2.2 Algorithmus 19
4.2.2.1 Punktwolkenerstellung 20
4.2.2.2 Vergleich 23
4.2.3 Ergebnis 32

5 Zusammenfassung und Ausblick **36**

Literatur **38**

Abbildungsverzeichnis **39**

Anhang A **I**

A.1 Punktwolkenerstellung - Quellcode II

A.2 Punktwolkenvergleich - Matplotlib - Quellcode V

A.3 Punktwolkenvergleich - Open3D - Quellcode IX

Abkürzungsverzeichnis

Abb.	Abbildung
bzw.	beziehungsweise
EoL	End of Line
etc.	et cetera
ICP	Iterative Closest Point
mind.	mindestens
o.ä.	oder ähnlichem
usw.	und so weiter
vgl.	vergleiche
z. B.	zum Beispiel

1 Einleitung

1.1 Problemstellung

Die Optimierung von End-of-Line (EoL) Testing Stationen ist ein kritischer Aspekt der Produktionsqualitätssicherung. In der Studienarbeit T3100 wurde bereits eine grundlegende Implementierung von 3D-Kameras zur Überwachung und Fehlererkennung in der EoL-Station vorgenommen. Trotz der erzielten Fortschritte besteht weiterhin Bedarf an der Verfeinerung und Verbesserung der Systeme, um eine noch höhere Genauigkeit und Zuverlässigkeit in der Qualitätskontrolle zu gewährleisten.

1.2 Aufgabenstellung

Die vorliegende Arbeit T3200 setzt sich zum Ziel, die in der Arbeit T3100 entwickelten Ansätze weiter zu verbessern und auszubauen. Die spezifischen Aufgaben umfassen:

- Distanzoptimierung: Um die Bildqualität zu steigern, ist es ratsam, die Entfernung zwischen dem Bauteil und der Kamera durch eine optimierte Positionierung zu ermöglichen.

- Integration eines neuen PoE-fähigen Switches oder einer Netzwerkkarte: Ein weiteres Ziel umfasst den Einbau eines leistungsstarken Power-over-Ethernet (PoE)-fähigen Switches oder einer Netzwerkkarte, die eine zuverlässige Leistung für beide Kameras sicherstellen.

- Tageslichtoptimierung: Implementierung von Maßnahmen zur Kontrolle der Lichtverhältnisse im Raum, um eine gleichmäßige Beleuchtung und somit eine bessere Bildqualität zu gewährleisten.

- Punktwolkeerstellung: Entwicklung und Implementierung eines Systems zur Erstellung präziser Punktwolken der Bauteile, um eine detaillierte 3D-Darstellung und Analyse zu ermöglichen.

- Bauteilidentifikation: Das Hauptziel ist die Entwicklung eines Systems zur automatischen Identifikation von Bauteilen und deren Speicherung als Muster.

- Defekterkennung: Nach der Identifikation sollen defekte Bauteile mit idealen Bauteilen verglichen werden, um Fehler zu erkennen.

1.3 Zielsetzung

Das übergeordnete Ziel der Studienarbeit T3200 besteht darin, die Effizienz und Zuverlässigkeit der EoL-Testing Station durch die Integration fortschrittlicher 3D-Kameratechnik und verbesserter Analysealgorithmen signifikant zu steigern. Durch die Optimierung der bestehenden Systeme und die Implementierung neuer Technologien soll die Qualität der hergestellten Produkte nachhaltig verbessert werden. Die Arbeit strebt an, durch systematische Tests und Experimente eine fortschrittliche und leistungsfähige Qualitätskontrolle am Ende der Produktionslinie zu etablieren, die den hohen Anforderungen moderner Produktionsprozesse gerecht wird.

2 Aktueller Stand der Station

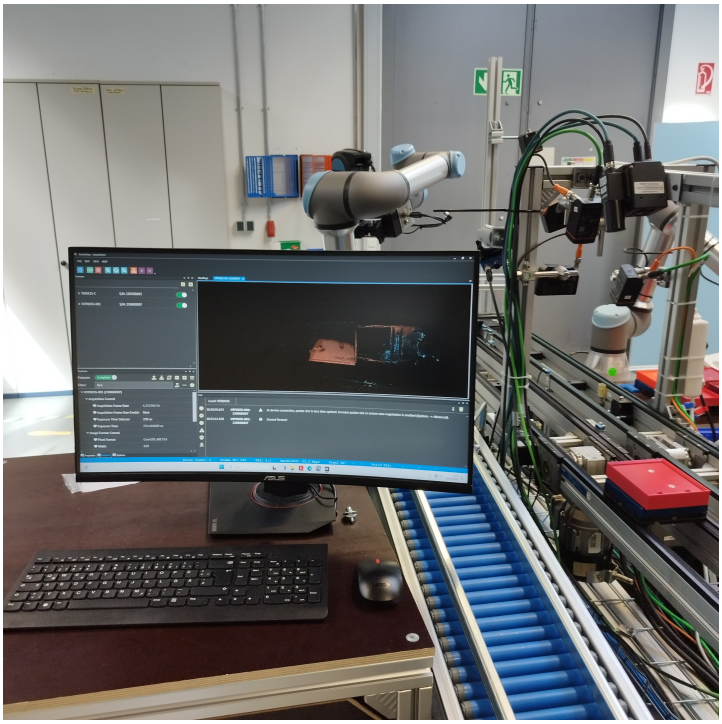


Abbildung 2.1: EOL Station

Die End-of-Line (EoL) Testing Station ist ein kritischer Bestandteil im Produktionsprozess, da sie die letzte Qualitätskontrolle vor dem Abschluss der Produktion darstellt. Diese Station gewährleistet, dass alle hergestellten Produkte die festgelegten Qualitätsstandards erfüllen und keine Fehler aufweisen, die die Funktionalität oder Sicherheit beeinträchtigen könnten. Im Rahmen der Studienarbeit T3100 wurden verschiedene Maßnahmen ergriffen, um die Funktionalität und Effizienz der EoL-Station zu verbessern. Hier werden die abgeschlossenen Arbeiten sowie die aufgetretenen Probleme und deren Auswirkungen beschrieben.

2.1 Abgeschlossene Arbeiten

Verbindung und Integration der Kamerasysteme

Eine der ersten Aufgaben war die Auswahl und Integration geeigneter Kamerasysteme. Die Helios2 Kamera (HLT003S-001) wurde erfolgreich in das System eingebunden und konfiguriert. Diese Kamera ist für ihre hohe Genauigkeit und Zuverlässigkeit bekannt und stellt sicher, dass detaillierte 3D-Bilder der überprüften Bauteile erfasst werden. Ebenso wurde die Triton 3,2MP Farbkamera (TRI032S-CC) erfolgreich installiert und in Betrieb genommen. Diese Kamera ergänzt die Helios2 durch ihre Fähigkeit, hochauflösende Farbbilder zu liefern, die für die optische Inspektion und Fehlererkennung essentiell sind.

Softwareintegration und Konfiguration in ArenaView

Ein wesentlicher Schritt war die Konfiguration der Kameras in der ArenaView Software. ArenaView bietet eine zentrale Plattform zur Steuerung und Auswertung der Kameradaten. Durch die erfolgreiche Konfiguration der Kameras in dieser Software konnten die erfassten Daten in Echtzeit verarbeitet und analysiert werden. Dies ermöglichte eine schnelle und präzise Fehlererkennung sowie die Dokumentation der Prüfergebnisse.

Kalibrierung und Orientierung der Kameras

Ein weiterer wichtiger Meilenstein war die präzise Kalibrierung der Triton und Helios Kameras. Die Kalibrierung ist entscheidend, um sicherzustellen, dass die Kameras akkurate und zuverlässige Daten liefern. Mithilfe spezialisierter Kalibrierungstools und -verfahren wurden die Kameras so eingestellt, dass Verzerrungen und Ungenauigkeiten minimiert wurden. Darüber hinaus wurden die Helios2 und Triton Kameras optimal ausgerichtet, um qualitativ hochwertige Bilder zu erfassen. Diese sorgfältige Ausrichtung gewährleistet, dass die Kameras konsistente und verwertbare Bilddaten liefern, die für die Fehlererkennung und Qualitätskontrolle unerlässlich sind.

Einrichtung des 3D und RGB-Streams

Ein zusätzlicher Erfolg war die Einrichtung eines kombinierten 3D und RGB-Streams in der ArenaView Software. Diese Einrichtung ermöglicht eine umfassende Analyse der erfassten Daten, da sowohl die dreidimensionalen Strukturen als auch die Farbinformationen der Bauteile erfasst und ausgewertet werden können. Der kombinierte Stream liefert eine detaillierte visuelle und geometrische Darstellung der Bauteile, was die Identifikation von Fehlern und Abweichungen erleichtert.

2.2 Aufgetretene Probleme

Einfluss von Tageslicht auf die Bildqualität

Während der Implementierung traten verschiedene Herausforderungen auf. Ein bedeutendes Problem war der Einfluss von Tageslicht auf die Bildqualität. Das natürliche Licht im Raum beeinträchtigte die Qualität der aufgenommenen Bilder erheblich. Insbesondere bei wechselnden Lichtverhältnissen, wie sie durch das Tageslicht verursacht werden, konnten die Kameras nicht immer konsistente und zuverlässige Daten liefern. Dies führte zu unzuverlässigen Ergebnissen und erschwerte die präzise Fehlererkennung.

Nicht optimale Distanz zwischen Bauteil und Kamera

Ein weiteres Problem war die nicht optimale Distanz zwischen Bauteil und Kamera. Die anfängliche Positionierung der Kameras war nicht ideal, was die Bildqualität negativ beeinflusste. Wenn die Kamera zu weit vom Bauteil entfernt war, wurden die Bilder unscharf und Details gingen verloren. War die Kamera zu nah, traten Verzerrungen auf, die ebenfalls die Qualität der Bilddaten beeinträchtigten. Diese unzureichende Positionierung erforderte mehrere Anpassungen und Tests, um die optimale Distanz zu finden, die klare und präzise Bilder gewährleistet.

Lange Antwortzeiten der ArenaView Software

Zusätzlich traten lange Antwortzeiten der ArenaView Software bei der Konfiguration und Parametrierung der Kameras auf. Diese Verzögerungen beeinträchtigten die Effizienz der gesamten Konfigurationsprozesse. Bei der Einstellung der Kameraparameter und der Anpassung der Konfigurationsoptionen zeigte die Software erhebliche Latenzen, was den Arbeitsfluss verlangsamte und zu Frustrationen führte. Diese langen Antwortzeiten machten es schwierig, schnelle Anpassungen vorzunehmen und erforderten häufige Pausen, um auf die Reaktion der Software zu warten.

Auswirkungen und Weiterentwicklung Die beschriebenen Probleme hatten unterschiedliche Auswirkungen auf die Effizienz und Genauigkeit der EoL-Station. Der Einfluss von Tageslicht führte zu variablen Ergebnissen bei der Bildaufnahme, was die Fehlererkennung unzuverlässig machte. Durch die nicht optimale Distanz zwischen Bauteil und Kamera wurden viele Bilder unscharf oder verzerrt, was ebenfalls die Fehlererkennung erschwerte. Die langen Antwortzeiten der ArenaView Software verzögerten den Konfigurationsprozess und führten zu ineffizienten Arbeitsabläufen.

Trotz dieser Herausforderungen konnten durch die abgeschlossenen Arbeiten und die Identifizierung der Problembereiche bedeutende Fortschritte erzielt werden. Die erfolgreiche Integration und Kalibrierung der Kameras sowie die Einrichtung des 3D und RGB-Streams legten eine solide Grundlage für die weitere Optimierung der EoL-Testing Station. Die kontinuierliche Weiterentwicklung der Fehlererkennungsalgorithmen und die Verbesserung der Bildqualität durch geeignete Maßnahmen werden die Effizienz und Genauigkeit der Qualitätskontrolle weiter steigern.

In dieser Arbeit wird der Fokus darauf liegen, die identifizierten Probleme vollständig zu beheben und die Leistungsfähigkeit der Station weiter zu verbessern. Dazu gehört die Implementierung von Maßnahmen zur Kontrolle der Lichtverhältnisse im Raum, die Optimierung der Kamerapositionierung und die Verbesserung der Softwareperformance. Diese Probleme müssen behoben werden, um eine erfolgreiche Punktwolkeerstellung zu ermöglichen und letztendlich defekte Bauteile zu erkennen und diese auszusortieren. Durch diese kontinuierlichen Verbesserungen soll die EoL-Testing Station den hohen Anforderungen moderner Produktionsprozesse gerecht werden und die Qualität der her-

gestellten Produkte nachhaltig sichern.

3 Umsetzung

3.1 Distanzoptimierung

Optimierung des Kamerawinkels Um die Bildqualität der Kameras zu verbessern und Reflexionen zu minimieren, wurde der Winkel der Kameras zur Säule, auf der sie befestigt sind, auf etwa 45° eingestellt. Diese Anpassung ermöglicht es den Kameras, das Bauteil auf dem Laufband aus einem optimalen Winkel zu erfassen, was zu schärferen und detaillierteren Bildern führt.

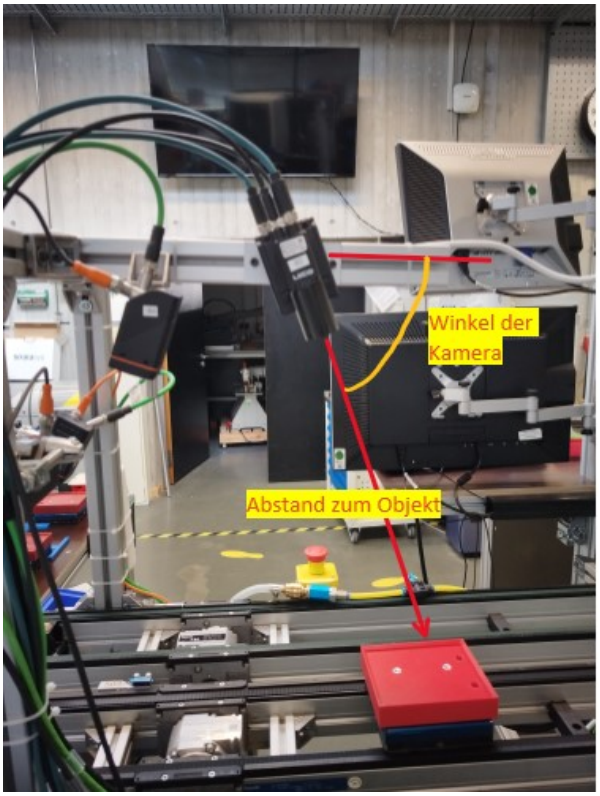


Abbildung 3.1: Winkeloptimierung

Anpassung der Höhe der Säule Ein weiterer wichtiger Schritt zur Optimierung der Bildqualität war die Anpassung der Höhe der Säule, auf der die Kameras montiert

sind. Ursprünglich befand sich die Kamera in einer Höhe von 30 cm zum Objekt auf dem Laufband. Diese Höhe erwies sich als suboptimal, da sie nicht nur die Bildschärfe beeinträchtigte, sondern auch die Lichtqualität und Reflexionen negativ beeinflusste.

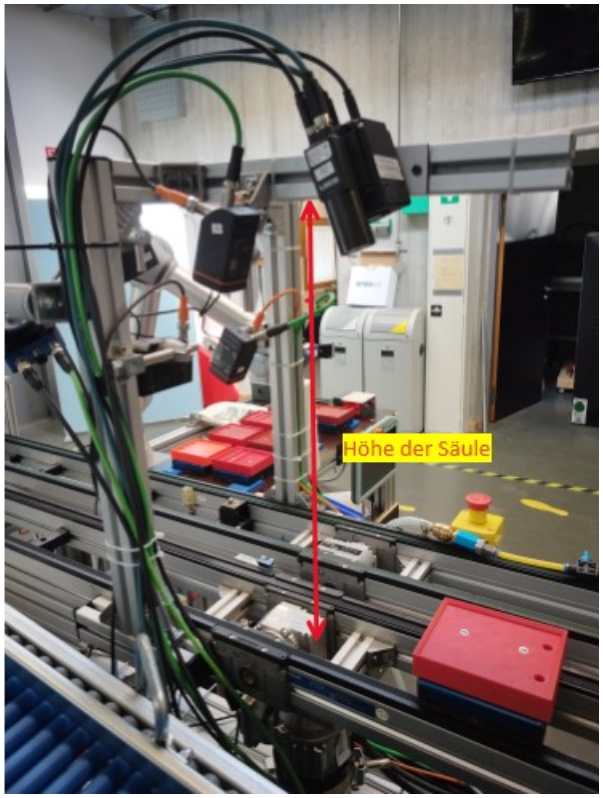


Abbildung 3.2: Abstandoptimierung

Durch eine Erhöhung der Säulenhöhe auf 50 cm zum Objekt auf dem Laufband konnte die Bildqualität erheblich verbessert werden. Diese neue Positionierung der Kameras sorgt dafür, dass die Lichtverhältnisse besser kontrolliert werden und unerwünschte Reflexionen reduziert werden. Zudem ermöglicht die größere Distanz eine schärfere Fokussierung auf das Objekt, was zu präziseren und detailreicheren Bildern führt.

Die Anpassung des Kamerawinkels und der Höhe der Säule hatte mehrere positive Auswirkungen auf die Bildqualität:

- Reflexionsverhinderung: Der neue Winkel und die angepasste Höhe halfen dabei, unerwünschte Reflexionen zu vermeiden. Diese Reflexionen hatten zuvor die Bild-

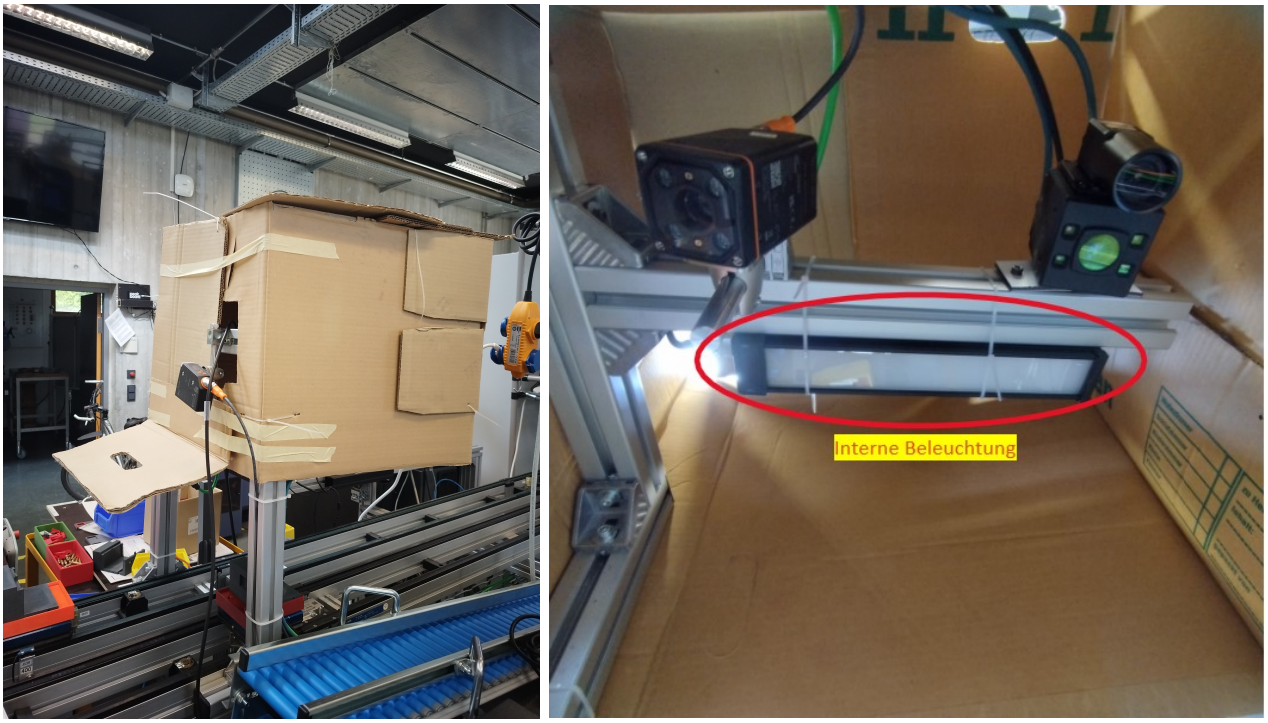
qualität beeinträchtigt und die Fehlererkennung erschwert. Mit den neuen Einstellungen werden reflektierende Oberflächen und Lichtquellen besser kontrolliert, was zu klareren Bildern führt.

- Scharfe Bilder: Die optimierte Distanz zwischen Kamera und Objekt ermöglicht eine präzisere Fokussierung. Dadurch werden schärfere und detailreichere Bilder erzeugt, was die Erkennung und Analyse von Fehlern erheblich erleichtert.

3.2 Tageslichtoptimierung

Die Optimierung des Tageslichteinfalls in der End-of-Line (EOL) Station stellt eine wichtige Aufgabe dar, um die Beleuchtung der Bauteile zu verbessern und die Prüfgenauigkeit zu erhöhen. In Produktionsumgebungen, in denen Präzision und Sichtbarkeit von entscheidender Bedeutung sind, kann unkontrolliertes Tageslicht die Erkennungsgenauigkeit beeinträchtigen. Dies gilt insbesondere für visuelle Inspektionsaufgaben, bei denen gleichmäßige Lichtverhältnisse erforderlich sind, um Fehler oder Unregelmäßigkeiten an den Bauteilen zu identifizieren.

Eine mögliche Maßnahme zur Regulierung des Tageslichteinfalls war die Verwendung eines Sonnenschirms, der das einfallende Licht abschirmen sollte. Bevor jedoch ein passender Sonnenschirm gefunden und bestellt wurde, sollte zunächst überprüft werden, ob diese Maßnahme tatsächlich eine Verbesserung bringen würde. Der erste Schritt bestand darin, einen provisorischen Abschirm aus Karton zu installieren, der mit einer internen Beleuchtung von Werner ausgestattet war. Diese Lösung sollte uns erste Einblicke in die Wirksamkeit einer dauerhaften Sonnenschirminstallation geben.



(a) Lichtschutz-Box **(b)** Box-Beleuchtung

Abbildung 3.3: Optimierung

Der provisorische Abschirm war zwar optisch nicht ansprechend und brachte technisch nur eine minimale Verbesserung, ermöglichte jedoch erste Erkenntnisse über die potenzielle Wirkung eines Sonnenschirms. Die interne Beleuchtung von Werner half, die Lichtverhältnisse besser zu kontrollieren, doch die Abschirmung aus Karton war weder langlebig noch besonders effektiv. Diese Testphase war dennoch entscheidend, um die grundsätzliche Machbarkeit der Lichtregulierung zu bewerten.

Es traten jedoch einige Herausforderungen auf, die die Umsetzung dieser Maßnahme erschwerten. Einerseits stellten die Kosten einen wesentlichen Faktor dar. Die Beschaffung und Installation eines geeigneten Sonnenschirms wären mit erheblichen finanziellen Aufwendungen verbunden gewesen. In einer Zeit, in der Budgets eng bemessen sind, war es schwierig, die notwendigen Mittel für eine solche Investition zu rechtfertigen.

Andererseits war die Station bereits mit anderen Geräten ausgestattet, deren Anordnung die Installation eines Sonnenschirms zusätzlich komplizierte. Die vorhandene Infrastruk-

tur bot wenig Flexibilität, und die räumlichen Gegebenheiten machten eine unkomplizierte Installation nahezu unmöglich. Eine umfassende Umstrukturierung der bestehenden Geräteanordnung wäre notwendig gewesen, um Platz für einen Sonnenschirm zu schaffen, was mit zusätzlichen Kosten und Aufwand verbunden gewesen wäre.

Zusätzlich wurde die Möglichkeit in Betracht gezogen, die Kamera an eine andere Position zu verlegen, um die Auswirkungen des Sonnenlichts zu minimieren. Diese Maßnahme hätte möglicherweise eine Verbesserung der Arbeitsbedingungen gebracht, da die Kamera dann weniger direkter Sonneneinstrahlung ausgesetzt wäre. Allerdings stellte sich heraus, dass eine solche Verlagerung aus zeitlichen Gründen nicht machbar war. Die Umsiedlung der Kamera hätte eine umfassende Neuplanung und Umsetzung erfordert, die den Rahmen unserer aktuellen zeitlichen Kapazitäten sprengte.

Insgesamt zeigte sich, dass die anfänglich favorisierte Lösung der Sonnenschirminstallation mit erheblichen Herausforderungen verbunden war. Angesichts der Kosten, der bestehenden Infrastruktur und der zeitlichen Einschränkungen war diese Maßnahme nicht praktikabel.

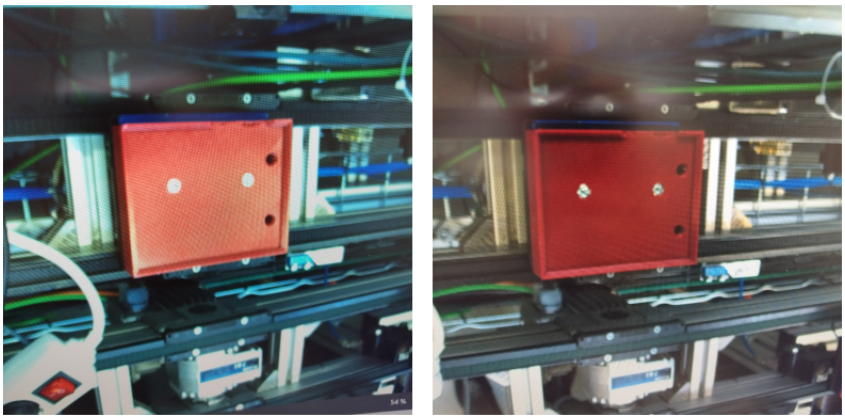


Abbildung 3.4: Bild links ohne Abdeckung, Bild rechts mit Abdeckung

Die Erfahrungen aus der Testphase mit dem provisorischen Abschirm haben jedoch wertvolle Einblicke geliefert, die in die Planung zukünftiger Maßnahmen einfließen können. Alternative Ansätze zur Tageslichtoptimierung werden nun geprüft. Dies könnte die Entwicklung und Implementierung weiterer innovativer Technologien zur Lichtsteuerung

oder die Anpassung der Arbeitsabläufe und -zeiten umfassen.

3.3 Software-Optimierung

In der vorliegenden Studienarbeit wurde untersucht, wie die Performance von Helios 3D Kameras in der Software Arenaview verbessert werden kann. Dabei wurden zunächst Softwareprobleme identifiziert, die zu Verzögerungen bei der Aktivierung von Operationen führten. Im Rahmen einer Reihe von Schritten und Tests wurde versucht, diese Probleme zu lösen und die Effizienz der Kameras zu steigern.

In einer vorherigen Studienarbeit (T3100) wurden die Helios Kameras auf unterschiedliche Weise an den PC angeschlossen. Eine Kamera wurde direkt über USB zu LAN mit eigener Spannungsversorgung an die PC-Netzwerkkarte angeschlossen, während die zweite Kamera über einen POE-fähigen Switch ohne eigene Spannungsversorgung verbunden wurde. Bei Versuchen, Daten von beiden Kameras zu streamen, traten erhebliche Verzögerungen auf, und die Software reagierte manchmal nicht mehr, was sogar zu Softwareabstürzen führte.

Ein Hauptproblem war die begrenzte Bandbreite des PCs, der nur über einen Netzwerkanschluss auf der Netzwerkkarte verfügte. Eine direkte Verbindung beider Kameras über einen Switch war aufgrund unzureichender Bandbreite nicht möglich. Um das Problem der Verzögerungen zu lösen, wurde zunächst ein neuer POE-fähiger Switch beschafft, um die Bandbreite zu erhöhen. Trotz des Austauschs des Switches blieben die Probleme bestehen. Die Kameras wurden in der Software (Arenaview) nicht erkannt, was darauf hinwies, dass das Problem nicht ausschließlich an der Bandbreite lag.

Um die Situation zu verbessern, wurde beschlossen, eine zusätzliche Netzwerkkarte zu beschaffen und in den PC einzubauen. Gleichzeitig wurde eine separate Spannungsversorgung für beide Kameras bereitgestellt, um sicherzustellen, dass sie über ausreichend Leistung und Bandbreite verfügten, um unabhängig voneinander zu arbeiten.

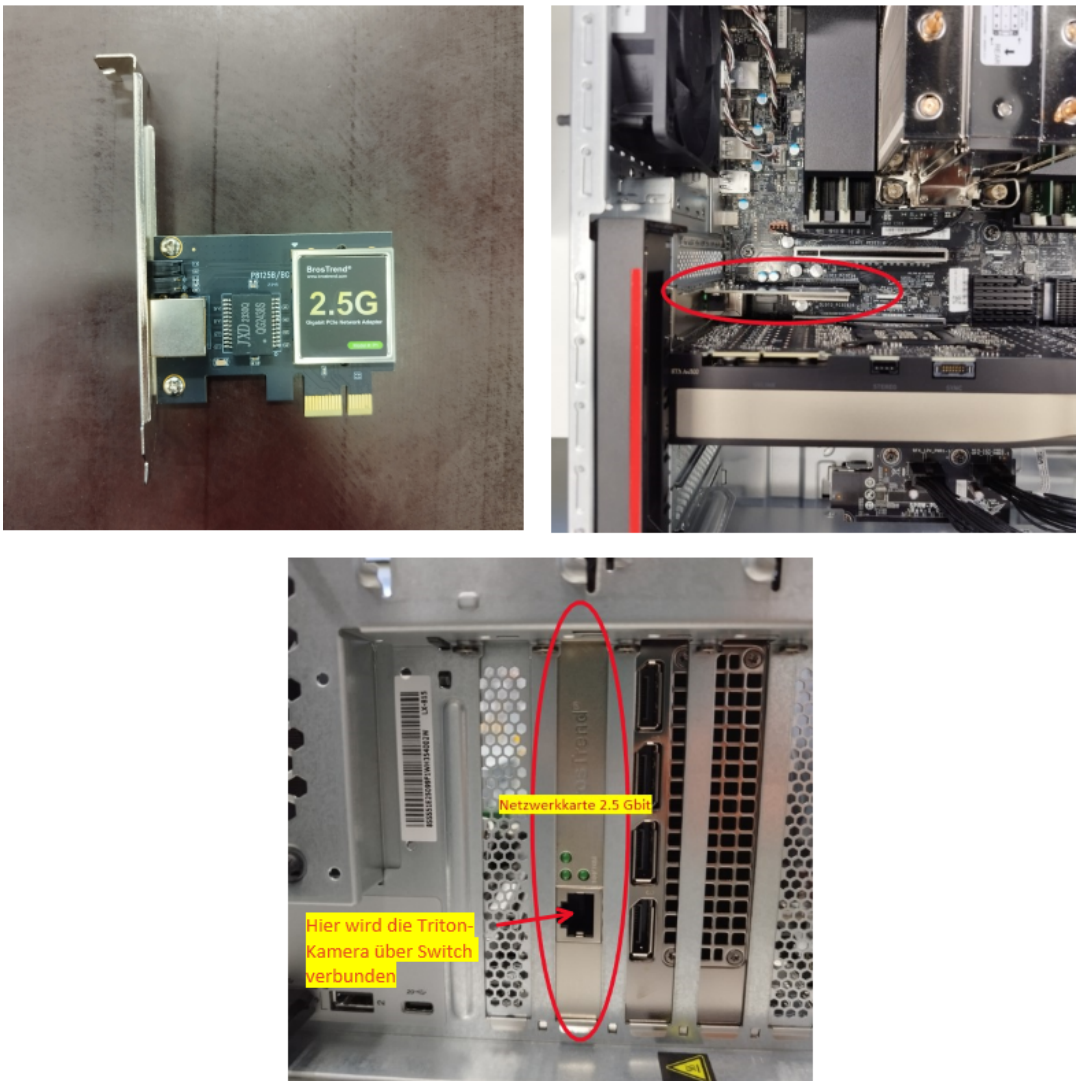


Abbildung 3.5: Netzwerkkarte

Diese Maßnahmen erwiesen sich als äußerst erfolgreich. Nach der Implementierung der zusätzlichen Netzwerkkarte und der separaten Stromversorgung wurden die Verzögerungen erheblich reduziert, und die Kameras konnten in der Software (Arenaview) schnell und zuverlässig erkannt werden. Die Operationen, wie das Einschalten und Streamen von Daten, erfolgten nun deutlich schneller und effizienter.

Diese erfolgreiche Softwareoptimierung zeigte, wie durch gezielte Anpassungen an der Hardwarekonfiguration und der Netzwerkinfrastruktur die Leistungsfähigkeit von Geräten wie den Helios 3D Kameras erheblich verbessert werden kann. Durch die Identifizie-

rung und Behebung von Engpässen konnten die Herausforderungen erfolgreich gemeistert und eine reibungslose Funktionsweise der Kameras sichergestellt werden.

4 Punktwolkenvergleich

In Kapitel 4 wird erläutert, wie ein Punktwolkenvergleich funktioniert und wie ein passender Algorithmus dafür entwickelt werden kann. Der Algorithmus soll pro Ausführung einen Snapshot in ArenaView von der Punktwolke des Bauteils aufnehmen. Diese wird als Polygon File Format (PLY) gespeichert. Die zwei Punktwolken sollen dann ausgelesen und verglichen werden. Der Output soll ein separates Fenster öffnen, in dem die zwei Punktwolken farblich voneinander getrennt visualisiert werden. Zudem soll angegeben werden, ob die überprüfte Punktwolke von der idealen Punktwolke abweicht.

4.1 Methoden

Der Vergleich von zwei Punktwolken kann auf verschiedene Weisen erfolgen. Im Allgemeinen betrachtet man dabei die räumliche Übereinstimmung der Punkte in den beiden Wolken. Die gängigsten Methoden zum Vergleich von Punktwolken sind:

1. Hausdorff-Abstand: Der Hausdorff-Abstand ist eine Metrik, die die maximale Distanz zwischen zwei Punktwolken misst. Er wird berechnet, indem man für jeden Punkt in einer Wolke den nächstgelegenen Punkt in der anderen Wolke findet und den größten dieser Abstände nimmt. Diese maximale Abweichung zeigt wie ähnlich oder unterschiedlich die beiden Punktwolken sind. Er ist gut für globale Vergleiche, aber empfindlich gegen Ausreißern.[1]

2. Cloud-to-Cloud: Die C2C-Methode berechnet die Distanzen zwischen zwei Punktwolken, die zu verschiedenen Zeitpunkten aufgenommen wurden. Dabei wird für jeden Punkt der zu vergleichenden Wolke die Distanz zur Referenzwolke berechnet. Das Ergebnis ist ein Distanzwert für jeden Punkt in der zu vergleichenden Wolke. Diese Werte geben Aufschluss über die räumlichen Unterschiede oder Ähnlichkeiten zwischen den beiden Punktwolken. [2]

3. Iterative Closest Point (ICP): Der Iterative Closest Point Algorithmus ermöglicht es, dass zwei ähnliche, aber unterschiedlich orientierte oder positionierte Punktwolken aufeinander abzubilden. Der Algorithmus basiert auf einem iterativen Ansatz, bei dem die Quellpunktwolke schrittweise an die Zielpunktwolke angepasst wird. Bei jedem Durchlauf wird für jeden Punkt der Quellwolke der nächstgelegene Punkt in der Zielwolke identifiziert. Basierend auf diesen Punktpaaren wird dann eine optimale Transformation ermittelt, welche auf die Quellpunktwolke angewendet wird, und dieser Prozess wiederholt sich, bis ein zufriedenstellendes Ergebnis dabei rauskommt.Das Endergebnis, die die Ähnlichkeit zwischen den Punktwolken maximiert. Der verbleibende Fehler nach der Anpassung gibt Aufschluss über die Übereinstimmung zwischen den verglichenen Punktwolken. [3]

4.2 Umsetzung

4.2.1 JupyterLab

Die Implementierung des Projekts basiert auf JupyterLab, einer webbasierten Entwicklungsumgebung, die sich durch ihre Fähigkeit zur interaktiven Verarbeitung von Notebooks, Code und Daten auszeichnet. JupyterLab ist bereits in dem Softwaretool Arenaview eingebettet, wodurch man es separat nicht nochmal installieren muss. Die Aktivierung von JupyterLab erfolgt über die Benutzeroberfläche von Arenaview: In den Konfigurationseinstellungen von Arenaview befindet sich der Reiter JupyterLab und dort die Option "Toggle the Jupyter Lab Server", deren Betätigung den automatischen Start des Systems initiiert.

Für die Benutzung bzw. für den Zugriff auf JupyterLab stehen dem Anwender zwei Optionen zur Verfügung:

1. Die direkte Nutzung innerhalb der Arenaview-Umgebung selbst, welche in einer integrierten Registerkarte geöffnet wird.

2. Der externe Zugriff über einen Browser unter Verwendung der Serveradresse in der Kommandozelle.

In dieser Entwicklungsumgebung kann der Quellcode für die Durchführung des Punktwolkenvergleichs nicht nur entwickelt, sondern auch unmittelbar ausgeführt werden. Dies ermöglicht eine effiziente und dynamische Herangehensweise an die Datenanalyse und -verarbeitung im Rahmen des Projekts.

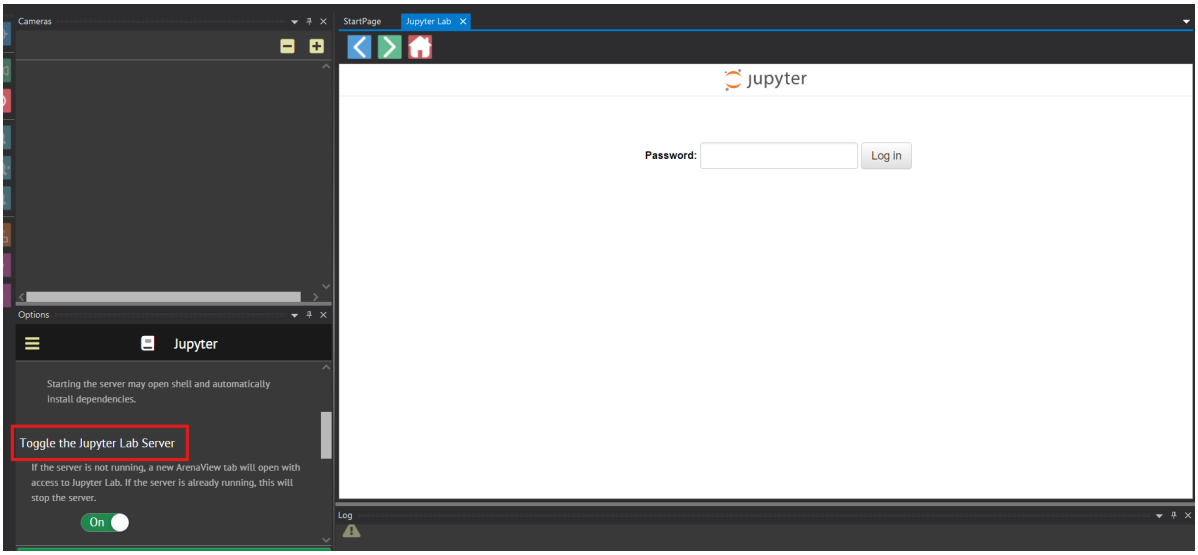


Abbildung 4.1: JupyterLab im Arenaview

4.2.2 Algorithmus

Aufgrund von Komplikationen war es nicht möglich, qualitativ hochwertige Aufnahmen der Bauteile und daraus resultierende Punktwolken zu erstellen. Um das Problem zu beheben, haben wir uns dazu entschlossen, eigene farbige Punktwolken im JupyterLab zu generieren. Das dient dazu, um unseren

4.2.2.1 Punktwolkenerstellung

Der folgende Quellcode soll zwei Punktwolken erstellen: eine ideale Punktwolke, die aus einer Kugel mit einem herausragenden Stab besteht und einer Fehlerpunktwolke, die nur aus der Kugel besteht.

Erklärung des Quellcodes:

1. Bevor wir die Bibliotheken importieren können, müssen diese zunächst installiert werden. Über die Kommandozeile oder direkt im Notebook kann der Befehl *pip install* verwendet werden, um die Bibliotheken zu installieren.

```
pip install numpy open3d
```

   Bei den Bibliotheken handelt es sich um *Numpy*, welche für die numerische Berechnung sowie Arrayoperationen dient und *Open3D* für die 3D-Datenverarbeitung.

```
import numpy as np
import open3d as o3d
```

2. In der Hauptfunktion werden die Punktwolken mit anpassbaren Parametern generiert. Bei *Num-points* handelt es sich um die Anzahl der Punkte für die Kugel. *Stick-points* gibt die Anzahl der Punkte für den Stab an. *Stick-length* ist die Länge des Stabs und *Stick-radius* ist der Radius des Stabs.

```
def create_sample_pointclouds(num_points=10000, stick_points=2000,
↪  stick_length=0.5, stick_radius=0.05):
```

3. Mithilfe der Parameter werden zunächst die Winkel Theta und Phi berechnet. Unter Verwendung des Radius werden die Kugelkoordinaten in kartesische Koordinaten umgewandelt. Diese Kugelpunkte werden dann in *sphere-points* gespeichert.

```
theta = np.random.uniform(0, np.pi, num_points)
phi = np.random.uniform(0, 2*np.pi, num_points)
r = 1  # Radius der Kugel
x = r * np.sin(theta) * np.cos(phi)
y = r * np.sin(theta) * np.sin(phi)
z = r * np.cos(theta)

sphere_points = np.column_stack((x, y, z))
```

4. Im nächsten Schritt wird der Stab erstellt. Dabei wird ähnlich wie bei der Kugel vorgegangen. Zuerst werden Theta und die Höhe berechnet, welche zur Bestimmung der kartesischen Koordinaten x, y und z dienen. Diese Punkte werden dann in *stick-points gespeichert.*

```
stick_theta = np.random.uniform(0, 2*np.pi, stick_points)
stick_height = np.random.uniform(0, stick_length, stick_points)
stick_x = stick_radius * np.cos(stick_theta)
stick_y = stick_radius * np.sin(stick_theta)
stick_z = stick_height + r  # Der Stick ragt aus der Kugel heraus

stick_points = np.column_stack((stick_x, stick_y, stick_z))
```

5. Anschließend wird für die ideale Punktwolke die Punkte von der Kugel und dem Stab kombiniert und in *ideal-points* gespeichert. Für die Fehlerpunktwolke werden nur die Punkte der Kugel gespeichert.

```
ideal_points = np.vstack((sphere_points, stick_points))

error_points = sphere_points.copy()
```

6. Jetzt wird über Open3D die Punktwolke erstellt. Dafür wird zunächst ein PointCloud-Objekt erstellt. Die *ideal-points* werden dann eingesetzt und abschließend wird für die visuelle Unterscheidung die Punktwolke gefärbt. Das Gleiche gilt für die Fehlerpunktwolke, wobei hier andere Punkte verwendet und eine andere Farbe zugewiesen werden.

```
pcd_ideal = o3d.geometry.PointCloud()
pcd_ideal.points = o3d.utility.Vector3dVector(ideal_points)
pcd_ideal.paint_uniform_color([0, 1, 0])  # Grün für die ideale
↪  Punktwolke

pcd_error = o3d.geometry.PointCloud()
pcd_error.points = o3d.utility.Vector3dVector(error_points)
pcd_error.paint_uniform_color([0, 0, 1])  # Blau für die
↪  Fehlerpunktwolke
```

7. Abschließend werden die Punktwolken als PLY-Datei gespeichert und die Funktion für die Punktwolkenerstellung ausgeführt.

```
    o3d.io.write_point_cloud("Ideal_PCD.ply", pcd_ideal)
    o3d.io.write_point_cloud("Error_PCD.ply", pcd_error)
```

```
    print("Punktwolken wurden erstellt und als 'Ideal_PCD.ply' und
    ↪  'Error_PCD.ply' gespeichert.")

#Iniitierung der Punktwolkenerstellung
if __name__ == "__main__":
    create_sample_pointclouds()
```

4.2.2.2 Vergleich

Für den Vergleich wurden zwei Ansätze entwickelt, die beide das gleiche Verfahren für den Punktwolkenvergleich verwenden: NearestNeighbor von scikit-learn, welches auf den *Cloud to Cloud Comparasion* basiert. Sie unterscheiden sich lediglich im Output. Vorab müssen zusätzlich zu den bereits installierten Bibliotheken noch weitere installiert werden, nämlich scikit-learn, plyfile und matplotlib. Über die Bibliothek scikit-learn wird der Nearest-Neighbors-Algorithmus implementiert. Plyfile dient zum Lesen und Schreiben von PLY-Dateien und matplotlib wird für die grafische Visualisierung der Punktwolken verwendet.

```
pip install scikit-learn plyfile matplotlib
```

Der erste Ansatz basiert auf die Visualisierung mit matplotlib:

1. Zu Beginn des Codes werden die notwendigen Bibliotheken importiert. Für die Verwendung des Qt5Agg-Renderers ist es erforderlich, dass PyQt5 installiert ist. Das kann durch Ausführung des Befehls pip install PyQt5 erfolgen. Alternativ kann auch der TkAgg-Renderer verwendet werden, falls PyQt5 nicht verfügbar ist.

```
import numpy as np
from sklearn.neighbors import NearestNeighbors
from plyfile import PlyData, PlyElement
import matplotlib
matplotlib.use('Qt5Agg')  #Alternativ TkAgg verwenden
import matplotlib.pyplot as plt
```

2. **load-ply:**

Die Funktion *load-ply* liest eine PLY-Datei ein und entnimmt die kartesischen Koordinaten der Punktwolke. Dabei wird überprüft, ob die Datei Farbinformationen enthält. Falls Farbinformationen vorhanden sind, werden diese ebenfalls entnommen. Anschließend werden alle Informationen in einem Array gespeichert.

```
def load_ply(filename):
plydata = PlyData.read(filename)
x = plydata['vertex']['x']
y = plydata['vertex']['y']
z = plydata['vertex']['z']

if 'red' in plydata['vertex'].data.dtype.names:
    r = plydata['vertex']['red']
    g = plydata['vertex']['green']
    b = plydata['vertex']['blue']
    return np.column_stack((x, y, z, r, g, b))
else:
    return np.column_stack((x, y, z))
```

3. **compare-point-clouds**:

Im nächsten Abschnitt werden über die Funktion *compare-point-clouds* zwei Punktwolken verglichen. Zunächst werden die Koordinaten wieder entnommen. Daraufhin wird ein Nearest-Neighbor-Klassifikator mit dem *ball-tree* Algorithmus erstellt. Der Ball Tree ist eine Baumstruktur, die den Raum rekursiv in Bälle unterteilt. Der Algorithmus beginnt mit allen Punkten in einem großen Ball. Anschließend wird dieser Ball in zwei kleinere Bälle aufgeteilt, wobei jeder Unterbaum etwa die Hälfte der Punkte enthält. Dies wird rekursiv so lange fortgesetzt, bis jeder Ball nur noch wenige Punkte enthält.

Daraufhin wird für jeden Punkt in *xyz1* der nächste Nachbar in *xyz2* gesucht. Danach werden die Punkte identifiziert, die innerhalb des Schwellenwerts liegen. Punkte mit einem Abstand kleiner als der Schwellenwert werden als übereinstimmend betrachtet. Anschließend werden die übereinstimmenden Punkte gezählt und der Prozentsatz berechnet. Abschließend wird die Differenz-Punktwolke erstellt, welche die nicht übereinstimmenden Punkte darstellt. Diese werden rot gefärbt.

```
def compare_point_clouds(cloud1, cloud2, threshold=0.001):
xyz1 = cloud1[:, :3]
xyz2 = cloud2[:, :3]

nn = NearestNeighbors(n_neighbors=1, algorithm='ball_tree').fit(xyz2)
distances, indices = nn.kneighbors(xyz1)

matching_indices = distances.flatten() < threshold
matching_points = np.sum(matching_indices)
percentage_matching = (matching_points / len(xyz1)) * 100

diff_cloud = np.copy(cloud1)

# Wenn die Farbfelder existieren, werden die nicht übereinstimmenden
↪  Punkte rot gefärbt
if cloud1.shape[1] > 3:
```

```
    diff_cloud[~matching_indices, 3:] = [255, 0, 0]

return diff_cloud, matching_points, percentage_matching
```

4. **save-ply:**

Diese Funktion speichert die Punktwolke im PLY-Format. Zunächst wird überprüft, ob die Punktwolke Farbwerte enthält, indem festgestellt wird, ob das Array mehr als drei Spalten besitzt. Anschließend wird ein strukturierter Array erstellt, in dem jeder Punkt in ein Tupel umgewandelt wird. Die Koordinaten x, y und z sind vom Typ f4, d.h. 32-bit Floating Point, und die Farben Rot, Grün und Blau sind vom Typ u1, d.h. 8-Bit Unsigned Integer. Aus dem Array wird dann ein PLY-Element erstellt, und im Anschluss wird aus diesem Element ein PLY-Datenobjekt erzeugt und gespeichert.

```
def save_ply(filename, points):
# Überprüfen, ob die Punkte Farbfelder haben
if points.shape[1] > 3:
    vertex = np.array([tuple(p) for p in points],
                      dtype=[('x', 'f4'), ('y', 'f4'), ('z', 'f4'),
                             ('red', 'u1'), ('green', 'u1'), ('blue',
                             ↪  'u1')])
else:
    vertex = np.array([tuple(p) for p in points],
                      dtype=[('x', 'f4'), ('y', 'f4'), ('z', 'f4')])
el = PlyElement.describe(vertex, 'vertex')
PlyData([el]).write(filename)
```

5. **visualize-point-clouds:**

Die Funktion visualize-point-clouds dient der grafischen Darstellung der drei Punktwolken (ideal, fehlerhaft und Differenz) in einer einzigen Abbildung. Sie nutzt Matplotlib, um eine Figur mit drei 3D-Subplots zu erstellen, wobei jeder Subplot eine der Punktwolken repräsentiert. Für jede Punktwolke gibt es ein 3D-Koordinatensystem, sowie einen Titel und beschriftete Achsen.

```
def visualize_point_clouds(cloud1, cloud2, diff_cloud):
plt.rcParams['figure.figsize'] = [10, 10]  # Vergrößert die Gesamtfigur
fig = plt.figure()

# Punktwolke Ideal
ax1 = fig.add_subplot(131, projection='3d')
ax1.scatter(cloud1[:, 0], cloud1[:, 1], cloud1[:, 2], c=(cloud1[:,
↪  3:]/255 if cloud1.shape[1] > 3 else 'b'), s=2)
ax1.set_title('PCD_ideal', fontsize=16)

# Punktwolke Error
ax2 = fig.add_subplot(132, projection='3d')
ax2.scatter(cloud2[:, 0], cloud2[:, 1], cloud2[:, 2], c=(cloud2[:,
↪  3:]/255 if cloud2.shape[1] > 3 else 'b'), s=2)
ax2.set_title('PCD_error', fontsize=16)

# Differenz-Punktwolke
ax3 = fig.add_subplot(133, projection='3d')
ax3.scatter(diff_cloud[:, 0], diff_cloud[:, 1], diff_cloud[:, 2],
↪  c=(diff_cloud[:, 3:]/255 if diff_cloud.shape[1] > 3 else 'b'), s=2)
ax3.set_title('PCD_Difference', fontsize=16)

for ax in [ax1, ax2, ax3]:
    ax.set_xlabel('X', fontsize=12)
    ax.set_ylabel('Y', fontsize=12)
    ax.set_zlabel('Z', fontsize=12)
    ax.tick_params(labelsize=10)
```

```
plt.tight_layout()
#plt.savefig('70.png', dpi=300, bbox_inches='tight')
plt.show()
```

6. **main:**

In der main-Funktion werden zuerst die erstellten Punktwolken geladen. Danach erfolgt ein Vergleich der Punktwolken sowie deren Visualisierung. Schließlich werden Informationen zu den Punktwolken ausgegeben, darunter die Anzahl der Punkte, die Anzahl der übereinstimmenden Punkte und der Prozentsatz der Übereinstimmung.

```
def main():
    cloud_ideal = load_ply("Ideal_PCD.ply")
    cloud_error = load_ply("Error_PCD.ply")

    diff_cloud, matching_points, percentage_matching =
    ↪  compare_point_clouds(cloud_ideal, cloud_error)

    save_ply("difference_cloud.ply", diff_cloud)

    print(f"Anzahl der Punkte in Punktwolke_ideal: {len(cloud_ideal)}")
    print(f"Anzahl der Punkte in Punktwolke_error2: {len(cloud_error)}")
    print(f"Anzahl der übereinstimmenden Punkte: {matching_points}")
    print(f"Prozentsatz der übereinstimmenden Punkte:
    ↪  {percentage_matching:.2f}%")

    if len(cloud_ideal) == len(cloud_error):
        print(f"Die Punktwolken haben die gleiche Anzahl von Punkten.")
        print(f"Anzahl der nicht übereinstimmenden Punkte:
        ↪  {len(cloud_ideal) - matching_points}")
```

```
    else:
        print(f"Die Punktwolken haben unterschiedliche Anzahlen von
        ↪ Punkten.")
        print(f"Differenz in der Punktanzahl: {abs(len(cloud_ideal) -
        ↪ len(cloud_error))}")

    visualize_point_clouds(cloud_ideal, cloud_error, diff_cloud)

if __name__ == "__main__":
    main()
```

Der zweite Ansatz basiert auf die Visualisierung mit Open3D:
Bis auf die Visualisierung, ist der Quellcode identisch zum ersten Ansatz

1. Importieren der Bibliotheken:

```
import numpy as np
import open3d as o3d
from sklearn.neighbors import NearestNeighbors
```

2. **compare-point-clouds**: Diese Funktion vergleicht zwei Punktwolken, indem sie den Ball-Tree-Algorithmus für eine effiziente Nächste-Nachbarn-Suche verwendet. Zunächst wird für jeden Punkt in der ersten Punktwolke (cloud1) der nächstgelegene Nachbar in der zweiten Punktwolke (cloud2) ermittelt. Dabei werden Punkte, die innerhalb eines bestimmten Schwellenwerts liegen, als übereinstimmend betrachtet. Anschließend berechnet die Funktion die Anzahl und den Prozentsatz der übereinstimmenden Punkte.

```
def compare_point_clouds(cloud1, cloud2, threshold=0.01):
nn = NearestNeighbors(n_neighbors=1, algorithm='ball_tree').fit(cloud2)
distances, _ = nn.kneighbors(cloud1)

matching_indices = distances.flatten() < threshold
matching_points = np.sum(matching_indices)
percentage_matching = (matching_points / len(cloud1)) * 100

return matching_indices, matching_points, percentage_matching
```

3. **visualize-point-clouds:**

Diese Funktion visualisiert die Punktwolken, indem sie eine Differenz-Punktwolke erstellt, bei der nicht übereinstimmende Punkte rot gefärbt werden. Die Punktwolken werden horizontal verschoben, um sie nebeneinander darzustellen, und für jede Punktwolke werden Koordinatensysteme hinzugefügt. Schließlich wird mithilfe von Open3D's mit *draw-geometries* gezeichnet, um eine interaktive 3D-Visualisierung aller Punktwolken und Koordinatensysteme zu ermöglichen.

```
def visualize_point_clouds(pcd_ideal, pcd_error, matching_indices):
# Erstellunf der Differenz-Punktwolke
pcd_diff = o3d.geometry.PointCloud()
pcd_diff.points = pcd_ideal.points
diff_colors = np.array(pcd_ideal.colors)
diff_colors[~matching_indices] = [1, 0, 0]  # Rot für nicht
↪  übereinstimmende Punkte
pcd_diff.colors = o3d.utility.Vector3dVector(diff_colors)

# Verschiebung der Punktwolken, sodass sie nebeneinander stehen
pcd_ideal.translate([-2.5, 0, 0])
pcd_error.translate([0, 0, 0])
pcd_diff.translate([2.5, 0, 0])
```

```
# KS für jede Punktwolke
coord_ideal = o3d.geometry.TriangleMesh.create_coordinate_frame(size=0.
↪  5).translate([-2.5, -1.5,
↪  -1.5])
coord_error = o3d.geometry.TriangleMesh.create_coordinate_frame(size=0.
↪  5).translate([0, -1.5,
↪  -1.5])
coord_diff = o3d.geometry.TriangleMesh.create_coordinate_frame(size=0.5
↪  ).translate([2.5, -1.5,
↪  -1.5])

# Visualisierung
o3d.visualization.draw_geometries([pcd_ideal, pcd_error, pcd_diff,
↪  coord_ideal, coord_error, coord_diff],
                                  zoom=0.5,
                                  front=[0, 0, -1],
                                  lookat=[0, 0, 0],
                                  up=[0, 1, 0],
                                  window_name="Point Cloud Comparision")
```

4. **main:**

Die main-Funktion startet den Ablauf des Punktwolkenvergleichs und dessen Visualisierung. Zunächst werden die Punktwolken geladen und in numpy-Arrays konvertiert, um den Vergleich zu ermöglichen. Anschließend werden die Vergleichs- und die Visualisierungsfunktion für die Punktwolken aufgerufen. Abschließend werden Informationen zu den Punktwolken, wie die Anzahl der Punkte und übereinstimmende Punkte, ausgegeben.

```
    def main():
    # Speicherung der Punktwolken
```

```
    pcd_ideal = o3d.io.read_point_cloud("Ideal_PCD.ply")
    pcd_error = o3d.io.read_point_cloud("Error_PCD.ply")

    ideal_points = np.asarray(pcd_ideal.points)
    error_points = np.asarray(pcd_error.points)

    # Vergleich der Punktwolken
    matching_indices, matching_points, percentage_matching =
    ↪  compare_point_clouds(ideal_points, error_points)

    print(f"Anzahl der Punkte in Punktwolke_ideal: {len(ideal_points)}")
    print(f"Anzahl der Punkte in Punktwolke_error2:
    ↪  {len(error_points)}")
    print(f"Anzahl der übereinstimmenden Punkte: {matching_points}")
    print(f"Prozentsatz der übereinstimmenden Punkte:
    ↪  {percentage_matching:.2f}%")

    # Aufruf zur Visualisierung der Punktwolken
    visualize_point_clouds(pcd_ideal, pcd_error, matching_indices)

if __name__ == "__main__":
    main()
```

4.2.3 Ergebnis

Erster Ansatz mittels Matplotlib:

In der Visualisierung sind alle drei Punktwolken erkennbar: Die ideale Punktwolke in grün, die fehlerhafte Punktwolke in blau und die Differenz-Punktwolke in rot. Jede Punktwolke besitzt ihr eigenes dreidimensionales Koordinatensystem. Es ist möglich, die Punktwolken zu rotieren oder die Darstellung mittels der Einstellungen oben links

zu verschieben. Ein Nachteil dieser Visualisierung ist die Performance. Wenn man versucht, die Punktwolken zu bewegen oder zu rotieren, beginnt es zu ruckeln und die Framerate sinkt. Generell ist diese Visualisierung daher nicht sehr stabil und kann bei der Darstellung zu erheblichen Leistungseinbußen führen. Aus diesem Grund gibt es noch den zweiten Ansatz.

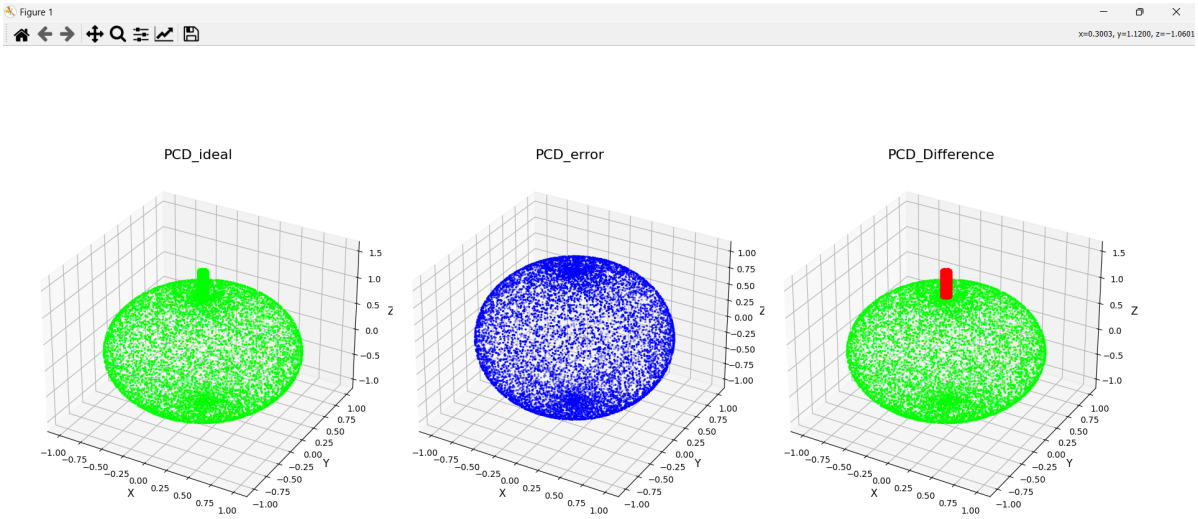


Abbildung 4.2: Punktwolkenvergleich - Matplotlib

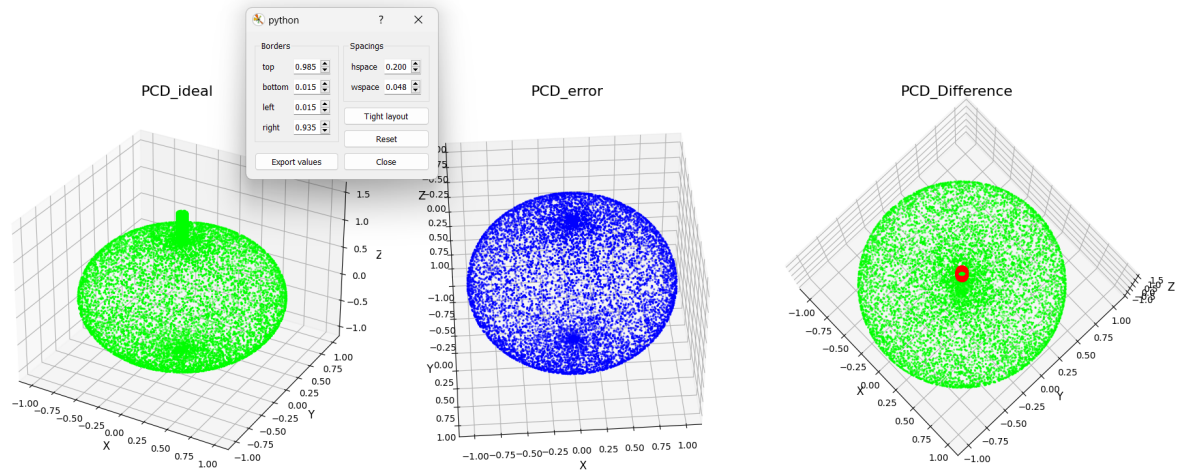


Abbildung 4.3: Punktwolkenvergleich mittles Matplotlib- Rotierte Ansicht

Zweiter Ansatz mittels Open3D:

In der Visualisierung sind alle drei Punktwolken zu erkennen: die ideale Punktwolke in grün, die fehlerhafte Punktwolke in blau und die Differenz-Punktwolke in rot. Jede Punktwolke besitzt ein eigenes kleines Koordinatensystem. Es ist möglich, die Punktwolken zu rotieren sowie die Ansicht zu vergrößern bzw. zu verkleinern. Allerdings fehlen zusätzliche Einstellungsmöglichkeiten. Ein Vorteil gegenüber der Matplotlib-Darstellung ist die verbesserte Leistung. Nachteile sind die fehlenden Beschriftungen und die generell schwierigere Strukturierung der Darstellung in Open3D.

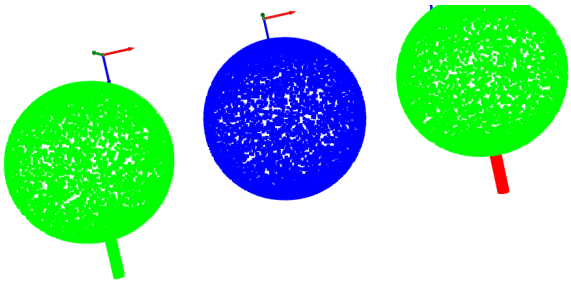


Abbildung 4.4: Vergleich der Punktwolken mittels Open3D

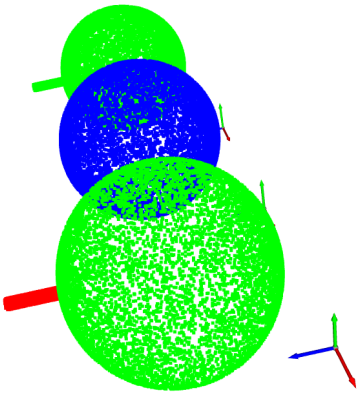


Abbildung 4.5: Vergleich der Punktwolken mittels Open3D - Rotierte Ansicht

```
Jupyter environment detected. Enabling Open3D WebVisualizer.
[Open3D INFO] WebRTC GUI backend enabled.
[Open3D INFO] WebRTCWindowSystem: HTTP handshake server disabled.
Anzahl der Punkte in Punktwolke_ideal: 12000
Anzahl der Punkte in Punktwolke_error2: 10000
Anzahl der übereinstimmenden Punkte: 10019
Prozentsatz der übereinstimmenden Punkte: 83.49%
```

Abbildung 4.6: Ausgabe des Punktwolkenvergleichs

5 Zusammenfassung und Ausblick

Diese Studienarbeit T3200 baute auf den Ergebnissen der vorherigen Arbeit T3100 auf und zielte darauf ab, die End-of-Line Teststation durch den Einsatz von 3D-Kameratechnik weiter zu optimieren. Die Hauptziele umfassten die Verbesserung der Bildqualität, die Entwicklung eines Programms zur Fehlererkennung und die Steigerung der Gesamtleistung der Station.

Folgende Errungenschaften wurden erzielt:

- Distanz- und Winkeloptimierung: Durch Anpassung der Kameraposition und des Winkels konnte die Bildqualität verbessert werden.

- Tageslichtoptimierung: Es wurden Versuche unternommen, den Einfluss von Tageslicht zu reduzieren, wobei sich eine vollständige Umsetzung aufgrund von Kosten- und Platzgründen als schwierig erwies.

- Software-Optimierung: Durch die Installation einer zusätzlichen Netzwerkkarte und den Anschluss einer separaten Stromversorgung konnte die Performance der Kameras deutlich gesteigert werden.

- Punktwolkenvergleich: Es wurde ein Algorithmus entwickelt, der zwei Punktwolken vergleicht und Unterschiede visualisiert. Dabei wurden zwei Ansätze verfolgt - einer basierend auf Matplotlib und einer auf Open3D.

Für zukünftige Arbeiten und Weiterentwicklungen ergeben sich folgende Ansatzpunkte:

- Tageslichtoptimierung: Der Lichtschutz kann noch weiter ausgebaut werden. Wenn möglich, sollte eine externe Lichtquelle eingebaut werden, um einen gleichmäßigen Lichteinfall zu jeder Zeit zu gewährleisten.

- Verbesserung des Algorithmus: Der Algorithmus für den Punktwolkenvergleich kann noch optimiert werden. Derzeit ist die Visualisierung noch ausbaufähig, und der Quellcode für die Aufnahme der Bauteile muss noch implementiert werden. Außerdem fehlen noch die Bezeichnungen für die Fehler.

- Distanz: Die Distanz kann noch weiter erhöht werden, um eine bessere Kalibrierung der Kameras zu ermöglichen.

- Equipment: Es könnten bzw. müssten weitere Netzwerkanschlüsse eingebaut werden, da es möglich ist, dass andere Gruppen diese Anschlüsse belegen.

Literatur

[1] Andrew Bell, Brad Chambers, Howard Butler u. a. *hausdorff*. 2. Januar 2024. URL: https://pdal.io/en/2.6.0/apps/hausdorff.html.

[2] Vitali Diaz u. a. „Comparison of Cloud-to-Cloud Distance Calculation Methods - Is the Most Complex Always the Most Suitable?“ In: *Recent Advances in 3D Geoinformation Science*. Lecture Notes in Geoinformation and Cartography. Springer Nature Switzerland, 2024, S. 329–330. DOI: 10.1007/978-3-031-43699-4_20.

[3] Wikepedia. *Iterative closest point*. 2. Juni 2024. URL: https://en.wikipedia.org/wiki/Iterative_closest_point.

Abbildungsverzeichnis

2.1 EOL Station 4

3.1 Winkeloptimierung 9
3.2 Abstandoptimierung 10
3.3 Optimierung 12
3.4 Bild links ohne Abdeckung, Bild rechts mit Abdeckung 13
3.5 Netzwerkkarte 15

4.1 JupyterLab im Arenaview 19
4.2 Matplotlib - Visualisierung 33
4.3 Matplotlib - Visualisierung2 33
4.4 Open3D - Visualisierung 34
4.5 Open3D - Visualisierung2 34
4.6 Open3D - Output 35

Anhang A

A.1 Punktwolkenerstellung - Quellcode

```
import numpy as np
import open3d as o3d

def create_sample_pointclouds(num_points=10000, stick_points=2000,
stick_length=0.5, stick_radius=0.05):

    # Erstellt eine Kugel als Punktwolke
    theta = np.random.uniform(0, np.pi, num_points)
    phi = np.random.uniform(0, 2*np.pi, num_points)
    r = 1  # Radius der Kugel
    x = r * np.sin(theta) * np.cos(phi)
    y = r * np.sin(theta) * np.sin(phi)
    z = r * np.cos(theta)

    sphere_points = np.column_stack((x, y, z))

    # Erstellt zusätzlich zu der Kugel einen Stick, der von der Kugel aus rausragt
    stick_theta = np.random.uniform(0, 2*np.pi, stick_points)
    stick_height = np.random.uniform(0, stick_length, stick_points)
    stick_x = stick_radius * np.cos(stick_theta)
    stick_y = stick_radius * np.sin(stick_theta)
    stick_z = stick_height + r  # Der Stick ragt aus der Kugel heraus
    stick_points = np.column_stack((stick_x, stick_y, stick_z))

    # Kombiniere Kugel und Stick für die ideale Punktwolke
    ideal_points = np.vstack((sphere_points, stick_points))

    # Erstellt die Fehlerpunktwolke (nur die Kugel)
    error_points = sphere_points.copy()
```

```
    # Erstellt über Open3D die Punktwolke
    pcd_ideal = o3d.geometry.PointCloud()
    pcd_ideal.points = o3d.utility.Vector3dVector(ideal_points)
    pcd_ideal.paint_uniform_color([0, 1, 0])  # Grün für die ideale Punktwolke

    pcd_error = o3d.geometry.PointCloud()
    pcd_error.points = o3d.utility.Vector3dVector(error_points)
    pcd_error.paint_uniform_color([0, 0, 1])  # Blau für die Fehlerpunktwolke

    # Speichert die Punktwolken als PLY-Dateien
    o3d.io.write_point_cloud("Ideal_PCD1.ply", pcd_ideal)
    o3d.io.write_point_cloud("Error_PCD2.ply", pcd_error)

    print("Punktwolken wurden erstellt und als 'Ideal_PCD.ply' und 'Error_PCD.ply'
    gespeichert.")

#Iniitierung der Punktwolkenerstellung
if __name__ == "__main__":
    create_sample_pointclouds()
```

A.2 Punktwolkenvergleich - Matplotlib - Quellcode

```
import numpy as np
from sklearn.neighbors import NearestNeighbors
from plyfile import PlyData, PlyElement
import matplotlib
matplotlib.use('Qt5Agg')  # Verwenden Sie 'Qt5Agg', falls Sie Qt installiert haben
import matplotlib.pyplot as plt

def load_ply(filename):
    plydata = PlyData.read(filename)
    x = plydata['vertex']['x']
    y = plydata['vertex']['y']
    z = plydata['vertex']['z']

    # Überprüfung Punktwolke, ob die PCD RGB-Werte besitzt
    if 'red' in plydata['vertex'].data.dtype.names:
        r = plydata['vertex']['red']
        g = plydata['vertex']['green']
        b = plydata['vertex']['blue']
        return np.column_stack((x, y, z, r, g, b))
    else:
        return np.column_stack((x, y, z))

def compare_point_clouds(cloud1, cloud2, threshold=0.001):
    xyz1 = cloud1[:, :3]
    xyz2 = cloud2[:, :3]

    nn = NearestNeighbors(n_neighbors=1, algorithm='ball_tree').fit(xyz2)
    distances, indices = nn.kneighbors(xyz1)

    matching_indices = distances.flatten() < threshold
    matching_points = np.sum(matching_indices)
    percentage_matching = (matching_points / len(xyz1)) * 100
```

```
    diff_cloud = np.copy(cloud1)

    # Wenn die Farbfelder existieren, werden die nicht
übereinstimmenden Punkte rot gefärbt
    if cloud1.shape[1] > 3:
        diff_cloud[~matching_indices, 3:] = [255, 0, 0]

    return diff_cloud, matching_points, percentage_matching

def save_ply(filename, points):
    # Überprüfen, ob die Punkte Farbfelder haben
    if points.shape[1] > 3:
        vertex = np.array([tuple(p) for p in points],
                dtype=[('x', 'f4'), ('y', 'f4'), ('z', 'f4'),
                    ('red', 'u1'), ('green', 'u1'), ('blue','u1')])
    else:
        vertex = np.array([tuple(p) for p in points],
                          dtype=[('x', 'f4'), ('y', 'f4'), ('z', 'f4')])
    el = PlyElement.describe(vertex, 'vertex')
    PlyData([el]).write(filename)

def visualize_point_clouds(cloud1, cloud2, diff_cloud):
    plt.rcParams['figure.figsize'] = [10, 10]  # Größe der Gesamtfigur
    fig = plt.figure()

    # Punktwolke Ideal
    ax1 = fig.add_subplot(131, projection='3d')
    ax1.scatter(cloud1[:, 0], cloud1[:, 1],
    cloud1[:, 2], c=(cloud1[:, 3:]/255 if cloud1.shape[1] > 3 else 'b'), s=2)
    ax1.set_title('PCD_ideal', fontsize=16)

    # Punktwolke Error
    ax2 = fig.add_subplot(132, projection='3d')
    ax2.scatter(cloud2[:, 0], cloud2[:, 1],
    cloud2[:, 2], c=(cloud2[:, 3:]/255 if cloud2.shape[1] > 3 else 'b'), s=2)
    ax2.set_title('PCD_error', fontsize=16)
```

```
    # Differenz-Punktwolke
    ax3 = fig.add_subplot(133, projection='3d')
    ax3.scatter(diff_cloud[:, 0], diff_cloud[:, 1], diff_cloud[:, 2],
c=(diff_cloud[:, 3:]/255 if diff_cloud.shape[1] > 3 else 'b'), s=2)
    ax3.set_title('PCD_Difference', fontsize=16)

    for ax in [ax1, ax2, ax3]:
        ax.set_xlabel('X', fontsize=12)
        ax.set_ylabel('Y', fontsize=12)
        ax.set_zlabel('Z', fontsize=12)
        ax.tick_params(labelsize=10)

    plt.tight_layout()
    #plt.savefig('70.png', dpi=300, bbox_inches='tight')
    plt.show()

def main():
    cloud_ideal = load_ply("Ideal_PCD1.ply")
    cloud_error = load_ply("Error_PCD2.ply")

diff_cloud, matching_points, percentage_matching = compare_point_clouds(cloud_ideal,

    #save_ply("difference_cloud.ply", diff_cloud)

    print(f"Anzahl der Punkte in Ideal_PCD: {len(cloud_ideal)}")
    print(f"Anzahl der Punkte in Error_PCD: {len(cloud_error)}")
    print(f"Anzahl der übereinstimmenden Punkte: {matching_points}")
    print(f"Prozentsatz der übereinstimmenden Punkte: {percentage_matching:.2f}%")

    if len(cloud_ideal) == len(cloud_error):
        print(f"Die Punktwolken haben die gleiche Anzahl von Punkten.")
        print(f"Anzahl der nicht übereinstimmenden Punkte:
{len(cloud_ideal) - matching_points}")
    else:
        print(f"Die Punktwolken haben unterschiedliche Anzahlen von Punkten.")
        print(f"Differenz in der Punktanzahl: {abs(len(cloud_ideal) -
len(cloud_error))}")

    visualize_point_clouds(cloud_ideal, cloud_error, diff_cloud)

if __name__ == "__main__":
```

A.3 Punktwolkenvergleich - Open3D - Quellcode

```
import numpy as np
import open3d as o3d
from sklearn.neighbors import NearestNeighbors

def compare_point_clouds(cloud1, cloud2, threshold=0.01):
    nn = NearestNeighbors(n_neighbors=1, algorithm='ball_tree').fit(cloud2)
    distances, _ = nn.kneighbors(cloud1)

    matching_indices = distances.flatten() < threshold
    matching_points = np.sum(matching_indices)
    percentage_matching = (matching_points / len(cloud1)) * 100

    return matching_indices, matching_points, percentage_matching

def visualize_point_clouds(pcd_ideal, pcd_error, matching_indices):
    # Erstelle die Differenz-Punktwolke
    pcd_diff = o3d.geometry.PointCloud()
    pcd_diff.points = pcd_ideal.points
    diff_colors = np.array(pcd_ideal.colors)
    diff_colors[~matching_indices] = [1, 0, 0]  # Rot für nicht
übereinstimmende Punkte
    pcd_diff.colors = o3d.utility.Vector3dVector(diff_colors)

    # Verschiebe die Punktwolken für nebeneinander liegende Darstellung
    pcd_ideal.translate([-2.5, 0, 0])
    pcd_error.translate([0, 0, 0])
    pcd_diff.translate([2.5, 0, 0])

    # Erstelle ein Koordinatensystem für jede Punktwolke
    coord_ideal = o3d.geometry.TriangleMesh.
    create_coordinate_frame(size=0.5).translate([-2.5, -1.5, -1.5])
    coord_error = o3d.geometry.TriangleMesh.
    create_coordinate_frame(size=0.5).translate([0, -1.5, -1.5])
    coord_diff = o3d.geometry.TriangleMesh.
    create_coordinate_frame(size=0.5).translate([2.5, -1.5, -1.5])
```

```
    # Visualisiere alle Punktwolken zusammen
    o3d.visualization.draw_geometries([pcd_ideal, pcd_error, pcd_diff,
coord_ideal, coord_error, coord_diff],
                                      zoom=0.5,
                                      front=[0, 0, -1],
                                      lookat=[0, 0, 0],
                                      up=[0, 1, 0],
                                      window_name="Point Cloud Comparison")

def main():
    # Speicherung der Punktwolken
    pcd_ideal = o3d.io.read_point_cloud("Ideal_PCD1.ply")
    pcd_error = o3d.io.read_point_cloud("Error_PCD2.ply")

    ideal_points = np.asarray(pcd_ideal.points)
    error_points = np.asarray(pcd_error.points)

    # Vergleiche die Punktwolken
    matching_indices, matching_points, percentage_matching =
compare_point_clouds(ideal_points, error_points)

    print(f"Anzahl der Punkte in Punktwolke_ideal: {len(ideal_points)}")
    print(f"Anzahl der Punkte in Punktwolke_error2: {len(error_points)}")
    print(f"Anzahl der übereinstimmenden Punkte: {matching_points}")
    print(f"Prozentsatz der übereinstimmenden Punkte: {percentage_matching:.2f}%")

    # Visualisiere die Punktwolken
    visualize_point_clouds(pcd_ideal, pcd_error, matching_indices)

if __name__ == "__main__":
    main()
```