

Entwicklung

3D Room Surveillance

ausgearbeitet von

David Alexander Kring, Feyza Keles, Lukas Martin Munz
Merle Struckmann, Merve Eser & Moritz Langer

vorgelegt an der

TECHNISCHEN HOCHSCHULE KÖLN
CAMPUS GUMMERSBACH
FAKULTÄT FÜR INFORMATIK UND
INGENIEURWISSENSCHAFTEN

im Studiengang

MEDIENINFORMATIK (M. Sc.)

Project Owner: Prof. Dr. Horst Stenzel
Technische Hochschule Köln

Betreuung: Jannis Malte Möller
Technische Hochschule Köln

Gummersbach, im September 2021

Inhaltsverzeichnis

1. Einleitung	3
2. Entscheidungsprozess	4
2.1. Gruppenaufteilung	4
2.2. Alternativen	4
2.2.1. Hardware	4
2.2.2. Software	5
2.3. Architektur	5
3. Kamera	6
3.1. Übertragung	6
3.2. Aufnahmen & Speicherung	7
3.3. Point Cloud Erstellung & Speicherung	7
4. Point Cloud Verarbeitung	8
4.1. Surface Matching	8
4.1.1. 3D Object Recognition mithilfe der Point Cloud Library	8
4.1.2. Surface Matching durch OpenCV	9
4.1.3. Surface Reconstruction	10
4.2. Point Cloud Matching	12
4.3. Point Cloud Verbesserung	14
4.3.1. PCL Installation unter Mac-Betriebssystem	14
4.3.2. OpenCV Installation unter Mac-Betriebssystem	15
4.3.3. OpenCV Implementierung	16
5. Interface	18
5.1. Programme	18
5.2. Navigation	18
5.3. Video Player	19
5.4. Login Bereich	19
5.5. Kamera Integration	20
6. Fazit	22
Abbildungsverzeichnis	23
Tabellenverzeichnis	24
Literaturverzeichnis	27
A. Arbeitsmatrix	28

1. Einleitung

Das Projekt *3D Room Surveillance* entsteht im Rahmen des Masterstudiengangs Medieninformatik M.Sc. an der Technischen Hochschule Köln. Dieses Dokument beinhaltet die Dokumentation der Entwicklungsphase, dass das zweite Modul innerhalb des Projektes darstellt. Ziel ist es, im Verlauf des Semesters ein Produkt basierend auf dem Konzept des vorherigen Semesters zu entwickeln.

Im Verlauf des Semesters hat sich das Team hierzu mit den bestehenden Technologien zur 3D-Raumüberwachung weiter auseinandergesetzt und das Konzept des vorangegangenen Moduls umgesetzt. Hierbei ging es um die Implementation eines Regalüberwachungssystems, dass 3D-Technologie nutzt, um Veränderungen im Kamerabereich festzustellen und dem Benutzer zu übermitteln. Dabei wird der Zeitabschnitt der Veränderung festgehalten, sodass diese auch zu einem späteren Zeitpunkt noch einsehbar sind.

Die Dokumentation beschäftigt sich nach einer kurzen Einleitung im Kapitel 1 mit dem Entscheidungsprozess in Kapitel 2. Dies zeigt auf welche Entscheidung zu Beginn des Projektes getroffen wurden und welche Hintergründe diese Entscheidungen hatten. Anschließend wird die Implementation in Kapitel 3 – 5 beschrieben. Zunächst wird in Kapitel 3 auf die Integration der Kamera mit der Azure Kinect SDK und die benötigten Funktionen für das Projekt eingegangen. Daraufhin fokussiert sich Kapitel 4 auf die eigentliche Point Cloud Verarbeitung und die Feststellung von Veränderungen im Kamerabereich. Für die Kommunikation mit dem Benutzer wurde ein Interface implementiert, auf das in Kapitel 5 eingegangen wird. In einem abschließenden Fazit, Kapitel 6, wird der bisherige Stand des Produktes noch einmal reflektiert und eingestuft.

2. Entscheidungsprozess

In diesem Kapitel wird vorab der Entscheidungsprozess aller, für das Projekt notwendigen, Entscheidungen dargelegt. Damit soll das Vorgehen während der Projektdurchführung nachvollziehbar aufgezeigt werden.

2.1. Gruppenaufteilung

Damit das arbeiten an dem Projekt effizient umgesetzt werden kann, unterteilte sich die Gruppe zu Anfang in drei Bereiche à zwei Mitglieder.

Tabelle 2.1.: Gruppenaufteilung

Kamerabild übertragen	David Alexander Kring, Merle Struckmann
Verbesserung des Bildes/ der Point Cloud	Feyza Keles, Merve Eser
Objekterkennung/ Tracking	Lukas Martin Munz, Moritz Langer

Die Gruppen in welchen die ersten Arbeiten aufgeteilt wurden, veränderten sich nicht im Verlauf des Projektes, jedoch wurden die Aufgabenbereiche angepasst. So haben sich Merve und Feyza nachdem die ursprüngliche Aufgabe nicht umsetzbar war, um das Interface gekümmert. Alexander und Merle erweiterten die Aufgabe auch auf das Abspeichern des Farbbildes und der Point Clouds. Zweiteres wurde als PLY gespeichert und benötigte im Laufe des Projektes noch sogenannte Faces welche durch das Team hinzugefügt wurden. Lukas und Moritz gingen von Tracking der Objekte auf das Surface Matching über, bevor dies dann auch durch das Point Cloud Matching ersetzt wurde.

2.2. Alternativen

Folgend werden die Entscheidungen und Alternativen bzgl. der Systemkomponenten Hardware und Software vorgestellt.

2.2.1. Hardware

Die Entscheidung für die Hardware fiel in der Bearbeitung der letzten Projektphase. Aufgrund der bereits vorhanden Kamera wurde diese Entscheidung übernommen. Die Alternativen waren die Intel RealSense LiDAR Camera L515 und ältere Versionen der Microsoft Kinectfamilie.

2.2.2. Software

Aus dem Projekt 1, welches drei der sechs Gruppenmitglieder im letzten Semester bearbeiteten gingen einige Entscheidungen hervor. So wurde zu Anfang die Point Cloud Library (PointCloudLibrary, b) übernommen. Diese wurde ausgewählt, da sie sich zu den Alternativen auf Point Cloud Verarbeitung spezialisiert. Nachdem die Umsetzung mit dieser Bibliothek nicht möglich war wurde die Alternative OpenCV (OpenCV) ausgewählt. Diese wurde ebenfalls letztes Semester betrachtet und bietet ein größeres Benutzungsumfeld und besseren Community Unterstützung. OpenCV würde im Endprodukt noch für die Darstellung und Speicherung der Kamera benutzt, während der Andere Teil des Programms mithilfe von Open3D (Open3D) umgesetzt wurde.

Eine andere Alternative war Cloud Compare, jedoch ist dies ein eigenes Programm, welches nicht sinnvoll in ein Back-End hinzugefügt werden kann.

Für die Entwicklung des Interfaces wurde sich auf das Programm Windows Presentation Foundation, kurz WPF, entschieden. Dies geschah, da die benutzte Kamera von Microsoft ist und das primäre Betriebssystem auf welchem das Programm entwickelt wird Windows ist. Zusätzlich gab es innerhalb der Gruppe bereits Erfahrungen mit der Benutzung des Tools.

2.3. Architektur

Aufgrund der wie oben beschriebenen Entscheidung PCL zu benutzen, welche eine Cpp Library ist und dem zusätzlichen Punkt, dass die SDK der Kamera ebenfalls in Cpp vorhanden ist, wurde sich entsprechen für diese als primäre Sprache entschieden. Die Idee hinter der Umsetzung war es, alle X-Sekunden ein Tiefenbild der Kamera zu verarbeiten und zu überprüfen, ob sich ein Objekt im Bild befindet. Das Ergebnis sollte dann mit dem des vorherigen Durchlaufs abgeglichen werden und so eine Veränderung erkannt werden. Multithreading sollte außerdem benutzt werden, um es zu ermöglichen mehrere Objekte gleichzeitig zu überprüfen. Diese Version würde einem Tracking bevorzugt, da die benötigte Rechenleistung geringer ist und die Umsetzung, gegeben der Zeit und Erfahrung in diesem Gebiet, realistischer war. Zusätzlich wurde geplant ein Interface zu erstellen, in welchem das Kamera- und Tiefenbild in Echtzeit angezeigt und Zugriff auf die gespeicherten Dateien ermöglicht wird.

Die Umsetzung erwies sich schwerer als gedacht, so wurde das Surface Matching durch Point Cloud Matching ersetzt, d.h. es werden keine Objekte mehr erkannt, sondern nur Veränderungen im ganzen Bild. Dies wird ebenfalls in individuellen Threads durchgeführt. Die Integration des Interface war nicht möglich (Abschnitt 5.5) deshalb wurde das Farbbild, das Tiefenbild und die Benachrichtigung über eine Veränderung einzeln ausgegeben.

3. Kamera

Das folgende Kapitel behandelt die Einbindung der Azure Kinect Kamera als Hardware und die einzelnen Schritte, die vorgenommen wurden, bevor die Daten weiterverarbeitet werden konnten. Die Azure Kinect wurde bereits in der vorherigen Projektphase ausgewählt. Diese Kamera ist nach der Xbox Kinect Serie von Microsoft mehr auf Entwickler ausgelegt und bietet eine umfassende SDK, um mit der Kamera zu interagieren. Zudem bietet sie eine sehr hohe 4k Auflösung und verschiedene *Field of View* Modi.

3.1. Übertragung

Der erste Schritt war, die verschiedenen Sensordaten, wie zum Beispiel die Farb- & Tiefenbilder, von der Kamera an ein Cpp - Programm zu übertragen. Dies ermöglichte die Weiterverarbeitung der Daten im Code. Hierzu stellt Microsoft bereits eine SDK bereit, die in Cpp benutzt werden kann, um die Kamera anzusprechen, zu konfigurieren und Sensordaten auszulesen. Diese Daten wurden dann zunächst mit OpenCV verarbeitet um diese als Bildstream zu rendern. Hierzu wurden zwei OpenCV Fenster angelegt (Vergleiche Listing 1), jeweils eins für Farb- bzw. Tiefenbild, in denen die Kamera-Streams gerendert wurden.

```
// Show Color
inline void kinect::show_color()
{

    if (color.empty())
    {
        return;
    }

    // Show Image
    const cv::String window_name = cv::format("color (kinect %d)", device_index);
    cv::imshow(window_name, color);
}
```

Listing 1: Beispiel Code um ein Fenster mit Farbbild in OpenCV zu generieren

Dies gab wichtiges Feedback, um die Kameraeinstellungen richtig festzulegen und zu validieren. Außerdem war diese Integration eine Voraussetzung für das Interfacing mit der Kamera und der allgemeinen Kommunikation mit der Hardware.

3.2. Aufnahmen & Speicherung

Der nächste Schritt war die Aufnahme von Kamerabildern, um diese zu speichern. Ein zusätzlicher Vorteil davon, nicht mit Echtzeitdaten zu arbeiten ist, dass es den Austausch von Testdaten, die dem realen Anwendungsfall gleichen, innerhalb der Gruppe deutlich erleichtert hat. So können alle Gruppenmitglieder Aufnahmen der Kamera verwenden ohne direkten Zugriff auf die Kamera zu benötigen.

Zur Speicherung von Videostreams bietet die AzureSDK die gut dokumentierte Methode `k4a_record_create` an, die Aufnahmen erstellt. In der Dokumentation der SDK finden sich auch Beispielimplementationen unter einer freien MIT Lizenz von Microsoft. Hierbei wird eine MKV-Videodatei (Matroska) erstellt, die mehrere Videokanäle beinhaltet. Jeder Videokanal enthält die Daten eines Sensors, also die Farb-, Tiefen- und Infrarot-Ansichten. Zusätzlich dazu gibt es einen Kanal für die IMU-Sensoren (Inertial Motion Units), also die Daten des Bewegungssensors und des Gyroskops. Die Daten der IMU sind allerdings für den vorliegenden Anwendungsfall nicht relevant und können nicht als Videostream interpretiert werden. Für die Tiefen- und Infrarot-Ansichten benutzt die SDK zusätzlich noch den speziellen Codec *b16g*, also 16bit Graustufen, Big-Endian), der von herkömmlichen Videoplayern nicht unterstützt wird. Deshalb können die meisten Videoplayer die erstellten Videodateien nicht öffnen. Die gespeicherten Daten sind dennoch akkurat und können weiterverarbeitet werden.

Die Aufnahme der Sensordaten lief weitestgehend reibungslos, allerdings muss angemerkt werden, dass das Dateiformat nicht optimal ist, da für den Anwendungsfall nicht alle Sensordaten benötigt wurden und die Datei hierdurch sehr groß werden und der Speicher nicht effizient genutzt werden kann.

3.3. Point Cloud Erstellung & Speicherung

Nachdem die Sensordaten gelesen und verarbeitet werden konnten, sollten diese nun in ein Point Cloud Format übertragen werden. Hierbei wurde sich für das PLY-Datenformat entschieden. Auch hierzu gab es bereits existierenden Beispielcode, der eine Point Cloud aus den Kameradaten erstellt und entsprechend speichert. Da die Daten der Kamera allerdings lediglich aus den Entfernungspunkten des Tiefensensors bestehen, hatten die Point Clouds keinen Mesh der Oberfläche, wie in den bisherigen Trainingsdaten üblich. Hierzu wurde später ein *Surface Reconstruction*-Algorithmus, der die Oberfläche der Objekte in der Point Cloud errechnet, implementiert (Vgl. Kapitel 4.1.3).

4. Point Cloud Verarbeitung

Im folgenden Kapitel wird genauer auf die Verarbeitung der Point Cloud eingegangen. Dabei werden die recherchierten, unterschiedlichen Alternativen bzw. Vorgehensweisen vorgestellt, wie eine Veränderung in einer Szene festgestellt werden kann. Am Ende dieses Kapitels wird auch auf die Point Cloud Verbesserung eingegangen.

4.1. Surface Matching

Die erste angedachte Vorgehensweise stellt das sogenannte *Surface Matching* dar, welches ebenfalls zwei alternative Umsetzungsmöglichkeiten lieferte. Die erste Möglichkeit war 3D Object Recognition mithilfe der Point Cloud Library. Aufgrund von Problemen, die bei der ersten Umsetzung auftreten wurde das Surface Matching mithilfe von OpenCV durchgeführt.

4.1.1. 3D Object Recognition mithilfe der Point Cloud Library

In dem Vorangegangenen Projekt 1 wurde sich für die Benutzung der Point Cloud Library (PointCloudLibrary, b) entschieden. Mithilfe dieser Library ist zu Anfang geplant gewesen einen technischen Prototyp zu erstellen. Dieser sollte in der Lage sein mithilfe der bereits vorhandenen 3D Object Recognition Funktion (PointCloudLibrary, a), Objekte in der Szene zu erkennen und Veränderungen auszugeben. Nach dem Download der PCL, mittels des Package-Managers vcpkg, wurden die ersten Setupbeispiele der PCL Dokumentation versucht auszuführen. Diese bereiteten bereits Probleme, da der Befehl “make” nicht richtig funktionierte und die benötigten Libraries nicht automatisch zu den Umgebungsvariablen des Path hinzugefügt wurden. Diese Probleme wurden jedoch verhältnismäßig schnell gelöst.

Die Dokumentation liefert des weiteren Code zur Objekterkennung mithilfe von Punktwolken mit. Dieser greift auf Header aus dem “visualization” Ordner zu, jedoch wurde dieser Ordner nicht mit Heruntergeladen, obwohl er im Github repository vorhanden ist. Deshalb wurde er manuell hinzugefügt, jedoch fehlte nun die vtk library, welche in den oben genannten Headern benutzt wurde. Diese Library musste ebenfalls manuell heruntergeladen, gebuildet und zu den Umgebungsvariablen hinzugefügt werden. Des weiteren musste die Cmake file angepasst werden, um die Position der vtk zu kennen, dies dauerte einigen an Zeit, da kein Gruppenmitglied Erfahrungen im Bereich Cmake mitbrachte. Trotz der Erfolgreichen Installation von vtk gab es Fehlermeldungen in der Header File, da einige Versionsabfragen nicht richtig auskommentiert waren.

Diese Probleme wurden jedoch mit der “AllInOne” Installation von PCL behoben, hier wurde lediglich OpenNI2 in einen “falschen” Ordner installiert, sodass er nicht richtig im Path angegeben war. Anschließend konnte das Programm ausgeführt werden. “3D

Object Recognition based on Correspondence Grouping”(PointCloudLibrary, a) funktionierte einwandfrei und das Beispiel Objekt wurde richtig erkannt. Die “Hypothesis Verification for 3D Object Recognition“(PointCloudLibrary, e) ist eine Verbesserung, die falsche Positive besser erkennen und eliminieren soll. Jedoch wurde hier das mitgelieferten Beispiele, fälschlicherweise als falsches Positiv erkannt.

4.1.2. Surface Matching durch OpenCV

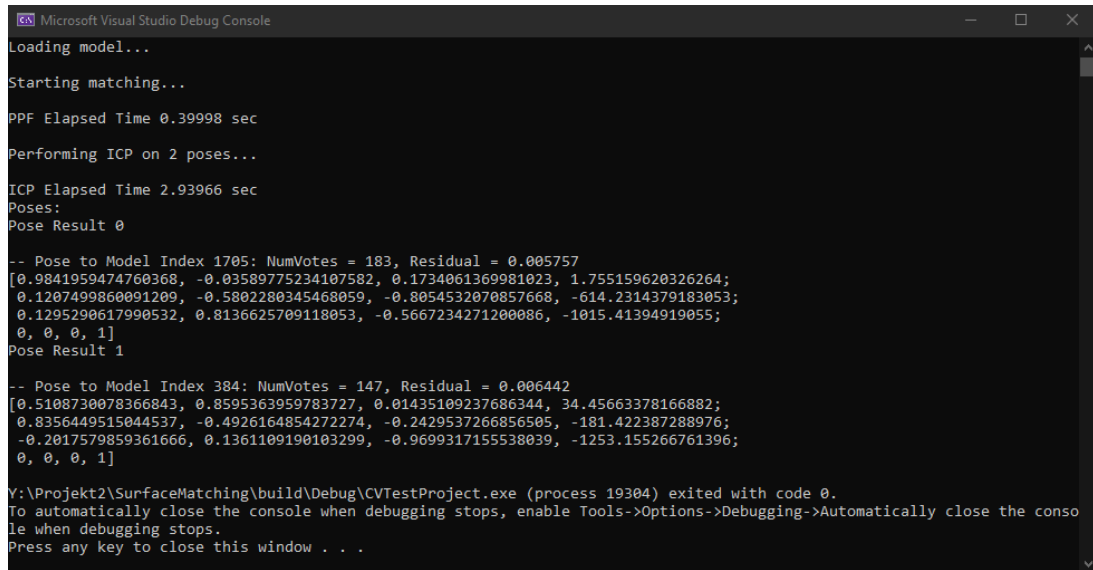
Die OpenCV (OpenCV) Library war bereits aus dem vorangegangenen Semester bekannt, da sie als Alternative für den Prototypen gehandelt wurde. Auch in dieser Library wurde eine Form von Surface Matching (PointCloudLibrary, a) für die Umsetzung genutzt. Genauer gesagt wurde hier ein Modul verwendet, welches teil der “opencv_contrib” ist, d.h. das es sollte im Umfeld der Library funktionieren ist aber noch nicht in der offiziell moderierten Version von OpenCV und erfordert einige erweiterte Installationsschritte. Das Github repository der Contib muss in den Ordner namens “modules” von OpenCV erstellt werden, bevor die gesamte Library neu gebaut werden muss.

Nach der Installation wurden das nächste Problemfeld festgestellt, Cmake. Da die Gruppe wenig bis keine Erfahrungen mit Cmake oder der benötigten CmakeList.txt hatte, ging bei den Versuchen die Richtigen Befehle anzuwenden einiges an Zeit verloren.

Das Surface Matching funktioniert, indem man eine Szene lädt und in dieser nach einem Modell sucht. Hierzu benötigt man diese 2 Dateien als .PLY und die Pfade zu jenen Dateien werden in das Programm geladen. Der in diesem Absatz erwähnte Code ist im Anhang zu finden. In den Folgenden Zeilen, wird zuerst das Modell geladen, bevor ein sog. Detector trainiert wird, das eingeladene Objekt zu erkennen. Dieser Prozess dauert (auf der von uns genutzten Hardware und mit den Beispiel Modellen) zwischen 40 und 50 Sekunden. Anschließend kann die Szene eingeladen und mithilfe des trainierten Detectors ein Matching ausgeführt werden.

Aus dem Matching werden nun nur die ersten N Resultate genommen (hier $N = 2$) und mithilfe des ICP (Iterative Closest Point) Algorithmus werden die Ergebnisse verbessert. Dies ist notwendig, da bislang nur die grobe Position des Objektes in der Szene bestimmt wurde und der Detector sehr anfällig für Noise ist. Laut Aussagen des Entwicklers hat das Ergebnis eine 10 Grad Fehlerschwelle. Dies ist darauf zurückzuführen, dass die Szene kein Vollständiges Modell beinhaltet, da Randpunkte und die Rückseite auf dem Kamerabild nicht vorhanden sind. Mit ICP wird versucht die Punkte des Modells mit den nächstgelegenen Punkten der Szene zu verbinden, je größer der Abstand, desto größer der Fehlerwert (opencv, 2014). Das aus dem Programm resultierende Ergebnis, zu sehen in Abbildung 4.5, liefert einmal eine “Pose to Model Index” anhand welchem erkannt werden kann, ob sich das gesuchte Objekt in der Szene befindet. Des Weiteren wird der “Residual” Wert mitgegeben, der als eine Art Fehlerüberprüfung fungiert, da er bei einem nicht vorhandenem Objekt sehr große Werte annimmt.

4. Point Cloud Verarbeitung



```
Microsoft Visual Studio Debug Console
Loading model...
Starting matching...
PPF Elapsed Time 0.39998 sec
Performing ICP on 2 poses...
ICP Elapsed Time 2.93966 sec
Poses:
Pose Result 0
-- Pose to Model Index 1705: NumVotes = 183, Residual = 0.005757
[0.9841959474760368, -0.03589775234107582, 0.1734061369981023, 1.755159620326264;
0.1207499860091209, -0.5802280345468059, -0.8054532070857668, -614.2314379183053;
0.1295290617990532, 0.8136625709118053, -0.5667234271200086, -1015.41394919055;
0, 0, 0, 1]
Pose Result 1
-- Pose to Model Index 384: NumVotes = 147, Residual = 0.006442
[0.5108730078366843, 0.8595363959783727, 0.01435109237686344, 34.45663378166882;
0.8356449515044537, -0.4926164854272274, -0.2429537266856505, -181.422387288976;
-0.2017579859361666, 0.1361109190103299, -0.9699317155538039, -1253.155266761396;
0, 0, 0, 1]
Y:\Projekt2\SurfaceMatching\build\Debug\CVTestProject.exe (process 19304) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Abbildung 4.1.: Ergebnis eines erfolgreichen Surface Matchings

Für die Verwendung in unserem Prototypen wurden die einzelnen Schritte des Surface Matchings in eigene Funktionen aufgeteilt. Damit muss das Trainieren des Detectors nur einmal pro Modell ausgeführt werden, während mehrere Szenen mit diesem Detector überprüft werden können. Folglich wurden mithilfe von Multithreading mehrere Durchläufe gleichzeitig durchgeführt, um mehrere Objekte erkennen zu können. Jedoch kam es hier zu einem Problem; da für mehrere Objekte nun nicht mehr das Beispiel verwendet werden sollte, wurden eigene .PLY-Dateien mithilfe der Azure Kinect aufgenommen. Diese gaben zu Anfang jedoch nur fehlerhafte Ergebnisse aus, da direkt aufgenommene Dateien, keine sogenannten Faces besaßen. Dieses Problem wurde wie in Unterabschnitt 4.1.3 beschrieben behoben.

Mit nur einem Objekt in der Szene konnte dieses erfolgreich erkannt werden, jedoch gab es nur noch falsche Positive, sobald mehrere Objekt in der Szene zu sehen waren. Nach einigem Testen und anpassen der Werte des Detectors und des ICP Algorithmus wurde festgestellt, dass durch die Aufnahmen und Konvertierung der Point Clouds, die Auflösung zu gering war, um ein erfolgreiches Surface Matching erstellen zu können. In einem der Meetings wurde vorgeschlagen das Matching mithilfe von ICP direkt auf den Point Clouds durchzuführen. Darauf basierend wurde diese Idee recherchiert.

4.1.3. Surface Reconstruction

Das Surface Matching wurde anfänglich nur mit Beispiel Point Clouds durchgeführt. Um die eigens erstellten Point Clouds für das Surface Matching nutzen zu können, mussten zu den erstellten Vertices Faces hinzugefügt werden. Die Idee war es eine Funktion zu erstellen, welche die Faces hinzufügt und mit diesen Point Clouds das Surface Matching aufruft. Zunächst wurden die Faces manuell in Meshlab (Cignoni u. a., 2008) in den Point Clouds hinzugefügt, so konnte der benötigte Funktionsablauf analysiert werden. Dieser sollte im späteren Verlauf im Code umgesetzt werden. Des

4. Point Cloud Verarbeitung

Weiteren konnten bereits eigene Point Clouds im Surface Matching getestet werden. Die Erstellung der Faces wurde nach der Anleitung von Shubham Wagh (Wagh) durchgeführt. Zuerst wurden die Normals der Vertices berechnet. Anschließend wurde die Poisson Surface Reconstruction durchgeführt, bei dem es sich um einen räumlich adaptiven Multiskalen-Algorithmus handelt (Kazhdan u. a., 2006). Da die Faces zumeist ungenaue Schätzungen des Algorithmus sind, muss das Mesh noch aufgeräumt werden. So werden Faces mit einer bestimmten Länge gelöscht, da diese meistens aus Vertices am Kamerarand entstanden sind. Zusätzlich werden isolierte Vertices gelöscht, sodass das Endprodukt eine durchgängige Fläche aufweist. Getestet wurde die Methode ebenfalls wenn vorab uninteressante Bereiche aus der Point Cloud gelöscht wurden, wie zum Beispiel die äußere Wand oder andere Regalbereiche.

Die Algorithmen zur Surface Reconstruction wurden mithilfe von Open3D (Zhou u. a., 2018) implementiert. Open3D ist eine Bibliothek zur 3D Datenverarbeitung in den Sprachen Python und Cpp. Open3D bietet bereits den Poisson Algorithmus an, der bei der manuellen Erstellung der Faces verwendet wurde. Außerdem bietet Open3D den Ball Pivoting Algorithmus an, der die Oberfläche eines Objektes mithilfe eines kleinen Balls rekonstruiert. Dem Ball wird ein Radius gegeben mit den er zwischen den Vertices rotiert sobald er drei Vertice berührt kann ein weiteres Triangle gebildet werden (Bernardini u. a., 1999). Für die Implementierung des Surface Reconstruction benötigt die Funktion eine Point Cloud. Zudem müssen die Normals der Point Cloud berechnet und in die richtige Richtung orientiert werden, dabei sollten die Normals in Richtung der Kamera zeigen. Anschließend können die Funktion des Poisson Algorithmus und des Ball Pivoting aufgerufen werden (siehe Listing 2). Der Algorithmus des Ball Pivoting muss zur Erstellung des Surface die Point Cloud und den Radius übergeben werden. Der Poisson Algorithmus benötigt zur Erstellung des Surface die Point Cloud, sowie Tiefen-, Weiten- und Skalierungswerte und ein Boolean zur Bestimmung der linearen oder nicht linearen Rekonstruktion.

```
geometry::PointCloud pcd;

io::ReadPointCloud(pcdFile, pcd);

const Eigen::Vector3d orientationPoint = Eigen::Vector3d(0.0, 0.0, -1.0);

// Generierung der Normals + zuordnen eines Orientierungspunkt
pcd.EstimateNormals();
pcd.NormalizeNormals();
pcd.OrientNormalsToAlignWithDirection(orientationPoint);

// Poisson Algorithmus
auto poisson = geometry::TriangleMesh::CreateFromPointCloudPoisson(pcd, 8, 0, 1.0f, false);

// Ball Pivoting
std::vector<double> radii = { 0.5, 1, 2, 4 };
auto ballPivoting = geometry::TriangleMesh::CreateFromPointCloudBallPivoting(pcd, radii);

io::WriteTriangleMesh(safeFile, *ballPivoting, true, true);
```

Listing 2: Poisson Algorithmus & Ball Pivoting

4. Point Cloud Verarbeitung

Da durch das Testing mit den bearbeiteten Point Clouds falsche Ergebnisse zustanden kamen, wurde probiert die erstellten Point Clouds mit dem Surface aufzuräumen, ähnlich wie es bereits bei der manuellen Erstellung erfolgte. Es erfolgte der Versuch unreferenzierte Vertices und degenerierte Triangles, sowie Faces ab einer bestimmten Länge zu löschen, sodass nur der Bereich von Interesse abgebildet wurde. Dies sollte mit der "RemoveTrianglesByMask"-Methode in Open3D berechnet werden. Mit dieser Methode sollten, ähnlich wie bereits bei der manuellen Erstellung in Meshlab, deutlich längere Faces aus dem Mesh gelöscht werden, weil diese als nicht relevant für die spätere Verarbeitung angesehen wurden. Die Funktion benötigt zur Löschung der Triangles eine vorher bestimmte Triangles Mask, die alle Triangles beinhaltet die gelöscht werden sollten. Abschließend sind Fehler bei der Berechnung der Triangle Mask aufgetreten, sodass entweder keine oder sogar alle Triangles gelöscht wurden. Das Problem wurde nicht behoben, da im Rahmen des Teams entschieden wurde eine alternative zum Surface Matching zu suchen.

Letztendlich konnte jedoch der Poisson Algorithmus gegenüber dem Ball Pivoting überzeugen. Zum einen war die konstruierte Oberfläche detaillierter und ohne jegliche Lücken zwischen den Faces, zum anderen war die Laufzeit des Algorithmus um einiges schneller. So benötigte das Ball Pivoting für eine annehmbare Rekonstruktion bereits mindestens 2 Minuten, diese beinhaltet jedoch Lücken zwischen den Faces. Der Poisson Algorithmus benötigte für die Rekonstruktion nur etwa 30 Sekunden.

Das Surface Reconstruction wurde mit dem Surface Matching letztendlich verworfen, da mit den eigens erstellten Point Clouds keine richtigen Ergebnisse, zumeist falsch Positive, aus der Analyse entstanden.

4.2. Point Cloud Matching

Aufgrund der zuvor genannten Problematik mit dem Surface Matching wurden Recherchen für eine alternative Vergleichsmethode durchgeführt. Die Recherche wurde dabei auch auf den direkten Vergleich zwischen zwei Punktwolken (zwei vollständigen Szene) ausgeweitet, da es nach der Recherche keine weitere sinnvolle Umsetzung gab. Bei dieser Methode können einzelne Objekte nicht direkt getrackt werden, aber Veränderungen in der gesamten Szene bzw. im kompletten Kamerabereich festgestellt und dem Nutzer mitgeteilt werden. Ein solcher Direktvergleich bietet zudem den Vorteil, dass keine zusätzlichen Funktionen zur Erstellung einer Oberfläche oder dem Bereinigen der Point Clouds benötigt werden. Dadurch ist der Vergleich weniger Zeitintensiv.

Die Recherche führte zur „ComputePointCloudDistance“-Methode (Open3D Dokumentation), welche von Open3D bereitgestellt wird. Diese Methode berechnet die Distanz zwischen zwei Punktwolken (Quellpunktwolke und Zielpunktwolke). Dabei wird für jeden Punkt in der Quellpunktwolke die Entfernung zum nächstgelegenen Punkt der Zielpunktwolke berechnet. Als Ergebnis liefert diese Methode den Verschiebungsvektor, mithilfe diesen dann festgestellt werden kann, ob eine Veränderung zwischen den beiden betrachteten Point Clouds vorliegt oder nicht.

Zuerst wurde mehrere Vergleiche mit multithreading ausgeführt, jedoch könnte der Code wegen verschiedenen Compiler requirements nicht direkt zu den anderen Codeteilen hinzugefügt werden. Aus diesem Grund wurde eine ausführbare Datei (executable)

4. Point Cloud Verarbeitung

erstellt, welche in einem eigenen Thread aufgerufen wird. Vor der Erstellung der executable wurde der Code auf nur das Nötigste gekürzt.

```
using namespace std;
using namespace cv;
using namespace open3d;

int matching(string file1, string file2)
{
    {
        open3d::geometry::PointCloud pcd1;
        open3d::geometry::PointCloud pcd2;

        open3d::io::ReadPointCloud(file1, pcd1);

        open3d::io::ReadPointCloud(file2, pcd2);

        auto dis_pcd1_pcd2 = pcd1.ComputePointCloudDistance(pcd2);

        if (dis_pcd1_pcd2[0] == 0 && dis_pcd1_pcd2[1] == 0) {

            return 0;
        }
        else if (dis_pcd1_pcd2[0] < 1 || dis_pcd1_pcd2[1] < 1) {

            return 1;
        }
        else {
            return 2;
        }
    }
}

int main(int argc, char* argv[]) {

    string file1 = argv[1];
    string file2 = argv[3];

    if (argc > 1) {
        return matching(file1, file2);
    }
    return 3;
}
```

Listing 3: Point Cloud Matching (ComputePointCloudDistance-Methode)

Das Listing 3 zeigt die Implementierung der "ComputePointCloudDistance"-Methode, bei der die Distanz zwischen zwei Point Clouds (pcd1 & pcd2) berechnet und abhängig von dieser ein Rückgabewert zurückgegeben wird. Anhand dieses Wertes werden dann die Konsolenausgabe, sowie das weitere Vorgehen des Programms bestimmt. Der executable werden zwei Pfade zu den Point Clouds mitgegeben. Der erste Pfad ist die Grundszenen (Quellpunktwolke) und der zweite Pfad ist die zu vergleichende

4. Point Cloud Verarbeitung

Szene (Zielpunktwolke). Falls eine kleine oder große Veränderung stattfindet, wird die Grundszenen auf die Aktuelle geändert und bei einer großen Veränderung erhält der Benutzer ebenfalls eine Ausgabe mit dem Namen der Datei, in welcher die Veränderung stattgefunden hat und der Zeit wann diese aufgetreten ist

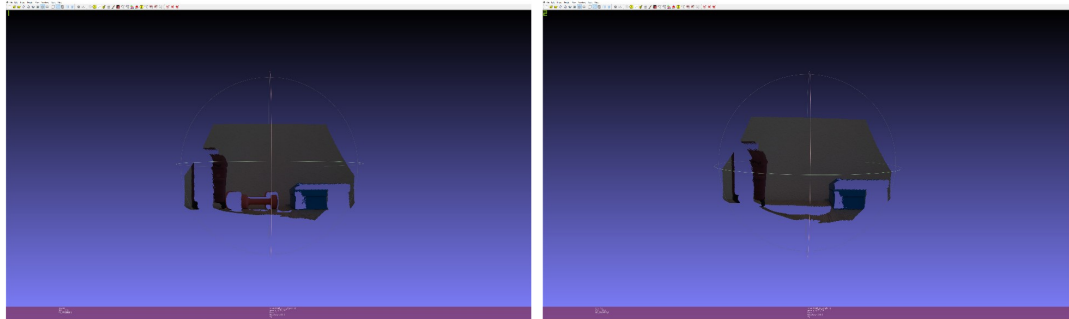


Abbildung 4.2.: Vergleich zweier Point Cloud Dateien

Abbildung 4.2 verbildlicht die Funktionsweise des Point Cloud Matchings bzw. der ComputePointCloudDistance-Methode. Dabei werden zwei Point Clouds, links (Quellpunktwolke) und rechts (Zielpunktwolke), miteinander verglichen. Die Abweichungen wird mithilfe der x- und y-Koordinaten des Verschiebungsvektors angegeben. Wie in der Abbildung 4.2 zu sehen ist, wurde ein Objekt, die Hantel, aus der Szene entnommen, sodass eine deutliche Veränderung der Koordinaten festzustellen ist.

4.3. Point Cloud Verbesserung

In diesem Bereich wurde mit dem Mac-Betriebssystem gearbeitet. Die Aufgabe bestand darin, das Bild bzw. die Point Cloud zu verbessern. Bei näherer Betrachtung sollte Ausreißer oder störende Messungen eliminiert werden (PointCloudLibrary, c). Da im Vorangegangenen Projekt 1 sich für die Benutzung der PCL entschieden wurde, wurde zu Anfang versucht PCL unter Mac-Betriebssystem zu installieren.

4.3.1. PCL Installation unter Mac-Betriebssystem

Folgendes Tutorial wurde befolgt, um die Point Cloud Library auf Computer mit dem Mac-Betriebssystem zu Installieren (PointCloudLibrary, d). Hierbei wurde versucht mithilfe der MacPorts die Point Cloud Library zu installieren bzw. zu kompilieren. Die Macports sind ein Paketverwaltungssystem, mit dem sich leicht Open-Source-Pakete installieren, updaten und auch wieder deinstallieren lassen. Um jede Komponente der PCL-Bibliothek zu kompilieren, mussten eine Reihe von Bibliotheksabhängigkeiten von Drittanbietern heruntergeladen und kompiliert werden.

Hierfür gibt es einige Bibliotheken die erforderlich sind, um PCL zu erstellen. Zuerst wird CMake-Version, eine Plattformübergreifendes Open-Source-Build-System, installiert. Dies konnte unter Mac einwandfrei installiert werden. Daraufhin wurde dann der Boost-Version installiert, der ebenfalls fehlerfrei eingerichtet wurde. Boost wird für

4. Point Cloud Verarbeitung

freigegebene Zeiger und Threading verwendet. Danach wurde der Eigen-Version, die als Matrix-Backend für SSE-optimierte Mathematik verwendet wird, installiert. Um die Installation erfolgreich abzuschließen, musste schließlich das Visualization Toolkit (VTK) installiert werden. Dies führte jedoch zu mehreren Problemen bei der Installation (s. Abbildung 4.3), die die Aufgabe erschwerte erfolgreich abzuschließen. Am

```
[ 59%] Building CXX object Rendering/OpenGL2/CMakeFiles/RenderingOpenGL2.dir/vtkCompositePolyDataMapper2.cxx.o
In file included from /Users/FeyzaKeles/Desktop/VTK-9.0.1/Rendering/OpenGL2/vtkCompositePolyDataMapper2.cxx:56:
In file included from /Users/FeyzaKeles/Desktop/VTK-9.0.1/Rendering/OpenGL2/vtkOpenGL.h:30:10: fatal error: 'GL/gl.h' file not found
#include <GL/gl.h> // Include OpenGL API.
^
1 error generated.
make[2]: *** [Rendering/OpenGL2/CMakeFiles/RenderingOpenGL2.dir/vtkCompositePolyDataMapper2.cxx.o] Error 1
make[1]: *** [Rendering/OpenGL2/CMakeFiles/RenderingOpenGL2.dir/all] Error 2
make: *** [all] Error 2
FeyzaKeles@Feyzas-MBP build %
```

Abbildung 4.3.: VTK Fehlermeldung unter Mac-Betriebssystem

Anfang lief die Installation gut, doch etwa bis zur Hälfte wurde die Installation abgebrochen. Beim Wiederstart störte sich das Programm an schon wegen vorhandenen Dateien. Die ursprünglich installierten Bibliotheken wurden dann deinstalliert. Hierbei wurde versucht andere Lösungen zu finden. Nach der Recherche im Internet wurde festgestellt, dass einige Veränderung, bei der Installation von VTK durchgeführt werden mussten (NIST). Die ersten Schritte wurden wie vorhin durchgeführt und erfolgreich abgeschlossen. Bei der Installation von VTK wurde der sechste Schritt vorgenommen und versucht mit diesem Schritt das Problem zu lösen. Der Versuch das VTK wieder zu installieren blieb erfolglos. Abbildung 4.4 zeigt, die Fehlermeldung die beim zweiten Versuch auftrat.

```
Scanning dependencies of target glew
[ 57%] Building C object ThirdParty/glew/vtkglew/CMakeFiles/glew.dir/src/glew.o
make[2]: *** No rule to make target '/opt/local/lib/libGL.dylib', needed by 'lib/libvtkglew-9.0.0-0.1.dylib'. Stop.
make[1]: *** [ThirdParty/glew/vtkglew/CMakeFiles/glew.dir/all] Error 2
make: *** [all] Error 2
FeyzaKeles@Feyzas-MBP build %
```

Abbildung 4.4.: Zweite VTK Fehlermeldung unter Mac-Betriebssystem

Leider zeigte es beim zweiten Versuch die gleiche Fehlermeldung wie beim ersten Versuch. Da auch andere Mitglieder Probleme bei der Installation von PCL hatten, entschied sich das Team, OpenCV weiter zu verwenden.

4.3.2. OpenCV Installation unter Mac-Betriebssystem

Wie auch in der PCL wurde hier MacPorts verwendet, um OpenCV zu installieren. Hierfür war ein Youtube-Video (Lupo) sehr hilfreich für die Installation. Zuerst wurde die neueste Version von OpenCV heruntergeladen. Nach dem Download wurde es an der gewünschten Stelle abgelegt. Denn wenn OpenCV einmal konfiguriert und der Speicherort später geändert wird, verliert es die gesamte Verbindung zum vorherigen Projekt. Dann wurde ein neuer Ordner namens build im OpenCV-Ordner erstellt. Dann wurde OpenCV kompiliert, damit es in XCode verwendet werden kann. XCode ist eine integrierte Entwicklungsumgebung von Apple für MacOS. Dies wurde mit der Befehl "cmake -G Unix Makefiles .." erstellt. Mit der Befehl "make" wurde die Bibliotheksdateien erstellt, die in XCode-Programmen verwendet werden kann.

4.3.3. OpenCV Implementierung

Als Nächstes wurde einen Cpp-Programmcode geschrieben, um eine Videodatei zu lesen. Mit einem so simplen Code wurde versucht herauszufinden, ob OpenCV auf dem Mac funktionierte. Zuerst wurden alle zuvor heruntergeladenen OpenCV-Bibliotheken hinzugefügt. Die wichtigste Klasse hier war VideoCapture. Mit den `open()` und `read()`-Funktionen dieser Klasse kann jede Videodatei geöffnet und gelesen werden. Zur Anzeige des Videos wurde die Funktion `imshow()` verwendet.

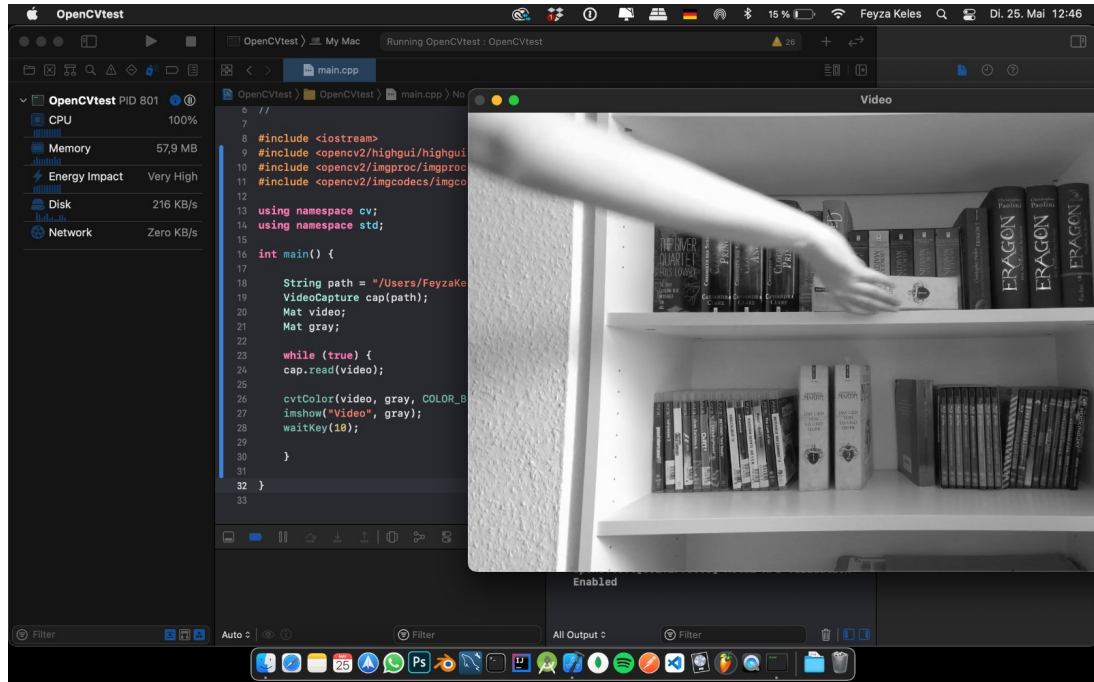


Abbildung 4.5.: OpenCV Implementierung unter Mac-Betriebssystem

Da alles einwandfrei funktionierte, konnte über den nächsten Schritt nachgedacht werden. Dabei wurde die Idee entwickelt, ein Controller (Spurleiste) zu erstellen mit dem das Bild angepasst werden kann. Hierbei wurde dies problemlos zum Cpp-Programm hinzugefügt. Dazu wurde eine einfache Code geschrieben, um zu überprüfen, ob die Spurleiste über das Video funktionierte. Bei diesen einfachen Code wurden zwei Spurleisten hinzugefügt, um dabei die Helligkeit und den Kontrast des Videos zu ändern. Die Spurleiste funktionierte beim Anpassen einwandfrei. Doch statt die Helligkeit und den Kontrast zu ändern wurde versucht mit diesem Spurleiste die eigentliche Aufgabe zu erfüllen, nämlich den Point Cloud zu verbessern. Jedoch wurde festgestellt, dass hierfür PCL-Bibliotheken benötigt werden (Ramsrigoutham).

In der Zwischenzeit haben die beiden Mitglieder von Herrn Prof. Stenzel Paper zugeschickt bekommen, um erfolgreich weiterzuarbeiten.

1. Paper – Scanning 3D Full Human Bodies Using Kinects (Tong u. a., 2012)
Bei der von Jing Tong, Jin Zhou, Ligang Liu, Zhigeng Pan und Hao Yan vorbereitete Dokumentation geht es um 3D scanning von einem menschlichen Körper, die

4. Point Cloud Verarbeitung

mit Kinects erstellt wurde. Dieses Dokument hat leider nicht das Team weitergebracht, da nur die Point Cloud Verbesserung von Kamera Auflösungen gebraucht war.

2. Paper - Single View 3D Reconstruction Based on Improved RGB-D Image (Cao u. a., 2020)

Dieses Dokument basiert auf die Einzelansicht 3D Rekonstruktion auf verbesserte RGB-Bild. Das von Mingwei Cao, Liping Zheng und Xiaoping Liu vorgelegte Dokument könnte für das Team hilfreich sein, aber da kein Code Beispiel vorgelegt war, war es für das Team auch nicht nützlich.

3. Paper - 3D Reconstruction Method of Rapeseed Plants in the Whole Growth Period Using RGB-D Camera (Teng u. a., 2021)

In diesem Dokument wurde eine Pflanze mit Kamera aufgenommen und die Farbkombinationen beachtet. Diese Verbesserung die für das Projekt gebraucht war, war nicht das was in diesem Dokument erklärt wurde. Also die von Teng Xiaowen, Zhou Guangsheng, Wu Yuxuan, Huang Chenglong, Dong Wanjing und Xu Shengyong erstellten Dokumentation war auch nicht weiterführend.

Alle drei Paper waren nicht für das Projekt 2 nützlich. Da Inhaltlich die Dokumente nicht hilfreich waren, wurde auch die Referenzen die in Papers hinterlegt wurden nicht betrachtet.

Das Team hat sehr viel Zeit für Bildbearbeitung investiert und nach einiger Zeit sich entschieden das Team bei eine andere Aufgabe zu unterstützen. Die zwei Team Mitglieder haben ihre Aufgabe mit Interface weitergeführt.

5. Interface

In diesem Kapitel wird die Interface programmierung behandelt. Darüber hinaus ist die Navigation integriert, so dass der Videoplayer und der Login Bereich erweitert wurden. Schließlich wird die Aufzeichnung nicht über die Interface, sondern über das integrierte Kameradisplay angezeigt.

5.1. Programme

Für das Interface wurde im Team entschieden, mit WPF zu arbeiten, weil in dem Projekt 1 das Interface mit WPF eingeplant war. WPF (Windows Presentation Foundation) ist eine Benutzeroberflächen-Framework und unterstützt in einem durchgängigen Bibliotheken verschiedene Arten von GUIs z.B. Videos. Mit diesem Framework können Desktopclientanwendungen wie Grafik, Layout, Anwendungsmodelle usw. erstellt werden. Um ein Modell mit Anwendungsprogrammierung bereitzustellen verwendet WPF XAML (Extensible Application Markup Language). WPF kann nur von Windows erstellt werden. Da für das Interface zuständige Teammitglieder MacBook Nutzer sind, trat das Problem auf, dass mit WPF nicht auf MacBook gearbeitet werden kann. Nach sehr viel Recherche wurde das Problem mit Virtualbox gelöst. Virtual Box (Hoffmann) ist ein Opensource-Tool, sodass weitere Betriebssysteme in einer virtuellen Umgebung auf einem MacBook/PC läuft. Die Virtualbox läuft immer auf dem aktuellen Version von Windows und besitzt zwei Speicheralternativen, 32 bit und 64 bit, die am Anfang der Installation eine Rolle spielen.

Für die XAML Erstellung in WPF wurde mit Visual Studio 2019 gearbeitet. In Visual Studio 2019 (Visual Studio 2019) können verschiedene Sprachen programmiert werden. Es ist eine Entwicklungsumgebung. Das Team hat die Sprache CSharp verwendet. Nachteile sind davon, dass auf dem MacBook die Virtualbox sehr langsam reagiert. Das hat den Teammitgliedern sehr viel Zeit gekostet, aber dennoch wurde es gelungen.

5.2. Navigation

In dem Projekt 1 wurde bereits geplant, wie im laufe des Projekt die Interface aussehen sollte (s. Abbildung B.1). Bei dem Interface wurde eine Navigationsleiste erstellt und in der Leiste befinden sich Kamera, Aufzeichnungen und Historie (s. Abbildung B.3). Diese Leiste kann durch anklicken geöffnet werden und zeigt entsprechend die Symbole und Beschriftungen der einzelnen Seiten. Ansonsten ist die Navigationsleiste geschlossen, es sind nur die Icons zu sehen. Zu dem wurden die passenden Icons mithilfe von Material Design Package angepasst. In der oberen rechten Seite befindet sich das Login und in der Mitte der Leiste ist der Firmenname zusehen. Im Allgemeinen ist die Interface in den Farben grün #1abc9c und grau #f1f1f1 gehalten.

Grundriss von Interface wurde vorbereitet. Die einzelnen Buttons sollten nach und nach bearbeitet werden. Zuerst wurde mit Kamera angefangen und für die Live Kamera Aufnahme ein Video Player überlegt. In den nächsten Kapitel wird das Video Player näher betrachtet.

5.3. Video Player

Zudem die anderen Teammitglieder den Code für das Backend weiter programmierten, hat das Interface Team ein Video Player programmiert. In dem Video Player gibt es PLAY, PAUSE, STOP und Ton Einstellung Button. Geplant war zum Schluss, Backend mit Frontend zu verbinden und die Aufnahmen mit den Buttons zusteuern (WPF). Schlussendlich konnten die beiden Seiten nicht verknüpft werden und für Aufnahmen wurde eine andere Alternative überlegt (s. Kapitel 5.5).

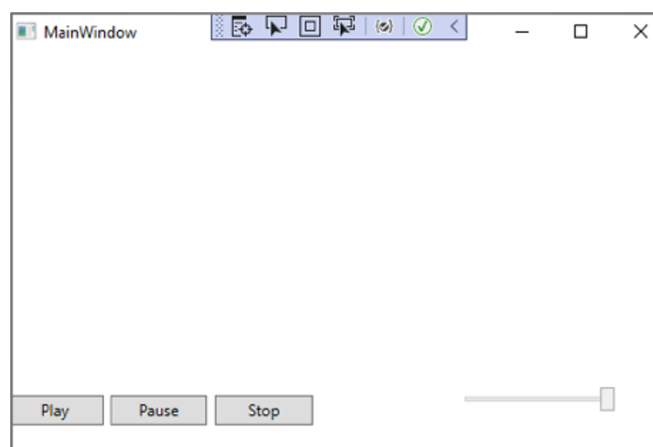


Abbildung 5.1.: Video Player

5.4. Login Bereich

Das Interface-Team hat die Login-Seite farblich passend zur Navigationsleiste erstellt (s. Abbildung 5.2). Wie üblich gab es einen Eingabebereich für Benutzername und Passwort. Es gab auch die Möglichkeit, das Passwort bei Bedarf zurückzusetzen. Es wurde versucht, Cpp-Programm mit der Datenbank zu verbinden. Dies war leider nicht erfolgreich.

5. Interface

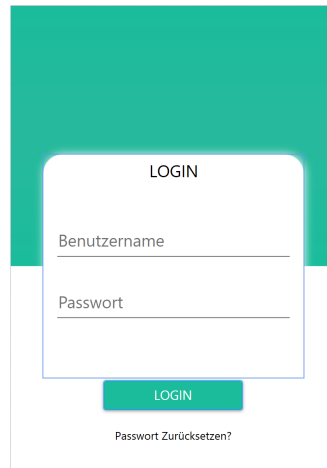


Abbildung 5.2.: Login Screen

5.5. Kamera Integration

Eine Überlegung die Kamera mit dem Interface zu integrieren war es, die direkte Implementierung innerhalb der Interface Struktur. Da die Azure Kinect SDK ebenfalls in CSharp verwendet werden kann, konnte die Implementierung ähnlich wie bereits in Kapitel 3 beschrieben eingebaut werden. Im Vergleich zu der vorherigen Kameraimplementierung lieferte die CSharp-Implementierung jedoch keine annehmbare Tiefenbilder von der Kamera. In den Tiefenaufnahmen konnte wenig Kontrast zwischen Objekten innerhalb des Kamerabereichs und der dahinter liegenden Wand erkannt werden. Daher wurde ein Algorithmus implementiert, der das Farbbild in ein Tiefenbild umwandelt. Dafür wird der Tiefenbuffer des Bildes verwendet, um die Pixel je nach Tiefenwert einzufärben (siehe Listing 4). Für das Tiefenbild wurde ein Graustufenbild des Originalbildes berechnet, dabei wird der Tiefenbuffer aus dem Tiefenbild entnommen und innerhalb der If-Abfrage abgeglichen. Je nach dem Wert des Tiefenbuffers wird das Outputbild eingefärbt. Zusätzlich zu der Kameraintegration wurde ein Button integriert, der als Toggle zwischen dem Farb- und Tiefenbild fungiert (siehe Abbildung 5.3).

```
Task<BitmapSource> createOutputColorBitmapTask = Task.Run(() =>
{
    transform.DepthImageToColorCamera(capture, transformedDepth);

    Span<ushort> depthBuffer = transformedDepth.GetPixels<ushort>().Span;
    Span<BGRA> colorBuffer = capture.Color.GetPixels<BGRA>().Span;
    Span<BGRA> outputColorBuffer = outputColorImage.GetPixels<BGRA>().Span;

    for (int i = 0; i < colorBuffer.Length; i++)
    {
        outputColorBuffer[i] = colorBuffer[i];
    }
});
```

5. Interface

```
if (depthBuffer[i] == 0)
{
    outputColorBuffer[i].R = 200;
    outputColorBuffer[i].G = 200;
    outputColorBuffer[i].B = 200;
}
else if (depthBuffer[i] > 10 && depthBuffer[i] <= 500)
{
    outputColorBuffer[i].R = 100;
    outputColorBuffer[i].G = 100;
    outputColorBuffer[i].B = 100;
}
else if (depthBuffer[i] > 500)
{
    outputColorBuffer[i].R = 10;
    outputColorBuffer[i].G = 10;
    outputColorBuffer[i].B = 10;
}
}

BitmapSource source = BitmapSource.Create(outputColorImage.WidthPixels,
    outputColorImage.HeightPixels, 96, 96, PixelFormats.Bgra32, null,
    outputColorImage.Memory.ToArray(), outputColorImage.StrideBytes);

source.Freeze();
return source;
});
```

Listing 4: Erstellung des Tiefenbildes

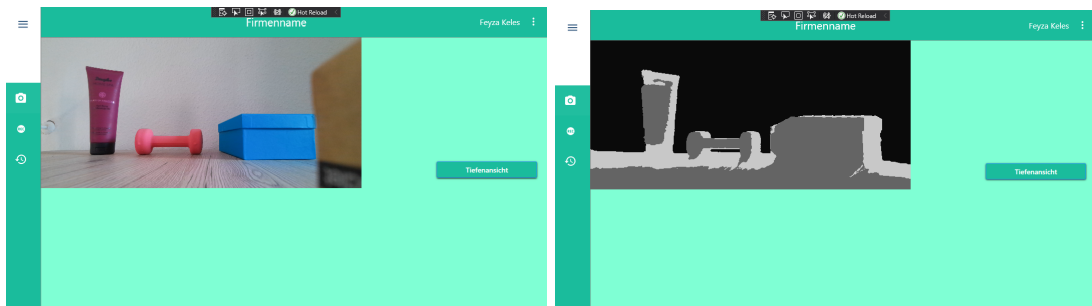


Abbildung 5.3.: Kamera Interface - Integration der Kamera in das Interface, über den Button kann zwischen Farb- und Tiefenbild gewechselt werden.

Schlussendlich wurde die Kamera Integrierung ins Interface wie hier beschrieben nicht übernommen, da die Kamera nicht in Frontend und Backend gleichzeitig agieren kann. Dies wird jedoch für das Erstellen der Point Clouds und der Aufnahmen benötigt. Gleichzeitig war die Umlegung von der genannten Backend Logik aus zeittechnischen Gründen nicht mehr möglich und es wurde nicht als sinnvoll erachtet diese in das Frontend zu verlagern. Dadurch ist das Interface für die finale Version des Projektes ungeeignet.

6. Fazit

Das Semester zur Entwurf und Implementierung einer Architektur hat ein guten Einblick gegeben, wie ein geplantes Regalüberwachungssystem, welches 3D Technologie nutzt, zustande gebracht wird. Die Veränderung im Kamerabereich wurde in Zeitabschnitten festgestellt und die gesammelten Daten an Benutzer übermittelt. Die Daten sind auch im nachhinein einsehbar.

Zuerst wurde ein Entscheidungsprozess durchgeführt, in welche die Entscheidung definiert und erläutert werden. Der nächste Schritt war die Integration der Kamera mit dem Azure Kinect SDK und den notwendigen Funktionen für das Projekt. Zeitgleich wurde die Verarbeitung von Point Clouds begonnen, unter anderem wurden hier Möglichkeiten zur Verbesserung der Point Clouds recherchiert sowie die eigentlich Methode zur Feststellung von Veränderungen implementiert. Zum Schluss wurde ein Interface mittels WPF erstellt, welches die Navigation, den Login-Screen sowie ein Ansatz zur Einbindung der Kamera enthält.

Das Endprodukt umfasst die Darstellung von Farb- und Tiefenbild der Azure Kinect. Zusätzlich werden nach festgelegten Zeitabschnitten Kameraauschnitte miteinander verglichen, um Veränderungen im Kamerabereich festzustellen. Diese Zeitabschnitte werden als Videoaufzeichnung festgehalten und nur gelöscht wenn keine Veränderung auftreten. Wenn eine Veränderung auftritt wird dies dem Benutzer über die Konsole mitgeteilt. Da bei der Zusammenführung von Frontend und Backend Probleme aufgetreten sind, wird zunächst nur das Backend, mit provisorischen Ausgaben, als fertig Lösung präsentiert. Zukünftig sollte die Integration des Frontends gearbeitet werden, sodass die Benutzerinteraktion über das Interface erfolgen kann. Dementsprechend soll die Live Übertragung der Kamera ähnlich wie bereits in Kapitel 5.5 erläutert durchgeführt werden. Zudem sollte die Mitteilung von Veränderung im gleichen Screen über eine Textbox erfolgen. Die Programmierung war durch all die äußeren Umstände, wie Installationen, oder fehlende Vorerfahrungen, wesentlich schwieriger als zuvor erwartet. Trotz dieser Probleme konnte die Kernidee aus Projekt 1, das Tracking von Veränderung mittels einer 3D Kamera, umgesetzt werden. Zudem bietet das Projekt eine gute Grundlage für die zukünftige Weiterarbeit.

Abbildungsverzeichnis

4.1. Ergebnis eines erfolgreichen Surface Matchings	10
4.2. Vergleich zweier Point Cloud Dateien	14
4.3. VTK Fehlermeldung unter Mac-Betriebssystem	15
4.4. Zweite VTK Fehlermeldung unter Mac-Betriebssystem	15
4.5. OpenCV Implementierung unter Mac-Betriebssystem	16
5.1. Video Player	19
5.2. Login Screen	20
5.3. Kamera Interface - Integration der Kamera in das Interface, über den Button kann zwischen Farb- und Tiefenbild gewechselt werden.	21
B.1. Kamera-Screen mit Pseudo-Problem	32
B.2. Interface	33
B.3. Navigation Leiste	33

Tabellenverzeichnis

2.1. Gruppenaufteilung	4
A.1. Arbeitsaufteilung	28

Literaturverzeichnis

- [Bernardini u. a. 1999] BERNARDINI, F. ; MITTLEMAN, J. ; RUSHMEIER, H. ; SILVA, C. ; TAUBIN, G.: The ball-pivoting algorithm for surface reconstruction. In: *IEEE Transactions on Visualization and Computer Graphics* 5 (1999), Nr. 4, S. 349–359. <http://dx.doi.org/10.1109/2945.817351>. – DOI 10.1109/2945.817351
- [Cao u. a. 2020] CAO, Mingwei ; ZHENG, Liping ; LIU, Xiaoping: Single View 3D Reconstruction Based on Improved RGB-D Image. In: *IEEE Sensors Journal* 20 (2020), Nr. 20, S. 12049–12056. <http://dx.doi.org/10.1109/JSEN.2020.2968477>. – DOI 10.1109/JSEN.2020.2968477
- [Cignoni u. a. 2008] CIGNONI, Paolo ; CALLIERI, Marco ; CORSINI, Massimiliano ; DELLEPIANE, Matteo ; GANOVELLI, Fabio ; RANZUGLIA, Guido: MeshLab: an Open-Source Mesh Processing Tool. In: SCARANO, Vittorio (Hrsg.) ; CHIARA, Rosario D. (Hrsg.) ; ERRA, Ugo (Hrsg.): *Eurographics Italian Chapter Conference*, The Eurographics Association, 2008. – ISBN 978–3–905673–68–5
- [Hoffmann] HOFFMANN, Christopf: *Windows 10 mit Virtualbox in Mac OS X einrichten*. <https://www.pcwelt.de/ratgeber/Windows-10-mit-Virtualbox-in-Mac-OS-X-einrichten-9839076.html>. – Zuletzt abgerufen 15. September 2021
- [Kazhdan u. a. 2006] KAZHDAN, Michael ; BOLITHO, Matthew ; HOPPE, Hugues: Poisson surface reconstruction. In: *Proceedings of the fourth Eurographics symposium on Geometry processing* Bd. 7, 2006
- [Lupo] LUPO, Tim: *OpenCV Installation Tutorial Mac*. <https://www.youtube.com/watch?v=37RvqZVddAw>. – Zuletzt abgerufen 26. September 2021
- [NIST] NIST: *Installing vtk 5.10.1 on OS X El Capitan with MacPorts*. <https://www.ctcms.nist.gov/oof/oof3d/vtk5elcap.html>. – Zuletzt abgerufen 26 September 2021
- [Open3D] OPEN3D: *Open3D Website*. <http://www.open3d.org/>. – Zuletzt abgerufen 26 September 2021
- [Open3D Dokumentation] OPEN3D DOKUMENTATION: *Point Cloud Distance*. <http://www.open3d.org/docs/latest/tutorial/Basic/pointcloud.html#Point-Cloud-Distance>. – Zuletzt abgerufen 17. September 2021
- [OpenCV] OPENCV: *OpenCV Website*. <https://opencv.org/>. – Zuletzt abgerufen 26 September 2021

- [opencv 2014] OPENCV, dev t.: *surface_matching. Surface Matching*. https://docs.opencv.org/3.0-beta/modules/surface_matching/doc/surface_matching.html. Version:2014. – Last accessed 19 September 2021
- [PointCloudLibrary a] POINTCLOUDLIBRARY: *3D Object Recognition based on Correspondence Grouping*. https://pcl.readthedocs.io/projects/tutorials/en/latest/correspondence_grouping.html#correspondence-grouping. – Zuletzt abgerufen 26 September 2021
- [PointCloudLibrary b] POINTCLOUDLIBRARY: *PointCloudLibrary Website*. <https://pointclouds.org/>. – Zuletzt abgerufen 26 September 2021
- [PointCloudLibrary c] POINTCLOUDLIBRARY: *Removing outliers using a StatisticalOutlierRemoval filter*. https://pcl.readthedocs.io/projects/tutorials/en/master/statistical_outlier.html. – Zuletzt abgerufen 26 September 2021
- [PointCloudLibrary d] POINTCLOUDLIBRARY: *Removing outliers using a StatisticalOutlierRemoval filter*. https://pcl.readthedocs.io/projects/tutorials/en/latest/compiling_pcl_macosx.html. – Zuletzt abgerufen 26 September 2021
- [PointCloudLibrary e] POINTCLOUDLIBRARY: *Tutorial: Hypothesis Verification for 3D Object Recognition*. https://pcl.readthedocs.io/projects/tutorials/en/latest/global_hypothesis_verification.html#global-hypothesis-verification. – Zuletzt abgerufen 26 September 2021
- [Ramsrigoutham] RAMSRIGOUTHAM: *Integrating PCL and OpenCV – Pass-Through Filter Example*. <https://ramsrigoutham.wordpress.com/2012/06/28/integrating-pcl-and-opencv-passthrough-filter-example/>. – Zuletzt abgerufen 26. September 2021
- [Teng u. a. 2021] TENG, Xiaowen ; ZHOU, Guangsheng ; WU, Yuxuan ; HUANG, Chenlong ; XU, Shengyong: Three-Dimensional Reconstruction Method of Rapeseed Plants in the Whole Growth Period Using RGB-D Camera. In: *Sensors* 21 (2021), 07, S. 4628. <http://dx.doi.org/10.3390/s21144628>. – DOI 10.3390/s21144628
- [Tong u. a. 2012] TONG, Jing ; ZHOU, Jin ; LIU, Ligang ; PAN, Zhigeng ; YAN, Hao: Scanning 3D Full Human Bodies Using Kinects. In: *IEEE Transactions on Visualization and Computer Graphics* 18 (2012), Nr. 4, S. 643–650. <http://dx.doi.org/10.1109/TVCG.2012.56>. – DOI 10.1109/TVCG.2012.56
- [Visual Studio 2019] VISUAL STUDIO 2019: *Windows 10 mit Virtualbox in Mac OS X einrichten*. <https://visualstudio.microsoft.com/de/downloads/>. – Zuletzt abgerufen 15. September 2021
- [Wagh] WAGH, Shubham: *Steps to create Textured Mesh from Point Cloud using Meshlab*. <https://gist.github.com/shubhamwagh/0dc3b8173f662d39d4bf6f53d0f4d66b>. – Zuletzt abgerufen 26 September 2021
- [WPF] WPF: *Tutorial: Simple video player using WPF/C#*. <https://www.youtube.com/watch?v=eCvPy30MWlw>. – Zuletzt abgerufen 15. September 2021

- [Zhou u. a. 2018] ZHOU, Qian-Yi ; PARK, Jaesik ; KOLTUN, Vladlen: Open3D: A Modern Library for 3D Data Processing. In: *arXiv:1801.09847* (2018)

A. Arbeitsmatrix

Im folgenden wird die Arbeitsaufteilung während der Projekt 2 Phase aufgelistet. Dabei werden auf der linken Spalte die einzelnen Arbeitspakete benannt und in den rechten Spalten die Bearbeitung der einzelnen Gruppenmitglieder in Stunden aufgezeigt.

Tabelle A.1.: Arbeitsaufteilung

	Alexander Kring	Merle Struckmann	Feyza Keles	Merve Eser	Moritz Langer	Lukas Munz
Kamera Live Übertragung	70	70	0	0	0	0
Kamera Aufnahmen	50	40	0	0	0	0
Point Cloud Erstellung	60	60	0	0	0	0
Point Cloud Verbesserung	0	0	75	75	0	0
Surface Matching (PCL)	0	0	0	0	50	50
Surface Matching (OpenCV)	30	30	0	0	110	100
Point Cloud Matching	5	10	0	0	30	35
Surface Reconstruction	60	60	0	0	0	0
Interface	0	0	60	60	0	0
Kamera Integration	20	20	0	0	0	0
Zusammenführung	35	40	35	35	40	35

B. Anhang

OpenCV Surface Matching

```
//  
// IMPORTANT: READ BEFORE DOWNLOADING, COPYING, INSTALLING OR USING.  
//  
// By downloading, copying, installing or using the software you agree to this license.  
// If you do not agree to this license, do not download, install,  
// copy or use the software.  
//  
//  
//  
// License Agreement  
// For Open Source Computer Vision Library  
//  
// Copyright (C) 2014, OpenCV Foundation, all rights reserved.  
// Third party copyrights are property of their respective owners.  
//  
// Redistribution and use in source and binary forms, with or without modification,  
// are permitted provided that the following conditions are met:  
//  
// * Redistribution's of source code must retain the above copyright notice,  
//   this list of conditions and the following disclaimer.  
//  
// * Redistribution's in binary form must reproduce the above copyright notice,  
//   this list of conditions and the following disclaimer in the documentation  
//   and/or other materials provided with the distribution.  
//  
// * The name of the copyright holders may not be used to endorse or promote products  
//   derived from this software without specific prior written permission.  
//
```

B. Anhang

```
// This software is provided by the copyright holders and contributors "as is" and
// any express or implied warranties, including, but not limited to, the implied
// warranties of merchantability and fitness for a particular purpose are disclaimed.
// In no event shall the Intel Corporation or contributors be liable for any direct,
// indirect, incidental, special, exemplary, or consequential damages
// (including, but not limited to, procurement of substitute goods or services;
// loss of use, data, or profits; or business interruption) however caused
// and on any theory of liability, whether in contract, strict liability,
// or tort (including negligence or otherwise) arising in any way out of
// the use of this software, even if advised of the possibility of such damage.
//
// Author: Tolga Birdal <tbirdal AT gmail.com>

#include "opencv2/surface_matching.hpp"
#include <iostream>
#include "opencv2/surface_matching/ppf_helpers.hpp"
#include "opencv2/core/utility.hpp"

using namespace std;
using namespace cv;
using namespace ppf_match_3d;

static void help(const string& errorMessage)
{
    cout << "Program init error : "<< errorMessage << endl;
    cout << "\nUsage : ppf_matching [input model file] [input scene file]"<< endl;
    cout << "\nPlease start again with new parameters"<< endl;
}

int main(int argc, char** argv)
{
    // welcome message
    cout << "*****" << endl;
    cout << "* Surface Matching demonstration : demonstrates the use of surface matching"
        << endl;
    cout << "  using point pair features." << endl;
    cout << "* The sample loads a model and a scene, where the model lies in a different"
        << endl;
    cout << "  pose than the training.\n* It then trains the model and searches for it in the"
        << endl;
    cout << "  input scene. The detected poses are further refined by ICP\n* and printed to the "
        << endl;
    cout << "  standard output." << endl;
    cout << "*****" << endl;

    if (argc < 3)
    {
        help("Not enough input arguments");
        exit(1);
    }

    #if (defined __x86_64__ || defined _M_X64)
        cout << "Running on 64 bits" << endl;
    #else
        cout << "Running on 32 bits" << endl;
    #endif

    #ifdef _OPENMP
        cout << "Running with OpenMP" << endl;
    #else
        cout << "Running without OpenMP and without TBB" << endl;
    #endif

    string modelFileName = (string)argv[1];
    string sceneFileName = (string)argv[2];

    Mat pc = loadPLYSimple(modelFileName.c_str(), 1);
```

```

// Now train the model
cout << "Training..." << endl;
int64 tick1 = cv::getTickCount();
ppf_match_3d::PPF3DDetector detector(0.025, 0.05);
detector.trainModel(pc);
int64 tick2 = cv::getTickCount();
cout << endl << "Training complete in "
    << (double)(tick2-tick1)/ cv::getTickFrequency()
    << " sec" << endl << "Loading model..." << endl;

// Read the scene
Mat pcTest = loadPLYSimple(sceneFileName.c_str(), 1);

// Match the model to the scene and get the pose
cout << endl << "Starting matching..." << endl;
vector<Pose3DPtr> results;
tick1 = cv::getTickCount();
detector.match(pcTest, results, 1.0/40.0, 0.05);
tick2 = cv::getTickCount();
cout << endl << "PPF Elapsed Time " <<
    (tick2-tick1)/cv::getTickFrequency() << " sec" << endl;

// Get only first N results
int N = 2;
vector<Pose3DPtr> resultsSub(results.begin(), results.begin()+N);

// Create an instance of ICP
ICP icp(100, 0.005f, 2.5f, 8);
int64 t1 = cv::getTickCount();

// Register for all selected poses
cout << endl << "Performing ICP on " << N << " poses..." << endl;
icp.registerModelToScene(pc, pcTest, resultsSub);
int64 t2 = cv::getTickCount();

cout << endl << "ICP Elapsed Time " <<
    (t2-t1)/cv::getTickFrequency() << " sec" << endl;

cout << "Poses: " << endl;
// debug first five poses
for (size_t i=0; i<resultsSub.size(); i++)
{
    Pose3DPtr result = resultsSub[i];
    cout << "Pose Result " << i << endl;
    result->printPose();
    if (i==0)
    {
        Mat pct = transformPCPose(pc, result->pose);
        writePLY(pct, "para6700PCTrans.ply");
    }
}

return 0;
}

```


Interface

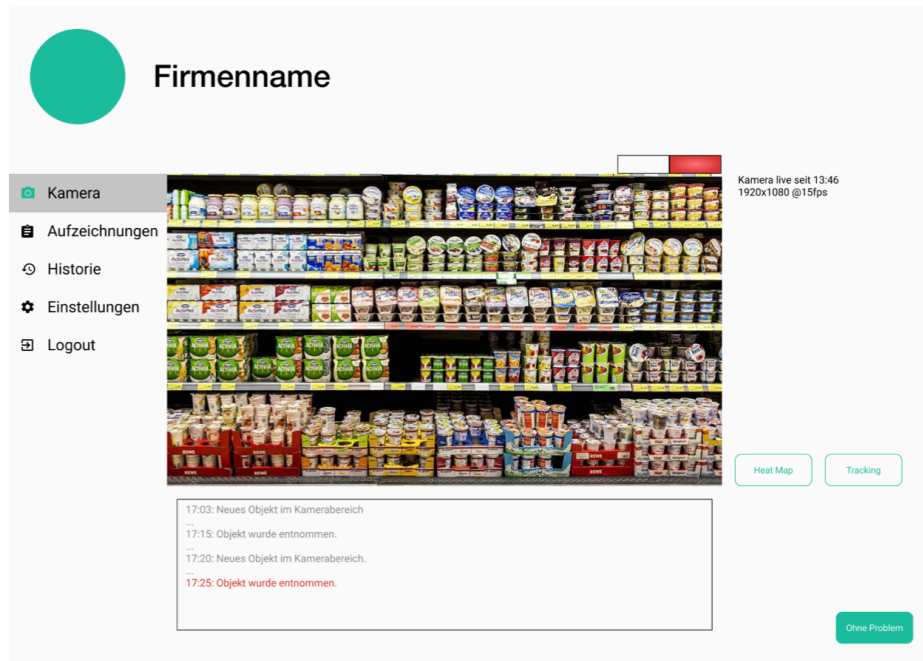


Abbildung B.1.: Kamera-Screen mit Pseudo-Problem

Interface

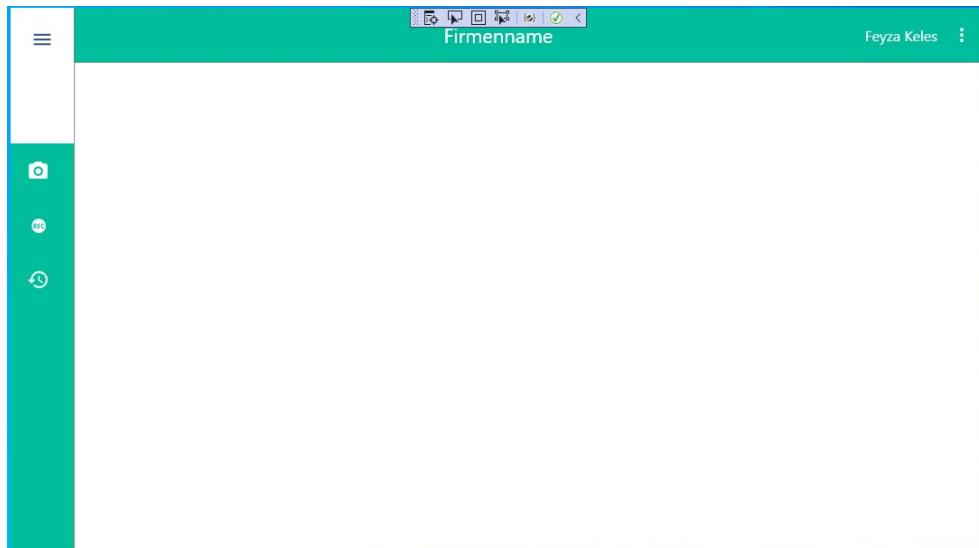


Abbildung B.2.: Interface

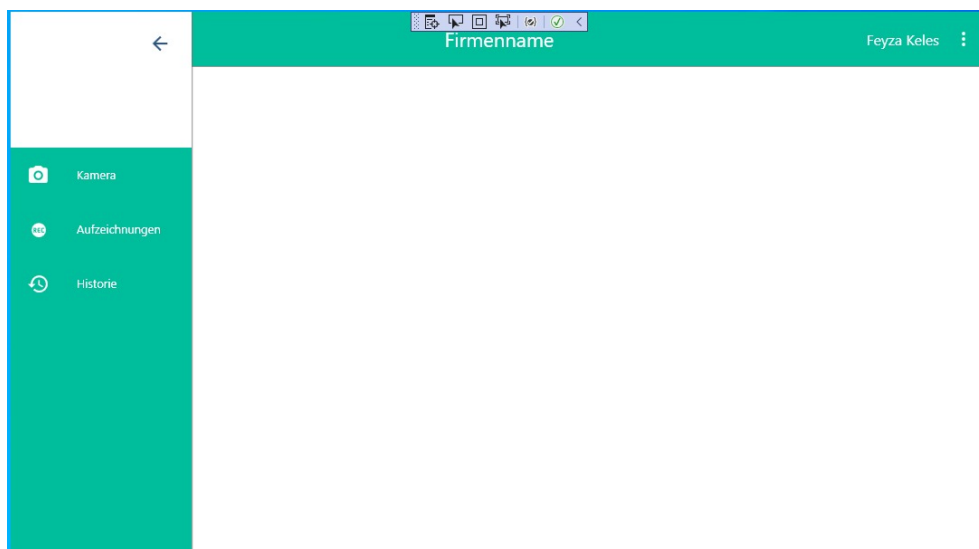


Abbildung B.3.: Navigation Leiste