Universitetet
i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

MASTEROPPGAVE

Studieprogram/spesialisering: Informasjonsteknologi - Kybernetikk og signalbehandling	Vårsemesteret, 2014 Åpen
Forfatter: Per Ivar Raugstad	………………………………………… (signatur forfatter)
Fagansvarlig: Karl Skretting Veileder: Arne Kristian Jansen	
Tittel på masteroppgaven: Gjenkjenning av objekter i tredimensjonale punktskyer skannet med Xbox Kinect Engelsk tittel: Object recognition in three-dimensional point clouds scanned with Xbox Kinect	
Studiepoeng: 30	
Emneord: Punktskyer, Gjenkjenning, 3D-skanning	Sidetall: 115 + vedlegg/annet: 96 + CD Stavanger, 16. juni 2014

Sammendrag

Skanning med 3D-skannere gir en mengde tredimensjonale koordinater som representerer punkter i rommet. Denne samlingen av punkter kalles for punktskyer. Denne oppgaven omhandler objektgjenkjenning i punktskyer. I oppgaven gjøres gjenkjenningen i punktskyer som er skannet med Xbox Kinect.

Point Cloud Library (PCL) er et C++ bibliotek som tilbyr mange algoritmer og funksjoner for behandling av punktskyer. For objektgjenkjenningen er det benyttet to algoritmer, ≪Correspondence Grouping≫ og ≪Implicit Shape Model≫, fra PCL-biblioteket. Disse algoritmene er blitt implementert i et program kalt Point Cloud Recognition Tool. Dette programmet er programmert i C++ og har et grafisk brukergrensesnitt for enkelt å kunne åpne, lagre og visualisere punktskyer. I programmet er det også implementert andre funksjoner for behandling av punktskyer. For skanning av punktskyer brukes applikasjonen Kinfu Large Scale, som er tilgjengelig i PCL-biblioteket.

I oppgaven blir de to algoritmene brukt til gjenkjenning mellom skannede punktskyer. Det blir også gjort gjenkjenning mellom tredimensjonale CAD-modeller og skannede punktskyer. Det viste seg at det var mulig å kjenne igjen objekter i begge tilfellene med de to algoritmene. De beste resultatene ble oppnådd under gjenkjenning mellom skannede punktskyer.

Forord

Med denne oppgaven fullfører jeg masterstudiet mitt i informasjonsteknologi, automatisering og signalbehandling ved Universitetet i Stavanger, våren 2014. Det har vært noen fine år der jeg truffet mange flotte mennesker.

Jeg vil gjerne benytte anledningen til å takke min veileder Arne Kristian Jansen for god veiledning, gode tilbakemeldinger og som til tross for en travel hverdag alltid har tatt seg tid til å hjelpe meg i situasjoner der jeg har stått fast. Jeg vil også rette en takk til Karl Skretting og Ståle Freyer.

Til slutt vil jeg takke min familie og min forlovede for å ha støttet og holdt ut med meg gjennom skrivingen av denne oppgaven.

Rennesøy 15. juni 2014

Per Ivar Raugstad

Innhold

1 Innledning **1**
- 1.1 Bakgrunn for oppgaven 1
- 1.2 Problemstilling 2
- 1.3 Litt historie om Kinect 3
- 1.4 Litt historie om Point Cloud Library(PCL) 5
- 1.5 Rapportens struktur 6

2 Maskinvare **9**
- 2.1 Kinect 9
  - 2.1.1 Spesifikasjoner 10
  - 2.1.2 Hvordan virker Kinect 12
  - 2.1.3 Dybdeoppløsning 14

3 Programmvare **17**
- 3.1 Point Cloud Library 17
  - 3.1.1 Underbiblioteker og drivere 18
  - 3.1.2 Datastrukturen ≪PointCloud≫ 18
  - 3.1.3 Filformater brukt i PCL 20
- 3.2 Bibliotekene Qt og VTK(Grafisk brukergrensesnitt og visualisering) 22
- 3.3 Kinfu Large Scale 22
- 3.4 Annen programvare brukt i prosjektet 24

4 Algoritmer **27**
- 4.1 Kd-tree 27
  - 4.1.1 Nærmeste-nabo-søk 29
- 4.2 Octree 30
  - 4.2.1 Deteksjon av forskjeller i rommet 31
- 4.3 Punktnormaler 32
- 4.4 Objektgjenkjenning i punktskyer 33
- 4.5 Beskrivelse av egenskaper til et 3D-punkt 34
  - 4.5.1 Point Feature Histogram(PFH) 37
  - 4.5.2 Fast Point Feature Histogram(FPFH) 38
  - 4.5.3 Signature of Histograms of Orientations(SHOT) 39
- 4.6 Hough-transform 42
  - 4.6.1 Hough-transform for rette linjer 43
  - 4.6.2 Generalisert Hough-transform 44
- 4.7 Iterative Closest Point(ICP) 46

4.8 Correspondence Grouping 47
4.9 Implicit Shape Model 51

5 Implementering **55**
5.1 Point Cloud Recognition Tool 55
5.1.1 Funksjonalitet 56
5.1.2 Programstruktur 63
5.2 Implementering av ≪Correspondence Grouping≫ 65
5.2.1 Initialisering 66
5.2.2 Gjennomgang av algoritmen 67
5.3 Implementering av ≪Implicit Shape Model≫ 71
5.3.1 Initialisering 72
5.3.2 Gjennomgang av algoritmen 73

6 Eksperimentelt og Resultater **77**
6.1 Forberedelser 77
6.1.1 Installering av Kinfu Large Scale 77
6.1.2 Maskinvare 78
6.1.3 Skanning med Kinfu Large Scale 78
6.1.4 Etterprosessering 81
6.2 Testing og resultater 84
6.2.1 Gjenkjenning mellom to skannede punktskyer 86
6.2.2 Gjenkjenning mellom CAD-modell og skannet punktsky . 93
6.2.3 Deteksjon av forskjeller mellom punktskyer 99
6.2.4 Sammenligning av tidsforbruk 100

7 Diskusjon og Konklusjon **103**
7.1 Skanning 103
7.2 Gjenkjenning 104
7.2.1 ≪Correspondence Grouping≫ 105
7.2.2 ≪Implicit Shape Model≫ 106
7.2.3 Parameterinnstilling 107
7.2.4 Deteksjon av forskjeller mellom punktskyer 107
7.3 Konklusjon 108

A Innhold på CD **116**

B Point Cloud Recognition Tool kildekode **117**
B.1 pcrtool.cpp 118
B.2 mainwindow.h 119
B.3 mainwindow.cpp 121
B.4 centralwidget.h 126
B.5 centralwidget.cpp 128
B.6 panels.h 134
B.7 panels.cpp 140
B.8 pcltools.h 168
B.9 pcltools.cpp 171
B.10 recognition.h 197
B.11 recognition.cpp 199
B.12 recognitionISM.cpp 203

B.13 recognitionISM.cpp 204
B.14 button.h 207
B.15 button.cpp 208
B.16 CMakeLists.txt 210

Kapittel 1

Innledning

1.1 Bakgrunn for oppgaven

Denne oppgaven har blitt gitt av Stormfjord Oil and Gas AS. Selskapet leverer løsninger for 3D-modellering og CAD rettet mot oljeindustrien og jobber med hovedmålsetning om å finne nye og lettere måter å ta i bruk 3D-data og gjøre informasjonen lettere tilgjengelig for operasjonell bruk. Under utvikling av industrielle verktøy eller installasjoner både i oljebransjen og andre områder, er det vanlig at det blir utarbeidet en 3D-modell av verktøyet som skal bygges. 3D-modellen brukes til referanse, oppmåling og planlegging av installasjon eller bruk av det spesifikke verktøyet. Under verktøyets levetid kan det hende at det er ønskelig med endring eller modifikasjoner på verktøyet. Det vil da oppstå et avvik mellom 3D-modellen og fysiske verktøyet. Oppdatering av 3D-modeller kan være tidkrevende og kostbart. Stormfjord ønsker derfor å se på om det er mulighet for forenkling av denne prosessen ved bruk av 3D-skanning og gjenkjenning for å detektere avvik mellom 3D-modell og fysisk objekt.

Figur 1.1.1: Eksempel på CAD-modeller av industrielle verktøy. Bildet er hentet fra Stormfjord. [1]

1.2 Problemstilling

Denne masteroppgaven kan deles opp i to hovedproblemer. Det ene er skanning av objekter med Xbox Kinect og det andre er gjenkjenning av skannede objekter. Dersom datasett scannet med Kinect kan brukes i gjenkjenning vil dette være en fordel siden Xbox Kinect er et rimelig og lett tilgjengelig alternativ til konvensjonelle 3D-skannere. Selve skanningen skal utføres med en åpen-kildekode applikasjon kalt Kinfu Large Scale. Denne skal brukes for å finne ut om den gir data som kan brukes videre til gjenkjenning i CAD-modeller eller andre scannede datasett.

Skanning av objekter i 3D resulterer i en mengde koordinater. Koordinatene som genereres er gitt i rommet og defineres med avstand fra origo i x,y og z-retning. Denne mengden med koordinater kalles for en punktsky. Til sammenligning med todimensjonale bilder som representerer en projeksjon av en scene, kan punktskyen ses på som et bilde av rommet der den ekstra dimensjonen som punktskyen har gir dybdeinformasjon til de opprinnelige to dimensjonene. Med sin ekstra dimensjon er punktskyene fortsatt underlagt reglene for euklidske rom der blant annet vektorprodukt og avstand gis av ligningene 1.2.1 og 1.2.2.

$$\boldsymbol{x} \cdot \boldsymbol{y} = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n \tag{1.2.1}$$

$$d(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\| = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{1.2.2}$$

Her er $\boldsymbol{x}$ og $\boldsymbol{y}$ vektorer og n er antallet dimensjoner i rommet, $\boldsymbol{R}^n$. [2] På grunn av disse sammenhengene vil mange av egenskapene som gjelder for todimensjonale bilder også gjelde for punktskyer. Dette fører til at mange algoritmer som i første omgang ble utviklet for todimensjonal bildebehandling, kan utvides til å fungere for tredimensjonal bildebehandling. Et eksempel er på dette er Hough-transformen som ofte brukes til deteksjon av linjer, sirkler og geometriske former i bilder men som også i flere tilfeller har blitt utvidet til å kunne brukes i tredimensjonale data. Hough-transformen er beskrevet nærmere i kapittel 4.6 og en utvidelse av hough-transformen til tre dimensjoner er beskrevet i slutten av kapittel 4.6.2.

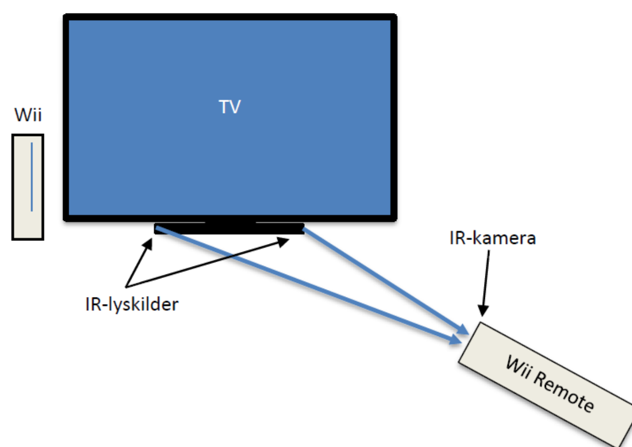
I gjenkjenningsdelen av prosjektet skal det undersøkes algoritmer som kan brukes for å gjenkjenne objekter i punktskyer. I tillegg vil det ses på om det er mulig å detektere forskjeller mellom punktskyene. Point Cloud Library (PCL), er et bibliotek i C++, som består av en stor samling av algoritmer for operasjoner i punktskyer. I denne oppgaven vil algoritmer fra PCL-biblioteket brukes for å løse de nevnte oppgavene. Her vil spesielt algoritmene ≪Correspondence

Grouping≫ og ≪Implicit Shape Model≫ brukes for å løse oppgaven med gjenkjenning.

1.3 Litt historie om Kinect

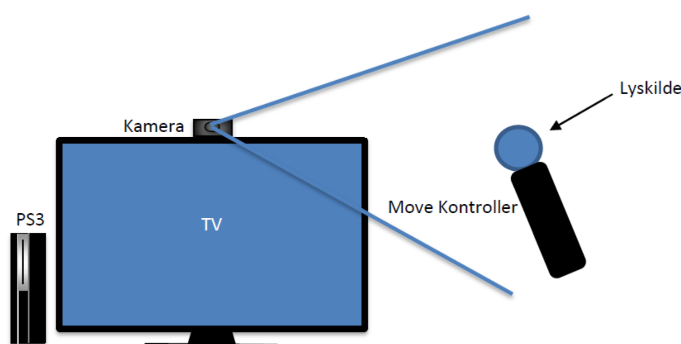
Spillindustrien har så lenge den har eksistert hatt en akselererende effekt på teknologiutvikling og nytenkning både innen programvare og maskinvare. Et godt eksempel på dette er det evige kappløpet om den beste og mest fotorealistiske grafikken i spill. Dersom en ønsker bedre grafikk kreves det kraftigere maskinvare samt mer optimalisert programvare. Et annet eksempel er spillerens interaksjon med spillet. Den tradisjonelle måten å løse dette på har lenge vært et sett med knapper som utfører forskjellige funksjoner, enten i form av tastatur eller håndholdte spillkontrollere.

Interaktiv spilling der spillerens fysiske bevegelser blir registrert og brukt til å kontrollere spillet er et eksempel på spillindustriens nytenkning. Sony var tidlig ute med denne typen spillkontroll da de lanserte EyeToy som et tillegg til Playstation 2 i 2003. EyeToy var hovedsakelig et lite kamera som kunne registrere spillerens bevegelser som igjen ble brukt til å kontrollere spillets fremgang. Da Nintendo tre år senere lanserte sin spillkonsoll, Nintendo Wii, mente mange at dette ville revolusjonere spillbransjen. Grunnen til dette var at Wii-konsollen ble levert med Wii Remote kontrolleren som standardutstyr. Dette var en trådløs peker-enhet som ved hjelp av et infrarødt kamera i kontrolleren og infrarøde lyskilder i nærheten av tv-skjermen, kunne registrere sin egen bevegelse i tre dimensjoner. I figur 1.3.1 vises en skisse av Wii-systemet.



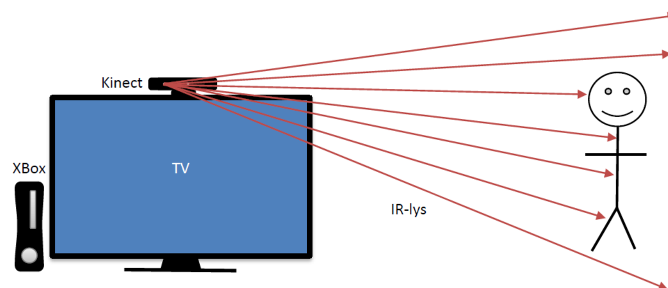
Figur 1.3.1: Bildet viser Wii-systemet. Her sendes infrarødt lys ut fra lyskilder plassert i nærheten av tv-skjermen. Dette lyset registreres av kameraet som sitter i Wii Remote kontrolleren og brukes til å bestemme posisjonen i rommet.

Sonys svar på Wii Remote ble Move til Playstation 3. Med lignende egenskaper som Wii Remote var dette er en håndholdt kontroller med en lysende kule i den ene enden. Denne kontrolleren brukes sammen med etterfølgeren av EyeToy, Playstation Eye. Spillerens bevegelser registreres av kameraet og kontrollens lyskule brukes som et referansepunkt (Figur 1.3.2). Microsoft valgte også å komme med et svar på konkurrentenes bevegelses kontrollerte spillteknologier. Microsoft valgte derimot en litt annen retning da de lanserte sin Kinect-sensor. [3]



Figur 1.3.2: Bildet viser Playstation Move. Her brukes kameraet Playstation Eye til å registrere den lysende kulen på Move-kontrolleren.

Microsoft annonserte for første gang Kinect under E3-messen i juni 2009. Da under kodenavnet ≪Poject Natal≫. I november året etter ble Kinect sluppet på markedet. Den ble gitt ut som et tillegg til Xbox 360 som på dette tidspunktet var Microsofts nyeste spillkonsoll. Her var det meningen at spilleren skulle kunne ta mer aktivt del i spillet og styre spillets fremgang ved hjelp av bevegelser og tale. Til sammenligning med konkurrentenes bevegelses-kontrollere, er Kinect-sensoren ikke avhengig av at spilleren har en håndholdt kontroller. Kinect-sensoren består av et RGB-kamera og et system for 3D-skanning ved bruk av strukturert lys (Figur 1.3.3), i tillegg til fire mikrofoner. Kinect kan dermed tilby både telegjenkjenning og ansiktgjenkjenning. Hver spiller blir også fulgt av Kinect-sensoren ved at spillerens ledd blir kartlagt. Både leddets posisjon og dets avstand fra andre ledd blir utregnet i real-time. [4]



Figur 1.3.3: Kinect-sensoren sender ut et mønster i infrarødt lys som fanges opp av en IR-sensor. Informasjonen som hentes inn brukes til å fastslå avstander i scenen.

I november 2010 ble det av selskapet ≪Adafruit Industries≫ utlovet en dusør på 2000 amerikanske dollar til den som kunne lage en åpen kildekode driver til Kinect. Microsofts første respons var å fordømme utfordringen som et forsøk på å modifisere deres produkt og at Kinect hadde innebygde sikkerhetsrutiner både i maskinvaren og programvaren for å forhindre tukling med deres produkt. [5] Dette resulterte i at ≪Adafruit Industries≫ økte dusøren til 3000 amerikanske dollar. Konkuransen resulterte i utviklingen av en linux driver som kunne benytte seg av Kinects RGB-kamera og dybdedata. [6] Etter hvert viste det seg at Microsoft første uttalelse var basert på en intern misforståelse, og at selskapets egentlige syn var at hacking eller tukling med deres fysiske produkt ikke kunne tolereres, men at usb-tilkoblingen var designet til å være åpen. [7] [8]

Våren 2011 lanserte Microsoft sin egen utviklerpakke kalt ≪Kinect for Windows SDK≫. Dette åpnet for for bruk av Kinects funksjoner i ikke-kommersielle applikasjoner skrevet i enten C++, C# eller Visual Basic. En kommersiell versjon ble utgitt senere i februar 2012.

I PCL-biblioteket brukes OpenNI-driveren til å kommunisere med Kinecten. OpenNI (Open Natural Interaction) er et åpne-kildekode alternativ til Microsofts ≪Kinect for Windows≫-rammeverk. OpenNI var opprinnelig en industriledet non-profit organisasjon bestående av flere medlemmer der noen av de ledende medlemmene var Willow Garage, Asus og PrimeSense. PrimeSense stod også bak maskinvaren Kinect-sensoren. Den 23. april 2014 ble PrimeSense kjøpt opp av Apple, som igjen medførte en nedstengning av domenet www.openni.org. [9] [10] OpenNI er for tiden tilgjengelig nettsiden `http://structure.io/openni`.

1.4 Litt historie om Point Cloud Library(PCL)

Utviklingen av Point Cloud Library ble påbegynt av Willow Garage i USA. Det California-baserte selskapet ble startet opp i 2006 og driver forskning på robo-

ter og roboter sin oppfatning av omverdenen. Selskapet er pådriver for utvikling av hardware og åpen kildekode programvare for personlige robotapplikasjoner. En av selskapets større satsinger er robotplattformen Personal Robot 2 (PR2). PR2 har to armer, begge med sju grader av frihet (DOF). Den har også et avansert persepsjonssystem bestående av et kamera og en time-of-flight laserscanner for tredimensjonal oppfatning av omgivelser. PR2 er ment til å være en felles plattform for utviklere til videre forskning og utvikling av robotteknologi. Blant selskapets øvrige produkter finnes også hobbyplatformen TurtleBot 2 og kommunikasjonsplatformen Texai. Willow Garage er derimot kanskje mest kjent for sitt åpen kildekode operativsystem for roboter, Robot Operating System(ROS). ROS kan lastes ned under BSD-lisens som tilsier at det kan fritt lastes ned endres på og bukes både privat og kommersielt. [11] [12]

I mars 2010 startet utviklingen av PCL basert på erfaringer fra det to måneder eldre prosjektet Point Cloud Mapping. Motivasjonen for PCL var å samle forskning på 3D-persepsjon og kombinere eksisterende 3D-algoritmer og flere års erfaring i feltet til et samlet rammeverk. Da Microsoft ga ut Kinect-sensoren i november 2010 fikk flere utviklere øynene opp for PCL. Dette resulterte i et raskt voksende utviklermiljø. Fokuset ble da skiftet mot å forbedre stabiliteten og brukervennligheten til biblioteket for å gjøre det enklere for andre å utvikle applikasjoner basert på PCL. Etter et års utvikling tok tidligere forsker ved Willow Garage, Radu B. Rusu, avgjørelsen om å gjøre PCL til et uavhengig prosjekt. Domenet www.pointclouds.org ble registrert og åpnet i mars 2011. Her ble den første versjonen av PCL (v1.0) sluppet i mai samme året. PCL drives i dag av selskapet Open Perception ledet av Radu B. Rusu og har et stort utviklermiljø som består av flere utviklere og bidragsytere fra hele verden. PCL finansieres som et samarbeid mellom noen av de største kommersielle aktørene på feltet. I februar 2014 har PCL nådd versjonsnummer 1.7.1. [13] [14] [15]

1.5 Rapportens struktur

Kapittel 2 - Maskinvare
I dette kapittelet beskrives Kinect-sensoren og teknologien som benyttes for å fange opp dybdedata.

Kapittel 3 - Programvare
I dette kapittelet beskrives programvaren som ble brukt i prosjektet. Her blir PCL-biblioteket beskrevet og de ulike C++ bibliotekene som ble brukt i implementeringen. Her beskrives også applikasjonen for skanning, Kinfu Large Scale.

Kapittel 4 - Algoritmer
I dette kapittelet gjennomgås teorien bak de viktigste algoritmene som ble brukt i oppgaven. Dette inkluderer en detaljert gjennomgang av de to algoritmene ≪Correspondence Grouping≫ og ≪Implicit Shape Model≫.

Kapittel 5 - Implementering
I dette kapittelet beskrives implementeringen og funksjonaliteten til programmet Point Cloud Recognition Tool. Det blir også gjennomgått hvordan de to algoritmene, ≪Correspondence Grouping≫ og ≪Implicit Shape Model≫, er implementert.

Kapittel 6 - Eksperimentelt og Resultater
I dette kapittelet beskrives først oppsett av utstyr og programvare for skanning. Deretter vises resultatene fra testing av algoritmene for gjenkjenning med forskjellige typer punktskyer. Deretter vises et resultat fra forskjellsdeteksjon og en sammenligning av tidsforbruket til de to algoritmene for gjenkjenning.

Tillegg A - Innhold på CD

Tillegg B - Point Cloud Recognition Tool kildekode

Kapittel 2

Maskinvare

I dette kapittelet blir det sett nærmere på Xbox Kinect. Her studeres maskinvaren som Kinecten bruker. Det blir også sett litt på hvordan Kinecten virker og hvordan dybdeinformasjonen blir til.

2.1 Kinect

Kinect-sensoren er opprinnelig et tilleggsutstyr til Xbox 360 og har en webkamera lignende design for plassering i nærheten av tv. Sensoren er et klasse 1 laserprodukt som tilsier at den ved normal bruk kan brukes sikkert under alle forhold.

Figur 2.1.1: Kinect for Windows. Foto hentet fra MSDN [16]

Kinect leveres i to versjoner. En beregnet på Xbox 360, kalt Kinect for Xbox, og

en beregnet for bruk sammen med Windows, kalt Kinect for Windows. Kinect for Xbox leveres med adapter for tilkobling via usb. Dette er fordi tidlige versjoner av Xbox 360 ikke hadde den nødvendige porten for Kinect. Selv om maskinvaren i de to versjonene er den samme, anbefaler Microsoft å bruke Kinect for Windows sammen med deres SDK. [17]. Dette er på grunn av at denne er optimalisert for Kinect for Windows SDK samtidig som Kinect for Xbox er optimalisert for Xbox 360. I tillegg oppfyller ikke applikasjoner utviklet med den sistnevnte kravet for lisensiering. Kinect for Windows-versjonen leveres også med kortere USB-kabel for mer stabil kjøring under intensiv bruk i tillegg til litt ekstra funksjonalitet. Den ekstra funksjonaliteten er listet i tabell 2.1.1.

SDK-versjon	**Funksjonalitet**
v1.0	Sette ≪DepthRange≫ til ≪Near≫
v1.5	Forbedrede farger i fargebilder
v1.6	Mulighet for å slå av IR-emitter

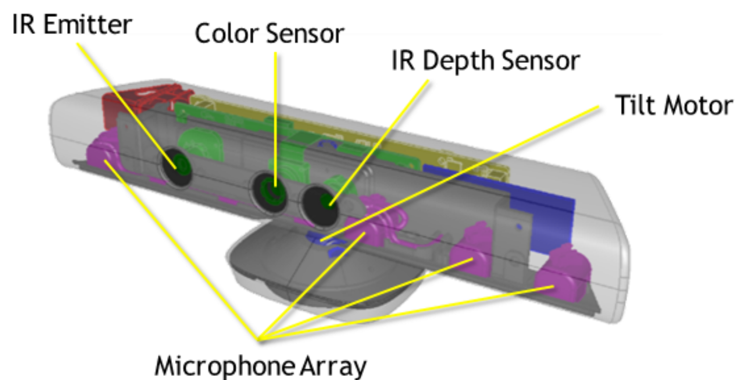
Tabell 2.1.1: Ekstra funksjonalitet tilgjengelig ved bruk av Kinect for Windows-versjonen av Kinect-sensoren sammen med SDK. Tabellen viser hvilken SDK-versjon som tilbyr disse funksjonene. [17]

På Microsoft sine hjemmesider [18], rapporteres det at Kinect-sensorens standardrekkevidde er minimum 800 mm og maksimum 4000 mm. Med en Kinect for Windows vil det derimot være mulig å sette ≪DepthRange≫ til ≪Near≫, som ifølge Microsoft vil gi en rekkevidde på minimum 500 mm og maksimum 3000 mm.

Kinect for Windows bruker egne drivere som følger med Kinect SDK-pakken. På grunn av dette, var denne versjonen i utgangspunktet ikke kompatibel med OpenNI rammeverket som brukes av PCL. Det ble laget en løsning på dette ved å lage en bro mellom SDK driverene og driverene brukt i OpenNI. [19] Kinect for Xbox er i dag et billigere alternativ til Kinect for Windows med en prisdifferanse på ca. 900 kr. Dersom det lages applikasjoner for Kinect for kommersiell bruk, må det gjøres med den kommersielle versjonen av Kinect for Windows SDK.

2.1.1 Spesifikasjoner

I Figur 2.1.2 vises et oversiktsbilde av komponentene som Kinect-sensoren inneholder og plasseringen av disse. De mest åpenbare komponentene er ett RGB-kamera, IR-emitter, IR-sensor, fire mikrofoner og en tiltmotor. En IR-emitter er en lyskilde som sender ut infrarødt lys. En IR-sensor er et kamera som detekterer infrarødt lys. I tillegg til disse komponentene har Kinecten også innebygd akselerometer.



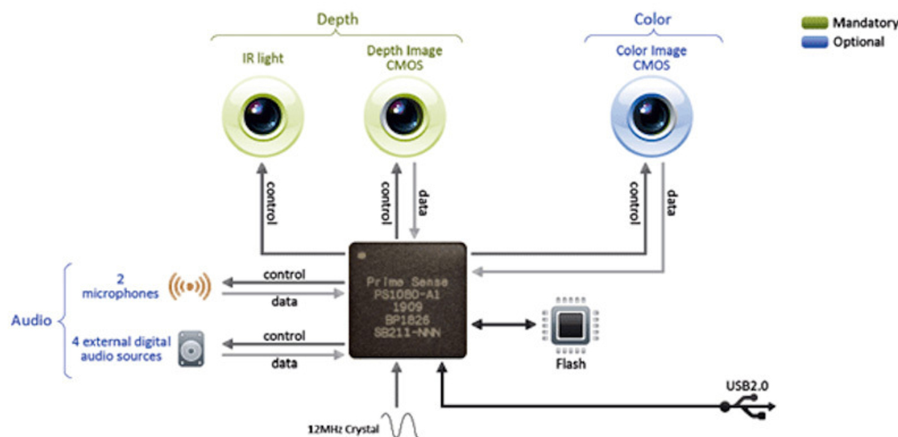
Figur 2.1.2: Kinect-sensorens komponenter. Figur hentet fra MSDN [20]

IR-emitteren brukes til å projisere et mønster av infrarødt lys ut i rommet. IR-sensoren registrerer det reflekterte infrarøde lyset. Informasjonen blir brukt til å lage dybde informasjon om scenen foran Kinecten. Mikrofonene kan brukes til å ta opp lyd og finne ut hvilken retning lyden kommer fra. Akselerometeret som sitter i Kinecten kan registrere aksellerajon langs tre akser og kan brukes blant annet til å finne ut hvordan Kinecten er orientert. RGB-kameraet som sitter i Kinecten har en standard oppløsning på video på 640x480 ved 30 bilder per sekund. Ved lavere bilderater kan både RGB-kameraet og IR-sensoren registrere data med en oppløsning på 1280x1024. [8]

Kinect	**Spesifikasjon**
Vertikal bildevinkel	43°
Horisontal bildevinkel	57°
Tiltvinkel	±27°
Oppløsning ved 30 fps (RGB og IR)	640x480
Oppløsning ved lavere fps (RGB og IR)	1280x1024
Aksellerometer rekkevidde	2G/4G/8G ($1G = 9.81m/s^2$)
Alselerometer nøyaktighet	1°

Tabell 2.1.2: Kinect-sensorens spesifikasjoner. [20] [8]

Hjernen i Kinect-sensoren er prosessoren PrimeSense PS1080. I figur 2.1.3 vises et diagram av en referanseplattform fra PrimeSense som viser hvordan PS1080 kan brukes. Diagrammet viser et lignende oppsett som det i Kinect-sensoren. Prosessoren leser både lyd, farge og dybde-data som prosesseres og sendes til usb. [21]



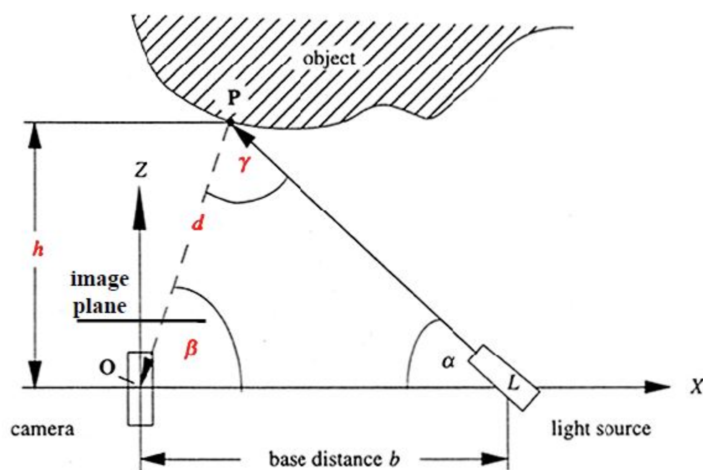
Figur 2.1.3: Figuren viser et diagram over Kinect-sensorens oppsett. [21]

2.1.2 Hvordan virker Kinect

For å kunne lage en tredimensjonal representasjon av en scene, konstruerer Kinect-sensoren et dybdekart. Dette skjer i prosessoren levert av PrimeSense. Detaljer om teknologien ikke tilgjengelig men i [22], gjøres det et forsøk på å forklare teknologien basert på patentsøknader fra PrimeSense [23] [24]. Her spekuleres det i at dybdekartet blir konstruert ved bruk av strukturert lys, dybdeinformasjon fra stereo og dybdeinformasjon fra fokus. Prinsippet om dybdeinformasjon fra fokus baseres på at ting som er mer uklare i et bilde har større avstand fra kameraet. I tillegg til dette tar Kinecten også i bruk maskinlæringsteori for gjenkjenning av personer og kroppsposisjon. [22] Området for denne rapporten er begrenset til skanning av stillestående objekter uten bevegelser. For mer informasjon om Kinect-sensorens gjenkjenning av kroppsposisjoner, se [22] og [25, s.476-485].

Strukturert lys og dybdeinformajon fra stereo

I stereovision brukes to kamera rettet mot scenen. Dybdeinformasjonen blir utregnet ved triangulering mellom korresponderende punkter i de to bildene. Et problem med denne metoden er å kunne finne korrespondansen mellom interessepunktene i de to bildene korrekt. Dette kan være både vanskelig og kostbart med tanke på regnekraft. Feil matching av punkter vil gi feil dybdeverdi [25, s.227]. Denne metoden kalles også passiv triangulering.

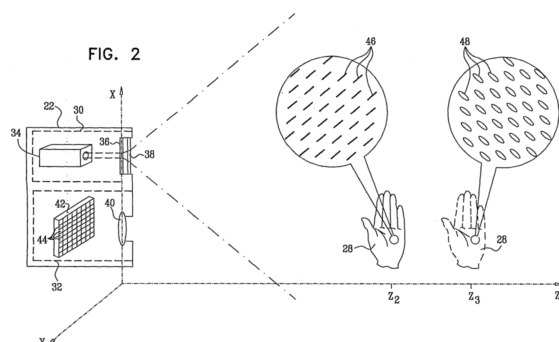


Figur 2.1.4: Figuren viser triangulering med et aktivt stereovision-system. Figuren er hentet fra [26]

Ved bruk av aktiv triangulering, fjernes ett av kameraene i stereovision-systemet og erstattes med en lyskilde. Lyskilden belyser objektet og kameraet fanger opp det reflekterte lyset. På denne måten elimineres korrespondanseproblemet fra den passive trianguleringen. Bruk av aktiv triangulering er vist i figur 2.1.4. Avstanden, d, fra kameraet til det reflekterte lyspunktet regnes ut ved å bruke formel 2.1.1. Her er b, avstanden mellom lyskilden og kameraet. [26] [25, s.452-454]

$$d = \frac{b \cdot \alpha}{sin(\alpha + \beta)} \tag{2.1.1}$$

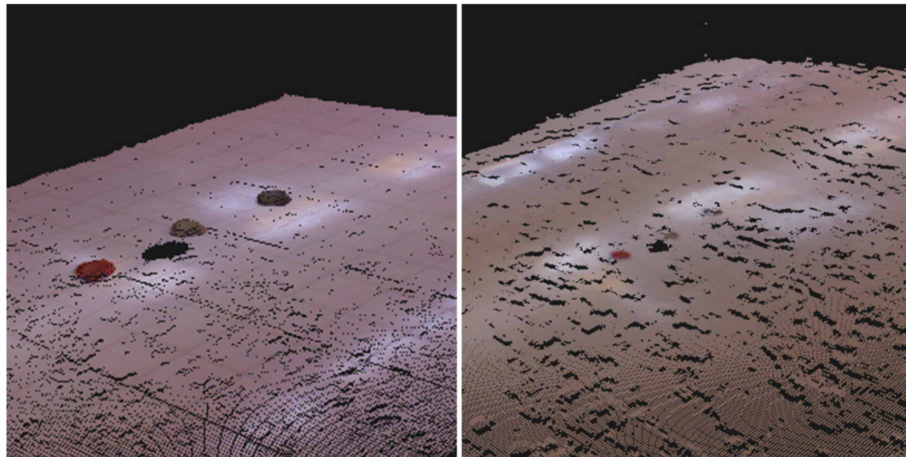
Ved bruk av strukturert lys, belyses scenen med et mønster som er lett gjenkjennelig og som skiller seg ut fra scenen. Ved å søke etter mønsteret i bildet og analysere deformasjonen av det kan dybdekartet regnes ut. I følge [22], brukes det i Kinect-sensoren en infrarød laser. Denne belyser scenen med et mønster av prikker (specklepattern). Dette prikkemønsteret sendes først gjennom en astigmatisk linse som har forskjellig karakteristikk i x og y retning. Dette gjør at prikkene ser ut som ellipser. Disse ellipsenes orientering er avhengig av avstanden til sensoren som vist i figur 2.1.5.



Figur 2.1.5: Figuren viser prikkemønsteret der de projiserte ellipsenes orientering er av avhengig av avstanden fra IR-emitteren. Figuren er hentet fra [24]

2.1.3 Dybdeoppløsning

Under innhenting av dybde data fra Kinect-sensoren vil økende avstander gi økende grad kvantifisering. Dette skjer på grunn av sensorens oppløsning og det kan føre til kvantifiseringsfeil i målingene. Figur 2.1.6 viser et eksempel på kvantifiseringsfeil.

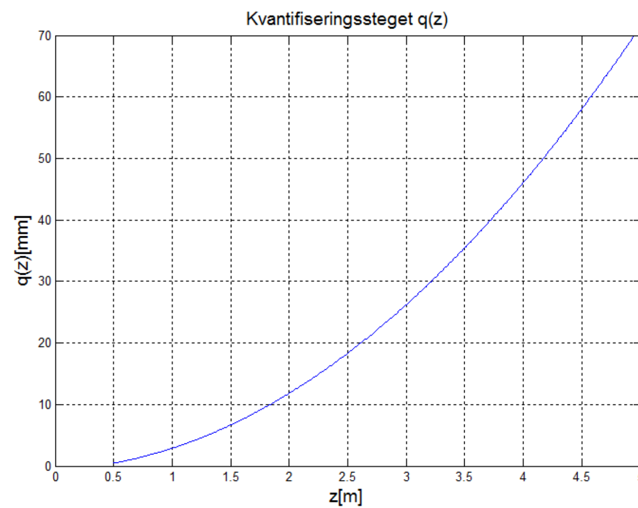


Figur 2.1.6: Figuren viser to punktskyer. Begge punktskyene stammer dybdestrømmen fra Kinect-sensoren. Skyene viser en overflate med tre referansepunkter som er skannet fra avstandene 0.5 meter (venstre) og 2 meter (høyre). I den venstre figuren er overflaten jevn. Til høyre vises effekten av kvantifiseringsfeilen som ujevnheter i overflaten.

I [27] ble Kinect-sensorens dybdeoppløsning testet. Her blir det målt hvor stort steget i kvantifiseringen er som funksjon av avstanden fra Kinect-sensoren. Her ble det testet med avstander fra 0.5 til 15 meter der det ble sett på målinger gjort i en bildevinkel på 5° rundt midten av bildet. Av resultatene ble økningen

i kvantifiseringen estimert til en funksjon $q(z)$ (Ligning 2.1.2) der z er avstanden til sensoren i meter. Ligning 2.1.2 gir en kurve som vist i figur 2.1.7.

$$q(z) = 2.73z^2 + 0.74z - 0.58[mm] \tag{2.1.2}$$



Figur 2.1.7: Her er kvantifiseringssteget, $q(z)$, plottet mot avstanden, z, fra 0.5 til 5 meter med formelen i 2.1.2. [27].

Kapittel 3

Programmvare

I dette kapittelet beskrives programvaren og kodebibliotekene som ble brukt i prosjektet. Først beskrive PCL-biblioteket. Her blir det sett på hvilke biblioteker som PCL er avhengig av. Deretter gjennomgås noen av de vanligste programmeringsteknikkene som er brukt i PCL-biblioteket. Videre blir applikasjonen som brukes til skanning, Kinfu Large Scale, beskrevet. Her blir det beskrevet kort hvordan denne virker og hvordan dybdedataene hentes fra Kinecten og lagres i minnet. Til slutt er det tatt med litt informasjon om resten av programvaren som er brukt i denne oppgaven.

3.1 Point Cloud Library

Point Cloud Library er et storskala, åpent prosjekt der alle som ønsker kan komme med bidrag. Biblioteket inneholder avanserte algoritmer for bilde- og punktskybehandling. Biblioteket er utviklet i C++ og er laget for å være kryssplattform kompatibelt. Biblioteket har mulighet for å kompileres for Windows, Linux, Mac OS og Android/iOS. Åpen kildekode blir vektlagt og biblioteket distribueres under en 3-klausul BSD-lisens. [15] Kort oppsummert vil dette si at biblioteket fritt kan distribueres så lenge informasjon om opphavsrett og ansvarsfraskrivelse er beholdt i kildekoden. I tillegg skal navn på bidragsyterne ikke bli brukt i arbeid som bygger på biblioteket, uten at det er gitt tillatelse til det. [28]

I dette prosjektet brukes PCL-versjon nummer 1.7.1 til programmeringen. Dette er den nyeste offisielle versjonen av PCL i skrivende stund. PCL-biblioteket er derimot stadig under utvikling og alle de nyeste oppdateringene er tilgjengelig i en versjon kalt PCL Master. Denne versjonen er fritt tilgjengelig for nedlasting fra GitHub. [29] PCL Master inkluderer funksjoner og applikasjoner som fortsatt

er på prototype-stadiet. Blant disse finnes også applikasjonen Kinfu Large Scale som åpner for skanning med Kinect-sensoren. Denne applikasjonen brukes i dette prosjektet og beskrives nærmere i kapittel 3.3.

3.1.1 Underbiblioteker og drivere

Point cloud library bygger på en samling av flere underbiblioteker. Disse implementerer forskjellige funksjoner som PCL er avhengig av for å kunne kjøre sine algoritmer. Fremgangsmåte for installering i Windows er beskrevet på PCL sine hjemmesider. Her er også en oversikt over hvilke versjoner av de ulike bibliotekene som er påkrevd. [30]

- **Boost** inneholder løsninger for grunnleggende programmering for blant annet pekere og threading. [31]
- **Eigen** inneholder metoder for lineær algebra, matrise og vektor operasjoner optimalisert for SSE. [32]
- **FLANN** brukes for sin implementasjon av datastrukturen kdtree. Kdtree er en trestruktur som er et nyttig verktøy for å utføre raske estimat av nærmeste-nabo-søk. [33]
- **Visualization ToolKit(VTK)** brukes til å visualisere punktskyer. Kan bygges til å fungere sammen med Qt. [34]
- **Qt** brukes for å lage GUI-applikasjoner basert på PCL. [35]
- **QHull** brukes i noen algoritmer som beregner på overflater. [36]
- **OpenNI** brukes for å kunne hente inn punktskyer fra enheter som støttes. [37]
- **Nvidia CUDA** er en parallellprosesseringplattform fra Nvidia som åpner for bruk av skjermkortprosessoren(GPU) til krevende operasjoner, på skjermkort med støtte for dette. [38]

3.1.2 Datastrukturen ≪PointCloud≫

Den mest brukte datastrukturen i PCL er punktskystrukturen ≪PointCloud≫. Strukturen er basert på programmeringsteknikken template. I C++ er et ≪template≫, en form for uspesifisert variabel type. Dette åpner for å kunne definere egne variabeltyper. En ≪PointCloud≫ kan opprettes med en PointT-template. Denne spesifiserer hvilken type punkter, punktskyen bruker. Dette gjør at punktskystrukturen er veldig generell og at det er mulig å lage sin egen type punktsky

ved å definere sin egen PointT-template. I C++ er en peker er en type variabel, som lagrer minneadressen til en verdi, og ikke selve verdien. ≪PointCloud≫-klassen implementerer også mulighet for å opprette pekere som kan inneholde minneadressen til en punktsky. Nedenfor vises et eksempel på opprettelse av en punktsky, punktsky-peker og en punktsky-konstantpeker.

```
PointCloud<PointT> punktsky;
PointCloud<PointT>::Ptr punktsky_peker;
PointCloud<PointT>::ConstPtr punktsky_konstant_peker;
```

Punktskyen

Det finnes to hovedtyper av punktskyer. Organiserte punktskyer og uorganiserte punktskyer. En punktsky kan kalles organisert dersom dataene utgjør matriselignende struktur. Det vil si punktene i punktskyen er organisert i rader og kolonner. Fordelen med organiserte punktskyer er at søkealgoritmer som nærmeste-nabo-søk vil fungere raskere. Skanningsutstyr som Kinect-sensoren vil ofte gi organiserte punktskyer. I uorganiserte punktskyer er det ingen struktur og punktene opptrer i tilfeldig rekkefølge i en liste. I punktsky-strukturen ≪PointCloud≫ er det noen hovedvariabler som beskriver skyen:

- `height`(int) er skyens høyde dersom det er en organisert punktsky. Da spesifiserer denne variabelen antallet kolonner i strukturen. For uorganiserte punktskyer settes denne verdien til 1.

- `width`(int) er skyens bredde dersom det er en organisert punktsky. Da spesifiserer denne variabelen antallet rader i strukturen. Altså hvor mange punkter det er i hver kolonne. Dersom det er en uorganisert punktsky inneholder denne antallet punkter i skyen.

For en organisert punktsky med `width=620` og `height=380` er antallet punkter gitt ved $620 \cdot 380 = 235600$. En uorganisert sky med samme antall punkter ville vært definert med `width=235600` og `height=1`. Antallet punkter i en punktsky er dermed i begge tilfeller gitt ved:

```
int antall = punktsky.width*punktsky.height;
```

Videre inneholder punktsky-strukturen ≪PointCloud≫, hovedvariablene:

- `points`(std::vector<PointT>) er en liste som inneholder alle punktene i skyen av den spesifiserte typen PointT.

- `isDense`(bool) er `true` dersom alle punkter i punktskyen har en endelig og gyldig verdi. Dersom punktskyen inneholder punkter med verdien NaN(Not a Number) er denne false.

PointT-malen

PointT kan defineres av en PCL-bruker til et ønsket formål. Dette gjør ≪PointCloud≫-strukturen svært allsidig og begrenser den ikke til å være en liste med xyz-koordinater med og uten farge, men også informasjon som overflatenormaler, interessepunkter og punktbeskrivelser med flere. Nedenfor er noen eksempler på PCLs mange ferdigdefinerte punkttyper.

- `PointXYZ` inneholder x, y og z verdien til et punkt. Punktene er referert til et kartesisk koordinatsystem ut fra med verdier gitt i forhold til origo.

- `PointXYZRGB` inneholder x, y og z verdien til et punkt i tillegg til fargekomponentene rød grønn og blå. Fargene spesifiseres i verdiområdet 0 til 255 for hver komponent.

- `Normal` blir brukt for å lagre informasjon om normalen til overflaten i et gitt punkt. Hvordan normaler kan finnes i punktskyer er beskrevet i kapittel 4.3

- `SHOT352` er en punkt-type som brukes for punktbeskrivelser av typen SHOT. SHOT er en algoritme for beskrivelse av punktegenskaper som studeres nærmere i kapittel 4.5.3.

```
PointCloud<PointXYZ>     punktsky;
PointCloud<PointXYZRGB> punktsky_farge;
PointCloud<Normal>       punktsky_normaler;
PointCloud<SHOT352>      SHOT_punktbeskrivelser;
```

3.1.3 Filformater brukt i PCL

Det finnes mange forskjellige filformater for lagring av tredimensjonale data. Alle har sine fordeler og ulemper. PCL støtter lesing og skriving til flere av disse men i dette prosjektet brukes i hovedsak Polygonal File Format (.ply) og Point Cloud Data (.pcd) for lagring av punktskydata.

Polygon File Format (.ply)

PCL støtter både lesing og skriving av filformatet Polygon File Format også kjent som Stanford Triangle Format. Filformatet ble utviklet på 90-tallet og er

et vanlig format for lagring av tredimensjonale data fra 3D-scannere. Filformatet inneholder enkle beskrivelser av et objekt bestående av flate polygoner. Et polygon er en figur definert av en sekvens av rette linjer som i enkleste form utgjør en trekant. [39] I tillegg til dette støtter det også andre egenskaper som farge, gjennomsiktighet, overflatenormaler, teksturkoordinater og verdier for usikkerhet. Formatet kan kodes i enten ren tekst referert til ASCII-kode eller binært. Filformatet starter med en header som gir informasjon om innholdet. I headeren brukes nøkkelord som ≪element≫ for å spesifisere typen data og ≪property≫ for å spesifisere dataenes egenskaper. Etter headeren følger en liste som inneholder alle dataene. Nedenfor vises et eksempel på en header i ply-formatet. [40]

```
ply
format ascii 1.0
comment eksempleheader
element vertex 458192
property float x
property float y
property float z
property float nx
property float ny
property float nz
element face 116736
property list uchar uint vertex_indices
end_header
0.406553 -1.399557 -1.605141 0.993687 0.054814 0.097883
0.406725 -1.402671 -1.605141 0.993687 0.054814 0.097883
0.406869 -1.402665 -1.606608 0.993687 0.054814 0.097883
0.406697 -1.399549 -1.606608 0.993687 0.054814 0.097883
...
```

Point Cloud Data (.pcd)

Point Cloud Data formatet er PCL-bibliotekets egenutviklede filformat. Dette filformatet er laget for å kunne støtte punktskydata generert med PCL. PCL forsvarer utviklingen av enda et filformat med at dette skal støtte alle funksjonene PCL-biblioteket tilbyr. I likhet med .ply formatet har også .pcd en header som spesifiserer dataenes type og egenskaper etterfulgt av en liste med punkter. [41]

3.2 Bibliotekene Qt og VTK(Grafisk brukergrensesnitt og visualisering)

I behandling av punktskyer er det viktig å kunne ha mulighet for visualisering av resultater. Spesielt ved gjenkjenning i punktskyer er visualiseringen viktig siden verifisering og vurdering av resultatene gjøres ved hjelp av det visuelle.

I dette prosjektet blir det utviklet et grafisk brukergrensesnitt(GUI) ved hjelp av Qt-biblioteket. Dette er et C++ bibliotek som åpner for enklere programmering av programmer med GUI. Biblioteket gir tilgang til enkle GUI-funksjoner som blant annet knapper, fil-menyer og fil-velgere men har også mulighet for mer avansert funksjonalitet. De ulike GUI-elementene blir i Qt-biblioteket omtalt som ≪widgets≫. Biblioteket tilbyr gode løsninger for oppsett og presentasjon av ≪widgets≫.

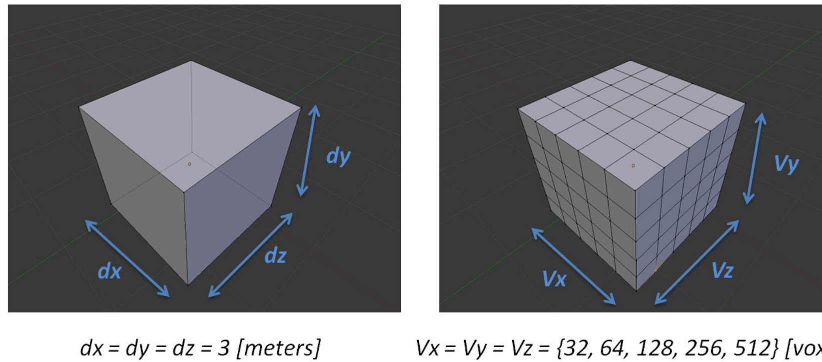
PCL-biblioteket tilbyr avansert funksjonalitet for visualisering av punktskyer gjennom klassen ≪PCLVisualizer≫. Med denne klassen opprettes det et interaktivt vindu der brukeren kan vise punktskyer og rotere kameraet for å skifte synsvinkel. Klassen kan også visualisere andre ting som for eksempel geometriske figurer, tekst og andre typer 3D-data. Denne klassen er avhengig av VTK-biblioteket for å fungere. For å gjøre det mulig å kunne bruke ≪PCLVisualizer≫-klassen sammen med et grafisk brukergrensesnitt laget i Qt må VTK-biblioteket kompileres med støtte for Qt. VTK-biblioteket kompilert med dette kalles QVTK.

3.3 Kinfu Large Scale

Kinfu Large Scale er en eksperimentell applikasjon som tilbys gjennom PCL Master, omtalt i kapittel 3.1. PCL Master kan lastes ned fra GitHub. [29] Kinfu Large Scale kan brukes til å skanne objekter ved hjelp av Kinect-sensoren. Dette gjøres ved å holde Kinect-sensoren rettet mot objektet som skal skannes og flytte den forsiktig rundt objektet. Kinfu LargeScale utnytter minnet på PCens skjermkort[1] og henter fortløpende punktskyer fra Kinecten og setter disse sammen til en komplett punktsky i form av et TSDF-volum[2]. TSDF-volumet lagres i minnet på skjermkortet i et voxel grid(rutenett av voksler). I et TSDF-volum blir avstanden bestemt som funksjon av voksel-koordinatene. I volumet blir det definert en overflate der fortegnet til verdien i hvert voksel-koordinat sier om koordinatet er foran eller bak overflaten. [43] [44]

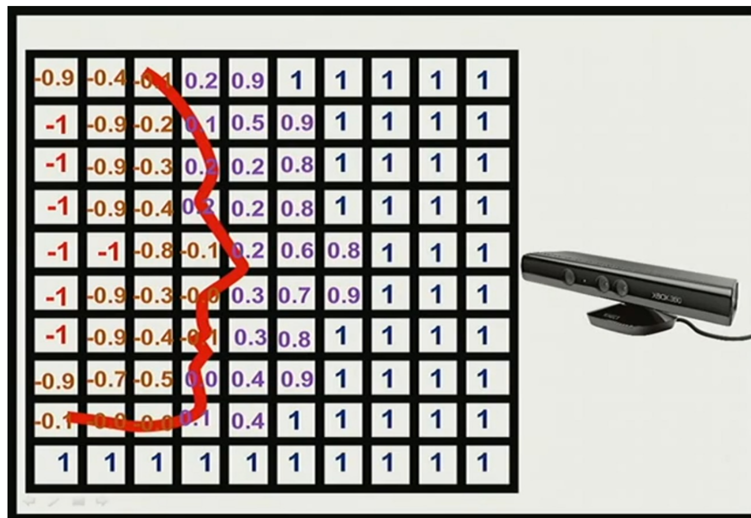
[1] Forutsetter et nVidia skjermkort med støtte for nVidia CUDA(Compute Unified Device Architecture), som åpner for bruk av GPU til prosessering [42]

[2] Truncated Signed Distance Function



Figur 3.3.1: En kube med sider på 3 meter(venstre), deles opp i voxler langs x, y og z-aksen(høyre). Figuren er hentet fra [44]

Det opprettes en kube med sidelengde gitt i meter. Kuben deles opp i et gitt antall voksler langs hver akse. Antallet voxler er satt til 512 som standard men kan også settes til 32, 64, 128 og 256. Lengden av kuben og antallet voxler gir oss oppløsningen på punktskyen. Disse parametrene avgjør også hvor mye minne som brukes av punktskyen på skjermkortet. [44]



Figur 3.3.2: Her vises TSDF-volumet som lagres i GPUen. Hver rute representerer en voxel. Hver voksel inneholder en verdi i området -1 til 1. Voksler som er på forsiden av overflaten får verdien 1 og voksler som ligger bak overflaten får verdien -1. Figuren er hentet fra [44]

Under innhentingen av data fra Kinect-sensoren brukes en teknikk kalt stråleavstøpning for å oversette dybdekartet til TSDF-volumet [45]. Når informasjonen skal hentes ut traverseres TSDF-volumet forfra og bakover der hver verdi i rutenettet sjekkes etter verdier i området mellom -1 og 0.98. Voksler med ver-

dien 1 representerer tomrom og ignoreres. På denne måten hentes kun verdier som ligger nært til overflaten.

For hvert dybdekart som hentes fra Kinecten, brukes blant annet bilateral filtrering og ICP-algoritmen[3] (Kapittel 4.7) for å sette sammen de ulike dybdekartene i sanntid [45]. Dersom Kinecten flyttes bort fra kuben som avgrenser TSDF-volumet, trigges en prosess kalt ≪skifting≫. Informasjonen i det forrige TSDF-volumet lagres da i den totale modellen av scenen i CPUen og TSDF-volumet i GPUen blir satt opp på nytt med den nye orienteringen til Kinecten.

Når skanningen er ferdig, lagres TSDF-punktskyen i filen world.pcd. For å kunne bruke denne som en punktsky må den først konerteres til et 3D-mesh. Kinfu Large Scale Mesh Output er en applikasjon som også er tilgjengelig i PCL Master. Denne applikasjonen laster inn TSDF-volmet i world.pcd og konverterer dette til 3D-mesh. Hver kube som ble skannet resulterer i en mesh.ply fil.

3.4 Annen programvare brukt i prosjektet

I dette prosjektet brukes tre gratisprogrammer til blant annet konvertering mellom filformater, filtrering av 3D-data og til å oppdrive CAD-data. Visual Studio 2008 brukes til programmering av punktsky-gjenkjenning med PCL-biblioteket.

MeshLab

MeshLab er et program som kan lastes ned fra [46]. Programmet er gratis og tilbyr mange funksjoner for redigering av 3D-data. I dette prosjektet blir MeshLab brukt til sammensetting og forenkling av datasett scannet med Xbox Kinect. I tillegg brukes noen filtre for å redusere støy i punktskyene.

Blender

Programmet blender er gratis og kan lastes ned fra [47]. Programmet har flere bruksområder for 3D-modellering. I dette prosjektet brukes blender til å generere testsett for algoritmene for gjenkjenning. Det brukes også til konvertering fra filtypen COLLADA (.dae) til det punktskyvennlige formatet .ply.

[3] Iterative Closest Point

SketchUp

SketchUp er gratis og kan brukes til 3D-modellering av konstruksjoner og objekter. Det kan lastes ned fra [48] Programmet inkluderer 3D Warehouse der en kan laste ned flere ferdiglagde modeller av forskjellige objekter. I dette prosjektet brukes modeller fra 3D Warehouse som en erstatning for CAD-data. Dette er nødvendig siden det er vanskelig å oppdrive ekte CAD-data fra bedrifter. Grunnen til dette er at CAD-data fra bedrifter ofte beskyttes for å bevare bedriftenes hemmeligheter.

Visual Studio 2008 og CMake

Visual Studio er et IDE[4] som leveres av Microsoft. PCL-biblioteket er laget i programspråket C++ og programmering som er gjort i dette prosjektet er derfor gjort i C++. I tillegg til å bruke Visual Studio 2008 er VisualAssist også brukt. Dette er et tilleggsverktøy til Visual Studio som forenkler programmeringen.

Programmet CMake er gratis og kan lastes ned fra [49]. Programmet brukes til å sette opp prosjekter i Visual Studio fra C++ kildekode. CMake sørger for å generere prosjektfiler og å koble de nødvendige linkene mellom PCL og de nødvendige bibliotekene som trengs for at PCL skal fungere. PCL versjonsnummer 1.5.1 for Visual Studio 2008 og 1.6.0 for Visual Studio 2010 kan installeres med ferdigkompilerte installasjonsfiler. Disse er tilgjengelige fra [50] og inneholder PCL og alle de nødvendige underbibliotekene. Nyere versjoner av PCL må settes opp manuelt ved å laste ned underbibliotekene separat. Oppsett av PCL-prosjekter med CMake er dokumentert gjennom veiledninger på PCLs hjemmesider [51] og [30].

[4]Integrated Development Enviroment

Kapittel 4

Algoritmer

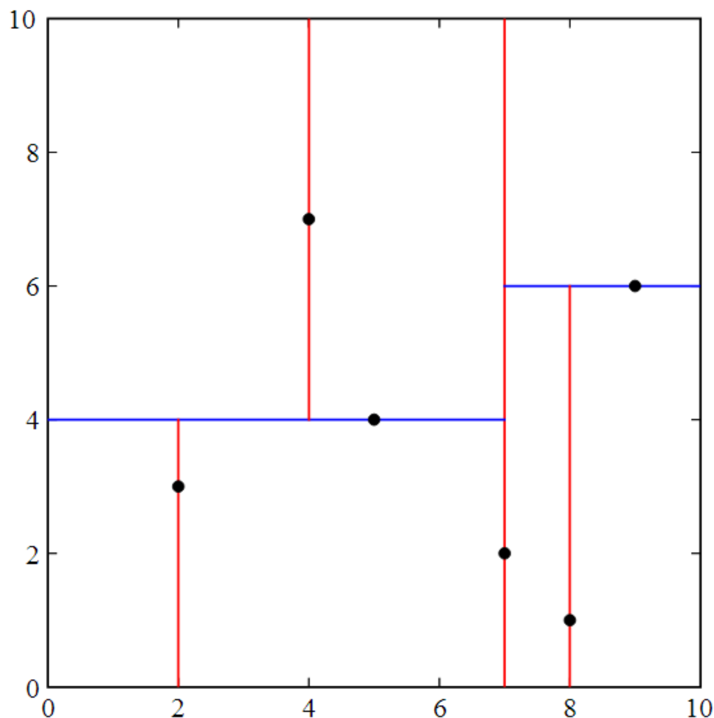
I dette kapittelet beskrives noen av de viktigste algoritmene som ble brukt i gjennomføringen av dette prosjektet. Først beskrives datastrukturene Kd-tree og Octree som brukes til forskjellige søkeoperasjoner i punktskyer. Deretter beskrives metoder for å utføre rigide transformasjoner og for å finne punktnormaler til overflater. Videre følger en litt generell beskrivelse av gjenkjenning i punktskyer etterfulgt av en detaljert gjennomgang av tre deskriptor-algoritmer. Mot slutten av kapittelet beskrives Hough-transformen først generelt og deretter hvordan den kan brukes ved gjenkjenning i tre dimensjoner. Deretter presenteres ICP-algoritmen som kan brukes for minimering av feil mellom to punktskyer. Til slutt følger en detaljert gjennomgang av de to algoritmene som brukes for gjenkjenning, ≪Correspondence Grouping≫ og ≪Implicit Shape Model≫.

4.1 Kd-tree

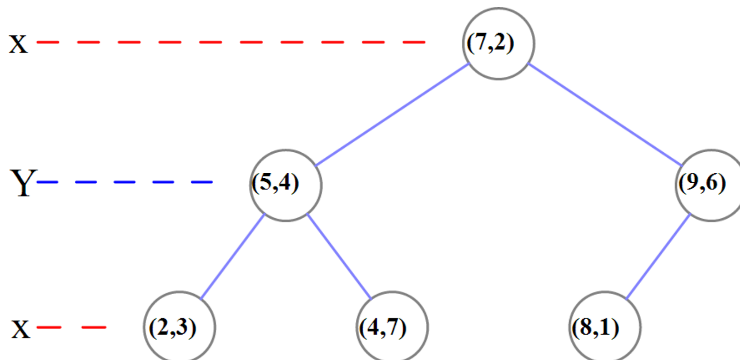
Kd-tree er en trestruktur som kan brukes for datasett med punkter i K dimensjoner(Kd). Strukturen åpner for raske tilnærminger til k-nærmeste-nabo-søk og radius-søk i datasettet. Kd-tree strukturen er et binært tre som tilsier at hver node har maks to undernoder(barn). Kd-tree strukturen er godt egnet til bruk i punktskyer ettersom det i flere algoritmer er bruk for raske nærmeste-nabo eller radius-søk. [52]

Ved opprettelsen av et Kd-tree splittes et datasett i to ved medianverdien til punktene, parallelt langs en av aksene. Datasettet er nå delt inn i to celler. Disse to cellene splittes igjen ved sine interne medianverdier langs en annen akse. Dette gjentas til det ikke er er flere punkter igjen i datasettet. For en nærmere beskrivelse av konstruksjon av Kd-tree strukturen se [52] eller [25, s.667]. I figur 4.1.1 vises et 2D eksempel på splitting av et datasett og i figur 4.1.2 vises den

resulterende trestrukturen.

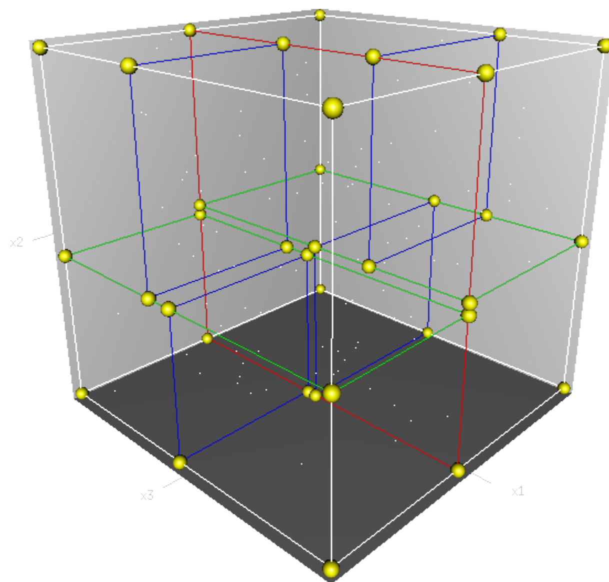


Figur 4.1.1: I figuren vises hvordan et datasett med 6 punkter splittes for å lage et Kd-tree. Siden antallet punkter er et partall, velges et av medianpunktene til å splittes først. Her velges punktet $(7, 2)$. Datasettet splittes dermed langs y-aksen i $x = 7$ og $(7, 2)$ settes som rotnode i treet. I de to cellene som oppstår splittes de resterende punktene i medianverdi langs y-aksen. Her blir dette punktet $(5, 4)$ og $(9, 6)$. Cellene splittes dermed i henholdsvis $y = 4$ og $y = 6$. Dette gjentas til det ikke gjenstår flere punkter i datasettet og den resulterende trestrukturen blir som vist i figur 4.1.2. Figuren er hentet fra Wikipedia. [53]



Figur 4.1.2: Her vises den resulterende trestrukturen fra splittingen som ble utført på datasettet i figur 4.1.1. Punktet $(7, 2)$ er her rotnoden siden det var den første som ble splittet. Figuren er hentet fra Wikipedia. [53]

Når Kd-tree strukturen utvides til tre dimensjoner skjer mye av det samme som i figur 4.1.1 og 4.1.2. Forskjellen er at det er en ekstra akse å ta hensyn til under splittingen av datasettet. Datasettet splittes derfor ved bruk av plan først perpendikulært langs x-aksen, deretter y-aksen og til slutt langs z-aksen. Dette repeteres til det ikke er flere punkter igjen i datasettet. [52] [25, s.667] I figur 4.1.3 vises et eksempel på splitting av et datasett i tre dimensjoner.



Figur 4.1.3: Figuren viser et tredimensjonalt eksempel. Hele datasettet utgjør rotnoden i treet. Rotnodens to barn blir til ved å splitte datasettet med et plan parallelt med x-aksen. De to nye cellene splittes parallelt med y-aksen og blir til rotnodens barnebarn. Disse splittes videre parallelt med z-aksen og de resulterende cellene blir til det neste nivået i treet. Denne prosessen repeteres som i de andre eksemplene. Figuren er hentet fra Wikipedia. [53]

4.1.1 Nærmeste-nabo-søk

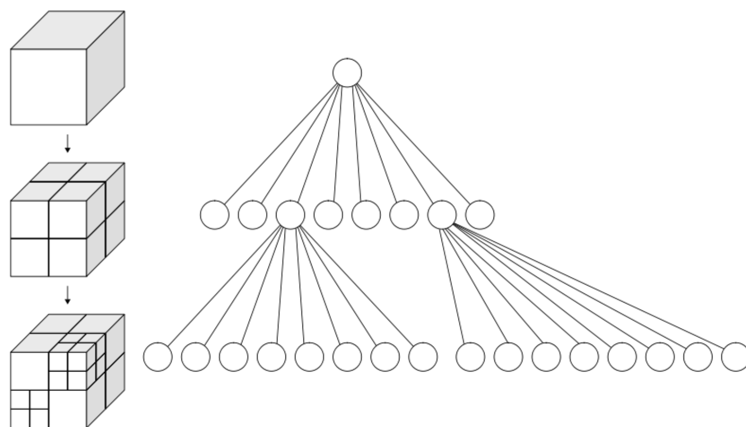
En av hovedfordelene med Kd-tree strukturen er at den kan brukes til å utføre raske nærmeste-nabo-søk. Søk etter en nabo i et datasett med n antall punkter ville med et lineært søk brukt $O(n)$ regneoperasjoner. Dette fordi det i et lineært søk må sjekkes avstanden mellom et punkt og alle andre punkt i datasettet for å finne naboen til dette punktet. Ved bruk av Kd-tree strukturen reduseres dette antallet til $O(log(n))$ i gjennomsnitt. [53]

I et nærmeste-nabo-søk med Kd-tree settes først et punkt som det ønskes å finne nærmeste nabo til. Deretter starter algoritmen i rotnoden og går til venstre eller høyre i treet avhengig om punktet er mindre eller større enn splitt-verdien i den gitte dimensjonen. Dette fortsetter ved at treet traverseres rekursivt og når en bladnode nås, settes denne til å være det ≪beste treffet≫. Deretter fortsetter algoritmen tilbake oppover i treet. For hver node sjekkes om avstanden til denne

noden er mindre enn det ≪beste treffet≫. I så fall er denne noden det nye ≪beste treffet≫. Deretter sjekkes det om en sirkel med radius lik avstanden fra punktet til det ≪beste treffet≫ krysser om et av planene som splitter datasettet krysses av sirkelen. Dersom dette skjer betyr det at det kan være bedre treff i den andre grenen til den nåværende noden. Denne må da traverseres på samme måte som resten av treet. Dersom sirkelen ikke krysser planet kan den grenen til den nåværende noden elimineres og algoritmen kan fortsette oppover i treet. Når algoritmen når rotnoden er prosessen ferdig og et estimat av det ≪beste treffet≫ er funnet. [53]

4.2 Octree

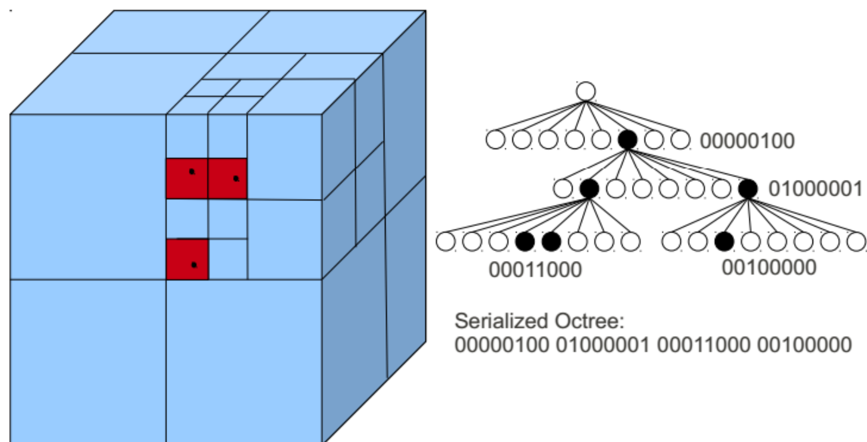
Et Octree er en annen trestruktur som er nyttig i operasjoner utført på 3D-data. Denne strukturen kan også benyttes for raske nabo-søk, men i denne oppgaven brukes den for deteksjon av forskjeller mellom punktskyer. Strukturen baseres på at datasettet deles opp i kuber. Et datasettet puttes inn i en kube. Kuben deles opp i åtte like store kuber. Hver av de åtte kubene som fortsatt inneholder punkter deles opp i åtte nye kuber. Dette gjentas og datasettet deles opp i mindre og mindre kuber helt til det nås en nedre grense for størrelsen på kubene. Denne oppdelingen resulterer i en trestruktur der hver node har enten åtte eller ingen barn. Et eksempel på dette er vist i figur 4.2.1. [54]



Figur 4.2.1: Et eksempel på en Octree-struktur. I eksempelet settes datasettet inn i en kube som blir til rotnoden i tre-strukturen. Denne kuben deles opp i åtte like store kuber, som representeres i treet som rotnodens barn. To av de åtte kubene deles opp i åtte nye kuber hver. Dette resulterer i at de to nodene, som representerer de to kubene, får åtte barn hver i treet. Figuren er hentet fra Wikipedia. [54]

At hver node i et octree har åtte barn er en fordel med tanke på minnehåndtering. Hver node kan da representeres med kun en Byte. Denne inneholder informasjon om nodens barn i form av 8 bits. Bit-verdien er binær. Dersom et bit har

verdien 1, indikerer dette at den korresponderende barnenoden, har egne barn. Et eksempel på Byte-representasjonen er vist i figur 4.2.2. [55]



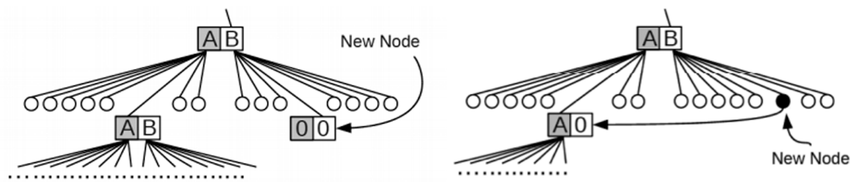
Figur 4.2.2: Figuren viser hvordan nodene i et octree blir representert ved en Byte. Her vises også hvordan treet kan lagres serielt i minnet. Figuren er hentet fra [55]

I figuren 4.2.2 representeres rotnoden med Byte-verdien `00000100`. Her har bit nummer 6 verdien 1, som indikerer at rotnodens sjette barnenode har sine egne barn. Denne noden får Byte-verdien `01000001`. Dette betyr at denne noden har to barnenoder som har egne barn. Denne verdisettingen forsetter nedover i treet.

4.2.1 Deteksjon av forskjeller i rommet

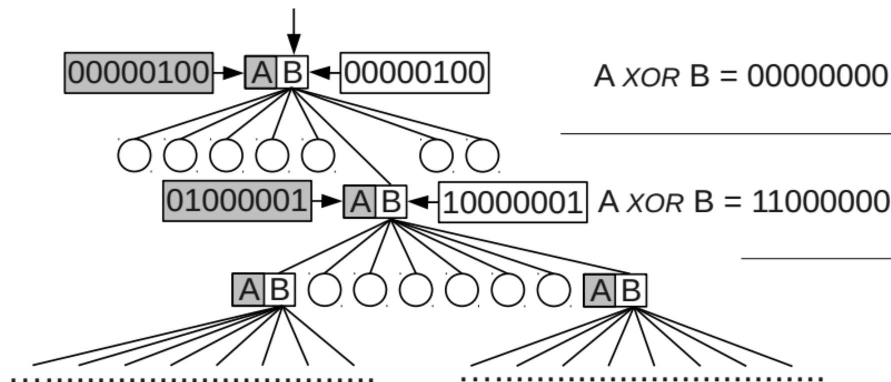
I dette prosjektet brukes octree i deteksjon av forskjeller mellom punktskyer. En funksjon for dette er implementert i pcl-biblioteket. Algoritmen forutsetter at objektene plassert i samme posisjon og med lik rotasjon referert til samme koordinatsystem. Dette kan brukes til å detektere forskjeller mellom to like objekter i to forskjellige punktskyer.

Metoden for deteksjon av forskjeller bruker en dobbel octree-struktur. Denne strukturen åpner for å lagre to octree-strukturer i minnet samtidig. Strukturen lagres parallelt ved å matche de ulike elementene som har samme posisjon i rommet. Dette er vist i figur 4.2.3



Figur 4.2.3: Figuren viser hvordan to octree kan lagres i minnet samtidig. Octree-strukturene A og B, lagres parallelt. Til venstre vises opprettelsen av barn til barnenode nummer seks i B-strukturen. Barnenode nummer seks er allerede aktiv i A-strukturen. De korresponderende nodene i A- og B-strukturen kan dermed lagres parallelt som vist til høyre. Figuren er hentet fra [55]

Forskjeller i rommet detekteres ved å bruke XOR-operatoren i den doble octree-strukturen. Et eksempel på dette er vist i 4.2.4. I eksempelet i figuren blir det detektert forskjeller i en av barnenodene. Her har barnenoden i A-strukturen har verdien `01000001` samtidig som den korresponderende noden i B-strukturen har verdien `10000001`. Her er det altså forskjeller i barn 1 og 2 mellom de to strukturene. Dette detekteres ved å bruke XOR-operatoren mellom A og B. Resultatet blir `11000000`, der forskjeller indikeres i barn nummer 1 og barn nummer 2, ved verdien 1. Oppløsningen på deteksjonsalgoritmen gis ved oppløsningen på det nederste nivået på octree-strukturen som blir brukt. Dette er sidelengden på de minste kubene, vist i figur 4.2.1. Dette setter grensen på hvor små forskjeller som kan detekteres med denne algoritmen.



Figur 4.2.4: Her vises hvordan romlige forskjeller kan detekteres ved hjelp av XOR-operasjoner mellom to octree-strukturene A og B. Figuren er hentet fra [55]

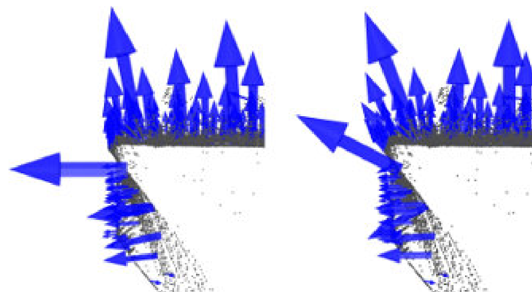
4.3 Punktnormaler

Punktene i punktskyer inneholder hver for seg lite informasjon om området de befinner seg i. Den eneste informasjonen som er beskrevet, er posisjonen, referert til koordinatsystemet. For å få en bedre beskrivelse av et område som et punkt befinner seg i kan det regnes ut punktnormal for punktet.

I punktskyer kan normalretningen i et punkt, $\boldsymbol{p_i}$, som er del av en overflate, finnes basert på informasjon om området rundt punktet. Punktet i seg selv består av et xyz-koordinat og gir lite informasjon om overflaten det er en del av. Normalretningen estimeres ved å bruke kovariansmatrisen til de k næremeste naboene til punktet.

$$\boldsymbol{\zeta} = \frac{1}{k}\sum_{i=0}^{k}(\boldsymbol{p_i} - \overline{\boldsymbol{p}})(\boldsymbol{p_i} - \overline{\boldsymbol{p}})^T \tag{4.3.1}$$

Punktet $\overline{\boldsymbol{p}}$ tilsvarer senterpunktet i området rundt $\boldsymbol{p_i}$ og k er antallet nabopunkter som er brukt. Ved egenverdidekomposisjon av kovariansmatrisen $\boldsymbol{\zeta}$ er normalretningen gitt av egenvektoren tilsvarende den minste egenverdien. Størrelsen på området rundt punktet $\boldsymbol{p_i}$ gis av de k nærmeste naboene til $\boldsymbol{p_i}$. Noen ekstra operasjoner må gjøres for å bestemme den positive retningen til normalen ettersom det ved denne beregningen er mer eller mindre tilfeldig. Dette gjøres i noen tilfeller ved å sette normalretningene slik at de er konsistente over hele punktskyen. Settes k høyt vil normal estimatet estimeres over et stort område. Normal estimatet kan da bli unøyaktig siden bruk av et stort område vil ha en glattende effekt. Settes k lavt, brukes et mindre område og estimatet kan da lett påvirkes av små støykomponenter i punktskyen. Figur 4.3.1 illustrerer to tilfeller med ulik k-verdi.



Figur 4.3.1: Her vises to tilfeller med ulik k-verdi. I bildet til venstre vises resultatet av en god k-verdi. Til høyre vises et eksempel på en dårlig k-verdi. Figuren er hentet fra [56]

For å få et best mulig resultat bør k-verdien settes til en verdi som er tilpasset detaljnivået på punktskyen som brukes. [56] [57]

4.4 Objektgjenkjenning i punktskyer

I objektgjenkjenningen gjort i denne oppgaven brukes det to datasett. Det ene er en modell av objektet som skal gjenkjennes og det andre er en scene der objektet befinner seg. Gjenkjenningen foregår gjennom to steg. Det første steget

er å trene opp gjenkjenningalgoritmen ved å bruke modellen som et datasett for trening. Det andre steget er å teste den trente modellen mot scene-punktskyen.

I dette prosjektet brukes to metoder, «Correspondence Grouping» og «Implicit Shape Model», som er litt ulike i tilnærmingen til gjenkjenningen. Felles for begge er at de bruker lokale deskriptorer. Det vil si at de ser på lokale egenskaper i form av små detaljer. Det finnes andre metoder for gjenkjenning der objektenes globale egenskaper blir brukt. Her blir objektene betraktet som en helhet. Her kan det derimot ikke være for mye støy i datasettet siden dett ville påvirke beskrivelsen direkte. [58]

Gjenkjenning med lokale deskriptorer kan fort være tidkrevende dersom alle punkt i datasettet skal beskrives. Derfor er det vanlig å begrense dette antallet ved å finne såkalte nøkkelpunkt. Ofte finnes nøkkelpunkt ved bruk av en detektoralgoritme som plukker ut punkter som har karakteristiske trekk som kan gjøre dem lette å beskrive. En annen metode er å bruke et utvalg av punkter fra det opprinnelige datasettet med lik avstand mellom punktene. Altså en nedsamplet versjon av datasettet.

De lokale egenskapene til punktskyen beskrives ved bruk av deskriptoralgoritmer. Disse algoritmene forsøker å lage punktbeskrivelser for nøkkelpunktene. Deskriptoralgoritmene har som oppgave å lage punktbeskrivelsene så reproduserbare som mulig. Dette må de være siden det er punktbeskrivelsene som sammenlignes for å finne ut om det er likheter mellom to punktskyer.

Måleenheten som brukes i punktskyer er gitt av skalaen til punktskyen som brukes. For eksempel kan en punktsky ha punkter som er spesifisert i centimeter. En x-verdi på 30 i et punkt i denne skyen vil da tilsvare 30 centimeter langs x-aksen. I dette prosjektet brukes punktskyer som kommer fra Kinect. Disse punktskyene er spesifisert i meter. På grunn av dette må også parametre, som definerer avstander, brukt i punktskyalgoritmer spesifiseres i meter.

4.5 Beskrivelse av egenskaper til et 3D-punkt

I en punktsky er alle punktene definert av sine koordinater x,y og z. To punkt i samme posisjon, referert til det samme koordinatsystemet kan stamme fra helt forskjellige kilder. Selv om punktene kan se like ut, er det umulig å avgjøre om punktene faktisk er like. Punktene kan for stamme fra målinger av to helt forskjellige overflater uten at dette framgår av punktenes koordinater. I punktene alene finnes det for lite informasjon for til å kunne sammenligne punktene. [57]

For å kunne sammenligne punkter trengs det en representasjon av punktene som sier mer om punktenes opprinnelse. For å gjøre dette må punktenes lokale nabolag studeres. Altså andre punkter som ligger nært til det opprinnelige punktet.

Ved å gjøre dette kan det hentes ut mer informasjon om punktet satt i sammenheng med opprinnelsen. For å lage disse representasjonene brukes «point feature descriptors». Dette er algoritmer som beskriver punkters egenskaper, videre omtalt som en «deskriptorer». Deskriptorer lager beskrivelser av punktene som inkluderer informasjon fra punktenes nabolag. Resultatet er «feature»-punkter(punkter der egenskapene er beskrevet). Dersom et punkt er godt beskrevet vil det kunne være mulighet til å sammenligne det med andre punkter for å fastslå om punktet kan ha samme opprinnelse. En god punktbeskrivelse er en kompakt representasjon som inneholder mye informasjon om et punkt. Gode punktbeskrivelser bør være mest mulig entydige slik at de ikke kan mistolkes. De bør kunne ha en høy grad av repeterbarhet slik at de kan reproduseres. Dette bør også kunne gjøres uavhengig av ulike datasett og med de samme karakteristikkene selv om en punktsky er blitt utsatt for rigide transformasjoner, ulik samplingstetthet eller støy. [57]

Det finnes i dag et stort utvalg av deskriptorer for 3D-punkter. Noen er opprinnelig beregnet for 2D-data men utvidet til å gjelde for 3D-data. Andre er laget fra grunnen for bruk i 3D-data. En oversikt over deskriptor algoritmer er listet opp i tabell 4.5.1.

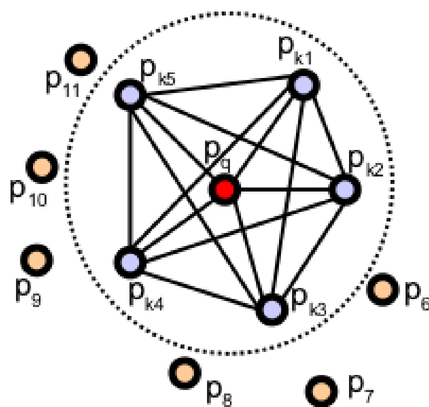
Deskriptor	Kategori	Unikt LKS	Tekstur
Struct. Indexing [Stein92]	Signatur	Nei	Nei
PS [Chua97]	Signatur	Nei	Nei
3DPF [Sun01]	Signatur	Nei	Nei
3DGSS [Novatnack08]	Signatur	Nei	Nei
KPQ [Mian10]	Signatur	Nei	Nei
3D-SURF [Knopp10]	Signatur	Ja	Nei
SI [Johnson99]	Histogram	RA	Nei
LSP [Chen07]	Histogram	RA	Nei
3DSC [Frome04]	Histogram	Nei	Nei
ISS [Zhong09]	Histogram	Nei	Nei
USC [Tombari10]	Histogram	Ja	Nei
PFH [Rusu08]	Histogram	RA	Nei
FPFH [Rusu09]	Histogram	RA	Nei
Tensor [Mian06]	Histogram	Nei	Nei
RSD [Marton11]	Histogram	RA	Nei
HKS [Sun09]	Annet	-	Nei
MeshHoG [Zaharescu09]	Signatur og Histogram	Ja	Ja
SHOT [Tombari10]	Signatur og Histogram	Ja	Ja

Tabell 4.5.1: En oversikt over noen dekriptoralgoritmer. Tabellen viser om deskriptoren er signatur eller histogram-basert, om den tar i bruk unike lokale koordinatsystem(LKS) og om den har mulighet for å ta i bruk teksturdata i beskrivelse av et punkt. (RA = referanse akse) Tabell hentet fra [58].

Det finnes to hovedgrupper av lokale 3D-punkt deskriptorer. De to gruppene signatur-baserte og histogram-baserte metoder. En signatur-basert deskriptor oppretter invariante lokale koordinatsystem(ikke nødvendigvis unike) for 3D-punktene som skal beskrives. En eller flere geometriske målinger blir gjort individuelt på hvert punkt i et utvalg av punkter i området rundt det gitte 3D-punktet. En sigantur-deskriptor gir en sterk beskrivelse for et nabolag rundt et 3D-punkt men er svært utsatt for små feil som kan føre til en betydelig endring av beskrivelsen til punktet. Histogram-baserte deskriptorer bruker geometriske eller topologiske mål av det lokale området som samles i histogrammer. Disse metodene gir mer robusthet men mindre grad av styrke i beskrivelsene av punkter enn signatur-baserte deskriptorer. [59] [60]

4.5.1 Point Feature Histogram(PFH)

Point feature histogram(PFH) [57] er en deskriptor-algoritme for beskrivelse av 3D-punkter. Algoritmen baserer seg på informasjon om punktnormaler i et område rundt nøkkelpunktet $\boldsymbol{p_q}$. Området blir definert av en kule med radius r. Innenfor dette området kobles hvert punkt til alle andre punkt i området som vist i figur 4.5.1.



Figur 4.5.1: Figuren viser hvordan alle punktene i området rundt nøkkelpunktet p_q kobles sammen. Figuren er hentet fra [61]

Forskjellene i normalretning regnes ut for hvert punkt-par innenfor området. For et punkt-par $\boldsymbol{p_s} \leftrightarrow \boldsymbol{p_t}$, defineres først normalretningen i begge punktene som beskrevet i kapittel 4.3. Normalretningene er da definert av vektorene $\boldsymbol{n_s}$ og $\boldsymbol{n_t}$ Deretter konstrueres det et referanse-koordinatsystem, definert av enhetsvektorene u(normalretningen), v og w, i det ene punktet. I dette tilfellet opprettes det i punktet $\boldsymbol{p_s}$. Enhetsvektorene u, v og w defineres ved:

$$u = n_s \tag{4.5.1}$$

$$v = u \times \frac{\boldsymbol{p_t} - \boldsymbol{p_s}}{\|\boldsymbol{p_t} - \boldsymbol{p_s}\|_2} \tag{4.5.2}$$

$$w = u \times v \tag{4.5.3}$$

Avstanden, d, mellom de to punktene regnes ut som den euklidske avstanden, $\|\boldsymbol{p_t} - \boldsymbol{p_q}\|_2$. Deretter blir vinkelen ϕ mellom normalretningen i punktet $\boldsymbol{p_s}$ og

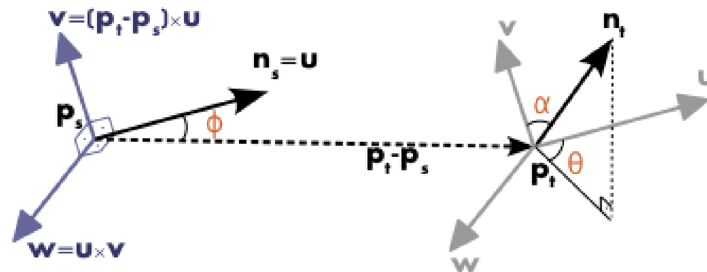
retningen til punktet $\boldsymbol{p_t}$ regnet ut. Rotasjonen mellom normalen n_t og referanse-koordinatsystemet blir gitt ved vinklene α og ϕ, vist i figur 4.5.2. Utregningene er gitt som følgende:

$$\alpha = v \cdot n_t \tag{4.5.4}$$

$$\phi = u \cdot \frac{\boldsymbol{p_t} - \boldsymbol{p_q}}{d} \tag{4.5.5}$$

$$\theta = arctan(w \cdot n_t, u \cdot n_t) \tag{4.5.6}$$

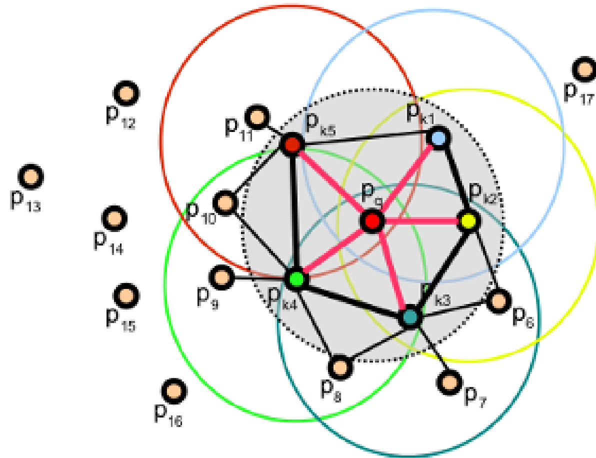
De fire verdiene d, α, ϕ og θ regnes ut for alle punkt-parene i området rundt $\boldsymbol{p_q}$ og samles i et histogram for hvert nøkkelpunkt. Disse histogrammene beskriver egenskapene til nøkkelpunktene. En stor ulempe med denne PFH algoritmen er kompleksiteten. Siden hvert punkt i området rundt $\boldsymbol{p_q}$ kobles til alle andre punkt der de samme regneoperasjonene utføres vil et område med k punkter gi $O(k^2)$ for et nøkkelpunkt. Altså $O(nk^2)$ for n nøkklåunkter. [61] [57]



Figur 4.5.2: Avstanden, d, mellom punktene og vinklene α, ϕ og θ regnes ut i forhold til referanse-koordinatsystemet i $\boldsymbol{p_s}$. Figuren er hentet fra [61]

4.5.2 Fast Point Feature Histogram(FPFH)

FPFH (Fast Point Feature Histogram) utledet fra PFH med ønske om å redusere kompleksiteten. For punktet $\boldsymbol{p_q}$ brukes en forenkling av algoritmen PFH kalt SPFH (Simplified Point Feature Histogram).



Figur 4.5.3: Figuren viser hvordan en SPFH blir utført i punktet $\boldsymbol{p_q}$ og hvordan en ny SPFH blir utført for hvert punkt, $\boldsymbol{p_k}$. Figuren er hentet fra [62]

SPFH regner ut vinklene α, ϕ og θ som i PFH, men for et begrenset antall punkt-par. Hver nabo, $\boldsymbol{p_k}$ i området rundt $\boldsymbol{p_q}$ kobles kun til punktet $\boldsymbol{p_q}$ og ingen andre naboer (Figur 4.5.3). Hvert punkt-par $\boldsymbol{p_q} \leftrightarrow \boldsymbol{p_k}$ blir vektet med en ny SPFH utført i punktet $\boldsymbol{p_k}$ der også avstanden ω_k blir brukt i vektingen. ω_k representerer avstanden fra punktet $\boldsymbol{p_q}$ til hver nabo som brukes i SPFH i et punkt $\boldsymbol{p_k}$. Denne prosessen kan vises ved ligningen:

$$FPFH(\boldsymbol{p_q}) = SPFH(\boldsymbol{p_q}) + \frac{1}{k}\sum_{i=1}^{k}\frac{1}{\omega_k} \cdot SPFH(\boldsymbol{p_k}) \qquad (4.5.7)$$

På denne måten reduseres kompleksiteten på algoritmen fra $O(nk^2)$ til $O(nk)$ for n nøkkelpunkter og k naboer. [62] [57]

4.5.3 Signature of Histograms of Orientations(SHOT)

I [60] foreslås det en algoritme for beskrivelse av 3D-punkter kalt ≪Signature of Histogram OrienTations≫ (SHOT). Dette er en algoritme som kombinerer egenskapene til signatur-baserte og histogram-baserte deskriptorer og kalles derfor en hybrid-deskriptor. Deskriptoren er inspirert av SIFT[1]-deskriptoren for 2D-data. SIFT-descriptoren plasserer et rutenett over et område rundt punktet som skal beskrives. Hver rute deles inn i mindre rutenett. For hver rute i de små rutenettene blir gradientinformasjonen estimert i midten av ruten. Disse estimatene er et vektlagt gjennomsnitt av nærliggende gradienter. Vektene blir valgt slik at gradientene utenfor ruten bidrar til resultatet. Gradientestimatene i hver

[1] Scale Invariant Feature Transform

rute blir samlet opp i histogrammer av retninger der hvert gradient-estimat bidrar med sin retning. De resulterende histogrammene blir satt sammen til en egenskapsvektor. Denne blir normalisert slik at summen av elementene blir 1 og verdier under en satt grense blir fjernet. Deretter normaliseres vektoren igjen. [25, s.187]

Unike lokale koordinatsystem

I 3D-punkt deskriptorer blir det ofte brukt lokale koordinatsystem i nøkkelpunktene som beskrives. Sammen med SHOT-deskriptoren foreslås det en kombinasjon av egenvektor dekomposisjon sammen med tiltak for å fjerne tvetydighet for å fastslå de lokale koordinatsystemene. Denne er basert på en metode brukt i [63] og [64] som ligner på prosessen som blir gjennomgått i kapittel 4.3. Her ble det brukt ≪total least squares≫(TLS) estimering for normalretningen ved egenverdi dekomposisjon av kovariansmatrisen $\boldsymbol{M}$:

$$\boldsymbol{M} = \frac{1}{k}\sum_{i=0}^{k}(\boldsymbol{p_i} - \hat{\boldsymbol{p}})(\boldsymbol{p_i} - \hat{\boldsymbol{p}})^T, \quad \hat{\boldsymbol{p}} = \frac{1}{k}\sum_{i=0}^{k} p_i \tag{4.5.8}$$

$\boldsymbol{M}$ regnes ut fra de k nærmeste naboer, $\boldsymbol{p_i}$, til punktet som skal beskrives. Her er $\hat{\boldsymbol{p}}$ senterpunktet i området av naboer. TLS-estimatet av normalretningen er dermed gitt ved egenvektoren som korresponderer til den minste egenverdien til matrisen $\boldsymbol{M}$. Normalenes fortegn blir satt ut fra ønske om konsekvent normalretning i datasettet. Denne metoden tar derimot bare hensyn til fortegnene i en akse(normalretningen) og er likegyldig til det to resterende aksene i et koordinatsystem. I tillegg til dette blir normalretningen gitt ut fra et globalt perspektiv. I scener med flere objekter kan ikke et globalt perspektiv brukes og det er ønskelig med en metode som bestemmer fortegnene lokalt for hvert punkt. I [60] foreslås det en forandring på matrisen $\boldsymbol{M}$ fra 4.5.8, der alle punkter innenfor en radius, R, brukes i utregningen. I tillegg byttes $\hat{\boldsymbol{p}}$ ut med nøkkelpunktet, $\boldsymbol{p}$, for å spare regnekraft. Matrisen $\boldsymbol{M}$ regnes dermed ut med lineær vekting:

$$\boldsymbol{M} = \frac{1}{\sum_{i:d_i \leq R}(R - d_i)} \sum_{i:d_i \leq R}(R - d_i)(\boldsymbol{p_i} - \boldsymbol{p})(\boldsymbol{p_i} - \boldsymbol{p})^T \tag{4.5.9}$$

der d_i er den euklidske avstanden mellom $\boldsymbol{p_i}$ og $\boldsymbol{p}$, $\|\boldsymbol{p_i} - \boldsymbol{p}\|_2$. Egenvektorene til $\boldsymbol{M}$ er ortogonale vektorer som har stor grad av repeterbarhet i datasett med støy. Disse brukes som et utgangspunkt til akser i koordinatsystemet. For å øke repeterbarheten til koordinatsystemet må det gjøres noen steg for å bestemme aksenes retning. Egenvektorene noteres med synkende egenverdi som $\boldsymbol{x}^+$, $\boldsymbol{y}^+$ og $\boldsymbol{z}^+$ som representasjon for aksene i koordinatsystemet. Her er $\boldsymbol{x}^-$, $\boldsymbol{y}^-$ og

$\boldsymbol{z}^-$ de samme vektorene med motsatt retning. For å bestemme retningen til x brukes:

$$S_x^+ \doteq \{i : d_i \leq R \wedge (\boldsymbol{p_i} - \boldsymbol{p}) \cdot \boldsymbol{x}^+ \geq 0\} \tag{4.5.10}$$

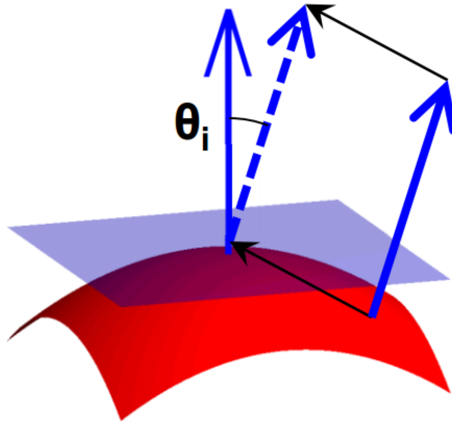
$$S_x^- \doteq \{i : d_i \leq R \wedge (\boldsymbol{p_i} - \boldsymbol{p}) \cdot \boldsymbol{x}^- > 0\} \tag{4.5.11}$$

$$\boldsymbol{x} = \begin{cases} \boldsymbol{x}^+ & , |S_x^+| > |S_x^-| \\ \boldsymbol{x}^- & , |S_x^+| < |S_x^-| \end{cases} \tag{4.5.12}$$

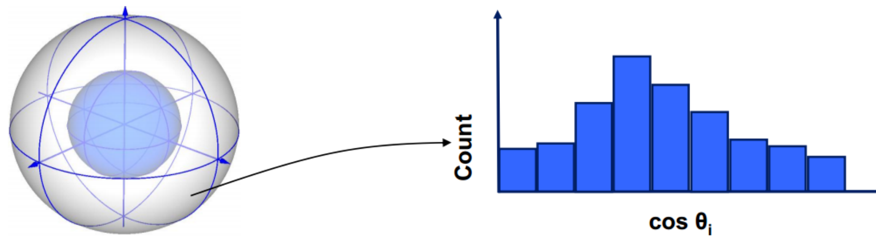
Retningen til $\boldsymbol{x}$-aksen blir altså gitt av 4.5.12. Samme metode blir brukt for å fastslå retning til $\boldsymbol{z}$-aksen. Deretter defineres y-aksens retning ved kryssproduktet $\boldsymbol{z} \times \boldsymbol{x}$. Noen ekstra steg gjøres for å behandle spesialtilfellet $|S_x^+| = |S_x^-|$. [59] [60] [65]

SHOT-deskriptoren

SHOT-deskriptoren baserer seg på normalene til punktene i det lokale koordinatsystemet, som defineres i forrige avsnitt. Området som er definert av det lokale koordinatssystemet avgrenses av en kule som deles opp i 32 seksjoner. Dette gjøres gjennom 8 asimut-divisjoner, 2 elevasjon-divisjoner og, 2 radiale divisjoner. Kule-strukturen vises i figur 4.5.5. For hvert punkt i kulen sjekkes vinkelen mellom normalretning i dette punktet og normalen til punktet p (Figur 4.5.4). Denne vinkelen kalles θ_i. Verdier for $cos(\theta_i)$ regnes ut for hver vinkel og resultatet samles opp i histogrammer for hver seksjon i kulen.



Figur 4.5.4: Vinkelen θ_i er vinkelen mellom normalen i nøkkelpunktet $\boldsymbol{p}$ og normalen til hvert punkt på kulen. Figuren er hentet fra [58].



Figur 4.5.5: For hvert punkt i hver seksjon av kulen regnes ut en verdi for $cos(\theta_i)$. Alle verdiene av $cos(\theta_i)$ samles i lokale histogrammer for hver seksjon. (I figuren er har kulen bare 16 seksjoner for enkelhets skyld.) Figuren er hentet fra [58].

For hvert punkt som samles i et lokalt histogram for en av kuleseksjonene utføres en interpolasjon med histogram-klassens naboklaser og de tilsvarende klassene i de andre lokale histogrammene. Dette gjøres for å unngå grenseeffekter mellom seksjonene. Hver telte verdi blir også vektet ut fra sin avstand til senterpunktet i kulen. Deretter normaliseres hele deskriptoren til å kunne summeres til en. Antallet klasser i histogrammene settes til å være 11 og den totale lengden på den resulterende egenskapsvektoren blir 352. [59] [60]

Det finnes også en utvidelse av SHOT-deskriptoren(SHOTb) som kan beskrive punktskyer med teksturinformasjon. Denne tar fargen til punktene med i beregningen for å beskrive et nøkkelpunkt. [59] I denne oppgaven brukes kun punktskyer uten teksturinformasjon. Derfor brukes den originale SHOT-deskriptoren.

4.6 Hough-transform

I både ≪Correspondence Grouping≫ og ≪Implicit Shape Model≫ brukes en tredimensjonal versjon av Hough-transformen til å finne best mulige resultater for gjenkjenning i punktskyer. I dette kapittelet beskrives Hough-transformen for rette linjer og hvordan denne er blitt utvidet til å gjelde for generaliserte former. Deretter beskrives en utvidelse av Hough-transformen til tre dimensjoner.

Hough-transform (HT) er en populær metode brukt for deteksjon av rette linjer i bildebehandling. Hough-transformen ble først foreslått i en patentsøknad av Hough i 1962. Senere implementasjoner har utvidet bruken av HT til å kunne detektere både sirkler, ellipser, hjørner og andre geometriske former. Også en generalisert versjon, kalt generalisert Hough-transform (GHT), finnes der metoden blir brukt til å detektere vilkårlige figurer. [66]

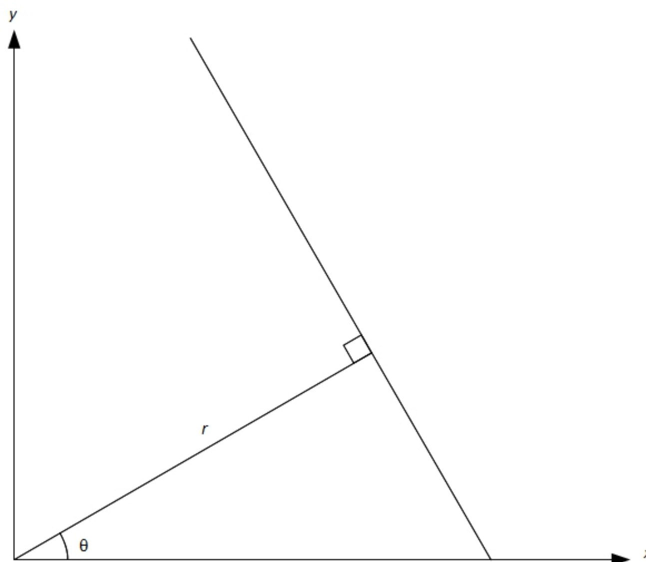
4.6.1 Hough-transform for rette linjer

Hough-transformen forutsetter at det på forhånd er gjort en form for kantdeteksjon i bildet. I den første implementasjonen av HT for deteksjon av rette linjer ble linjene parameterisert ved ligningen:

$$y = mx + c \tag{4.6.1}$$

Hvert punkt langs en kant plottes som en linje gjennom punktet for alle mulige verdier av parametrene (m, c). For hver linje av parameterne (m, c) vil andre kantpunkter langs linjen avgi en stemme. Verdier av (m, c) som genererer flest stemmer indikerer en rett linje i bildet. Denne formen var derimot vanskelig brukt i datamaskiner på grunn av at parameterne, (m, c), tilnærmes uendelig for vertikale linjer.

En annen tilnærming til problemet ble foreslått av Duda og Hart i 1972. Denne metoden baserte seg normalparameterisering av rette linjer ved parameterne (θ, r). Med denne metoden kna vertikale linjer også representeres, uten at parametrene tilnærmes uendelig. Her brukes en rett strek fra origo som står normalt på den rette linjen som en representasjon. θ er normalens vinkel fra x-asken og r beskriver radiusen fra origo og ut til den perpendikulære linjen. I figur 4.6.1 vises normalparameterisering til en rett linje. [66]

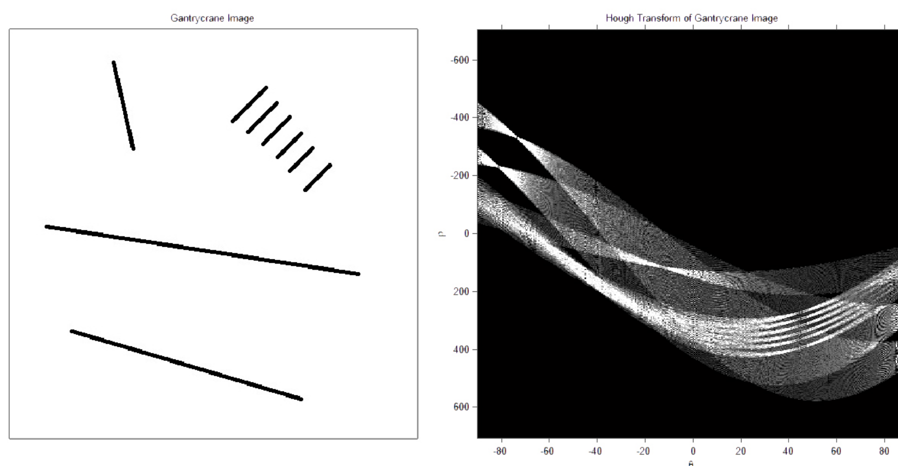


Figur 4.6.1: Normalenparameterisering med parametrene (θ, r) av en rett linje. [66, s.267]

Hvert par av (θ, r) utgjør en unik linje. Med ligningen 4.6.2 for ett sett med punkter, (x, y) fra et bilde, kan disse plottes i en akkumulatormatrise.

$$xcos(\theta) + ysin(\theta) + r = 0 \tag{4.6.2}$$

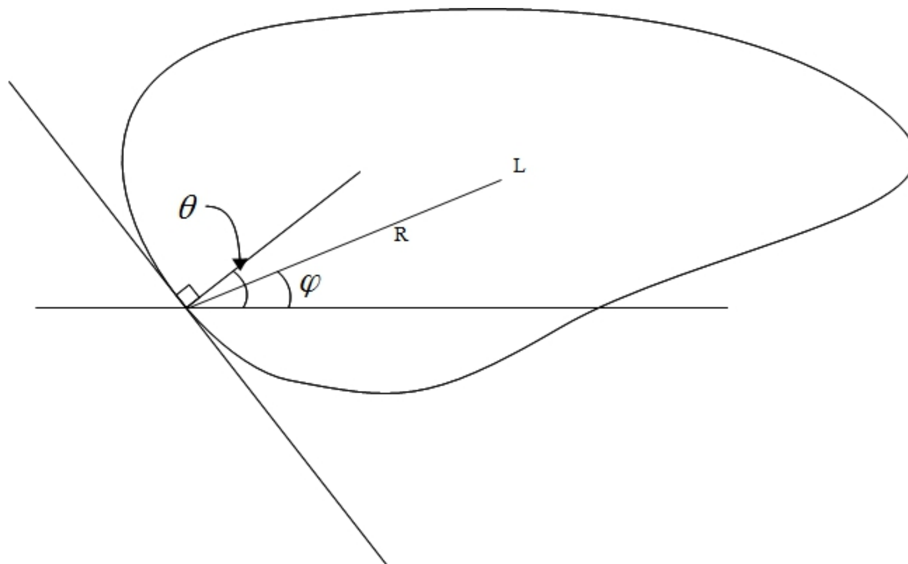
Her plottes kurvene i samme koordinatsystem for alle verdier av (θ, r). Kurvene i akkumulatormatrisen kan ses på som stemmer fra de ulike punktene i bildet(x, y). Flere kurver som krysses indikerer at det er samsvarende stemmer fra flere punkter(x, y) for verdi av θ og r. Altså indikeres det at det er en linje i bildet gjennom disse punktene. [25, s.320] I figurene 4.6.2 vises et eksempel på deteksjon av rette linjer ved bruk av akkumulatormatrise.



Figur 4.6.2: Til venstre vises punktene i bildet og til høyre vises akkumulatormatrisen. Kurvene krysses i akkumulatormatrisen for de verdiene av (θ, r) som gir normaler til linjene i bildet. Figuren er laget med kode fra [67]

4.6.2 Generalisert Hough-transform

Den generaliserte Hough-transformen(GHT) ble til gjennom arbeid gjort i 1975 av Merlin og Farber og Ballard i 1981. I teorien kan den generaliserte Hough-transformen brukes til å detektere figurer med tilfeldig form i bilder. I bilder kan GHT brukes dersom en har informasjon om gradientretningene til kantene i bildet. Dette kan f.eks finnes ved hjelp av en Sobel-operator [68]. Det første som gjøres er å finne et referansepunkt L i en mal av figuren som skal detekteres. For hvert punkt langs kanten til figuren, hentes gradientinformasjonen og tilhørende vinkel θ. Deretter dras en linje R fra punktet til referansepunktet L. Lengden på R og den tilhørende vinkelen φ er begge variable med hensyn på θ. $R(\theta)$ og $\varphi(\theta)$ lagres for hvert punkt langs kanten av malen.



Figur 4.6.3: Utregning av den generaliserte Hough-transformen for en tilfeldig figur [66, s.316].

Ved hjelp av tabellene $R(\theta)$ og $\varphi(\theta)$ kan referansepunktet L nås for ulike verdier av θ. For tilfeldige figurer kan $R(\theta)$ inneholde flere verdier for en verdi av θ på grunn av kanter som har samme retning på gradientene. I disse tilfellene vil bare en av verdiene føre til referansepunktet L. Referansepunktet kan nås gjennom ligningene 4.6.3 og 4.6.3. [66, s.316] [69]

$$L_x = x + R(\theta)cos(\varphi(\theta)) \tag{4.6.3}$$

$$L_y = x + R(\theta)cos(\varphi(\theta)) \tag{4.6.4}$$

Dette gir et parameterrom på to dimensjoner bestående av $R(\theta)$ og $\varphi(\theta)$. For hvert kantpunkt i bildet avgis det en stemme for hvert plan i parameterrommet i en posisjon gitt av det forventede objektet. Etter på kan det søkes etter toppunkter i akkumulatormatrisen som indikerer tilfeller av objektet. Dersom orientasjonen til objektet er ukjent må det gjøres ekstra beregninger for dette som betyr at det trengs enda en dimensjon i parameterrommet. Det samme gjelder dersom størrelsen på objektet er ukjent. Problemet er da firedimensjonalt og vil kreve stor regnekraft for å løse. For å spare regnekraft blir parameterrommet ofte kvantifisert i små grupper av nærliggende punkter. Større grupper vil minske antallet regneoperasjoner, men også føre til dårligere nøyaktighet. [66, s.316-317]

Generalisert Hough-transform i tredimensjonale data

I [70] blir det gjort forsøk på å utvide den generaliserte hough-transformen til å kunne brukes på tredimensjonale data. Her blir gradientretningen fra GHT byttet ut med punktnormaler i 3D-data. Siden en normalretning kan beskrives med to vinkler φ og ψ brukes den todimensjonale listen $R(\varphi, \psi)$ til å lagre lengden på vektorene fra hvert punkt til referansepunktet. To ligninger brukes for å finne hver orienteringen til hver vektor. Denne metoden er derimot krevende regnemessing og den oppgis å et antall regneoperasjoner på $O(M)$ uten å ta hensyn til rotasjon og translasjon. M er her antallet punkter i punktskyen. Dersom transformasjon blir tatt med i beregningen, oppgis det et antall regneopperasjoner på $O(M \cdot N^4)$ der N er størrelsen på intervallene i hver akse i akkumulatormatrisen. [70]

4.7 Iterative Closest Point(ICP)

Dersom to datasett, med tredimensjonale punkter, der det ene datasettet inneholder punkter som er en rigid transformasjon av punktene i det andre datasettet kan disse rettes opp ved hjelp av ≪registrering≫. En algoritme for registrering kalt ≪Iterative Closest Point≫ ble foreslått i 1992 av Besl og McKay. Denne algoritmen forsøker å iterativt minimere feilen mellom datasettene.

I algoritme 1, beskrives algoritmen med pseudokode. Algoritmen tar inn to datasett og returnerer den estimerte transformasjonen mellom disse. De to datasettene kalles Modell og Scene. Det opprettes punkt-par mellom de punktene som er nærmest hverandre i de to datasettene. Algoritmen forsøker å minimere E, som er feilen mellom punkt-parene.

Input: Modell, Scene
Output: $(\mathcal{R}, t)$
$E' \leftarrow +\infty$
$(\mathcal{R}, t) \leftarrow$ Initialiser-Registrering(Scene, Modell)
repeat
    $E \leftarrow E'$
    Registrert Scene $\leftarrow$ Utfør-Registrering(Scene, Modell)
    Punkt-Par $\leftarrow$ Estimer-Nærmeste-Par(Registrert Scene, Modell)
    $(\mathcal{R}, t, E') \leftarrow$ Oppdater-Registrering(Scene, Modell, Punkt-Par, $\mathcal{R}, t$)
until $|E' - E| < \tau$
return $(\mathcal{R}, t)$

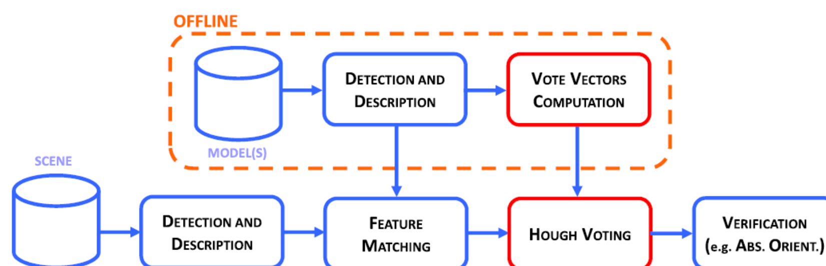
Algoritme 1: Iterative Closest Point [25, s.465]

Det første som gjøres er å initialisere feilen E' til positiv uendelig. Deretter initialiseres rotasjonen, $\mathcal{R}$ og translasjonen, t, ved hjelp av et røft estimat av den rigide transformasjonen mellom modellen og scenen(*Initialiser-Registrering*). Deretter

startes løkken der det første som skjer er at feilen E' kopieres til E som representerer den forrige feilen. Metoden *Utfør-Registrering* lager en kopi av scenen, ≪Registrert Scene≫, og transformerer denne med $\mathcal{R}$ og t. Deretter brukes metoden *Estimer-Nærmeste-Par* for å finne de parene av punkter som er nærmest hverandre mellom ≪Registrert Scene≫ og modellen. Til slutt i løkken kjøres metoden *Oppdater-Registrering*. Denne metoden oppdaterer den totale transformasjon mellom modellen og scenen og estimerer den nye feilen E'. Løkken stoppes når feilen har konvergert innen en feilmargin gitt ved τ. Algoritmen returnerer den totale rigide transformasjonen mellom scenen og modellen. I noen tilfeller vil feilen konvergere uten at riktig transformasjon er oppnådd. I noen tilfeller skjer det at feilen ikke konvergerer innenfor den oppgitte feilmarginen. Det er da vanlig å gi opp etter ett gitt antall iterasjoner. Det er heller ikke sikkert at algoritmen gir det riktige estimatet for transformasjon der feilen konvergerer. [25, s.464-465]

4.8 Correspondence Grouping

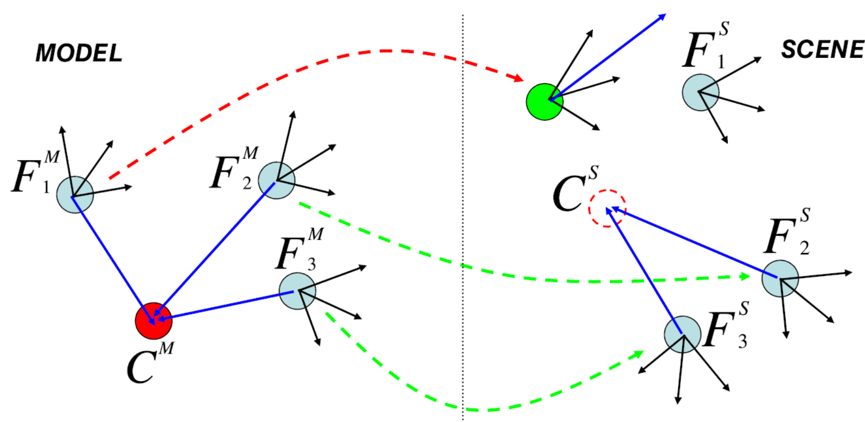
I dette prosjektet brukes det en algoritme for gjenkjenning, kalt ≪Correspondence Grouping≫. En implementering av denne presenteres på PCL-biblioteket sine hjemmesider. [71] Algoritmen er basert på artikkelen ≪Object recognition in 3D scenes with occlusions and clutter by Hough voting≫ [72], skrevet av F. Tombari og L. Di Stefano. I denne artikkelen blir det blir det foreslått en metode for gjenkjenning i tredimensjonale scener ved bruk av Hough-stemming. Her brukes lokale beskrivelser av nøkkelpunkter kombinert med lokale koordinatsystem for å redusere størrelsen på Hough-rommet.



Figur 4.8.1: Prosess for objektgjenkjenning i punktskyer ved bruk av to hovedtrinn, foreslått i [72]. Figur hentet fra [72].

Metoden består av to hovedtrinn, offline og online(Se figur 4.8.1). Med begrepet ≪offline≫ menes i dette tilfellet at dette trinnet kan gjøres på forhånd, og trenger ikke nødvendigvis repeteres for hvert forsøk på gjenkjenning. I dette trinnet skjer initialiseringen av gjenkjenningsalgoritmen. Selve gjenkjenningen foregår i ≪online≫-steget. Med begrepet ≪online≫ menes selve prosessen med gjenkjenning. Her testes den beskrevne modellen mot en scene for å finne forekomster av modellen.

Det første som gjøres er å finne nøkkelpunkter i begge punktskyene. Dette kan gjøres enten ved å velge punkter tilfeldig eller ved å bruke en nøkkelpunktdetektor. Deretter lages en beskrivelse av punktene ved bruk av en deskriptor-algoritme. En oversikt over noen deskriptor-algoritmer er vist i 4.5.1 i kapittel 4.5. Deskriptoralgoritmen resulterer i et sett med punktbeskrivelser. Når punktbeskrivelser er opprettet i både modell og scene, startes ett søk etter korrespondanser. Dette gjøres ved å se på den euklidske avstanden mellom hver punktbeskrivelse i modellen og hver punktbeskrivelse i scenen. Den euklidske avstanden avgrenses med en øvre grense. Avstander som kommer under denne grensen, indikerer en korrespondanse mellom modell og scene.



Figur 4.8.2: Eksempel på 3D Hough-stemming basert på lokale koordinatsystem. Figur hentet fra [72].

Eksempel på korrespondanser mellom punktbeskrivelser vises i figur 4.8.2 som grønne linjer. På grunn av støy eller lignende i datasettene, kan det også oppstå feil i disse korrespondansene. Den røde linjen i figur 4.8.2 er et eksempel på dette. Korrespondansesøket resulterer i et sett av korrespondanser. Disse korrespondansene brukes senere til å fastslå orienteringen til detekterte objekt. I likhet med den generaliserte Hough-transformen settes det, som en del av ≪offline≫-trinnet, et unikt referansepunkt, C^M, i modellen. C^M vises som en rød sirkel i figur 4.8.2. I følge [72] har plasseringen av C^M ingen påvirkning på resultatet. I eksperimentene utført i [72], ble C^M satt til å være senterpunktet i modellen. Etter at et referansepunkt er funnet regnes det ut vektorer fra hvert nøkkelpunkt, F_i^M, til referansepunktet C^M. Disse vektorene vises som blå linjer i figur 4.8.2. Nøkkelpunktene er nå avhengige av det globale koordinatsystemet til modellen. Dette medfører problemer med rotasjon og translasjon når nøkkelpunktene introduseres til scenens koordinatsystem som kan være helt forskjellig fra modellens.

Siden det ønskes at nøkkelpunktene og vektorene skal være invariante mot rotasjon og translasjon, introduseres lokale koordinatsystem. For hvert nøkkelpunkt, F_i^M, lages det et koordinatsystem lokalt for punktet. I [72] brukes en entydig løsning der det utføres en egenverdidekomposisjon av korrelasjonsmatrisen til

et avstands-vektlagt område rundt hvert nøkkelpunkt. Etterpå gjøres koordinatsystemet unikt ved at det blir utført en prosess for å fjerne tvetydighet i koordinatsystemets fortegn.

Gitt at alle punktene i modellen er referert til det samme globale koordinatsystemet, kan prosessen med å regne ut vektorene i modellen foregå som følgende. Vektorer, $V_{i,G}^M$, regnes ut mellom referansepunktet og hvert nøkkelpunkt F_i^M ved ligningen:

$$V_{i,G}^M = C^M - F_i^M \tag{4.8.1}$$

Hver vektor, $V_{i,G}^M$, transformeres dermed til sitt korresponderende lokale koordinatsystem assosiert med det tilhørende nøkkelpunktet F_i^M (Figur 4.8.3). Dette gjøres ved bruk av matriseproduktet mellom hver vektor og matrisen R_{GL}^M:

$$V_{i,L}^M = R_{GL}^M \cdot V_{i,G}^M \tag{4.8.2}$$

Her er R_{GL}^M en rotasjonsmatrise der hver rad representerer en enhetsvektor for det lokale koordinatsystemet til hvert nøkkelpunkt F_i^M:

$$R_{GL}^M = [L_{i,x}^M L_{i,y}^M L_{i,z}^M]^T \tag{4.8.3}$$

Til slutt i «offline»-trinnet assosieres de roterte vektorene $V_{i,L}^M$ til sine respektive nøkkelpunkter F_i^M.

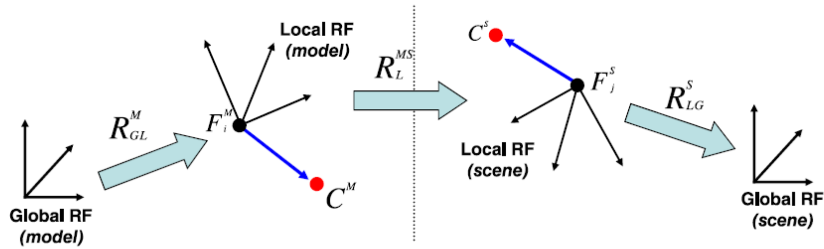
Når «online»-trinnet startes finnes først nøkkelpunktene i scenen, F_j^S. Deretter finnes korrespondansene mellom punktbeskrivelsene i disse til punktbeskrivelsene for nøkkelpunktene i modellen, som er beskrevet tidligere. Når dette er gjort startes stemmeprosessen. Hvert nøkkelpunkt i scenen der det finnes korrespondanse med modellen, $F_j^S \leftrightarrow F_i^M$, gir en stemme til et mulig referansepunkt i scenen, C^S(figur 4.8.2). Denne stemmen baseres på informasjonen i vektorene $V_{i,L}^M$, som gir retningen til referansepunktet fra hvert nøkkelpunkt i modellen. Fra disse stemmene lages det vektorer som beskriver retningen og avstanden til det tenkte referansepunktet C^S. For nøkkelpunktene i scenen, F_j^S, brukes også lokale koordinatsystem. Dette gjør at også disse nøkkelpunktene er invariante mot rotasjon og translasjon. På grunn av dette kan informasjonen i vektorene $V_{i,L}^M$ og $V_{i,L}^S$ brukes til å finne rotasjonsmatrisen R_L^{MS}(figur 4.8.3) som gir $V_{i,L}^M = V_{i,L}^S$. Denne matrisen beskriver nå rotasjonen fra vektorene referert til de lokale koordinatsystemene i modellen til de lokale koordinatsystemene i scenen. Til slutt transformers vektorene, $V_{i,L}^S$, til det globale koordinatsystemet i scenen

ved ligningen:

$$V_{i,G}^S = R_{LG}^S \cdot V_{i,L}^S + F_j^S \tag{4.8.4}$$

Matrisen R_{LG}^S består av enhetsvektorene som oppstår fra utregningen av lokale koordinatsystem for hvert nøkkelpunkt, F_j^S, i scenen:

$$R_{LG}^S = [L_{i,x}^S L_{i,y}^S L_{i,z}^S] \tag{4.8.5}$$



Figur 4.8.3: Transformasjoner mellom det globale og de lokale koordinatsystemene. Figur hentet fra [72].

På denne måten oppstår ikke problemet med det store Hough-rommet som i 3D-utvidelsen av den generaliserte Hough-transformen(GHT) [70] omtalt i 4.6.2. I stedet for å bruke egne parameterrom og hough-stemming for å finne objektets orientering fra modell til scene, brukes matrisene R_{GL}^M, R_L^{MS} og R_{LG}^S til å rotere vektorene fra $V_{i,G}^M$ til $V_{i,G}^S$ som vist i figur 4.8.3. Stemmene fra nøkkelpunktene F_j^S kan da gis med hensyn til vektorene $V_{i,G}^S$, for mulige referansepunkt C^S. Disse stemmene akkumuleres for å finne beviser for tilfeller av objektet som det søkes etter. Forekomster av objektet finnes ved å søke etter topper i de akkumulerte stemmene. Dersom det kun søkes etter et objekt brukes den toppen med høyest verdi.

Siden Hough-rommet ofte kvantifiseres i bokser for å minske kompleksiteten, foreslås det i [72] at stemmeprosessen gjøres mer robust ved at verdien i hver boks Hough-rommet adderes med verdien til sine seks nærmeste naboer. Dette gjøres på grunn av at korrespondansene mellom nøkkelpunktene i scenen og modellen ofte vil falle i feil boks i Hough-rommet.

4.9 Implicit Shape Model

I dette prosjektet brukes algoritmen kalt ≪Implicit Shape Model≫ for gjenkjenning i punktskyer. Denne er foreslått i [73] og en implementering av algoritmen er demonstrert i [74]. Denne algoritmen bruker en kombinasjon av lokale punktbeskrivelser og generalisert Hough-transform. Algoritmen er beregnet for å kunne trenes opp med flere forskjellige objektklasser og deretter fungere som en klassifiserer. Det kan også brukes flere datasett for opptrening av samme klasse. Algoritmen trener en modell av hvert objekt, som senere brukes for detektering av senterpunkter til mulige forekomster av samme objekt i andre punktskyer.

Algoritmen må først trenes opp. Her brukes et eller flere datasett som inneholder objekter som skal gjenkjennes senere. Hvert objekt som brukes i treningen har en egen objektklasse, c. Det første som gjøres er å finne nøkkelpunkter, enten ved å bruke en algoritme for detektering av nøkkelpunkter eller ved å gjøre en uniform sampling fra datasettet. Deretter opprettes punktbeskrivelser ved hjelp av en deskriptor-algoritme. I [73] beskrives en utvidelse av 2D-deskriptoren, SURF[2] [75] , til 3D, kalt 3D-SURF (tabell 4.5.1). Denne foreslås som en aktuell deskriptor for beskrivelse av nøkkelpunktene. 3D-SURF er ikke implementert i PCL-biblioteket. For implementasjonen som beskrives i kapittel 5.3, brukes FPFH-deskriptoren, som er beskrevet i 4.5.2. Etter at nøkkelpunktene er beskrevet med en deskriptor-algoritme, grupperes de ved å bruke ≪k-means≫-algoritmen. Denne algoritmen forsøker å gruppere punkter i et datasett, slik at den kvadrerte summen av punktene i hver gruppe minimeres. [76] De resulterende gruppene blir kalt for ≪visuelle ord≫. Antallet ≪visuelle ord≫ defineres til å være 10% av antallet nøkkelpunkter. Hvert nøkkelpunkt har nå en tilhørighet til et ≪visuelt ord≫.

For å bygge modellen bruker algoritmen plasseringen til de ≪visuelle ordene≫, relativt til objektklassens senterpunkt. Denne informasjonen blir samlet og og lagret sammen med hvert av de ≪visuelle ordene≫. For et ≪visuelt ord≫, v, assosieres en liste med stemmer fra hvert nøkkelpunkt i v. Stemmene består av nøkkelpunktets klasse, c, en vektor til senterpunktet i objektet, (x', y', z'), størrelsen til punktbeskrivelsen i nøkkelpunktet, σ', og størrelsen til objektet, σ. Hvert av de ≪visuelle ordene≫ kan dermed gi stemmer for ulike klasser eller flere ganger i samme klasse.

På grunn av at forskjellige klasser kan ha ulikt antall nøkkelpunkter, utføres en statistisk vekting for hvert ≪visuelle ord≫. Vektingen gis av formelen $W_{st}(c_i, v_j)$ som er vist i ligning 4.9.1. Denne vektingen vektlegger alle stemmene som gis fra et ≪visuelt ord≫, v_j til en klasse, c_i. Den statistiske vektingen i 4.9.1 består av variablene $n_{vot}v_j$, $n_{vot}c_i, v_j$, $n_{vw}(c_i)$ og n_{ftr}. Variabelen $n_{vot}(v_j)$ er det totale antallet stemmer fra det aktuelle ≪visuelle ordet≫, v_j. Variabelen $n_{vot}(c_i, v_j)$ er antallet stemmer fra det ≪visuelle ordet≫, v_j, til den aktuelle klassen, c_i. Antallet ≪visuelle ord≫ som gir stemmer til en klasse, c_i, gis av $n_{vw}(c_i)$. Verdien, $n_{ftr}(c_i)$, er antallet nøkkelpunkter som ble brukt til å trene opp klassen c_i. I ligning 4.9.1

[2] Speeded Up Robust Features

er alle klassene inkludert i C.

$$W_{st}(c_i, v_j) = \frac{1}{n_{vw}(c_i)} \cdot \frac{1}{n_{vot}(v_j)} \cdot \frac{\frac{n_{vot}(c_i, v_j)}{n_{ftr}(c_i)}}{\sum_{c_k \in C} \frac{n_{vot}(c_k, v_j)}{n_{ftr}(c_k)}} \tag{4.9.1}$$

I tillegg til den statistiske vektingen for hvert visuelle ord, blir også hvert nøkkelpunkt vektet med en «lært vekt». Denne vektingen gjøres for å normalisere stemmene basert på hvor ofte de treffer den korrekte objektklassens senterpunkt. For å lære vektingen for et nøkkelpunkt, defineres det en stemme, λ_{ij}. Denne defineres som en stemme gitt av et nøkkelpunkt i et «visuelt ord», v_j, til et treningsdatasett av klassen c_i. Stemmen gis i form av en avstand fra nøkkelpunktet i v_j til senterpunktet i treningsdatasettet der det «visuelle ordet» ble funnet. Stemmen, definert som λ_{ij}, gis fra alle nøkkelpunkt i det «visuelle ordet», v_j, til alle treningsdatasett i klassen, c_i. For hver stemme regnes det ut en Gaussisk funksjon av avstanden som er definer λ_{vj}. Denne funksjonen vises i ligning 4.9.2. Her er standardavviket, σ, gitt til å være 10% av størrelsen på treningsdatasettet. I ligning 4.9.2 er A et sett med nøkkelpunkt som assosieres med det «visuelle ordet», v_j for et treningsdatasett for klassen c_i. Funksjonen f defineres som medianverdien og $d_a(\lambda_{ij})$ er den euklidske distansen til stemmen, λ_{ij}.

$$W_{lrn}(\lambda_{ij}) = f\left(\left\{ e^{-\frac{d_a(\lambda_{ij})^2}{\sigma^2}} \;\middle|\; a \in A \right\}\right) \tag{4.9.2}$$

Resultatet av vektingen gjort med ligning 4.9.2 er at nøkkelpunkter som er nærmere det faktiske senterpunktet, vektes høyere. Denne vektingen gjøres for å oppnå korrekt stemming over klasse-senterpunkt uavhengig av treningsdatasett. Den totale vektingen for en stemme, λ_{ij}, er dermed gitt ved ligning 4.9.3.

$$W(\lambda_{ij}) = W_{st}(c_i, v_j) \cdot W_{lrn}(\lambda_{ij}) \tag{4.9.3}$$

Etter at treningen av objektklassene er gjennomført kan gjenkjenningen startes. Et testsett introduseres til algoritmen som vil prøve å finne forekomster av en klasse i testsettet. I testsettet detekteres nøkkelpunkter. Disse beskrives med deskriptor-algoritmen. For hvert nøkkelpunkt blir det gjort et søk etter det nærmeste «visuelle ordet» i de trente modellklassene. For hvert nøkkelpunkt gis det en stemme i Hough-rommet Dette rommet har fem dimensjoner som består av de tre koordinatene som gir senterpunktet til objektet, størrelsen på objektet og objektets klasse. Alle stemmene vektes med W fra ligning 4.9.3. Deteksjon av objekter blir gjort ved å se på toppunkter i Hough-rommet.

I [73] foreslås det to metoder for gruppering av Hough-stemmene. I den ene metoden, diskretiseres Hough-rommet ved å gruppere stemmene i bokser. Hver stemme bidrar til hver boks med basert på den Gaussian-veide distansen til disse. Det detekterte senterpunktet er gitt av den boksen som får høyest poengsum. Dette gjør stemmeprosessen mer robust og det trengs ikke et støyfritt testsett. En annen metode som foreslås er å regne ut objektsenteret i testsettet på forhånd. Dermed reduseres søket til den mest aktuelle klassen, gitt det kjente senterpunktet. Denne metoden vil fungere bra for feilfrie testsett men mindre bra for testsett der objektet er i en scene. [74] [73]

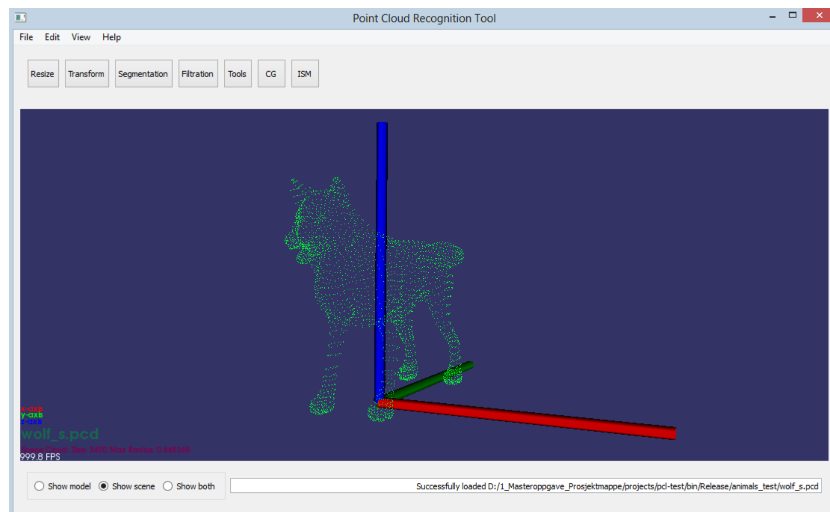
Kapittel 5

Implementering

I dette kapittelet beskrives funksjonaliteten til Point Cloud Recognition Tool og hvordan det er implementert. Deretter beskrives implementeringen av de to algoritmene ≪Correspondence Grouping≫ og ≪Implicit Shape Model≫.

5.1 Point Cloud Recognition Tool

I denne oppgaven ble det utviklet et program kalt ≪Point Cloud Recognition Tool≫(PCRTool). Programmet er programmert i C++ og er laget med et grafisk brukergrensesnitt for å enkelt kunne laste inn og visualisere to punktskyer av filtypen .pcd eller .ply.



Figur 5.1.1: Point Cloud Recognition Tool

Programmet inneholder funksjoner for skalering, transformasjon, filtrering, segmentering og gjenkjenning mellom punktskyer. Motivasjonen for å lage dette programmet var å kunne ha de nevnte funksjonene lett tilgjengelig for eksperimentering og testing av ulike datasett.

5.1.1 Funksjonalitet

Point Cloud Recognition Tool er tenkt til å være et verktøy brukt for gjenkjenning mellom punktskyer. Derfor er det laget for å håndtere to punktskyer samtidig. Programmet laster inn punktskyer fra ply eller pcd-filer. Disse kan visualiseres og det kan velges de skal visualiseres en og en eller om begge skal visualiseres i samme koordinatsystem. I tillegg til algoritmer for gjenkjenning, inneholder PCRTool også en del funksjonalitet for redigering av punktskyer. Dette er nyttig siden det ofte er nødvendig med litt prosessering av datasettene i forkant av gjenkjenningen.

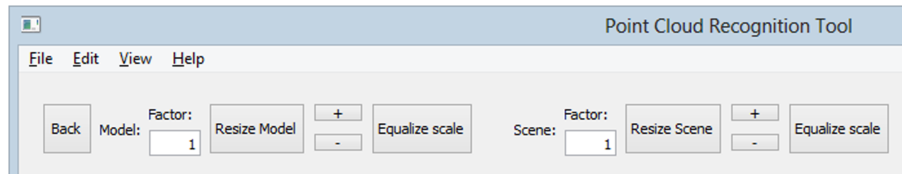
Importering og eksportering av punkstkyer

PCRTool kan åpne filer av typen ply og pcd. Dette gjøres via *File* → *Import...*→ *Import Model File...* eller *File* → *Import...*→ *Import Scene File...* På denne måten spesifiseres det om den innlastede filen er modellen eller scenen. Denne spesifiseringen er viktig siden mange algoritmer fungerer i en retning. Et eksempel på dette er ICP-algoritmen, som flytter modellen mot scenen, og ikke motsatt. Punktskyer kan lagres i formatene ply og pcd. Lagring skjer på samme måten som åpning, via *File* → *Export...*→ *Export model...* eller *File* → *Export...*→ *Export scene...*.

Visualisering i PCRTool

Punktskyene som importeres i programmet PCRTool visualiseres i programvinduet. Hvilken punktsky som skal vises, velges nede til venstre. Velges *Show model*, vises bare modell-punktskyen. Velges *Show scene* vises bare scene-punktskyen. Begge skyene kan vises i samme koordinatsystem ved å velge *Show both*. For å endre på kameravinkelen brukes musen til å dra bildet slik at kameraet roteres. Ved å bruke musehjulet kan kameravinkelen zoomes inn og ut. For å flytte kameraet sidelengs, holdes skifttasten nede samtidig som bildet dras til siden med musen. Rotering av kameraet, skjer på samme måte ved å holde inne kontrolltasten. Kameravinkelen kan nullstilles ved å trykke på **r**-tasten. For å øke eller minske størrelsen på punktene i punktskyen, brukes + og − på tastaturet.

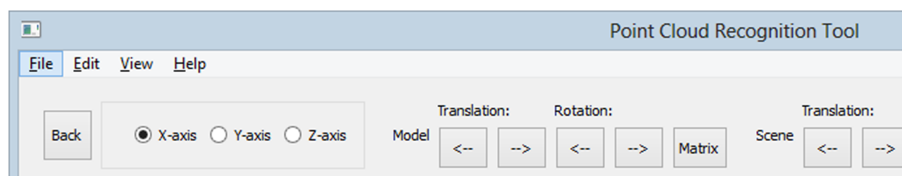
Skalering



Figur 5.1.2: Figuren viser Resize-panelet i PCRTool.

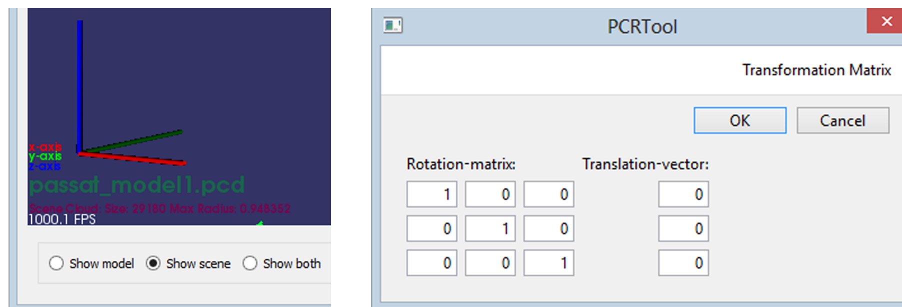
Panelet for skalering av de innlastede punktskyene vises ved å trykke på *Resize*. Her kan modell-skyen og scene-skyen skaleres med en faktor. For eksempel dersom en ønsker å øke størrelsen på punktskyen med 10% brukes faktoren 1, 1. Knappene med pluss og minus vil henholdsvis øke og minske størrelsen på den gjeldende skyen med 1%. Funksjonen *Equalize scale* er nyttig i noen situasjoner dersom en har to punktskyer med helt forskjellige skalaer. For eksempel kan den ene er punktskyen ha punkter som er gitt i meter der den andre er gitt i centimeter. Denne funksjonen vil da prøve å utjevne de største forskjellene. Dette gjøres ved å sammenligne lengden mellom gjennomsnittlig punktverdi (senterpunktet) og punktet som ligger lengst vekk fra senterpunktet i de to punktskyene. Deretter regnes det ut en faktor som vil gjøre at denne lengden blir lik i begge punktskyene. Dersom *Equalize scale* blir brukt for modell-skyen vil den bli skalert i henhold til scene-skyens skala. Det motsatte gjelder dersom funksjonen blir brukt for scene-skyen.

Transformasjon



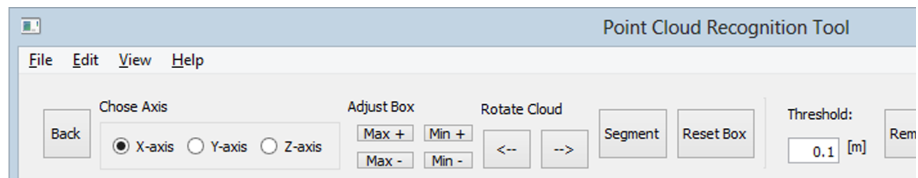
Figur 5.1.3: Figuren viser panelet for transformasjon i PCRTool.

For å rotere og flytte punktskyene brukes rigide transformasjoner. Ved å klikke på *Transform* vises panelet (figur 5.1.3) for transformasjon av punktskyer. Her kan skyene enten roteres rundt eller flyttes langs den spesifiserte aksen. Aksene spesifiseres av referanseaksene som vises i visualiseringen. Her er x-aksen vist i rødt, y-aksen i grønt og z-aksen i blått. Referanseaksene er vist til venstre i figur 5.1.4. Vet å klikke på knappen kalt *Matrix*, åpnes et nytt vindu der transformasjonsmatrisen kan legges inn direkte. Dette vinduet er vist i figur 5.1.4



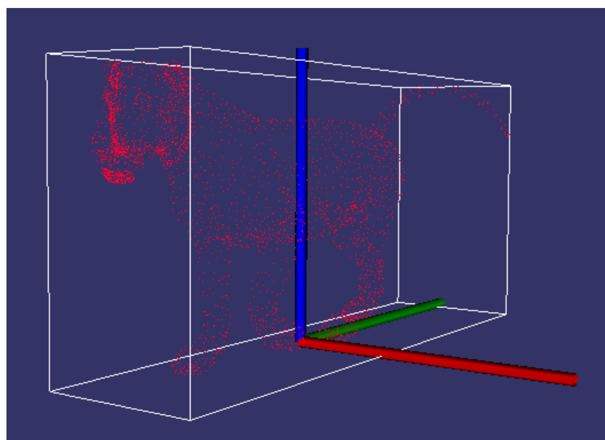
Figur 5.1.4: Til venstre vises referanseaksene fra visualiseringen. Til høyre vises vinduet som brukes for å kunne legge inn rotasjonsmatrisen og translasjonsvektoren direkte.

Segmentering



Figur 5.1.5: Figuren viser panelet for segmentering i PCRTool.

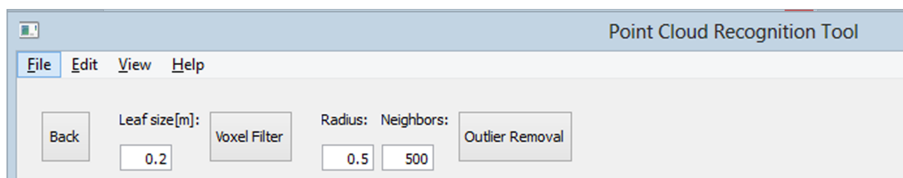
Av og til er det nødvendig å kunne bruke bare en liten del av punktskyene i gjenkjenningen. Da kan punktskyen segmenteres ved å bruke panelet for segmentering, som er vist i figur 5.1.5. Dette panelet vises ved å klikke på *Segmentation*. Under bruk av panelet, kan bare en av punktskyene vises om gangen. Punktskyen som skal segmenteres velges ved å klikke på *Show model* eller *Show scene*, nede i venstre hjørne i programvinduet. Punktskyen omsluttes av en boks, som har sider som er ortogonale i forhold til referanseaksene (figur 5.1.6). For å endre størrelsen på denne velges først i hvilken akse boksen skal endres. Knappene *Max +* og *Max -* gjør boksen større eller mindre i ene enden langs aksen som er valgt. Knappene *Min +* og *Min -* gjør boksen større eller mindre i andre enden langs den valgte aksen. Skyen kan også roteres rundt den valgte aksen på samme måte som i panelet for transformasjon, ved å bruke pilene. Ved å klikke *Segment*, fjernes de delene av punktskyen som ligger utenfor boksen.



Figur 5.1.6: Figuren viser boksen som brukes for segmentering av punktskyer.

Noen ganger inneholder punktskyer store flater. Disse kan oppstå under skanning på grunn av gulv eller vegger som er i bakgrunnen av objektene som skannes. For at dette ikke skal påvirke gjenkjenningen, kan det være en fordel å segmentere bort disse flatene. Dette kan gjøres automatisk ved å trykke på knappen *Remove Dominant Plane*. Da forsøkes det å fjerne det største planet i punktskyen. Parameteren *Threshold* spesifiserer tykkelsen på planet som fjernes.

Filtrering



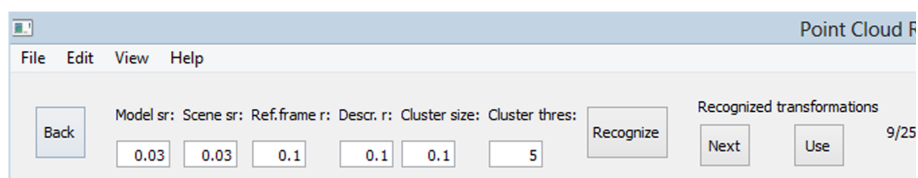
Figur 5.1.7: Figuren viser filterpanelet i PCRTool.

I Point Cloud Recognition Tool er det implementert to typer filtre. Disse vises i filterpanelet ved å klikke *Filtration*. Panelet er vist i figur 5.1.7. De to filtrene er et Voxelgrid-filter og et Outlier-Removal-filter. Voxelgrid-filteret brukes for nedsampling av punktskyene. Når filteret kjøres, deles punktskyen opp i et tredimensjonalt rutenett der kun et punkt fra hver rute blir tatt vare på. Parameteren *Leaf Size* spesifiserer, i meter, størrelsen på rutene som brukes. Outlier-Removal-filteret brukes for å fjerne støy i form av små felter med punkter som ikke er del av punktskyen. Dette gjøres ved å sette parametrene *Radius* og *Neighbours*. I hvert punkt sjekkes det innenfor den gitte radiusen, om det er flere nabopunkter enn det antallet som er spesifisert i parameteren, *Neighbours*. Dersom det ikke er tilfellet fjernes alle punktene som ligger innenfor den gitte radiusen.

Andre verktøy

Under panelet kalt *Tools*, finnes noen flere verktøy som kan være nyttige. Ved å klikke på *Model Density* eller *Scene Density* startes en algoritme for å regne ut gjennomsnittlig avstand mellom alle punkt i punktskyen. Resultatet vises i tekstboksen, nede til høyre i programvinduet. Ved å trykke på *Merge Cloud*, kopieres modell-punktskyen inn i scene-punktskyen. Deretter kan scene-punktskyen lagres. Denne funksjonen kan brukes til å sette sammen to punktskyer.

Gjenkjenning med ≪Correspondence Grouping≫



Figur 5.1.8: Figuren viser panelet for gjenkjenning med ≪Correspondence Grouping≫ i PCRTool.

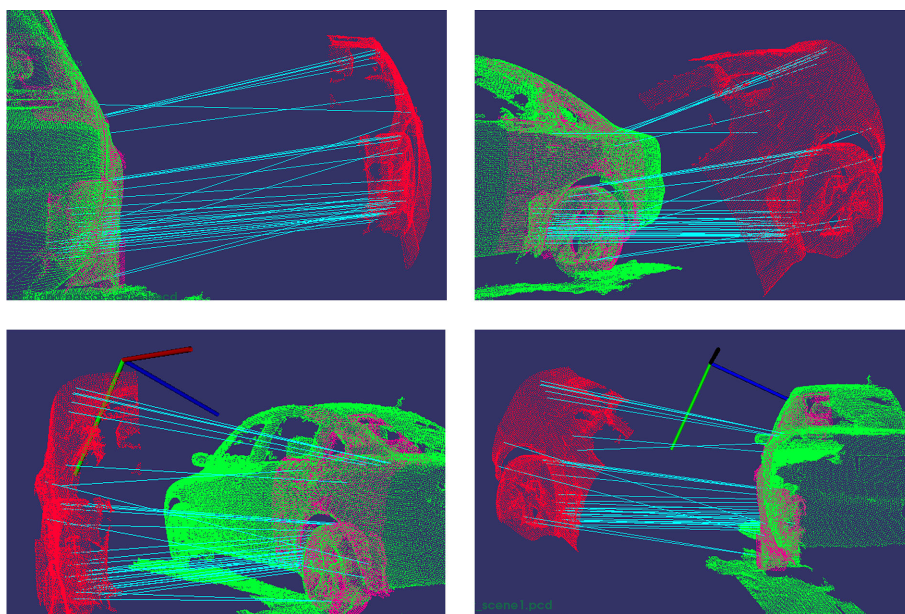
Ved å trykke på *CG*, vises panelet for gjenkjenning med ≪Correspondence Grouping≫ i PCRTool. Fra dette panelet settes parametrene for algoritmen, som er beskrevet i 4.8. Algoritmen som er implementert i PCRTool, bruker uniform sampling av nøkkelpunkter. Den beskriver nøkkelpunktene med SHOT-deskriptoren (kapittel 4.5.3). Mulige forekomster av modell i scene finnes ved bruk av Hough-metoden, som er beskrevet i kapittel 4.8. Parametre for disse tre elementene kan settes i panelet for gjenkjenning:

- **Model sr:** Denne parameteren setter hvor tett nøkkelpunktene i modell-punktskyen skal samples. Verdien gis i meter.
- **Scene sr:** Denne parameteren setter hvor tett nøkkelpunktene i scene-punktskyen skal samples. Verdien gis i meter.
- **Ref.frame r:** Parameteren bestemmer hvor stor radius som skal brukes under konstruksjon av de lokale koordinatsystemene for SHOT-deskriptoren. Koordinatsystemene blir konstruert ved egenverdidekomposisjon av kovariansmatrisen for et område. Radiusen definerer området rundt hvert punkt. For mer informasjon om dette, se kapittel 4.5.3. Verdien gis i meter.
- **Descr. r:** Parameteren bestemmer radiusen som skal brukes i beskrivelsen av nøkkelpunktene. Denne brukes til å definere størrelsen på kulen som opprettes rundt hvert nøkkelpunkt av SHOT-deskriptoren. Se kapittel 4.5.3.

- **Cluster size:** Med denne parameteren kan størrelsen på diskretiseringen av Hough-rommet stilles. Dette åpner for en mer robust stemmeprosess men også redusert nøyaktighet.

- **Cluster thresh:** Parameteren definerer en nedre grense på antallet stemmer fra korrespondanser mellom modell og scene, som må være til stede for at modell-forekomsten skal bli tatt med i resultatet.

Når alle parametrene er satt, kan algoritmen startes ved å klikke *Recognize*. Informasjon om kjøringen kan ses i kommandovinduet. Algoritmen vil da forsøke å detektere mulige forekomster av modellen i scenen. Dersom mer enn en mulig forekomst blir detektert, kan de vises ved å klikke på *next*. Transformasjonsmatrisen for hver forekomst, skrives ut i kommandovinduet. Modell-punktskyen kan flyttes til en av de detekterte forekomstene ved å klikke *Use*.

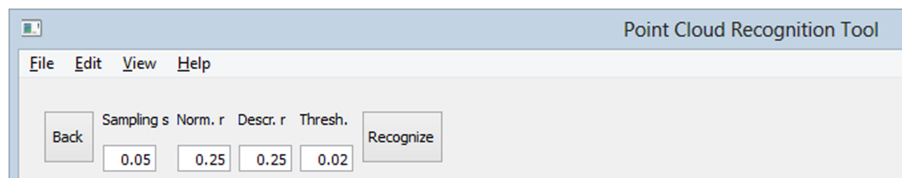


Figur 5.1.9: Figuren viser et vellykket forsøk på gjenkjenning fra fire forskjellige kameravinkler. Her vises modell-punktskyen i rødt og scene-punktskyen i grønt. Den rosa punktskyen indikerer den mulige forekomsten av modellen i scenen. De turkise linjene viser hvilke og hvor mange korrespondanser, som ble detektert mellom modellen og scenen.

Etter at modellen er flyttet det korresponderende området i scenen, kan resultatet finjusteres med ICP-algoritmen beskrevet i kapittel 4.7. Denne kan kjøres ved å klikke på *Iterative Closest Point* i panelet for gjenkjenning. Algoritmen trenger kun en parameter kalt *Max Iterations*. Denne spesifiserer det maksimale antallet iterasjoner algoritmen skal utføre. Ved å sette denne til verdien 0, brukes algoritmens egen deteksjon av konvergens for å stoppe iterasjonen.

Det er også mulighet til å detektere forskjeller mellom to punktskyer. Dette gjøres med algoritmen ≪Spatial Change Detection≫, som er beskrevet i kapittel 4.2.1. Denne kjøres ved å klikke *Show Differences* i panelet for gjenkjenning. Parameteren *Octree Resolution* spesifiserer sidelengden på de minste kubene i Octree-strukturen. Verdien gis i meter. Dette gir oppløsningen til deteksjonen av forskjeller.

Gjenkjenning med ≪Implicit Shape Model≫



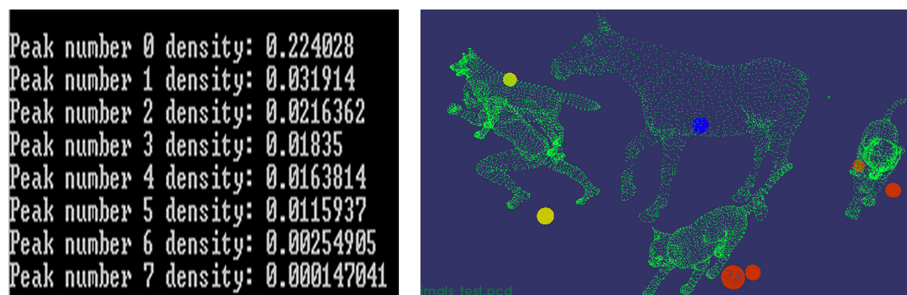
Figur 5.1.10: Figuren viser panelet for gjenkjenning med ≪Implicit Shape Model≫ i PCRTool.

Ved å trykke på *ISM*, vises panelet for gjenkjenning med ≪Implicit Shape Model≫ i PCRTool. Fra dette panelet settes parametrene for algoritmen, som er beskrevet i 4.9. Algoritmen som er implementert i PCRTool, beskriver nøkkelpunktene med FPFH-deskriptoren (kapittel 4.5.2). Algoritmen detekterer senterpunktene til mulige forekomster av objektet i scenen. Dette gjøres ved bruk av Hough-stemming som beskrives i kapittel 4.9. Algoritmen trenger følgende parametre for å kunne kjøre:

- **Sampling s:** Denne parameteren spesifiserer hvor tett nøkkelpunktene i modell-punktskyen og scene-punktskyen skal samples. Verdien gis i meter.
- **Norm. r:** Parameteren bestemmer radiusen som skal brukes i utregningen av punktnormalene. Denne metoden beskrives i 4.3.
- **Descr. r:** Variabelen definerer radiusen på området som brukes under beskrivelse av nøkkelpunktene.
- **Thresh:** Parameteren definerer en grense for hvor stor tetthet av Hough-stemmer det må være for et punkt, for at punktet skal regnes som et mulig senterpunkt. Denne grensen brukes under visualiseringen av senterpunktene for å bestemme hvilken farge senterpunktene får

Når parametrene er satt kan algoritmen kjøres ved å trykke på *Recognize*. Når kjøringen er ferdig blir de detekterte toppene i Hough-rommet skrevet ut i kommandovinduet som vist til venstre i figur 5.1.11 . Til høyre i figuren vises detekterte senterpunkter i visualiseringen. Her blir det beste treffet visualisert med en blå kule, treff som har en tetthet, høyere enn grensen satt av `Thresh`-parameteren, visualiseres med en gul kule. De resterende treffene visualiseres

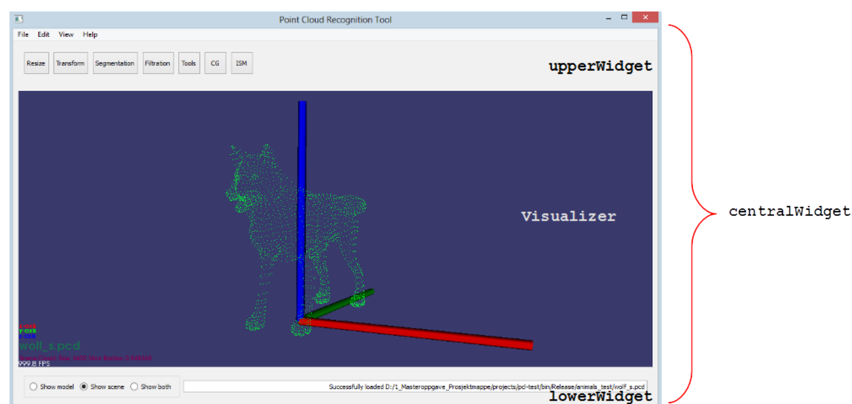
med røde kuler.



Figur 5.1.11: Til venstre vises utskriften av de detekterte toppene i Hough-rommet. Til høyre vises visualiseringen av de detekterte toppene.

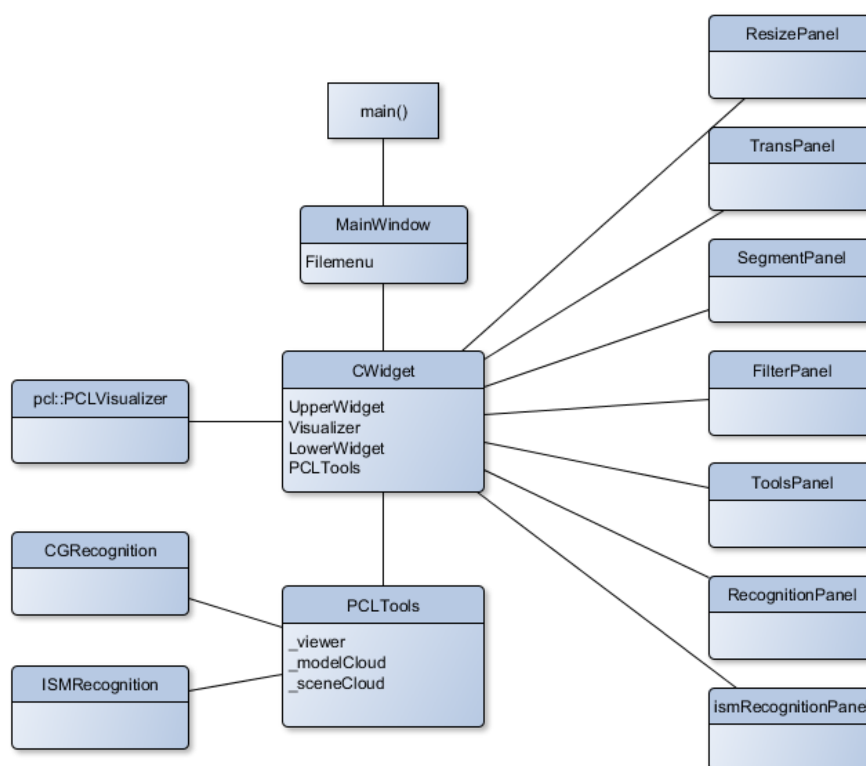
5.1.2 Programstruktur

Brukergrensesnittet i Point Cloud Recognition Tool er programmert med funksjoner som er tilgjengelige i Qt-biblioteket. Qt-biblioteket tilbyr en rekke ≪widgets≫(elementer) som kan arrangeres i layouter for å lage brukergrensesnitt. Programmets punktsky-funksjonalitet er programmert med PCL-bibliotekets funksjoner. PCRTool er sammensatt av tre hovedklasser. Dette er klassene `MainWindow`, `CWidget` og `PCLTools`. Når `main`-metoden kjøres opprettes en Qt-applikasjon ved å initialisere et objekt av `MainWindow`. I denne klassen opprettes programvinduet og fil-menyen. Deretter initialiseres et objekt av klassen `CWidget`, kalt `centralWidget`. Denne klassen arver metodene til klassen `QWidget` som gjør at den kan brukes som en `QWidget`. Objektet `centralWidget` settes til å være det sentrale elementet i programvinduet. Objektet `centralWidget` inneholder tre elementer som arrangeres i en vertikal layout. De tre elementene er `upperWidget`, `visualizer` og `lowerWidget` og disse vises i figur 5.1.12.



Figur 5.1.12: Figuren viser programmet Point Cloud Recognition Tool. Her vises sammensetningen av `centralWidget` med de tre elementene `upperWidget`, `Visualizer` og `lowerWidget`.

Det øverste elementet er upperWidget. Dette elementet inneholder en QStackedWidget som fungerer som en stabel av paneler der det kan byttes mellom hvilket panel som skal vises til en hver tid. upperWidget-elementet brukes til å vise de forskjellige knappene som styrer punktsky-funksjonaliteten i programmet. Funksjonaliteten kategoriseres i kategoriene Resize, Transform, Segmentation, Filtration, Tools og Recognition. Hver kategori av funksjoner har sitt eget panel som legges i stabelen av elementer. Hvert panel har sin egen klasse og det er centralWidget som oppretter objekter av panelene som skal brukes. Det neste elementet som utgjør centralWidget er en Visualizer. Denne er initialisert som et objekt av klassen pcl::PCLVisualizer fra PCL-biblioteket. Dette vinduet er interaktivt og kan i tillegg til annen funksjonalitet brukes til å visualisere en eller flere punktskyer. Det siste elementet i centralWidget er lowerWidget. I dette panelelementet kan det velges hvilken punktsky som skal visualiseres i vinduet. Her er også en tekst-linje der det blir vist informasjon om kjøringen til programmet.



Figur 5.1.13: Figuren viser en forenklet visualisering av strukturen mellom klassene i PCRTool

I centralWidget opprettes objektet pclTools av klassen PCLTools. Av hensyn til visualiseringen av punktskyer, passeres en peker til visualizer-vinduet ned til pclTools. Klassen inneholder alle metoder i programmet som utfører operasjoner på og visualisering av punktskyer. pclTools er satt til å være en privat variabel for centralWidget. Alle metoder andre steder i programmet må gå gjennom centralWidget for å bruke metoder i pclTools. På grunn av

dette, får alle paneler som initialiseres i centralWidget, passert en peker til denne klassen som kalles parentCWidget. Dette gjør at panelene får tilgang til punktsky-funksjonaliteten ved følgende metodekall:

```
parentCWidget->getPCL();
```

Metoden getPCL returnerer da en peker til pclTools-klassen, som gir tilgang til klassens public-metoder. På denne måten oppstår ikke situasjoner der metoder i pclTools blir forsøkt brukt fra flere steder på samme tid. Programmet har mulighet for å laste inn to ulike punktskyer. Punktskyene som lastes inn er modell-skyen og scene-skyen. Etter at punktskyene er lastet inn blir private pekere til punktskyene lagret i objektet pclTools, med navnene _modelCloud og _sceneCloud. Dette vil si at det er kun klassen pclTools som skal ha tilgang til punktskyene. For å vite hvilken punktsky som det skal utføres operasjoner på brukes std::string-argumenter som har verdien "model", "scene" eller "both". Nedenfor vises et eksempel på et metodekall fra ResizePanel for å endre størrelse på model-punktskyen:

```
parentCWidget->getPCL()->resizeCloud("model", factor);
```

På grunn av størrelsen til ≪Correspondence Grouping≫-algoritmen, er denne flyttet ut i en egen klasse. Denne klassen er kalt CGRecognition. Algoritmen som er implementert i denne klassen er basert på kode som er tilgjengelig fra [71]. pclTools initialiserer denne klassen når gjenkjenningen skal kjøres. Det samme er tilfellet for ≪Implicit Shape Model≫-algoritmen. Denne er implementert i klassen ISMRecognition. Denne er også implementert basert på implementeringen som er beskrevet på PCL sine hjemmesider. [74] Denne klassen blir også initialisert fra pclTools.

5.2 Implementering av ≪Correspondence Grouping≫

Algoritmen ≪Correspondence Grouping≫, omtalt i kapittel 4.8, er implementert i klassen CGRecognition. I denne klassen er mye av koden basert på implementeringen som er tilgjengelig på PCL-bibliotekets hjemmesider. [71] Her brukes uniform sampling av nøkkelpunkter. I [72] ble flere mulige deskriptor-algoritmer foreslått. I dette prosjektet beskrives nøkkelpunktene med SHOT-deskriptoren, som er beskrevet i kapittel 4.5.3. Lokale koordinatsystem for Hough-stemmingen lages på samme måte som er foreslått for SHOT-deskriptoren. Denne metoden omtales også i kapittel 4.5.3. Hough-stemmingen som brukes, er omtalt i kapittel 4.8.

Klassen PCLTools har en privat global peker til klassen CGrecognition, kalt rec. Metoden recognize i pclTools kjøres for å bruke algoritmen. I recognize-metoden opprettes et nytt objekt av CGRecognition ved å re-instansiere pekeren

`rec`. Punktskyene som skal brukes og parametrene som algoritmen trenger, settes via metodene `setInputModel()`, `setInputScene()` og `setParameters()`. Deretter kjøres metoden `compute()`, som starter algoritmen.

```
rec.reset(new CGRecognition());
rec->setInputModel(_modelCloud);
rec->setInputScene(_sceneCloud);
rec->setParameters(model_ss, scene_ss, rf_rad, descr_rad,
    cg_size, cg_tresh);
rec->compute();

transformations = rec->getTransformations();
correspondences = rec->getCorrespondences();
```

Algoritmen returnerer `transformations` og `correspondences`. `transformations` er en vektor som inneholder de rigide transformasjonene som er detektert mellom modellen og scenen. `correspondences` inneholder en liste over korrespondanser mellom punktskyene for hver detekterte forekomst av modellen i scenen.

5.2.1 Initialisering

I `CGRecognition` defineres først noen typer som blir mye brukt i resten av koden. `PointType` defineres til å være et vanlig `pcl::PointXYZ`, som kun inneholder et 3D-koordinat. Deskriptortypen, `DescriptorType`, defineres til å være `pcl::SHOT352`. Normaltypen, `NormalType`, defineres til `pcl::Normal` og typen for de lokale koordinatsystemene, `RFType` defineres til å være `pcl::ReferenceFrame`.

Klassen `CGRecognition` bruker åtte ≪PointCloud≫-strukturer for å holde på den nødvendige informasjonen, algoritmen trenger. To liste-strukturer brukes til å lagre resultatene fra algoritmen. De ulike strukturene er listet opp nedenfor.

```
pcl::PointCloud<PointType>::Ptr model;
pcl::PointCloud<PointType>::Ptr model_keypoints;
pcl::PointCloud<PointType>::Ptr scene;
pcl::PointCloud<PointType>::Ptr scene_keypoints;
pcl::PointCloud<NormalType>::Ptr model_normals;
pcl::PointCloud<NormalType>::Ptr scene_normals;
pcl::PointCloud<DescriptorType>::Ptr model_descriptors;
pcl::PointCloud<DescriptorType>::Ptr scene_descriptors;

std::vector<Eigen::Matrix4f,
    Eigen::aligned_allocator<Eigen::Matrix4f> > rototranslations;
std::vector<pcl::Correspondences> clustered_corrs;
```

Disse initialiseres under kjøring av konstruktøren til klassen. `model` og `scene` er de originale punktskyene som det skal gjenkjennes i. Disse overføres til klassen ved å bruke metodene, `setInputModel()` og `setInputScene()`. `model_keypoints` og `scene_keypoints` opprettes for å inneholde nøkkelpunkter som hentes fra

model og scene. model_normals og scene_normals opprettes for å inneholde normalretninger for model og scene. model_descriptors og scene_descriptors opprettes for å inneholde punktbeskrivelsene for nøkkelpunktene. rototranslations er en liste med 4x4 matriser som opprettes for å inneholde de detekterte transformasjonene. clustered_corrs er en liste som inneholder korrespondanser mellom punktbeskrivelser.

Algoritmen bruker seks parametre under kjøring. Disse settes utenfra ved å bruke metoden setParameters() i klassen CGRecognition. De seks parametrene er listet opp nedenfor.

```
float model_ss_;
float scene_ss_;
float rf_rad_;
float descr_rad_;
float cg_size_;
float cg_thresh_;
```

Her er model_ss_ og scene_ss_, oppløsningen på den uniforme samplingen av punktskyene, for å plukke ut nøkkelpunktene. rf_rad_ er radiusen som brukes under konstruksjonen av de lokale koordinatsystemene som algoritmen trenger. Denne radiusen sier hvor stort område som skal tas hensyn til når kovariansmatrisen skal opprettes. Opprettelsen av de lokale koordinatsystemene er omtalt i kapittel 4.5.3. descr_rad_ er radiusen som brukes under punktbeskrivelsene. Her brukes SHOT-deskriptoren, som betyr at denne radiusen definerer kulen som brukes her (kapittel 4.5.3). Variabelen cg_size_ definerer størrelsen på boksene som brukes for å diskretisere Hough-rommet. cg_thresh_ brukes for å sette en nedre grense for antallet stemmer fra korrespondanser som trengs for å definere et resultat som en deteksjon.

5.2.2 Gjennomgang av algoritmen

Selve algoritmen kjøres i klassens compute-metode. Det første som gjøres er å regne ut normalene til punktskyene. Utregning av normalverdiene er vist nedenfor.

```
pcl::NormalEstimationOMP<PointType, NormalType> norm_est;
norm_est.setKSearch (10);

norm_est.setInputCloud (model);
norm_est.compute (*model_normals);

norm_est.setInputCloud (scene);
norm_est.compute (*scene_normals);
```

Her opprettes det først et objekt, norm_est, av pcl-klassen, NormalEstimatorOMP, med forhåndsdefinert punkttype og normaltype. Søkeavstanden settes med me-

toden `setKSearch()` til å være 10. Dette vil si at de 10 nærmeste naboene tas med i beregningen av normalene i et punkt. I [71] omtales dette som ok verdi for flere datasett. Normalene regnes ut ved å sette punktskyen `model` som input til `norm_est`-objektet. Deretter kjøres metoden `compute(*model_normals)`. Her gis pekeren `model_normals` som input. Pekeren spesifiserer hvor algoritmen som finner normalretninger i punktskyen, skal lagre resultatet. Etterpå repeteres det samme for punktskyen, `scene`.

Det neste som gjøres er å finne nøkkelpunkter i punktskyene. Dette gjøres ved å bruke uniform sampling av skyene. Nedenfor vises koden for uniform sampling av modellen, `model`.

```
pcl::PointCloud<int> sampled_indices;

pcl::UniformSampling<PointType> uniform_sampling;
uniform_sampling.setInputCloud (model);
uniform_sampling.setRadiusSearch (model_ss_);
uniform_sampling.compute (sampled_indices);
pcl::copyPointCloud (*model, sampled_indices.points,
    *model_keypoints);
```

En liste over indekser, `sampled_indices`, opprettes. Deretter opprettes et objekt av klassen `pcl::UniformSampling`. Dette objektet brukes til å utføre samplingen. Her settes først punktskypekeren, `model`, og parameteren `model_ss_`. Metoden `compute(sampled_indices)`, kjøres og henter ut indeksene til nøkkelpunktene fra punktskyen. Deretter kopieres de indekserte punktene fra punktskyen, `model`, til `model_keypoints`. Det samme gjøres for punktskyen, `scene`.

Det neste som gjøres er å regne ut punktbeskrivelsene for nøkkelpunktene. Her brukes SHOT-deskriptoren. Koden som brukes for å lage punktbeskrivelser for nøkkelpunktene i modellen er vist nedenfor.

```
pcl::SHOTEstimationOMP<PointType, NormalType, DescriptorType>
    descr_est;
descr_est.setRadiusSearch (descr_rad_);

descr_est.setInputCloud (model_keypoints);
descr_est.setInputNormals (model_normals);
descr_est.setSearchSurface (model);
descr_est.compute (*model_descriptors);
```

Et objekt, `descr_est`, opprettes av klassen `pcl::SHOTEstimationOMP`. Her settes radiusen til deskriptoren, nøkkelpunktene, og normalretningene ved å bruke metodene `setRadiusSearch()`, `setInputCloud()` og `setInputNormals()`. Punktbeskrivelsene opprettes ved å kjøre `compute()`-metoden med pekeren `model_descriptors` som input. Den samme metoden brukes for å lage punktbeskrivelser for scenen.

Det neste som gjøres er å finne korrespondansene mellom punktbeskrivelsene for modellen og punktbeskrivelsene for scenen. For å gjøre dette brukes det

en kdtree-struktur, som brukes til å gjøre et nærmeste-nabo-søk i punktbeskrivelsene. For å lagre detekterte korrespondansene opprettes et objekt av klassen `pcl::Correspondences`, kalt `model_scene_corrs`. Dette objektet kan holde avstanden mellom to korrespondanser og korrespondansenes indekser. Nedenfor vises opprettelsen av `model_scene_corrs` og kdtree-strukturen, `matc_search`.

```
pcl::CorrespondencesPtr model_scene_corrs (new
    pcl::Correspondences ());

pcl::KdTreeFLANN<DescriptorType> match_search;
match_search.setInputCloud (model_descriptors);

std::vector<int> neigh_indices (1);
std::vector<float> neigh_sqr_dists (1);
```

Kdtree-strukturen blir opprettet for punktbeskrivelsene i `model_descriptors`, ved å bruke metoden `setInputCloud()`. To lister, med en lengden 1, opprettes for å kunne lagre resultatene fra nabosøket. Disse er `neigh_indices` og `neigh_sqr_dists`. Den første lagrer det detekterte nabopunktets indeks i `model_descriptors`. Altså hvilket nummer det detekterte punktet er i listen over modell-punktbeskrivelser. Den andre listen lagrer den kvadrerte avstanden mellom det detekterte nabopunktet og punktet som det letes etter. Altså avstanden mellom de detekterte korrespondansene. Grunnen til at lengden er 1 er at bare det nærmeste treffet i søket er interessant. For hver punktbeskrivelse i `scene_descriptors`, søkes gjennom kdtree-strukturen etter det nærmeste treffet. Siden kdtree-strukturen er bygget fra datasettet, `model_descriptors`, søkes det dermed etter de nærmeste treffene mellom modell-beskrivelsene og scene-beskrivelsene. Treffene som søket gir, er mulige korrespondanser mellom modell og scene. Nedenfor vises hvordan det søkes etter korrespondanser til et av punktene fra `scene_descriptors`.

```
int found_neighs = match_search.nearestKSearch
    (scene_descriptors->at (i), 1, neigh_indices,
    neigh_sqr_dists);
```

I algoritmen, itereres det over variabelen `i`, som betyr at samme søk blir utført for alle scene-punktbeskrivelsene. Metoden `nearestKSearch` starter søket og returnerer antallet naboer som ble funnet. Metodens første inputvariabel er et av punktene fra `scene_descriptors`. Den neste inputvariabelen spesifiserer hvilken nabo det skal søkes etter. Den nærmeste naboen spesifiseres med verdien 1. De to neste variablene er listene som ble initialisert tidligere. Her lagres indeks og kvadrert avstand til det detekterte treff.

For hvert treff, sjekkes den kvadrerte avstanden mellom søkepunktet, `scene_descriptors->at(i)` og søketreffet. Denne avstanden må være mindre enn grensen på 0.25 for at det skal være et gyldig treff. På denne måten avgjøres det om de to punktbeskrivelsene er like nok for å kunne defineres som en korrespondanse. Nedenfor vises hvordan dette gjøres i koden.

```
if(found_neighs == 1 && neigh_sqr_dists[0] < 0.25f)
```

```
{
        pcl::Correspondence corr (neigh_indices[0],
            static_cast<int> (i), neigh_sqr_dists[0]);
        model_scene_corrs->push_back (corr);
}
```

Dersom et treff kan defineres som en korrespondanse, opprettes det et midlertidig korrespondanseobjekt, kalt `corr`. Dette initialiseres med tre variabler. Den første er `neigh_indices[0]`, som nå inneholder modell-punktbeskrivelsens indeks. Den andre er for-løkkens iterasjonsvariabel, `i`, som spesifiserer indeksen til scene-punktbeskrivelsen. Den tredje variabelen er avstanden mellom punktbeskrivelsene i søket. Denne midlertidige korrespondansen legges inn i `model_scene_corrs`. Etter at alle punktbeskrivelsene er gjennomsøkt, inneholder `model_scene_corrs` ett sett med korrespondanser.

Det neste som gjøres, er å lage lokale koordinatsystem for nøkkelpunktene. Dette gjøres ved å opprette ett objekt, `rf_est` av klassen `pcl::BOARDLocalReferenceFrameEstimation`. Denne klassen oppretter lokale koordinatsystem med metoden beskrevet i kapittel 4.5.3. Koden for konstruksjonen for `model_keypoints` er vist nedenfor.

```
pcl::BOARDLocalReferenceFrameEstimation<PointType, NormalType,
    RFType> rf_est;
rf_est.setFindHoles (true);
rf_est.setRadiusSearch (rf_rad_);

rf_est.setInputCloud (model_keypoints);
rf_est.setInputNormals (model_normals);
rf_est.setSearchSurface (model);
rf_est.compute (*model_rf);
```

Ved å bruke `setFindHoles()`, med inputparameter `true`, spesifiseres det at algoritmen skal lete etter og ta hensyn til huller i overflaten når koordinatsystemene konstrueres. Med metoden `setRadiusSearch()`, settes radiusen, R, til området som skal brukes for opprettelse av matrisen $\boldsymbol{M}$ fra ligning 4.5.9 i kapittel 4.5.3. Klasseobjektet `rf_est`, trenger også nøkkelpunktene, normalene og den opprinnelige punktskyen. Disse settes ved å bruke metodene `setInputCloud()`, `setInputNormals()` og `setSearchSurface()`. Grunnen til at den opprinnelige punktskyen, `model` brukes er at denne inneholder alle punktene. Koordinatsystemene konstrueres kun i nøkkelpunktene, men trenger et område av punkter rundt nøkkelpunktet under opprettelsen. Disse hentes fra punktskyen `model`. Konstruksjonen startes ved bruk av metoden `compute()` og koordinatsystemene lagres i `model_rf`. Den samme prosedyren utføres for `scene_keypoints` og koordinatsystemene lagres i `scene_rf`.

All informasjonen som trengs for gjenkjenning er nå klar og algoritmen for Hough-stemming kan startes. Dette er algoritmen som ble omtalt i kapittel 4.8. Koden for denne algoritmen er vist nedenfor. Det første som gjøres er å opprette et objekt, `clusterer`, av klassen `pcl:Hough3DGroupin`. Deretter settes algo-

ritmeparametrene, `cg_size_` og `cg_thresh_`. Disse er omtalt tidligere i kapittelet og spesifiserer størrelsen på boksene, som diskretiserer Hough-rommet, og nedre grense for antall korrespondanser. Ved å bruke `setUseInterpolation()` med input `true`, spesifiseres det at det skal brukes interpolering mellom nærliggende bokser i Hough-rommet. Ved å bruke metoden `setUseDistanceWeight()` med input `false`, spesifiseres det at avstanden mellom korrespondansene ikke skal brukes som en veiefaktor for Hough-stemmene.

```
pcl::Hough3DGrouping<PointType, PointType, RFType, RFType>
    clusterer;
clusterer.setHoughBinSize (cg_size_);
clusterer.setHoughThreshold (cg_thresh_);
clusterer.setUseInterpolation (true);
clusterer.setUseDistanceWeight (false);

clusterer.setInputCloud (model_keypoints);
clusterer.setInputRf (model_rf);
clusterer.setSceneCloud (scene_keypoints);
clusterer.setSceneRf (scene_rf);
clusterer.setModelSceneCorrespondences (model_scene_corrs);

clusterer.recognize (rototranslations, clustered_corrs);
```

For å kunne kjøre trenger algoritmen nøkkelpunkter for modell og scene. Nøkkelpunktene for modellen settes med metoden `setInputCloud()` og nøkkelpunktene for scenen settes med `setSceneCloud()`. Deretter settes koordinatsystemene med `setInputRf()` og `setSceneRf()`. Til slutt settes de detekterte korrespondansene, `model_scene_corrs`, med metoden `setModelSceneCorrespondences()`. Algoritmen startes ved å bruke metoden `recognize()`, som lagrer estimerte transformasjonsmatriser i listen, `rototranslations` og de tilhørende korrespondansene i listen `clustered_corrs`.

5.3 Implementering av ≪Implicit Shape Model≫

Algoritmen ≪Implicit Shape Model≫ (ISM), beskrevet i kapittel 4.9 ble foreslått i artikkelen [73]. I dette prosjektet implementeres algoritmen i klassen `ISMRecognition`. I denne klassen er mye av koden for selve algoritmen basert på koden som ligger på PCL-bibliotekets hjemmesider. [74] Her brukes nøkkelpunkter som er en nedsamplet versjon av de opprinnelige punktskyene. Det foreslås bruk av deskriptoren 3D-SURF i [73]. Denne er derimot ikke implementert i PCL-biblioteket og FPFH-deskriptoren, omtalt i kapittel 4.5.2 brukes i stedet.

I klassen `PCLTools` kan ISM-algoritmen kjøres ved å bruke metoden `ismRecognition()`. I denne metoden opprettes et nytt objekt, `ismrec`, av klassen `ISMRecognition`. Punktskyene for modell og scene settes ved å bruke metodene `setInputModel()` og `setInputScene()`. Parametrene settes gjennom metoden `setParameters()`. Algoritmen startes ved å kjøre metoden `compute()`. Når algoritmen er ferdig

opprettes en liste over senterpunkter, `centerpoints` og en liste over de høyeste toppene i Hough-rommet, `strongest_peaks`. Listene brukes til å lagre resultatet fra algoritmen ved å bruke metodene `getCenterPoints()` og `getStrongestPeaks()`.

```
ISMRecognition ismrec;
ismrec.setInputModel(_modelCloud);
ismrec.setInputScene(_sceneCloud);
ismrec.setParameters(ss,descrr,normr);
ismrec.compute();
std::vector<pcl::ISMPeak,
    Eigen::aligned_allocator<pcl::ISMPeak>> strongest_peaks;
std::vector<pcl::PointXYZ> centerpoints;
strongest_peaks=ismrec.getStrongestPeaks();
centerpoints=ismrec.getCenterPoints();
```

ISM-algoritmen har mulighet til å trenes opp med flere punktskyer hvert objekt. Denne funksjonen er ikke utnyttet i denne implementeringen. Her brukes kun en punktsky for trening og en punktsky for testing.

5.3.1 Initialisering

Algoritmen initialiseres med fire parametere. Disse er `ss_`, `norm_rad_` og `descr_rad_`. Parameteren `ss_`, definerer størrelsen på samplingen av både modell og scene for å opprette nøkkelpunkter. Parameteren `norm_rad_` er radiusen som brukes for å konstruere normalretningene. Dette blir beskrevet i kapittel 4.3, der radiusen brukes til å opprette en kovariansmatrise. Parameteren `descr_rad_` spesifiserer radiusen i området som brukes for å opprette punktbeskrivelsene i nøkkelpunktene. Dette er beskrevet i kapittel 4.5.2.

```
float ss_;
float norm_rad_;
float descr_rad_;
```

Under initialiseringen av algoritmen opprettes det to punktsky-pekere for modellen og scenen. To lister opprettes. I den ene, `strongest_peaks`, lagres de beste treffene fra Hough-stemmingen. I den andre, `detected_centerpoints`, lagres senterpunktene som regnes ut basert på resultatet i `strongest_peaks`. Koden for å opprette de nevnte punktskyene og listene er vist nedenfor.

```
pcl::PointCloud<pcl::PointXYZ>::Ptr model;
pcl::PointCloud<pcl::PointXYZ>::Ptr scene;
std::vector<pcl::PointXYZ> detected_centerpoints;
std::vector<pcl::ISMPeak, Eigen::aligned_allocator<pcl::ISMPeak>
    > strongest_peaks;
```

5.3.2 Gjennomgang av algoritmen

ISM-algoritmen har mulighet til å trenes opp med flere objektklasser. Hver objektklasse kan også trenes av flere punktskyer. På grunn av dette brukes det lister for å lagre punktskyer for trening, normalretningene for hver punktsky og objektklassenummeret. Initialiseringen av disse listene vises nedenfor.

```
std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> training_clouds;
std::vector<pcl::PointCloud<pcl::Normal>::Ptr> training_normals;
std::vector<unsigned int> training_classes;
```

For hver punktsky som brukes til å trene opp ISM-modellen, finnes normalretningene. Dette gjøres ved å bruke et objekt kalt `normal_estimator`, av klassen `pcl::NormalEstimation`. Her settes radiusen, `norm_rad_`, som brukes for konstruksjon av normalene, ved å kjøre metoden `setRadiusSearch()`. Det opprettes også en peker til en punktbeskriver kalt `feature_estimator`, av klassen `pcl::Feature`. Denne initialiseres til å være en punktbeskriver av typen `FPFH` (kapittel 4.5.2). Dette gjøres ved å opprette et objekt kalt `fpfh`, av klassen `pcl::FPFHEstimation`, der radiusen `descr_rad_`, settes ved å bruke metoden `setRadiusSearch()`. Opprettelsen av objektene `normal_estimator` og `feature_estimator` er demonstrert i koden nedenfor.

```
pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal>
    normal_estimator;
normal_estimator.setRadiusSearch (norm_rad_);

pcl::FPFHEstimation<pcl::PointXYZ, pcl::Normal,
    pcl::Histogram<153> >::Ptr fpfh(new
    pcl::FPFHEstimation<pcl::PointXYZ, pcl::Normal,
    pcl::Histogram<153> >);
fpfh->setRadiusSearch (descr_rad_);
pcl::Feature< pcl::PointXYZ, pcl::Histogram<153> >::Ptr
    feature_estimator(fpfh);
```

For hver punktsky som brukes under treningen regnes det ut normalverdier. Dette gjøres ved å bruke objektet, `normal_estimator`, som er nevnt tidligere. For hver punktsky, legges selve punktskyen, normalretningene og objektklassenummeret inn i listene som ble opprettet tidligere. Disse listene heter `training_clouds`, `training_normals` og `training_classes`. Den nødvendige informasjonen legges bakerst i listen ved å bruke metoden `push_back()`. I dette prosjektet brukes kun en punktsky under opptreningen. Derfor legges punktskyen `model`, dens normalretninger, og objektklassenummer, direkte inn i listen. Her brukes objektklassenummeret 0.

```
pcl::PointCloud<pcl::Normal>::Ptr tr_normals = (new
    pcl::PointCloud<pcl::Normal>)->makeShared ();
normal_estimator.setInputCloud (model);
normal_estimator.compute (*tr_normals);

training_clouds.push_back (model);
```

```
training_normals.push_back (tr_normals);
training_classes.push_back (0);
```

Det neste som gjøres er å opprette et objekt av klassen `pcl::ism::ImplicitShapeModelEstimation`. Det er dette objektet som utfører treningen av ISM-modellen og søking etter mulige senterpunkter til et objekt. Objektet som blir opprettet kalles `ism`. For at algoritmen skal kunne kjøres må en punktbeskriver og treningsinformasjon passeres til objektet. Dette gjøres med metodene som er vist i koden nedenfor. Her passeres først punktbeskriveren `feature_estimator`, deretter de tre listene med treningsinformasjon. Til slutt settes parameteren, `ss_`, som spesifiserer størrelsen på samplingen av nøkkelpunkter fra punktskyene.

```
pcl::ism::ImplicitShapeModelEstimation<153, pcl::PointXYZ,
    pcl::Normal> ism;
ism.setFeatureEstimator(feature_estimator);
ism.setTrainingClouds (training_clouds);
ism.setTrainingNormals (training_normals);
ism.setTrainingClasses (training_classes);
ism.setSamplingSize (ss_);

pcl::ism::ImplicitShapeModelEstimation<153, pcl::PointXYZ,
    pcl::Normal>::ISMModelPtr ismModel =
    boost::shared_ptr<pcl::features::ISMModel>
       (new pcl::features::ISMModel);
ism.trainISM (ismModel);
```

Før algoritmen kan starte må det også opprettes et objekt for å lagre ISM-modellen. Opprettelsen av dette objektet kalt `ismModel`, vises i koden ovenfor. Når dette er gjort kan treningen startes ved å bruke metoden `trainISM()`.

Når treningen er ferdig kan testdelen av algoritmen kjøres. Før selve testingen kan begynne, må det regnes ut normalretninger for punktskyen som ISM-modellen skal testes mot. Punktskyen som skal testes her er scenen. Her regnes det ut normaler på samme måte som det ble gjort tidligere. Dette er vist i koden nedenfor. Normalene lagres i ≪PointCloud≫-strukturen `testing_normals`.

```
pcl::PointCloud<pcl::Normal>::Ptr testing_normals = (new
    pcl::PointCloud<pcl::Normal>)->makeShared ();
normal_estimator.setInputCloud (scene);
normal_estimator.compute (*testing_normals);
```

Når dette er gjort, brukes den samme ISM-estimatoren, `ism`, som ble opprettet tidligere til å utføre testingen. Metoden, `findObjects()`, brukes for å starte testalgoritmen. Denne kjøres med ISM-modellen, punktskyen, normalretningene og testklassenummeret som input. I dette prosjektet testes det mot kun en klasse. Verdien til `testing_class` er dermed 0. Det vil si at vi tester mot objektklasse nummer 0.

```
boost::shared_ptr<pcl::features::ISMVoteList<pcl::PointXYZ> >
    vote_list = ism.findObjects (
       ismModel,
       scene,
       testing_normals,
       testing_class);
```

Resultatet fra testalgoritmen er en liste av typen `ISMVoteList` kalt `vote_list`. Denne inneholder alle stemmene fra Hough-stemmingen. For å finne toppene brukes en metode som undertrykker elementer som ikke er toppunkter. Dette gjøres ved å kjøre metoden `findStrongestPeaks()` fra listen med stemmer, `vote_list`. Resultatet fra denne lagres i listen, `strongest_peaks`. Bruk av metoden er vist nedenfor.

```
double radius = ismModel->sigmas_[testing_class] * 10.0;
double sigma = ismModel->sigmas_[testing_class];
vote_list->findStrongestPeaks (strongest_peaks, testing_class,
    radius, sigma);
```

Metoden bruker størrelsen på objektet gitt av radiusen og variabelen `sigma` som brukes under trening av modellen. Disse hentes fra en liste kalt `sigmas_` for den aktuelle objektklassen, i den trente modellen `ismModel`. Verdien som hentes, er den samme som brukes ved vektingen $W_{lrn}(\lambda_{ij})$ fra ligning 4.9.2 i kapittel 4.9. Her er denne notert med tegnet σ. Denne defineres under treningen til å være 10% av størrelsen til objektet gitt av radiusen. Variabelen `radius` finnes dermed ved å gange `sigma` med 10, som vist i koden ovenfor.

For å hente ut de detekterte senterpunktene, kjøres en for-løkke over antallet detekterte topper i listen `strongest_peaks`. Her opprettes et punkt kalt `point`. For hvert toppunkt i `strongest_peaks` hentes x, y og z verdien fra toppunktet og legges i punktet `point`. Dette legges dermed i listen over detekterte senterpunkter, kalt `detected_centerpoints`. For-løkken som kjøres er vist nedenfor.

```
pcl::PointXYZ point;
for (size_t i_vote = 0; i_vote < strongest_peaks.size ();
    i_vote++)
{
       point.x = strongest_peaks[i_vote].x;
       point.y = strongest_peaks[i_vote].y;
       point.z = strongest_peaks[i_vote].z;
       detected_centerpoints.push_back(point);
}
```

Kapittel 6

Eksperimentelt og Resultater

I dette kapittelet blir det først gjennomgått noen forberedelser som utføres før Kinect kan brukes til skanning. Her beskrives oppsettet av Kinfu Large Scale og systemspesifikasjonene. Selve prosessen med skanning er også beskrevet. Deretter vises det hvordan etterprosessering av de skannede filene utføres. I den neste delen vises resultater fra gjenkjenning i punktskyer. Her demonstreres også forskjellsdeteksjon før det til slutt gjøres en sammenligning av tidsforbruket til de to gjenkjenningsalgoritmene.

6.1 Forberedelser

I denne seksjonen vil det bli gjennomgått hvordan skanning med Kinfu Large Scale utføres. Deretter vises en metode som kan brukes for å sette sammen og forenkle de skannede punktskyene.

6.1.1 Installering av Kinfu Large Scale

Kinfu Large Scale applikasjonen er ikke tilgjengelig i den siste versjonen av PCL. For å kunne bruke denne, må kildekoden til PCL Master lastes ned fra GitHub [29]. Når kildekoden er lastet ned brukes CMake til å sette opp et prosjekt i Visual Studio. På pointclouds.org er det beskrevet hvordan dette kan gjøres. [77] Når prosjektet er satt opp, kan det åpnes i Visual Studio. Det neste som må gjøres er å kompilere løsningene

pcl_kinfu_largeScale og
pcl_kinfu_largeScale_mesh_output.

De to applikasjonene vil da kompileres og være tilgjengelige i Release-mappen for prosjektet under navnene
pcl_kinfu_largeScale_release.exe og
pcl_kinfu_largeScale_mesh_output_release.exe

For at Kinfu Large Scale skal fungere må en ha installert OpenNI [37] og Nvidia CUDA [38]. Som nevnt tidligere er Kinfu Large Scale avhengig av at skjermkortet som applikasjonen skal kjøres på har støtte for CUDA.

6.1.2 Maskinvare

I dette prosjektet ble Xbox-versjonen av Kinect, Kinect for Xbox, brukt til skanning. Kinfu Large Scale ble kjørt på en bærbar datamaskin med følgende spesifikasjoner:

OS: Windows 8 64bit
CPU: Intel Core i7 - 3630CM, 2.4GHz
Minne: 8GB
Skjermkort: Nvidia Geforce GT 635M 2GB

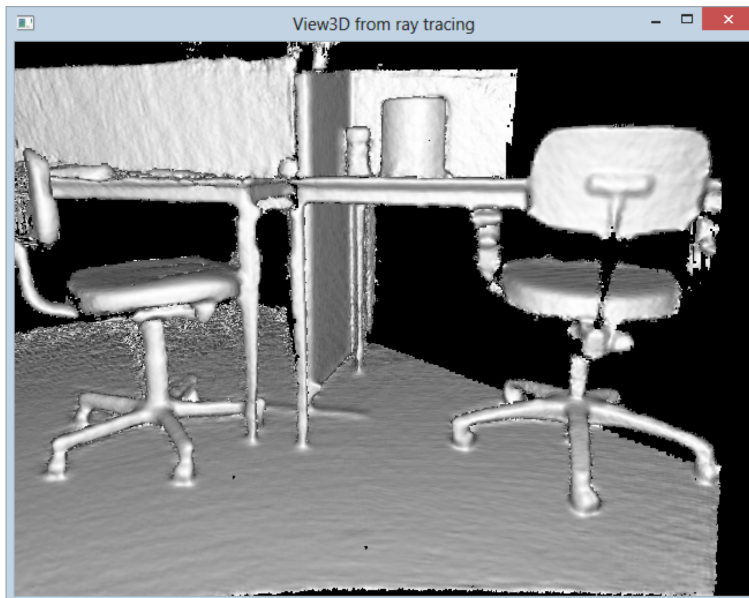
Under skanning ble de beste resultatene opplevd når datamaskinen var tilkoblet strømuttak. Dette kan komme av at datamaskinen setter noen begrensninger på maskinvaren under batteridrift for bedre strømsparing.

6.1.3 Skanning med Kinfu Large Scale

Kinfu Large Scale er en konsollapplikasjon. Dette betyr at applikasjonen startes fra kommandovinduet i Windows, CMD.exe. For å kjøre Kinfu Large Scale, må kommandovinduet åpnes for den adressen som applikasjonen ligger under. Dette kan gjøres ved holde inne skift og høyreklikke på mappen som inneholder applikasjonene og deretter velge «Åpne kommandovindu her».

For å starte Kinfu Large Scale skrives navnet på applikasjonen inn i kommandovinduet etterfulgt av enter. Det er mulig å sette programparametere ved oppstart. Tilgjengelige programparametere vises ved å skrive inn:

```
pcl_kinfu_largeScale_release.exe -h
```



Figur 6.1.1: Kinfu Large Scale: Vinduet viser progresjonen i skanningen. Det som vises er det nåværende innholdet i det gjeldende TSDF-volumet. Kameravinkelen i vinduet er et estimat av den fysiske kameravinkelen til Kinecten. Denne estimeres ut fra registrering i real-time.

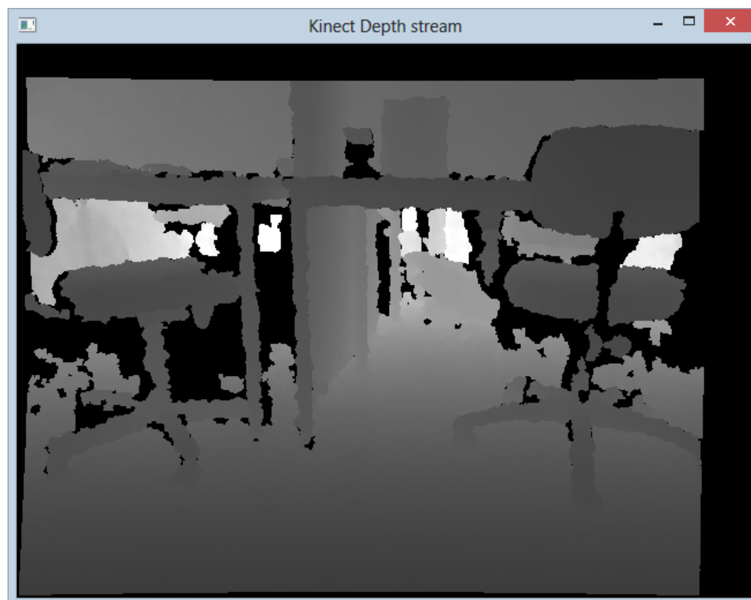
I dette prosjektet er parameterne `volume_size` og `shifting_distance` interessante. Parameteren `volume_size` har en standardverdi på 3 meter. Denne definerer lengden på kuben som ble omtalt i kapittel 3.3. Denne settes for å definere størrelsen på kuben som lagres i skjermkortets minne og derav oppløsningen på det som blir skannet. Parameteren `shifting_distance` har en standardverdi på 1.5 meter. Denne variabelen spesifiserer hvor langt senterpunktet av Kinectens kameravinkel kan flyttes før ≪skiftingen≫(Kapittel 3.3) trigges. Parametrene kan endres til andre verdier en standardverdiene ved oppstart. For eksempel kan `volume_size` settes til to meter og `shifting_distance` til 1 meter ved å skrive følgende i kommandovinduet:

```
pcl_kinfu_largeScale_release.exe --volume_size 2 --shifting_distance 1
```

Når Kinfu Large Scale har startet vises tre nye vinduer på skjermen i tillegg til kommandovinduet. Disse er vist i figurene 6.1.1, 6.1.2 og 6.1.3. Vinduet som vises oppe til venstre (Figur 6.1.1) i skjermen viser det nåværende innholdet i TSDF-volumet, som blir omtalt i kapittel 3.3). Dette viser progresjonen av skanningen i real-time.

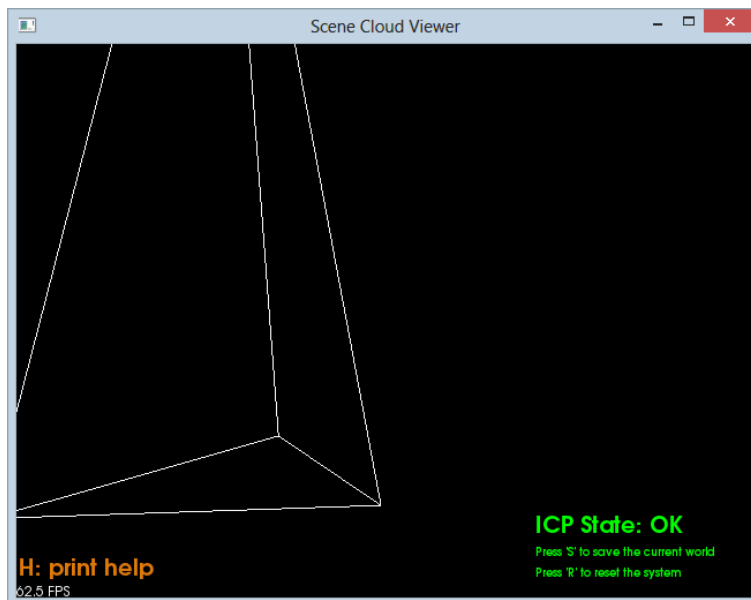
Vinduet som vises oppe til høyre (Figur 6.1.2) viser strømmen med dybdeinformasjon som kommer fra Kinecten. Her vil ting som står nærmere Kinecten være mørke samtidig som ting lenger vekk er lysere.

Det siste vinduet, som er vist i figur 6.1.3, viser tilstanden til registreringen.



Figur 6.1.2: Kinfu Large Scale: Vinduet viser strømmen med dybdeinformasjon som kommer fra Kinecten. Det er dette Kinecten ≪ser≫ til en hver tid.

Registreringen skjer ved bruk av ICP-algoritmen. Dersom tilstanden vises som ≪OK≫, betyr dette at ICP-algoritmen klarer å henge med under skanningen. Algoritmen forsøker å hele tiden justere dybdedataene som kommer fra Kinecten slik at disse lagres riktig i forhold til hverandre. Dersom algoritmen ikke lenger klarer å holde følge vil tilstanden skifte til ≪LOST≫. Den siste brukbare posisjonen, som Kinecten hadde, vises da i dette vinduet. Applikasjonen vil forsøke å gjenoppta skanningen dersom Kinecten flyttes tilbake til denne posisjonen. Det finnes en del kommandoer som kan gis til applikasjonen under kjøring. For å få opp en liste over tilgjengelige kommandoer kan en trykke på ≪h≫. Vinduet, som viser tilstanden til ICP-algoritmen, må være det aktive vinduet for at tastaturkommandoer skal virke.



Figur 6.1.3: Kinfu Large Scale: Vinduet viser tilstanden til registreringen av dybdeinformasjonen. Dersom algoritmen ikke klarer å registrere lenger vil Kinectens siste brukbare posisjon, vises i dette vinduet.

Skanningen starter når applikasjonen startes opp. Kinecten må da være rettet mot det som skal skannes. Når Kinecten flyttes bortover må dette skje forsiktig. Det er også en fordel å følge med på om den estimerte kameravinkelen (figur 6.1.1) er lik den reelle kameravinkelen (figur 6.1.2). Dersom det oppstår avvik mellom disse er dette et tegn på at registreringen sliter med å holde følge. Når skanningen er ferdig, gis kommandoen «s» på tastaturet. Det totale TSDF-volumet lagres da i samme mappen som applikasjonen ligger. Denne lagres med filnavnet `world.pcd`

6.1.4 Etterprosessering

Resultatet av skanning med Kinfu Large Scale er et TSDF-volum lagret i en fil kalt `world.pcd`. For at dette volumet skal kunne brukes som en punktsky, må det utføres litt etterprosessering. Applikasjonen Kinfu Large Scale Mesh Output konverterer og deler opp TSDF-volumet til mindre filer i filformatet .ply. Dette gjøres ved å åpne et kommandovindu, på samme måte som i forrige delkapittel. Når et kommandovindu er åpnet for mappen som inneholder applikasjonen, kan den kjøres ved å gi følgende kommando:

```
pcl_kinfu_largeScale_mesh_output_release.exe world.pcd --volume_size 2
```

Her spesifiseres parametrene `world.pcd` og `volume_size`. Den første gir fil-

navnet til TSDF-volumet som skal konverteres. Den siste spesifiserer hvilken `volume_size` som ble brukt under skanning. I dette tilfellet er `volume_size` satt til 2 meter. For å få riktig skalering på punktskyene, er det viktig at `volume_size` settes til rett verdi.

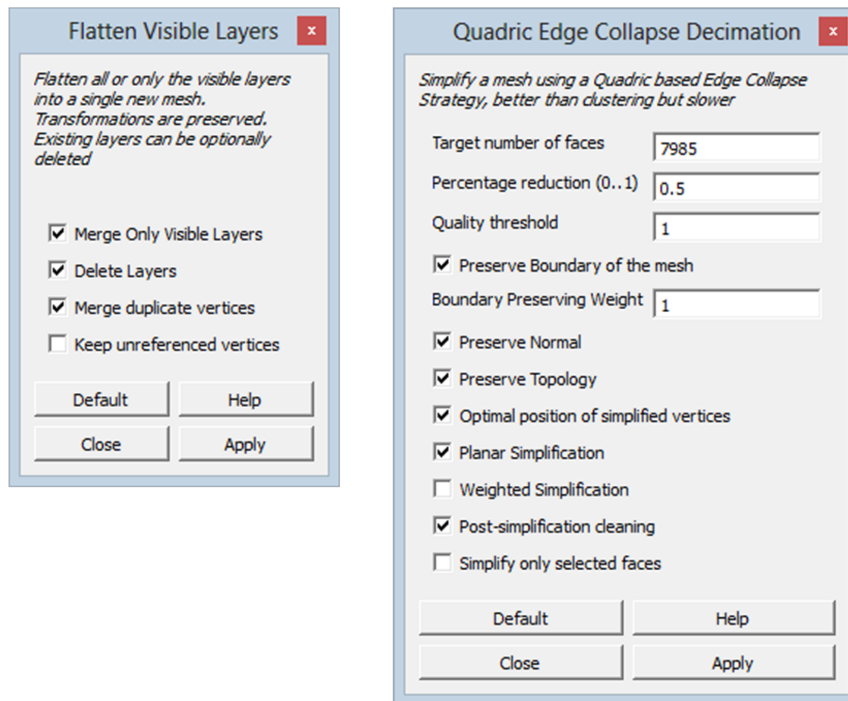
Under konverteringen, genereres en ply-fil for hver kube i det totale TSDF-volumet. Ply-filene blir lagret med filnavnene `mesh_1.ply`, `mesh_2.ply` osv. Disse lagres også i samme mappen som applikasjonen.

Forenkling og sammensetting av skannede punktskyer

Den skannede scenen er nå delt opp i flere ply-filer som i utgangspunktet inneholder for mange punkter for å kunne behandles. For å sette sammen og forenkle ply-filene som ble generert fra Kinfu Large Scale Mesh Output, brukes MeshLab [46]. Etter at MeshLab er åpnet kan ply-filene importeres ved å velge *File → Import Mesh...*, eller ved å dra filene fra mappen og direkte inn i MeshLab-vinduet.

- **Sammensetting:** For å sette sammen filene velges *View → Show Layer Dialog*. Dette åpner liste med oversikt over de importerte filene. Ved å høyreklikke i listen og velge *Flatten Visible Layers*. I dialogboksen som åpnes, klikk *Apply* med standardvalgene, som vist i 6.1.4.

- **Forenkling:** Skanningen resulterer i et stort antall punkter. For å redusere antallet punkter i filen kan punkter, som enten er duplikater eller som ligger nær hverandre, fjernes. Dette gjøres ved å velge *Filters → Remeshing, Simplification and Reconstruction → Quadric Edge Collapse Decimation.* Vinduet som åpnes vises til høyre i figur 6.1.4. Her settes settes *Percentage Reduction* til 0.5 og *Quality Threshold* til 1. Deretter sjekkes alle tekstboksene utenom *Weighted Simplification* og *Simplify Only Selected Faces.*



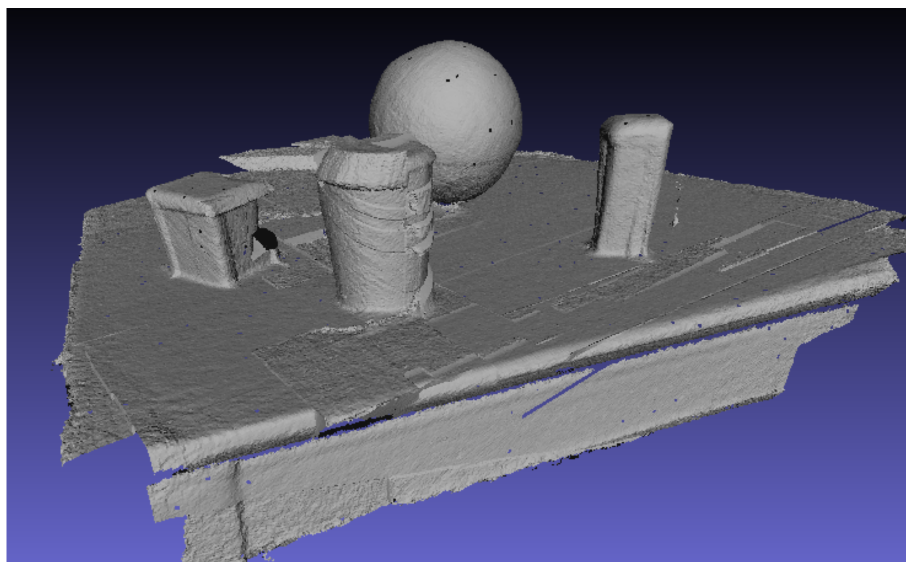
Figur 6.1.4: MeshLab: Vinduet til venstre viser *Flatten Visible Layers*-dialogboksen. Vinduet til høyre viser dialogboksen for *Quadric Edge Collapse Decimation*.

Når både sammensetting og forenkling er utført kan filen eksporteres i ply-format fra MeshLab ved å klikke *File* → *Export Mesh As...*, og så velge hvor den skal lagres.

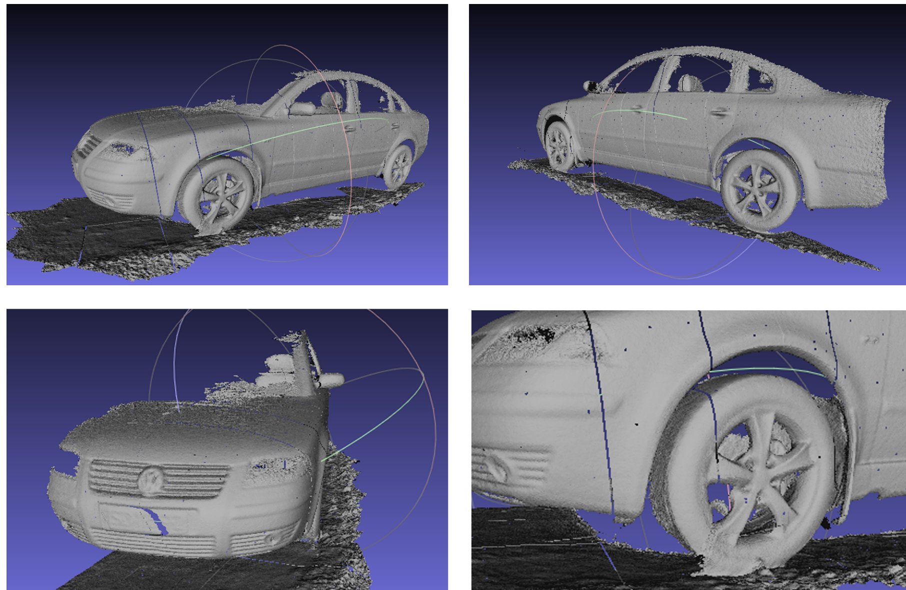
6.2 Testing og resultater

I denne seksjonen testes gjenkjenning med algoritmene ≪Correspondence Grouping≫ (CG) og ≪Implicit Shape Model≫ (ISM). Disse blir gjennomgått i kapitlene 4.8 og 4.9. I noen tilfeller ved bruk av CG-algoritmen kan resultatene finjusteres med ≪Iterative Closest Poin≫, som er omtalt i kapittel 4.7.

Her brukes to scener til testing av algoritmene. Begge scenene er skannet med Kinfu Large Scale. I den ene scenen er forskjellige geometriske figurer plassert på et bord. Den andre scenen inneholder en bil. I figur 6.2.1, vises de skannede geometriske figurene. Figur 6.2.2 viser en Volkswagen Passat. Under skanning ble `volume_size` satt til å være 0.8 meter for figurene og 1.5 meter for bilen. De to punktskyene representerer to forskjellige kategorier av størrelsesorden. Scenen med de geometriske figurene inneholder objekter med små størrelser i forhold til scenen med bilen.



Figur 6.2.1: Figuren viser geometriske figurer på et bord, skannet med Kinfu Large Scale. Her er `volume_size` satt til å være 0.8 under skanning.



Figur 6.2.2: Denne figuren viser en Volkswagen Passat, skannet med Kinfu Large Scale. Her er `volume_size` satt til 1.5 meter. I figuren er punktskyen visualisert med programmet MeshLab.

I testene i kapittel 6.2.1, utføres gjenkjenning mellom to skannede punktskyer, både med CG-algoritmen og med ISM-algoritmen. De skannede punktskyene avbilder de samme objektene men er skannet på ulike tidspunkt. Deretter, i kapittel 6.2.2, testes det gjenkjenning mellom objekter fra tredimensjonale CAD[1]-modeller og skannede punktskyer. CAD-modellene er tredimensjonale modeller som er designet med dataprogrammer som for eksempel Blender [47] eller SketchUp [48]. Deteksjon av forskjeller mellom to punktskyer testes i 6.2.3. Til slutt testes tidsforbruket til de to algoritmene i 6.2.4. Alle testene utføres med programmet Point Cloud Recognition Tool. Punktskyene som er brukt i dette kapittelet er vedlagt på CD-platen.

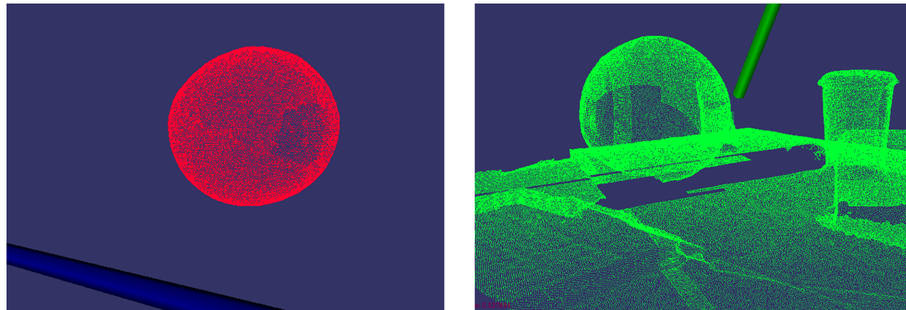
[1] Computer-aided Design

6.2.1 Gjenkjenning mellom to skannede punktskyer

I de følgende testene brukes to skannede punktskyer av de samme objektene til å teste CG-algoritmen og ISM-algoritmen.

Gjenkjenning av kule med ≪Correspondence Grouping≫

I figur 6.2.3 vises punktskyene som skal brukes i denne testen. De to punktskyene i figuren er skannet av den samme scenen som er vist i figur 6.2.1. I punktskyen til venstre er alt utenom kulen segmentert bort. Denne brukes som modell i denne testen. Punktskyen til høyre brukes som scene i denne testen. Denne inneholder litt støy og er ufullstendig noen steder.

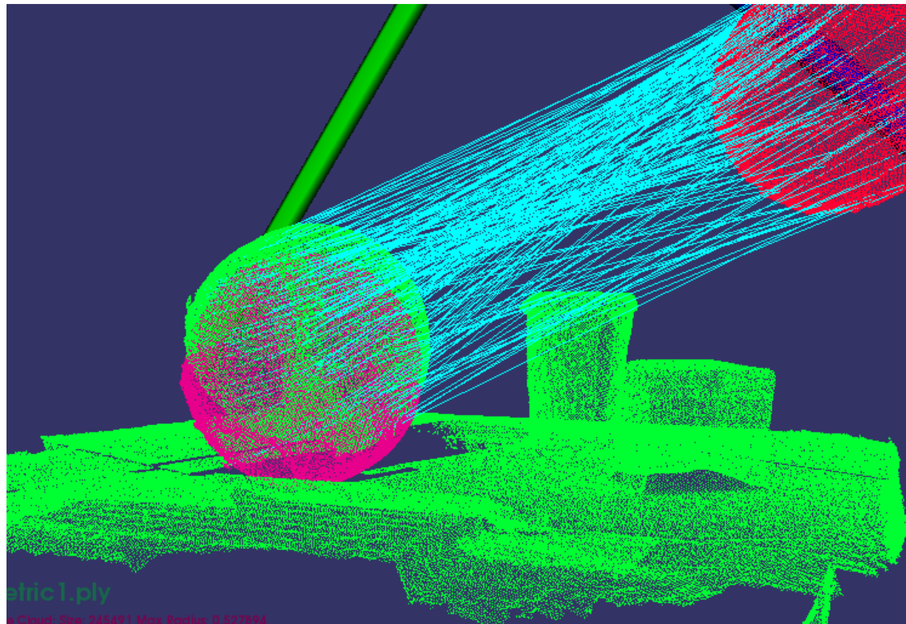


Figur 6.2.3: Til venstre vises modellen som det skal søkes etter. Til høyre vises scenen som det skal søkes i.

Parameter	Verdi
Model sr:	0.01 [m]
Scene sr:	0.01 [m]
Ref.frame r:	0.03 [m]
Descr. r	0.03 [m]
Cluster size	0.05 [m]
Cluster thres	35

Tabell 6.2.1: Tabellen viser parametrene brukt i testen.

For gjenkjenning med CG-algoritmen, brukes parametrene som er vist i tabell 6.2.1. I figur 6.2.4 vises resultatet fra CG-algoritmen. Her vises de detekterte korrespondansene mellom modellen og scenen med turkise linjer. Den rosa punktskyen viser den estimerte transformasjonen til modellen. Den estimerte transformasjonen og antallet korrespondanser, skrives ut i kommandovinduet. Dette er vist i figur 6.2.5. Det viser seg at det detekteres 264 korrespondanser mellom de to punktskyene, noe som gir en stor sikkerhet for et riktig resultat.



Figur 6.2.4: Den estimerte transformasjonen og antallet detekterte korrespondanser.

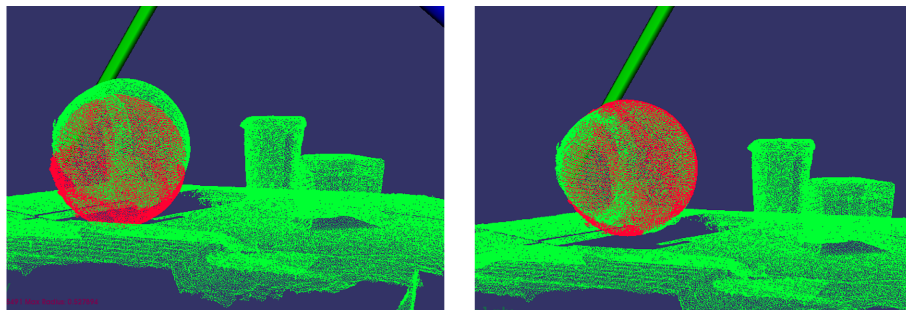
```
Showing transformation 0of1:
    Correspondences belonging to this instance: 264

        |  0.576  0.512  0.637 |
    R = | -0.818  0.351  0.456 |
        |  0.010 -0.784  0.621 |

    t = < -0.094, 0.253, 0.162 >
```

Figur 6.2.5: Her vises den estimerte transformasjonen til modellen. 264 korresponderende punktbeskrivelser er brukt under estimeringen.

For å finjustere resultatet brukes algoritmen kalt ≪Iterative Closest Point≫ (ICP), omtalt i kapittel 4.7. Resultatet av denne vises i figur 6.2.6. Ved å bruke denne, forbedres resultatet. Den estimerte transformasjonen fra ICP-algoritmen skrives ut i kommandovinduet. Dette er vist i figur 6.2.7.



Figur 6.2.6: Til venstre vises modellen, flyttet til den estimerte posisjonen fra 6.2.5. Til høyre vises resultatet, justert med ICP-algoritmen.

```
Model has converged: 1
Fitness score: 0.000233974
Transformation matrix:

        |  0.999 -0.053 -0.011 |
    R = |  0.054  0.998  0.017 |
        |  0.010 -0.018  1.000 |

    t = < 0.024, -0.059, -0.011 >
```

Figur 6.2.7: Figuren viser den estimerte transformasjonen som ble brukt under justeringen av resultatet.

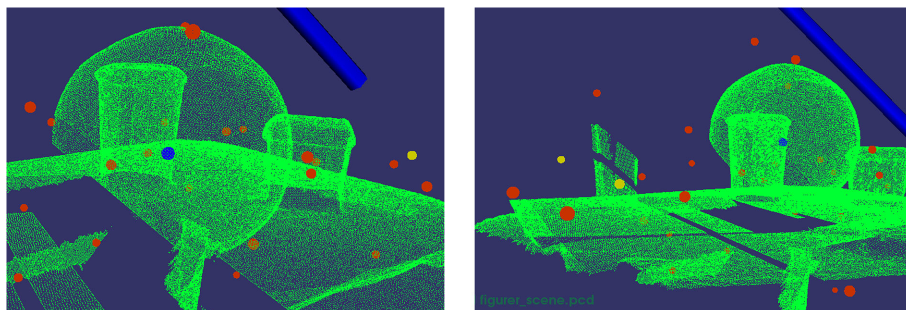
Gjenkjenning av kule med «Implicit Shape Model»

I denne testen brukes de samme punktskyene som fra forrige test. Disse er vist i figur 6.2.3. «Implicit Shape Model»-algoritmen (ISM) trenes opp med modell-punktskyen og detekterer mulige senterpunkter i scene-punktskyen. Parametrene brukt i denne testen er vist i tabell 6.2.6.

Parameter	Verdi[m]
Sampling s:	0.03 [m]
Norm. r:	0.03 [m]
Descr. r:	0.06 [m]
Thresh.	0.01

Tabell 6.2.2: Tabellen viser parametrene brukt i testen.

Resultatet fra ISM-algoritmen er vist i figur 6.2.8. Den blå prikken viser punktet som fikk mest stemmer under Hough-stemmingen. Gule prikker viser punkter, som fikk et antall stemmer, som gir en tetthet høyere enn tettheten spesifisert av parameteren *Thresh*, i tabell 6.2.6. Røde prikker har en tetthet som er lavere enn *Thresh*. Senterpunktet i kulen ble detektert som det beste treffet, selv om kulen i scenen var ufullstendig.

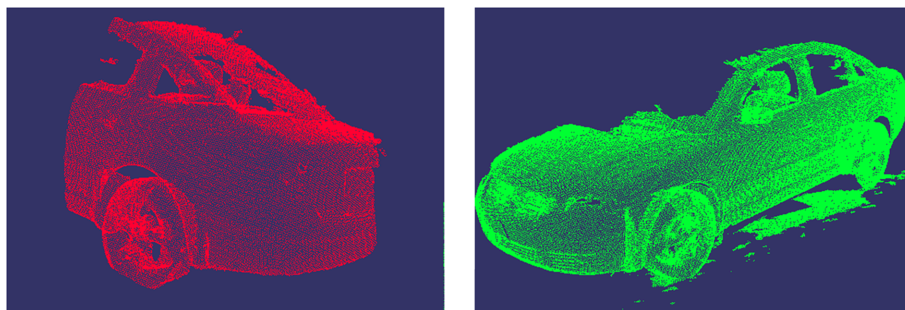


Figur 6.2.8: Resultatet fra ISM-algoritmen vist fra to kameravinkler.

Resultatet i figur 6.2.8 viser at ISM-algoritmen estimerer korrekt senterpunkt modell-punktskyen i scene-punktskyen.

Gjenkjenning av bil-segment med ≪Correspondence Grouping≫

I denne testen brukes punktskyer skannet med Kinfu Large Scale på to forskjellige tidspunkt. Begge punktskyene inneholder deler av en Volkswagen Passat. Den ene punktskyen inneholder mer av fronten og den andre inneholder mer av bakenden til bilen. Den punktskyen som inneholder mer av bakenden, er segmentert som vist i figur 6.2.9. Begge punktskyene er filtrert med et VoxelGrid-filter med voxelstørrelse på 0.01 meter. Dette gjøres for å redusere antallet punkter.

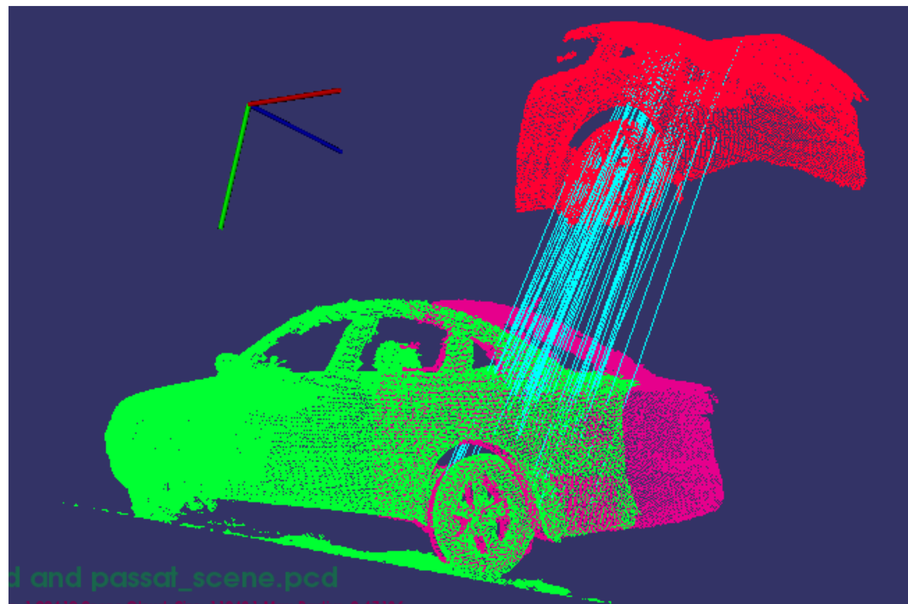


Figur 6.2.9: Til venstre vises modell-punktskyen. Punktskyen til høyre er scenen.

I denne testen brukes ≪Correspondence Grouping≫ med parametere som vist i tabell 6.2.3. Resultatet av CG-algoritmen er vist i figur 6.2.10. Dette resultatet var et av 17 detekterte forekomster av modellen i scenen. Av de 17 bar kun dette resultatet var det riktige. Det ble detektert 60 korrespondanser for denne forekomsten. Den estimerte transformasjonen er vist i figur 6.2.11. I dette tilfellet inneholder modell-punktskyen mange punkter som ikke finnes i scene-punktskyen. På grunn av dette kan ikke ≪Iterative Closest Point≫ brukes. Dersom denne brukes, forverres resultatet.

Parameter	Verdi
Model sr:	0.03 [m]
Scene sr:	0.03 [m]
Ref.frame r:	0.1 [m]
Descr. r	0.1 [m]
Cluster size	0.09 [m]
Cluster thres	5

Tabell 6.2.3: Tabellen viser parametrene brukt i testen.



Figur 6.2.10: Figuren viser resultatet fra CG-algoritmen. Her er modellen vist i rødt, scenen i grønt og den estimerte transformasjonen til modellen i rosa. De turkise linjene viser korrespondansene som ble brukt.

```
Showing transformation 10of17:
    Correspondences belonging to this instance: 60

        |  0.983 -0.025 -0.181 |
    R = | -0.019  0.971 -0.238 |
        |  0.182  0.237  0.954 |

    t = < -0.546, 1.640, -0.996 >
```

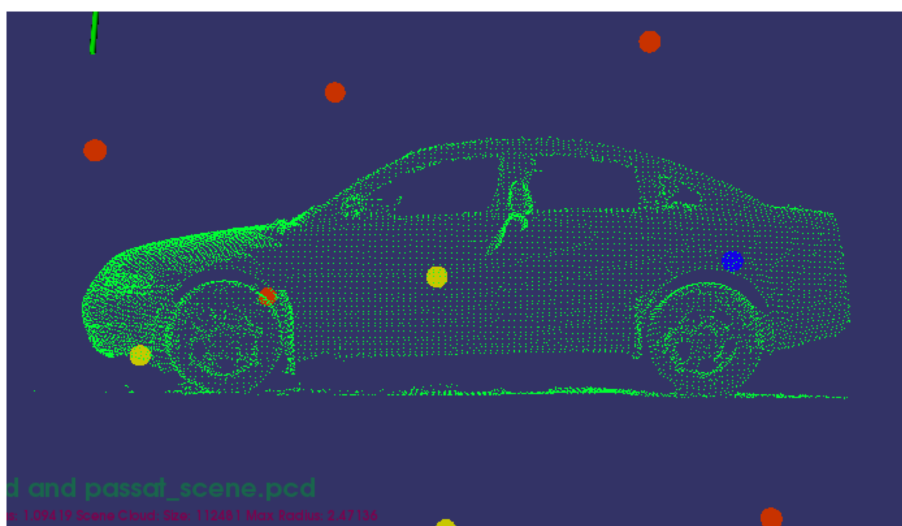
Figur 6.2.11: Her vises den estimerte transformasjonen til modellen. 60 korresponderende punktbeskrivelser ble brukt for å estimere transformasjonen.

Gjenkjenning av bil-segment med ≪Implicit Shape Model≫

I denne testen brukes de samme punktskyene som vises i figur 6.2.9. Her filtreres begge punktskyene med et VoxelGrid-filter med en voxelstørrelse på 0.03 meter. Dette gjøres for å redusere antallet punkter. Modell-punktskyens senterpunkt estimeres ved bruk av ISM-algoritmen. Parametrene brukt i denne testen er vist i tabell 6.2.4. Resultatet er vist i 6.2.12. Den blå prikken i figuren viser det beste treffet for et mulig senterpunkt for modellen i scenen. Dette er et godt estimat av senterpunktet til modellen.

Parameter	**Verdi[m]**
Sampling s:	0.05 [m]
Norm. r:	0.25 [m]
Descr. r:	0.25 [m]
Thresh.	0.02

Tabell 6.2.4: Tabellen viser parametrene brukt i testen.



Figur 6.2.12: Resultat av gjenkjenning mellom bil-segment og scene med ≪Implicit Shape Model≫. Den blå prikken indikerer det beste treffet for et mulig senterpunkt.

6.2.2 Gjenkjenning mellom CAD-modell og skannet punktsky

I dette kapittelet ble det brukt CAD-modeller for gjenkjenning punktskyer fra skannede scener. Punktskyene som brukes her er de samme som er brukt i testene i det forrige kapittelet. En av modell-punktskyene ble laget med programmet Blender. Den andre modell-punktskyen er fra en modell lastet ned fra 3D warehouse i programmet SketchUp. [78] Fra denne er det hentet ut et hjul-segment som brukes i gjenkjenningen.

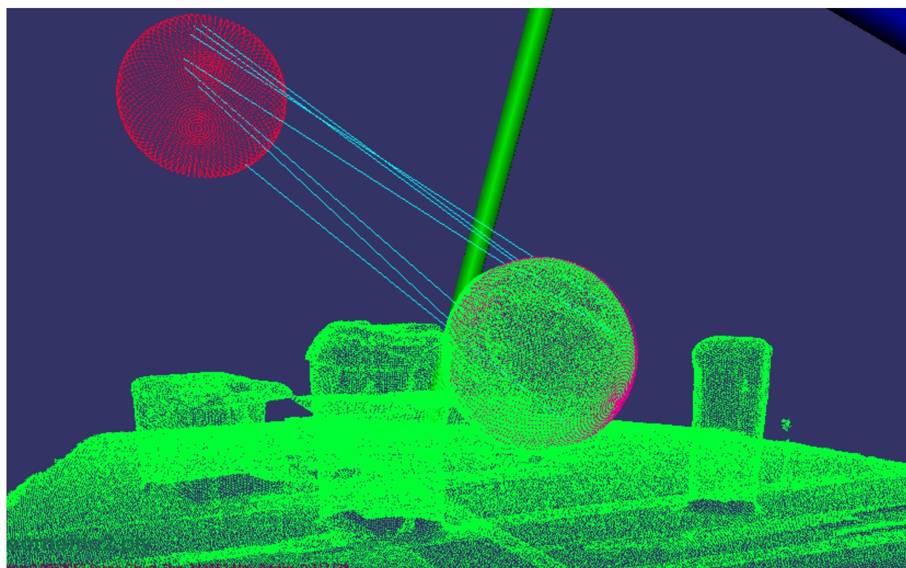
Gjenkjenning av kule med ≪Correspondence Grouping≫

I denne testen ble det forsøkt å kjenne igjen en CAD-modellert kule i scenen med de geometriske figurene i figur 6.2.1. Kulen er laget med programmet Blender og har en radius på 9.5 centimeter. Scene-punktskyen stammer fra den samme scenen som ble skannet og brukt i kapittel 6.2.1. Her er derimot scenen fullstendig. Parametrene som ble brukt er vist i tabell 6.2.5.

Parameter	**Verdi**
Model sr:	0.01 [m]
Scene sr:	0.02 [m]
Ref.frame r:	0.03 [m]
Descr. r	0.02 [m]
Cluster size	0.03 [m]
Cluster thres	5

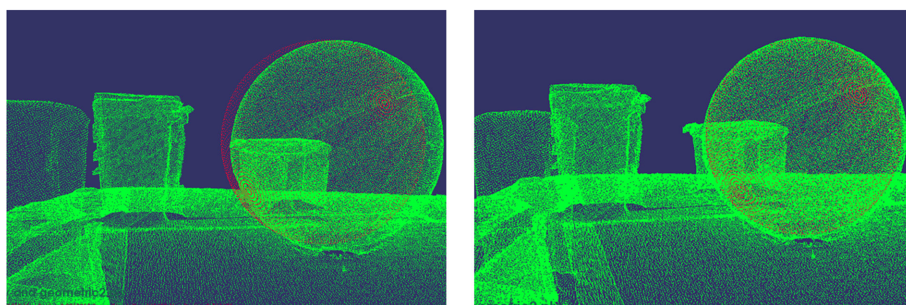
Tabell 6.2.5: Tabellen viser parametrene som ble brukt i testen.

Resultatet av gjenkjenningen er vist i figur 6.2.13. Figuren viser et riktig resultat, men sammenlignet med testen gjort i kapittel 6.2.1, ble færre korrespondanser detektert her. I kapittel 6.2.1 ble så mange som 264 korresponderende punktbeskrivelser detektert. Her ble bare 7 korrespondanser detektert.



Figur 6.2.13: Figuren viser resultatet av gjenkjenning med CG-algoritmen. Modell-punktskyen er en kule laget med programmet Blender. Scene-punktskyen er scannet med Kinfu Large Scale.

Etter at modellen er detektert i scenen, flyttes modellen til den estimerte posisjonen. Deretter brukes «Iterative Closest Point» for å finjustere resultatet. Dette er vist i figur 6.2.14.



Figur 6.2.14: Bildet til venstre viser punktskyen før ICP-algoritmen ble kjørt. Bildet til høyre viser resultatet etter kjøring.

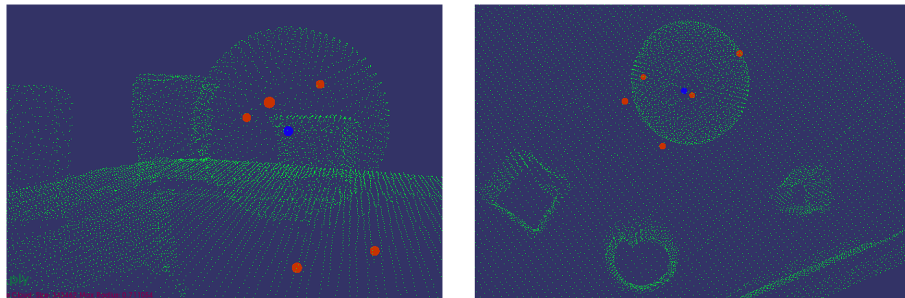
Gjenkjenning av kule med «Implicit Shape Model»

Her brukes de samme testsettene som ble brukt i forrige avsnitt. Den eneste forskjellen er at scene-punktskyen filtreres med et VoxelGrid-filter med voxelstørrelse på 0.01. Dette gjøres for å redusere antallet punkter i punktskyen og dermed reduser tidsforbruket til algoritmen. Antallet reduseres fra 343462 punkter til 28730 punkter. Her brukes gjenkjenning med «Implicit Shape Model» og parametrene for algoritmen er vist i 6.2.6.

Parameter	**Verdi[m]**
Sampling s:	0.01 [m]
Norm. r:	0.02 [m]
Descr. r:	0.03 [m]
Thresh.	0.02

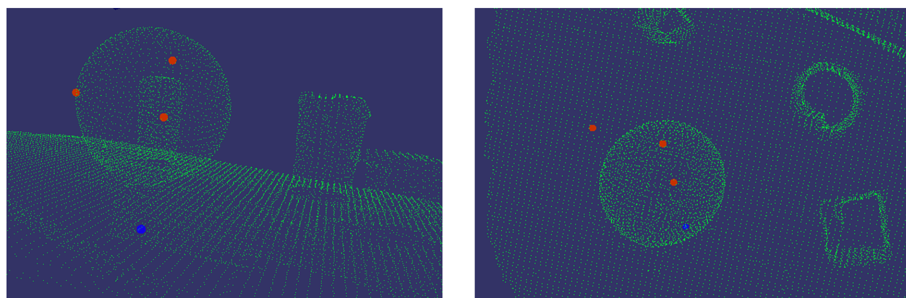
Tabell 6.2.6: Tabellen viser parametrene for ISM-algoritmen.

I figur 6.2.15 vises resultatet. Det beste treffet for et mulig senterpunkt, er indikert med en blå prikk. Resultatet vist i figuren er et av de beste resultatene fra flere gjennomkjøringer.



Figur 6.2.15: Figuren viser resultatet fra ISM-algoritmen fra to forskjellige kameravinkler.

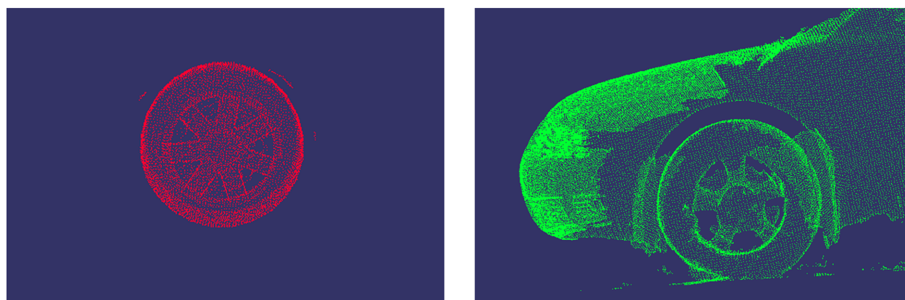
Det er verdt å merke seg at resultatet i dette tilfellet ikke er konstant og at det kan hende at feildeteksjoner av senterpunkter forekommer. Et eksempel på en feildeteksjon er vist i figur 6.2.16. Her blir det beste treffet detektert til å være utenfor kulen. Alle treffene er derimot konsentrert rundt kulen.



Figur 6.2.16: Figuren viser resultatet fra en annen gjennomkjøring av ISM-algoritmen med de samme parametrene. Det beste treffet blir detektert til å være utenfor kulen. Treffene er likevel konsentrert i området rundt kulen.

Gjenkjenning av hjul med ≪Correspondence Grouping≫

I denne testen brukes den skannede punktskyen av bilen som scene. Modellen er et hjul, segmentert ut fra en CAD-modell av en bil. I scenen-punktskyen er litt av bakken, som bilen står på, segmentert bort. Det er også brukt et VoxelGrid-filter med voxelstørrelse på 0.01 meter. CAD-modellen er hentet fra 3D Warehouse i SketchUp. [78] Modellen ble først importert i Blender for å øke antallet punkter. Dette ble gjort med ≪Subdivision Surface≫-funksjonen. Deretter er hjulet segmentert ut fra modellen og filtrert med et VoxelGrid filter med voxelstørrelse på 0.01. Dette er gjort for å få en jevnere avstand mellom punktene i modellen. Modellen og scenen er vist i figur 6.2.17. Legg merke til at hjulfelgen i modellen er forskjellig fra hjulfelgen i scenen. Her kunne resultatet ha blitt justert litt ved å bruke ≪Iterative Closest Point≫. På grunn av at hjulfelgene er ulike, ville dette hatt liten effekt.

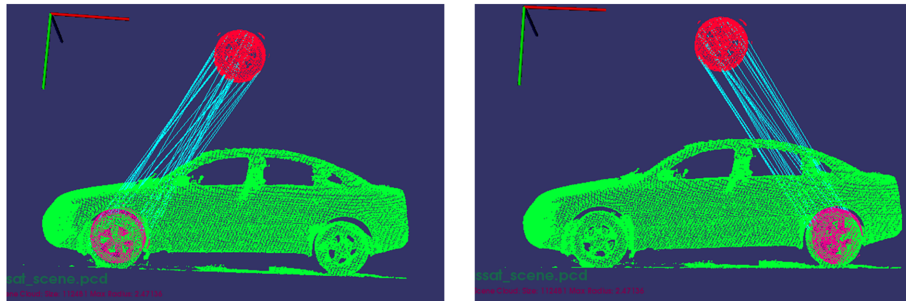


Figur 6.2.17: Figuren viser modell-punktskyen og scene-punktskyen som er brukt i denne testen.

Parameter	Verdi
Model sr:	0.01 [m]
Scene sr:	0.03 [m]
Ref.frame r:	0.07 [m]
Descr. r	0.06 [m]
Cluster size	0.1 [m]
Cluster thres	35

Tabell 6.2.7: Tabellen viser parametrene som ble brukt i testen.

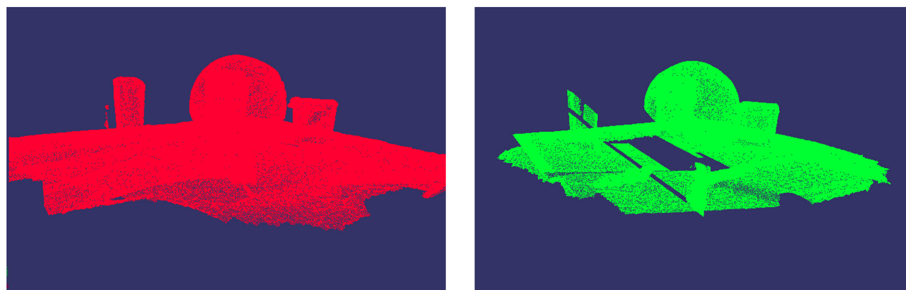
Ved å kjøre CG-algoritmen med de oppgitte parametrene estimeres fem mulige transformasjoner for modellen. De to beste treffene er vist i figur 6.2.18.



Figur 6.2.18: Her vises de to beste resultatene fra gjenkjenningen mellom de to punktskyene i denne testen. Den røde punktskyen er modellen. Og de rosa punktskyene er modellen plassert i de estimerte posisjonene. De turkise linjene viser korrespondansene som er detektert i de to tilfellene.

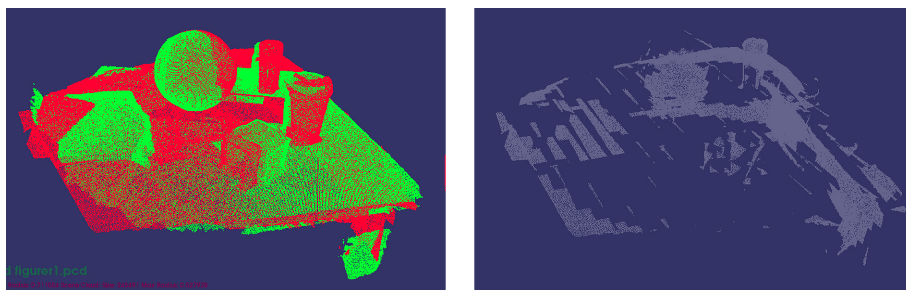
6.2.3 Deteksjon av forskjeller mellom punktskyer

I dette kapittelet demonstreres deteksjon av forskjeller mellom punktskyer med algoritmen beskrevet i kapittel 4.2.1. Punktskyene som er brukt er skannet med Kinfu Large Scale. Scenen består av geometriske figurer på et bord. Punktskyene har noen få forskjeller som er synlige i 6.2.19. Her har modell-punktskyen flere punkter som ikke eksisterer i scene-punktskyen. Først brukes ≪Correspondence Grouping≫ for å legge den ene skyen oppå den andre. Deretter finjusteres resultatet med ≪Iterative Closest Point≫.



Figur 6.2.19: Her vises punktskyene som ble brukt i denne testen. Til venstre vises modellen og til høyre vises scenen. Modell-punktskyen har punkter som ikke finnes i scene-punktskyen.

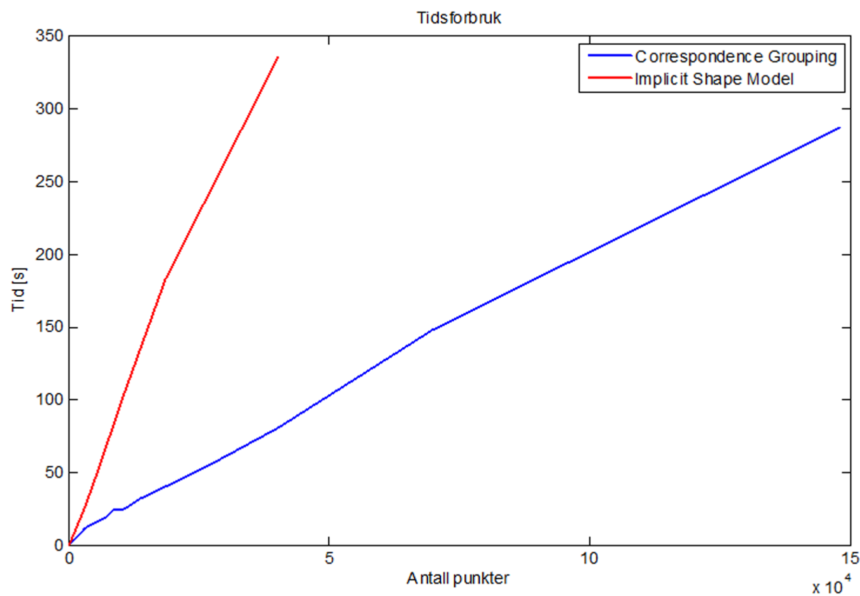
Når punktskyene er lagt oppå hverandre, kan algoritmen for forskjellsdeteksjon kjøres. Her brukes en octree-oppløsning på 0.02 meter. Dette er sidelengden på den minste voxelen i trestrukturen. Resultatet er vist i 6.2.20. I algoritmen tas det utgangspunkt i modell-punktskyen. Punkter som blir detektert som forskjeller vises med lyseblått. Dette er punkter som finnes i modell-punktskyen men som ikke finnes i scene-punktskyen innenfor området til en voxel med sidelengde 0.02 meter. Resultatet viser de mange punktene som mangler i scene-punktskyen.



Figur 6.2.20: Til venstre vises de to punktskyene lagt oppå hverandre. Til høyre vises de detekterte forskjellene.

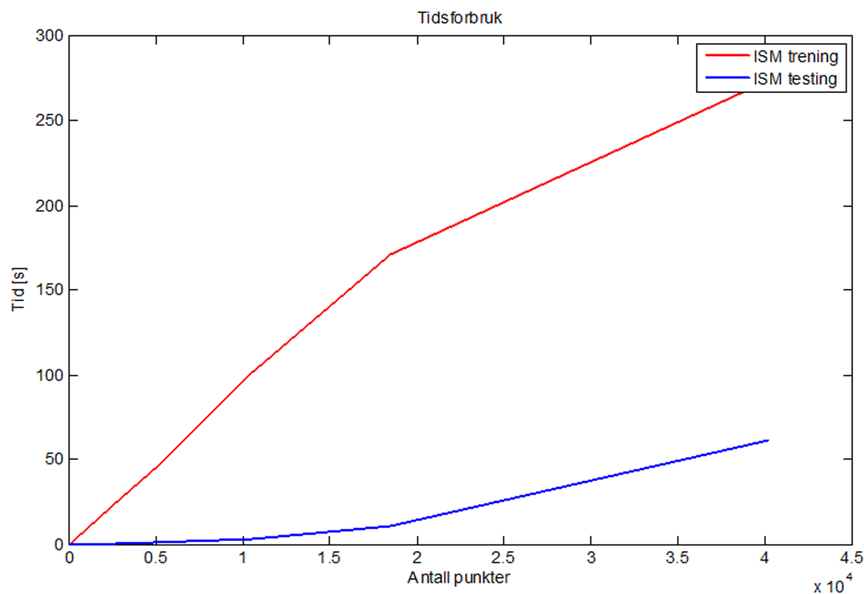
6.2.4 Sammenligning av tidsforbruk

I dette kapittelet testes og sammenlignes tidsforbruket til de to algoritmene ≪Correspondence Grouping≫ og ≪Implicit Shape Model≫. I testen ble det brukt modell-punktsky og scene-punktsky med likt antall punkter. Deretter ble antallet punkter i hver punktsky økt for å teste algoritmene. Algoritmens parametre ble satt til å være konstante under testene. Tidsforbruket til de to algoritmene er vist i figur 6.2.21.



Figur 6.2.21: Sammenligning av tidsforbruket til de to algoritmene ≪Correspondence Grouping≫ (blå) og ≪Implicit Shape Model≫ (rød).

Fra figur 6.2.21, blir det klart at ISM-algoritmen har betydelig større tidsforbruk enn CG-algoritmen. På grunn av måten ISM-algoritmen er implementert på, kan også tidsforbruket for trening og for testing sammenlignes. Dette er vist i figur 6.2.22.



Figur 6.2.22: Sammenligning av tidsforbruket fra treningen (rød) og testingen (blå) som skjer i ≪Implicit Shape Model≫.

Fra grafen figur 6.2.22 blir det klart at det er treningen i algoritmen som står for den største delen av tidsforbruket. Tidsforbruket for testingen begynner først å øke etter at størrelsen på punktskyene har nådd rundt 20000 punkter.

På grunn av måten ≪Correspondence Grouping≫-algoritmen er implementert på, blir det vanskelig å teste tidsforbruket på samme måte som i figur 6.2.22. Dette er fordi treningen og testingen skjer mer eller mindre parallelt. Fra beskrivelsen av algoritmen i figur 4.8.1 i kapittel 4.8 kan det derimot være rimelig å anta at det er testingen som står for det største tidsforbruket. Her blir treningen (offline-steget) beskrevet som en kortere prosess enn testingen (online-steget).

Kapittel 7

Diskusjon og Konklusjon

7.1 Skanning

I denne oppgaven blir skanning av punktskyer gjort med Kinect for Xbox og applikasjonen Kinfu Large Scale. Denne applikasjonen ble valgt fordi den forenkler skanneprosessen ved at den setter sammen dybdedata fra Kinect-sensoren i real-time. Bilder av to skannede punktskyer er vist i kapittel 6.2, i figurene 6.2.1 og 6.2.2. I Kinfu Large Scale kan parameteren `volume_size` benyttes for å stille størrelsen på TSDF-volumet som er omtalt i kapittel 3.3. Denne parameteren stiller sidelengden på volumet og er satt til en standardverdi på 3 meter.

Det viser seg at det er mulig å oppnå bedre kvalitet på skannede datasett ved å sette denne parameteren lavere. Samtidig som kvaliteten øker, øker også minneforbruket for hvert TSDF-volum. Under ≪skiftingen≫ (kapittel 3.3), lagres det gjeldende TSDF-volumet i CPUen. Etter en stund med skanning klarer ikke applikasjonen å registrere dybdedataene lenger og ICP-tilstanden, omtalt i kapittel 6.1.3, skifter til ≪LOST≫. Med `volume_size` satt til 1.5 meter, skjer dette vanligvis etter 10 til 15 ≪skift≫. Med lavere verdier av `volume_size` skjer dette tidligere. Det er mulig at dette skjer fordi minneforbruket til det totale TSDF-volumet blir for stort. Best resultat oppnås ved å sette variabelen `volume_size` til en verdi som er tilpasset scenen som skal skannes. I de to punktskyene i kapittel 6.2, ble verdien 0.8 meter brukt for figurene i figur 6.2.1 og 1.5 meter for bilen i figur 6.2.2.

Som nevnt i kapittel 2.1, er rekkevidden til Kinect for Xbox, som er brukt i denne oppgaven, oppgitt til å være 800 mm til 4000 mm. Dette begrenser alternativene for innstilling av `volume_size`-variabelen. Under skanning ble verdier fra 0.7 meter og opp til 3 meter testet. En verdi på 0.7 meter var i minste laget. Da måtte sensoren holdes ganske nær scenen for at dybdeinformasjon i det

hele tatt skulle bli fanget opp. Samtidig kunne den ikke holdes for nær siden en da ville havne utenfor rekkevidden til til sensoren. Med `volume_size` satt til 3 meter ble dybdeinformasjonen litt unøyaktig og det ble vanskeligere for applikasjonen å holde ICP-tilstanden til ≪OK≫. Under skanning av bilen i figur 6.2.2, skiftet ICP-tilstanden til ≪LOST≫ etter 15 ≪skift≫. Dette setter en begrensning for hvor store objekter som kan skannes med Kinfu Large Scale. For å skanne større scener, må skanningen gjøres i flere omganger og punktskyene må deretter settes sammen manuelt eller ved bruk av registreringsalgoritmer. På PCL sine hjemmesider finnes mer informasjon om registrering av punktskyer. [79]

Under forsøkene gjort med Kinfu Large Scale ble det gjort noen observasjoner om hvordan best mulig resultat kan oppnås. Applikasjonen bruker mye minne så det er en fordel å lukke alle andre programmer under skanning. Dersom applikasjonen kjøres fra en bærbar pc bør det brukes strømkabel. Dette for å unngå eventuelle tiltak for strømsparing på datamaskinen. Med høyere bilderate under skanning er det lettere å oppnå et godt resultat ettersom registreringen går fortere. Skanning bør gjøres innendørs, etter solnedgang eller i overskyet vær ettersom mye sollys skaper en del støy for det infrarøde mønsteret som Kinecten bruker. Glassflater og sorte områder er vanskelige å skanne. I glassflatene slippes mye av det infrarøde lyset igjennom samtidig som sorte områder absorberer mye av det infrarøde lyset. Store flater med lite detaljer er vanskelige å skanne ettersom registreringen trenger referansepunkter for å kunne sette sammen dybdedataene. Det er også vanskelig å få scannet scener med små detaljer korrekt. Dette problemet framgår litt av figurene i figur 6.2.1. Dette er i utgangspunktet feilfrie geometriske figurer. Under skanning oppstår det en del deformering. Dette oppstår sannsynligvis som følge av filtrering og feilregistrering. Dette problemet er mindre ved større scener med færre små detaljer.

7.2 Gjenkjenning

I kapitlene 6.2.1 og 6.2.2 er det testet gjenkjenning mellom to skannede punktskyer og mellom CAD-modell og skannet punktsky. Både ≪Correspondence Grouping≫ og ≪Implicit Shape Model≫ har blitt testet og gitt gode resultater i begge tilfellene. Generelt sett er de beste resultatene gjort mellom to skannede punktskyer. Gjenkjenning mellom CAD-modeller og skannede punktskyer er litt vanskeligere å få til. Dette kommer av at deskriptorene i begge algoritmene prøver å beskrive lokale punkter der området rundt punktet blir brukt i beskrivelsen. Dette byr på litt problemer dersom vi ser på forskjellene mellom CAD-data og skannet data.

Skannet data består stort sett av jevnt fordelte punkter som dekker en overflate av den skannede scenen. Dette er en fordel i bruk av lokale deskriptorer, siden det ligger mye informasjon lagret rundt hvert punkt i punktskyen. I CAD-data brukes kun de mest nødvendige punktene for å lage en tredimensjonal modell av et objekt. Som et eksempel på dette, kan en CAD-modell av en kube, lages

med bare åtte punkter. Et punkt i hvert hjørne. På grunn av dette, kan ikke CAD-data brukes direkte i algoritmene. I denne oppgaven har det blitt forsøkt å løse dette problemet ved å bruke en funksjon kalt «Subdivision Surface», i programmet Blender. Denne funksjonen oppretter flere punkter ved å legge inn ekstra punkter mellom de opprinnelige. Dette gir flere punkter i punktskyen, men disse er ikke nødvendigvis jevnt fordelt. Noe som kan hjelpe litt på dette er å bruke et VoxelGrid-filter. Dette filteret resulterer i en jevnere avstand mellom punktene.

Det er også andre grunner til at denne typen gjenkjenning er vanskelig. CAD-data er ofte en veldig nøyaktig beskrivelse av et objekt. Virkeligheten er ikke alltid like nøyaktig som dette. I tillegg utgjør Kinect-sensoren en feilkilde med sine begrensninger. Eksempler på dette er dybdeoppløsningen som er omtalt i kapittel 2.1.3. En annen feilkilde er feilene som oppstår under skanning og som er tydelig i figur 6.2.1.

Objekter som skal forsøkes gjenkjent bør ha former som det er mulig å beskrive. For at gjenkjenningen skal bli vellykket bør objektene ha elementer som er enten unike eller skiller seg ut fra resten av scenen.

7.2.1 «Correspondence Grouping»

Gjenkjenning med «Correspondence Grouping» gir mulighet for å estimere transformasjon mellom modell-punktsky og scene-punktsky. Dette vil ofte være en fordel ettersom det åpner for å bruke «Iterative Closest Point» (kapittel 4.7) og forskjellsdeteksjon (kapittel 4.2.1) mellom punktskyene. Av og til kan resultatene fra algoritmen være litt unøyaktige. Dette er sannsynligvis et resultat av diskretiseringen som gjøres av Hough-rommet. Hough-rommet diskretiseres ved å stille parameteren *Cluster size*. Denne stiller størrelsen på stegene som Hough-rommet diskretiseres i. Større verdi her gir mer unøyaktighet, men gjør også stemme-prosessen mer robust ved at flere stemmer fra et større område samles for å utgjøre en sterkere stemme. Vanligvis kan også de unøyaktige resultatene justeres ved å bruke ICP-algoritmen. ICP-algoritmen fungerer best dersom modell-punktskyen har færre punkter enn scenen-punktskyen. Er det store områder av punkter i modellen som ikke finnes i scenen vil ICP-algoritmen gjøre resultatet dårligere. Dette skjer fordi algoritmen forsøker å minimere feil mellom punkter og disse store områdene vil bidra til en stor feil, som algoritmen vil forsøke å fjerne.

Med like datasett og like parametere gjennom flere gjennomkjøringer, gir CG-algoritmen stort sett konsekvente resultater. Dersom en eller begge punktskyer roteres eller flyttes, påvirkes derimot resultatene. Disse vil fortsatt kunne være riktige, men forskjellige fra de forrige resultatene. I implementeringen gjort i denne oppgaven er det ikke tatt hensyn til antallet korrespondanser som detekteres mellom modell og scene. Dette kunne ha blitt gjort for å bedre kunne skille falske resultater fra riktige.

CG-algoritmen har ikke mulighet for å detektere forskjellige størrelser av modellen. Objektet som skal detekteres må være i samme skala som objektene i scenen. Dersom forskjellen i størrelsen på objektene i modell og scene er for stor, vil forskyvningen av stemmene i Hough-rommet bli for stor. Dette gjør at stemmene fra korrespondansene ikke klarer å lage et toppunkt i Hough-rommet. Dersom objektets størrelse i scenen er ukjent kan en mulig løsning være å bruke modeller av flere forskjellige størrelser i flere gjennomkjøringer av algoritmen.

Fra kapittel 6.2.4, blir det klart at CG-algoritmens tidsforbruk er rimeligere enn ISM-algoritmens tidsforbruk. Dersom det skal søkes etter samme objekt i flere scener, kan tidsforbruket kortes ned ved å implementere mulighet for å bruke samme modellinformasjon flere ganger.

7.2.2 ≪Implicit Shape Model≫

Gjenkjenning med ≪Implicit Shape Model≫ gir mulighet til å estimere senterpunkter til mulige forekomster av modell-punktskyen i scene-punktskyen. I tester gjort med ISM-algoritmen, viser det seg ofte at de detekterte senterpunktene ikke er konsistente mellom flere kjøringer. Dette skjer i tilfeller der toppunktene i Hough-rommet består av en lav tetthet av stemmer. I disse tilfellene er det vanskelig å reprodusere resultater. En mulig løsning kunne vært å se på et gjennomsnittlig toppunkt fra flere kjøringer.

I testing av tidsforbruket til ISM-algoritmen i kapittel 6.2.4, viser det seg at tidsforbruket fort blir veldig høyt. Allerede ved 40000 punkter er tidsforbruket på rundt 300 sekunder. Det viser seg også at det er treningen av algoritmen som har det største tidsforbruket. Det skal nevnes at det i testen ble brukt like store punktskyer for trening og testing. Dersom det er et mindre objekt som skal kjennes igjen i en større scene, er det sannsynlig at modell-punktskyen er noe mindre enn scenen. ISM-algoritmen har innebygget mulighet for å lagre modellinformasjonen fra treningen, som kunne ha bidratt til å redusere tiden for gjenkjenning av like objekter. Dette er ikke implementert i denne oppgaven.

ISM-algoritmen har mulighet for å trenes opp med flere modell-punktskyer for et objekt. Dette inkluderer muligheten for å kunne bruke modeller av forskjellige størrelser for samme objekt. Dette åpner for at objekter av forskjellige størrelser kan gjenkjennes med algoritmen. Dette er derimot ikke implementert i denne oppgaven. Her brukes kun en punktsky for trening og en punktsky for testing.

Det høye tidsforbruket til treningen kan skyldes bruk av FPFH-deskriptoren, omtalt i kapittel 4.5.2. I FPFH-deskriptoren finnes normalenretningene til alle punktene som brukes i beskrivelsen av et punkt. Til sammenligning bruker SHOT-deskriptoren, omtalt i kapittel 4.5.3, kun en normalretning som regnes ut i punktet som skal beskrives. Dette kan kanskje forbedres litt ved å implementere algoritmen på en annen måte. I denne implementeringen regnes det ut

normaler for alle punktene i punktskyen uavhengig av verdien gitt som *Sampling size*.

7.2.3 Parameterinnstilling

Parameterinnstilling for både CG-algoritmen og ISM-algoritmen må gjøres med tanke på objektene som skal gjenkjennes. I begge algoritmene brukes lokale punktbeskrivelser som baserer seg på normalretningen til et område rundt punktet som skal beskrives. Lengden på radiusen som blir brukt i utregning av normaler må derfor stilles inn med tanke på egenskapene til området. Denne problemstillingen ble også beskrevet i kapittel 4.3. Et for stort område brukt i utregningen vil gi en mer unøyaktig retning på normalen. Samtidig vil utregning med et for lite område lett kunne påvirkes av støy. I CG-algoritmen blir denne typen utregning gjort ved utregning av selve SHOT-deskriptoren og de lokale koordinatsystemene. Radiusen for disse utregningene spesifiseres av parametrene *Descr. r* og *Ref.frame r* i *CG*-panelet i PCRTool. I ISM-algoritmen gjøres denne typen utregning for normaler som blir brukt i FPFH-deskriptoren. Radius for denne utregningen er spesifisert av *Norm. r* i *ISM*-panelet i PCRTool.

Radiusen for FPFH-deskriptoren er litt annerledes. Denne spesifiserer et område rundt punktet som skal beskrives. Punktene innenfor dette området tas med i beskrivelsen. Radiusen for denne bør også settes med tanke på størrelsen av objektet, men er litt mer uavhengig av den nevnte problemstillingen. Radius for denne utregningen er spesifisert av *Descr. r* i *ISM*-panelet i PCRTool.

Generelt sett må de nevnte parametrene være små nok for å kunne lage et sett av flere beskrivelser for et objekt. Samtidig må verdiene være større enn oppløsningen på punktskyene. Dersom de er lavere, risikeres det at det ikke er nok punkter for å opprette punktbeskrivelser. Dette bør vurderes når parametrene *Model sr* og *Scene sr* settes i CG-algoritmen. Dette fordi disse reduserer oppløsningen på punktskyen.

7.2.4 Deteksjon av forskjeller mellom punktskyer

I kapittel 6.2.3, ble deteksjon av forskjeller demonstrert. Denne metoden baseres på octree-strukturene som ble beskrevet i 4.2.1. For deteksjon av forskjeller er det en parameter, *Octree Resolution*. Denne parameteren spesifiserer sidelengden på den minste voxelen i octree-strukturen. Dette gir da oppløsningen på forskjellsdeteksjonen. Stilles denne parameteren for lavt vil det kunne detekteres forskjeller der punkttettheten er ulik mellom punktskyene. Forskyvninger som følge av unøyaktighet under skanning vil også kunne detekteres. Dersom parameteren settes for høyt, kan deteksjonen bli mer unøyaktig. Forskjeller som skulle blitt detektert kan bli utelatt.

7.3 Konklusjon

I denne oppgaven har det blitt utført skanning med Xbox Kinect. Dette er gjort med applikasjonen Kinfu Large Scale. Deretter er det blitt utført objektgjenkjenning mellom punktskyer. Det ble også gjort deteksjon av forskjeller mellom punktskyer. For gjenkjenning ble det brukt to forskjellige algoritmer. Disse algoritmene er kalt ≪Correspondence Grouping≫ (CG) og ≪Implicit Shape Model≫ (ISM). For å lettere kunne teste algoritmene, ble disse implementert i et program kalt Point Cloud Recognition Tool. Dette programmet har et grafisk brukergrensesnitt for enkel innlasting og lagring og visualisering av punktskyer. I tillegg til dette ble også noen nyttige ekstrafunksjoner lagt til. Dette er funksjoner for segmentering, transformasjon, skalering og filtrering av punktskyer. Algoritmen ≪Iterative Closest Point≫ (ICP) og en algoritme for forskjellsdeteksjon ble også implementert.

I oppgaven ble to typer gjenkjenning testet. Den ene typen var gjenkjenning mellom skannede punktskyer. Her ble det brukt punktskyer fra Kinfu Large Scale. Den andre typen var gjenkjenning mellom tredimensjonale CAD-modeller og skannede punktskyer. Her ble det forsøkt å øke antallet punkter i CAD-modellene for å kunne bruke disse i gjenkjenning. Med riktig parameterinnstilling var det mulig å få til gode resultater for gjenkjenning mellom skannede punktskyer med både CG-algoritmen og ISM-algoritmen. Dette var også mulig ved gjenkjenning mellom skannede punktskyer og CAD-modeller. Her var derimot resultatene svakere og vanskeligere å få til på grunn av ujevn tetthet mellom punktene.

ICP-algoritmen kan detektere senterpunkter fra en trent modell-punktsky, i en scene punktsky. I oppgaven ble det gjort flere vellykkede deteksjoner av senterpunkter. CG-algoritmen estimerer den rigide transformasjonen mellom to punktskyer. Dette er også blitt testet i oppgaven. Med en korrekt estimert transformasjonsmatrise kunne den ene punktskyen flyttes til riktig posisjon på den andre. Etter flytting ble resultatet forbedret ved å kjøre ICP-algoritmen. Deteksjon av forskjeller mellom punktskyer er også blitt utført.

Algoritmene testet i denne oppgaven, viser at det er mulig å utføre både gjenkjenning og deteksjon av forskjeller mellom punktskyer. Dette er mulig enten punktskyene stammer fra tredimensjonale CAD-modeller eller er skannet med Xbox Kinect. I punktskyer fra CAD-modeller oppstår det derimot utfordringer med å oppnå god nok punkttetthet og nok punkter for å kunne utføre gjenkjenning.

Bibliografi

[1] "Visualization." Stormfjord.no, http://stormfjord.no/solutions/sales-marketing/visualization. Lastet ned 03.06.2014.

[2] "Euclidean space." Wikipedia, http://en.wikipedia.org/wiki/Euclidean_space. Lastet ned 03.06.2014.

[3] "History of video games." Wikipedia, http://en.wikipedia.org/wiki/History_of_video_games. Lastet ned 10.05.2014.

[4] S. Crawford, "How microsoft kinect works." HowStuffWorks.com, http://electronics.howstuffworks.com/microsoft-kinect.htm, juli 2010. Lastet ned 10.05.2014.

[5] D. Terdiman, "Bounty offered for open-source kinect driver." CNET, http://www.cnet.com/news/bounty-offered-for-open-source-kinect-driver/, november 2010. Lastet ned 10.05.2014.

[6] "The open kinect project." Adafruit, https://www.adafruit.com/blog/2010/11/04/the-open-kinect-project-the-ok-prize-get-...
...1000-bounty-for-kinect-for-xbox-360-open-source-drivers/, november 2010. Lastet ned 10.05.2014.

[7] T. Bishop, "Microsoft: Kinect wasn't hacked, usb port left open 'by design'." Puget Sound Business Journal, http://www.bizjournals.com/seattle/blog/techflash/2010/11/microsoft-kinect-not-hacked-left.html?page=all, november 2010. Lastet ned 10.05.2014.

[8] "Kinect." Wikipedia, http://en.wikipedia.org/wiki/Kinect. Lastet ned 06.05.2014.

[9] A. Armstrong, "Openni to close." i-programmer, http://www.i-programmer.info/news/194-kinect/7004-openni-to-close-.html, februar 2014. Lastet ned 10.05.2014.

[10] "Openni." Wikipedia, http://en.wikipedia.org/wiki/OpenNI. Lastet ned 10.05.2014.

[11] "Willow garage." Wikipedia, http://en.wikipedia.org/wiki/Willow_Garage. Lastet ned 05.05.2014.

[12] "Ros." Willow Garage, http://www.willowgarage.com/pages/software/ros-platform. Lastet ned 05.05.2014.

[13] R. B. Rusu, "PointClouds.org: A new home for Point Cloud Library (PCL)." http://www.willowgarage.com/blog/2011/03/27/point-cloud-library-pcl-moved-pointcloudsorg. Lastet ned 05.05.2014.

[14] "Media." PointClouds.org, http://pointclouds.org/media/. Lastet ned 05.05.2014.

[15] "About." PointClouds.org, http://pointclouds.org/about/. Lastet ned 05.05.2014.

[16] "Kinect for windows sensor." MSDN, http://msdn.microsoft.com/en-us/library/hh855355.aspx. Lastet ned 11.05.2014.

[17] "Fequently asked questions." Kinect for Windows, http://www.microsoft.com/en-us/kinectforwindows/Faq.aspx. Lastet ned 03.06.2014.

[18] "Kinect sensor." MSDN, http://msdn.microsoft.com/en-us/library/hh438998.aspx. Lastet ned 14.06.2014.

[19] R. B. Rusu, "Kinect for windows with pcl." PointClouds.org, http://pointclouds.org/news/2013/05/13/kinect-for-windows-with-pcl/, mai 2013. Lastet ned 12.05.2014.

[20] "Kinect for windows sensor components and specifications." MSDN, http://msdn.microsoft.com/en-us/library/jj131033.aspx. Lastet ned 11.05.2014.

[21] "Microsoft kinect teardown." iFixit, http://www.ifixit.com/Teardown/Microsoft+Kinect+Teardown/4066. Lastet ned 12.05.2014.

[22] J. MacCormick, "How does the kinect work?." Presentert ved Dickinson College, 6. november 2011, tilgjengelig fra http://users.dickinson.edu/~jmac/selected-talks/kinect.pdf. Lastet ned 12.05.2014.

[23] A. Shpunt and Z. Zalevsky, "Depth-varying light fields for three dimensional sensing," May 8 2008. US Patent App. 11/724,068.

[24] B. Freedman, A. Shpunt, and Y. Arieli, "Distance-varying illumination and imaging techniques for depth mapping," Nov. 18 2010. US Patent App. 12/522,176.

[25] D. A. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*. Pearson Education Limited, 2 ed., 2012.

[26] G. Gerig, "Structured lighting." Forelesningsmateriell, 3D Computer Vision, University of Utah, tilgjengelig fra http://www.sci.utah.edu/~gerig/CS6320-S2012/Materials/CS6320-CV-S2012-StructuredLight.pdf, 2012. Lastet ned 13.05.2014.

[27] J. Smisek, M. Jancosek, and T. Pajdla, "3d with kinect," in *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pp. 1154–1160, Nov 2011.

[28] "Bsd-license." Wikipedia, http://en.wikipedia.org/wiki/BSD_licenses. Lastet ned 13.05.2014.

[29] "Pcl master." GitHub, https://github.com/PointCloudLibrary/pcl. Lastet ned 04.06.2014.

[30] A. Placitelli and M. Boufarguine, "Compiling pcl from source on windows." PointClouds.org, http://pointclouds.org/documentation/tutorials/compiling_pcl_windows.php. Lastet ned 13.05.2014.

[31] "Boost." Boost, http://www.boost.org/. Lastet ned 04.06.2014.

[32] "Eigen." Eigen, http://eigen.tuxfamily.org/index.php?title=Main_Page. Lastet ned 04.06.2014.

[33] "Flann." FLANN, http://www.cs.ubc.ca/research/flann/.

[34] "Vtk." VTK, http://www.vtk.org/. Lastet ned 04.06.2014.

[35] "Qt." QT Project, http://qt-project.org/. Lastet ned 04.06.2014.

[36] "Qhull." Qhull.org, http://www.qhull.org/. Lastet ned 04.06.2014.

[37] "Openni." Structure, http://structure.io/openni.

[38] "Nvidia cuda." Nvidia Cuda Zone, https://developer.nvidia.com/cuda-downloads. Lastet ned 04.06.2014.

[39] "Polygon." Wikipedia, http://en.wikipedia.org/wiki/Polygon. Lastet ned 04.06.2014.

[40] "Ply (file format)." Wikipedia, http://en.wikipedia.org/wiki/PLY_(file_format). Lastet ned 15.05.2014.

[41] R. B. Rusu, "The pcd (point cloud data) file format." PointClouds.org, http://pointclouds.org/documentation/tutorials/pcd_file_format.php. Lastet ned 15.05.2014.

[42] "Cuda." Wikipedia, http://en.wikipedia.org/wiki/CUDA. Lastet ned 20.05.2014.

[43] "Signed distance function." Wikipedia, http://en.wikipedia.org/wiki/Signed_distance_function. Lastet ned 20.05.2014.

[44] F. Heredia and R. Favier, "Using kinfu large scale to generate a textured mesh." PointClouds.org, http://pointclouds.org/documentation/tutorials/using_kinfu_large_scale.php. Lastet ned 20.05.2014.

[45] "How kinect and kinect fusion (kinfu) work." Razor Vision, http://razorvision.tumblr.com/post/15039827747/how-kinect-and-kinect-fusion-kinfu-work. Lastet ned 21.05.2014.

[46] "Meshlab." Programmvare tilgjengelig fra http://meshlab.sourceforge.net/. Lastet ned 02.06.2014.

[47] "Blender." Programmvare tilgjengelig fra http://www.blender.org/. Lastet ned 02.06.2014.

[48] "Sketchup." Programmvare tilgjengelig fra http://www.sketchup.com/. Lastet ned 02.06.2014.

[49] "Cmake." CMake, http://www.cmake.org/. Lastet ned 05.06.2014.

[50] "Downloads." PointClouds.org, http://pointclouds.org/downloads/

windows.html. 05.06.2014.

[51] N. Sallem, "Using pcl in your own project." PointClouds.org, http://pointclouds.org/documentation/tutorials/using_pcl_pcl_config.php. Lastet ned 05.06.2014.

[52] G. OLeary, "How to use a kdtree to search." PointClouds.org, http://pointclouds.org/documentation/tutorials/kdtree_search.php. Lastet ned 21.05.2014.

[53] "Kd-tree." Wikipedia, http://en.wikipedia.org/wiki/Kd-tree. Lastet ned 21.05.2014.

[54] "Octree." Wikipedia, urlhttp://en.wikipedia.org/wiki/Octree. Lastet ned 22.05.2014.

[55] J. Kammerl, "Octree compression." Presentert under ICRA, St. Paul, Minnesota, USA. 18. mai 2012, tilgjengelig fra http://www.pointclouds.org/assets/icra2012/search.pdf. Lastet ned 10.06.2014.

[56] R. B. Rusu, "Estimating surface normals in a pointcloud." PointClouds.org, http://pointclouds.org/documentation/tutorials/normal_estimation.php. Lastet ned 26.05.2014.

[57] R. B. Rusu, *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments.* PhD thesis, Computer Science department, Technische Universitaet Muenchen, Germany, October 2009.

[58] F. Tombari, "How does a good feature look like?." Presentert under ICRA, Karlsruhe, Germany. 10. mai 2013, tilgjengelig fra http://www.pointclouds.org/assets/icra2013/pcl_features_icra13.pdf. Lastet ned 16.05.2014.

[59] S. Salti, F. Tombari, and L. D. Stefano, "Shot: Unique signatures of histograms for surface and texture description," *Computer Vision and Image Understanding*, Akseptert manuskript, 2014.

[60] F. Tombari, S. Salti, and L. D. Stefano, "Unique signatures of histograms for local surface description," in *11th European Conference on Computer Vision (ECCV)*, 5.-11. september 2010.

[61] R. B. Rusu, "Point feature histograms (pfh) descriptors." PointClouds.org, http://pointclouds.org/documentation/tutorials/pfh_estimation.php. Lastet ned 26.05.2014.

[62] R. B. Rusu, "Fast point feature histograms (fpfh) descriptors." PointClouds.org, `http://pointclouds.org/documentation/tutorials/fpfh_estimation.php`. Lastet ned 27.05.2014.

[63] H. Hoppe, T. DeRose, T. Duchamp, J. Mcdonald, and W. Stuetzle, "Surface reconstruction from unorganized points," in *COMPUTER GRAPHICS (SIGGRAPH 92 PROCEEDINGS)*, pp. 71–78, 1992.

[64] N. J. Mitra, A. Nguyen, and L. Guibas, "Estimating surface normals in noisy point cloud data," in *Special issue of International Journal of Computational Geometry and Applications*, vol. 14, pp. 261–276, 2004.

[65] A. Petrelli and L. D. Stefano, "On the repeatability of the local reference frame for partial shape matching," in *13th International Conference on Computer Vision (ICCV)*, 2011.

[66] E. R. Davies, *Machine Vision: Theory, Algorithms, Practicalities.* Morgan Kauffmann, 3 ed., 2005.

[67] "Hough-transform." Mathworks, `http://www.mathworks.se/help/images/ref/hough.html`. Lastet ned 19.05.2014.

[68] "Sobel operator." Wikipedia, `http://en.wikipedia.org/wiki/Sobel_operator`. Lastet ned 18.05.2014.

[69] "Hough transform." `http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm`. Lastet ned 18.05.2014.

[70] K. Khoshelham, "Extending generalized hough transform to detect 3d objects in laser range data," in *ISPRS Workshop on Laser Scanning*, 2007.

[71] T. Cavallari and F. Tombari, "3d object recognition based on correspondence grouping." PointClouds.org, `http://pointclouds.org/documentation/tutorials/correspondence_grouping.php`. Lastet ned 30.05.2014.

[72] F. Tombari and L. Di Stefano, "Object recognition in 3d scenes with occlusions and clutter by hough voting," in *Image and Video Technology (PSIVT), 2010 Fourth Pacific-Rim Symposium on*, pp. 349–355, Nov 2010.

[73] J. Knopp, M. Prasad, G. Willems, R. Timofte, and L. V. Gool, "Hough transform and 3d surf for robust three dimensional classification," in *Proceedings of the European Conference on Computer Vision*, 2010.

[74] S. Ushakov, "Implicit shape model." PointClouds.org, http://pointclouds.org/documentation/tutorials/implicit_shape_model.php. Lastet ned 09.06.2014.

[75] "Surf." Wikipedia, http://en.wikipedia.org/wiki/SURF. Lastet ned 14.06.2014.

[76] "K-means clustering." Wikipedia, http://en.wikipedia.org/wiki/K-means_clustering. Lastet ned 09.06.2014.

[77] A. Placitelli and M. Boufarguine, "Building pcl's dependencies from source on windows." PointClouds.org, http://pointclouds.org/documentation/tutorials/compiling_pcl_dependencies_windows.php. Lastet ned 05.06.2014.

[78] best lemming, "vw passat." Sketchup modell, tilgjengelig for nedlasting fra https://3dwarehouse.sketchup.com/model.html?id=d077353d10798501c9faca5495e1b7e3. Lastet ned 13.06.2014.

[79] D. Holz, R. B. Rusu, and J. Sprickerhof, "The pcl registration api." PointClouds.org, http://pointclouds.org/documentation/tutorials/registration_api.php. Lastet ned 14.06.2014.

Tillegg A

Innhold på CD

Den vedlagte CD-platen inneholder fire mapper.

I mappen *Datasett for testing* ligger alle punktskyene som ble brukt i testene i denne rapporten.

- *deteksjon av forskjeller* - inneholder punktskyene som ble brukt i kapittel 6.2.3.
- *original datasett* - inneholder originalene av alle datasettene.
- *skann-cad* - inneholder punktskyene som ble brukt i kapittel 6.2.2
- *skann-skann* - inneholder punktskyene som ble brukt i kapittel 6.2.1

Mappen *PCRTool*, inneholder programmet Point Cloud Recognition Tool. Programmet kjøres fra filen PCRTool.exe.

Mappen *PCRTool source*, inneholder kildekoden til programmet.

Mappen *Rapport* inneholder denne rapporten i PDF-format

Tillegg B

Point Cloud Recognition Tool kildekode

I dette tillegget ligger kildekoden til programmet Point Cloud Recognition Tool. Programmet er delt opp i følgende filer.

B.1 pcrtool.cpp - inneholder main-metoden.
B.2 mainwindow.h - header-fil for mainwindow.cpp.
B.3 mainwindow.cpp - inneholder klassen MainWindow.
B.4 centralwidget.h - header-fil for centralwidget.cpp.
B.5 centralwidget.cpp - inneholder klassen CWidget.
B.6 panels.h - header-fil for panels.cpp.
B.7 panels.cpp - inneholder alle panelklassene.
B.8 pcltools.h - header-fil for pcltools.cpp.
B.9 pcltools.cpp - inneholder klassen PCLTools.
B.10 recognition.h - header-fil for recognition.cpp.
B.11 recognition.cpp - inneholder klassen CGRecognition.
B.12 recognitionISM.h - header-fil for recognitionISM.cpp.
B.13 recognitionISM.cpp - inneholder klassen ISMRecognition.
B.14 button.h - header-fil for button.cpp.
B.15 button.cpp - inneholder klassen Button.
B.14 CmakeLists.txt - inneholder cmake-filen for prosjektet.

B.1 pcrtool.cpp

```
#include <QApplication>

#include <pcl/io/pcd_io.h>
#include <pcl/features/normal_3d.h>
#include <pcl/sample_consensus/sac_model_plane.h>
#include <pcl/visualization/cloud_viewer.h>
#include <pcl/common/common.h>
#include <vtkRenderWindow.h>

#include <QVTKWidget.h>
#include "mainwindow.h"

int main(int argc, char** argv)

{
        QApplication app(argc, argv);
        MainWindow myApp;
        myApp.show();
        app.exec();

  return EXIT_SUCCESS;
}
```

B.2 mainwindow.h

```
#pragma once
#include "pcltools.h"
#include <QMainWindow>
#include <QtGui>
#include <QAction>
#include <QKeySequence>
#include <QWidget>
#include <QString>
#include <QMessageBox>

class QAction;
class QMenu;
class QPlainTextEdit;
class CWidget;

class MainWindow : public QMainWindow
{
      Q_OBJECT

public:
      MainWindow();
private slots:
      //File Menu-----
      void newProject();
      void importModel();
      void importScene();
      void save();
      void exportModel();
      void exportScene();
      void exit();
      void undo();
      void screenshot();
      void help();
      void about();
private:
      //File Menu---------------------
      void createActions();
      void createMenus();
      void createCentralWidget();
      int saveQuestion(QString text);
      CWidget *centralWidget;

      QMenu *fileMenu;
      QMenu *saveAsMenu;
      QMenu *importMenu;
      QMenu *exportMenu;
      QMenu *editMenu;
      QMenu *viewMenu;
      QMenu *helpMenu;
      QAction *newAction;
      QAction *importModelAction;
      QAction *importSceneAction;
      QAction *saveAction;
```

```
        QAction *exportModelAction;
        QAction *exportSceneAction;
        QAction *exitAction;
        QAction *undoAction;
        QAction *screenshotAction;
        QAction *redoAction; // Kanskje
        QAction *helpAction;
        QAction *aboutAction;
};
```

B.3 mainwindow.cpp

```
#include "mainwindow.h"
#include "centralwidget.h"

MainWindow::MainWindow()
{
      centralWidget = new CWidget(this);
      setCentralWidget(centralWidget);
      setWindowTitle("Point Cloud Recognition Tool");
      createActions();
      createMenus();
}
// Menu -------------------------------------------------------------------
void MainWindow::createActions()
{
      newAction = new QAction(tr("&New Empty Project..."), this);
      newAction->setShortcuts(QKeySequence::New);
      newAction->setStatusTip(tr("New Empty Project"));
      connect(newAction, SIGNAL(triggered()), this, SLOT(newProject()));

      importModelAction = new QAction(tr("&Import Model File..."), this);
      importModelAction->setStatusTip(tr("Import Model File"));
      connect(importModelAction, SIGNAL(triggered()), this,
SLOT(importModel()));

      importSceneAction = new QAction(tr("&Import Scene File..."), this);
      importSceneAction->setStatusTip(tr("Import Scene File"));
      connect(importSceneAction, SIGNAL(triggered()), this,
SLOT(importScene()));

      saveAction = new QAction(tr("&Save Project..."), this);
      saveAction->setShortcuts(QKeySequence::Save);
      saveAction->setStatusTip(tr("Save current project"));
      connect(saveAction, SIGNAL(triggered()), this, SLOT(save()));

      exportModelAction = new QAction(tr("&Export model..."), this);
      exportModelAction->setStatusTip(tr("Export model as..."));
      connect(exportModelAction, SIGNAL(triggered()), this,
SLOT(exportModel()));

      exportSceneAction = new QAction(tr("&Export scene..."), this);
      exportModelAction->setStatusTip(tr("Export scene as..."));
      connect(exportSceneAction, SIGNAL(triggered()), this,
SLOT(exportScene()));

      exitAction = new QAction(tr("&Exit"), this);
      exitAction->setStatusTip(tr("Exit program"));
      connect(exitAction, SIGNAL(triggered()), this, SLOT(exit()));

      undoAction = new QAction(tr("&Undo"), this);
      undoAction->setShortcuts(QKeySequence::Undo);
```

```
	undoAction->setStatusTip(tr("Undo last operation"));
	connect(undoAction, SIGNAL(triggered()), this, SLOT(undo()));

	screenshotAction = new QAction(tr("&Take Screenshot"), this);
	screenshotAction->setStatusTip(tr("Undo last operation"));
	connect(screenshotAction, SIGNAL(triggered()), this,
SLOT(screenshot()));

	helpAction = new QAction(tr("&Help"), this);
	helpAction->setStatusTip(tr("Undo last operation"));
	connect(helpAction, SIGNAL(triggered()), this, SLOT(help()));

	aboutAction = new QAction(tr("&About"), this);
	aboutAction->setStatusTip(tr("Undo last operation"));
	connect(aboutAction, SIGNAL(triggered()), this, SLOT(about()));
}
void MainWindow::createMenus()
{
	fileMenu = menuBar()->addMenu(tr("&File"));
	fileMenu->addAction(newAction);
	importMenu = fileMenu->addMenu(tr("&Import..."));
	importMenu->addAction(importModelAction);
	importMenu->addAction(importSceneAction);
	exportMenu = fileMenu->addMenu(tr("&Export..."));
	exportMenu->setEnabled(false);

	exportMenu->addAction(exportModelAction);
	exportMenu->addAction(exportSceneAction);
	exportModelAction->setEnabled(false);
	exportSceneAction->setEnabled(false);
	fileMenu->addSeparator();
	fileMenu->addAction(exitAction);

	editMenu = menuBar()->addMenu(("&Edit"));
	editMenu->addAction(undoAction);
	editMenu->addSeparator();
	editMenu->addAction(screenshotAction);

	viewMenu = menuBar()->addMenu(tr("&View"));

	helpMenu = menuBar()->addMenu(tr("&Help"));
	helpMenu->addAction(helpAction);
	helpMenu->addAction(aboutAction);
}

// Menu callbacks ----------------------------------------------------------
int MainWindow::saveQuestion(QString text)
{
	QMessageBox msgBox;
	msgBox.setText(text);
	msgBox.setInformativeText("Do you want to save your changes?");
	msgBox.setStandardButtons(QMessageBox::Save | QMessageBox::Discard |
QMessageBox::Cancel);
	msgBox.setDefaultButton(QMessageBox::Save);
	int answer = msgBox.exec();
	return answer;
}
```

```
void MainWindow::newProject()
{
	bool newProject = false;

	if(centralWidget->getPCL()->_modelIsModified)
	{
		int answer = saveQuestion("The model pointcloud is modified.");
		if(answer==QMessageBox::Save)
		{
			//save model
		}
		else if(answer==QMessageBox::Cancel)
		{
			return;
		}
	}
	if(centralWidget->getPCL()->_sceneIsModified)
	{
		int answer = saveQuestion("The scene pointcloud is modified.");
		if(answer==QMessageBox::Save)
		{
			//save scene
		}
		else if(answer==QMessageBox::Cancel)
		{
			return;
		}
	}

}

void MainWindow::importModel()
{
	QString path = QFileDialog::getOpenFileName(this, tr("Load Model
file..."), "/", tr("PointCloud files (*.pcd *.ply)"));
	int cond = centralWidget->getPCL()-
>loadCloud(path.toStdString(),true);
	if(cond==0)
	{
		centralWidget->setDisplay(tr("Successfully loaded ")+path);
		centralWidget->enableButtons(tr("model"));
		centralWidget->enablePanels("model");
		centralWidget->setChecked("model");
		centralWidget->getPCL()->initialShow("model");
		exportMenu->setEnabled(true);
		exportModelAction->setEnabled(true);
	}
	else
		centralWidget->setDisplay(tr("Error! Was not able to load
")+path);
}
void MainWindow::importScene()
{
	QString path = QFileDialog::getOpenFileName(this, tr("Load Scene
file..."), "/", tr("PointCloud files (*.pcd *.ply)"));
	int cond = centralWidget->getPCL()-
>loadCloud(path.toStdString(),false);
	if(cond==0)
```

```
	{
		centralWidget->setDisplay(tr("Successfully loaded ")+path);
		centralWidget->enableButtons(tr("scene"));
		centralWidget->enablePanels("scene");
		centralWidget->setChecked("scene");
		centralWidget->getPCL()->initialShow("scene");
		exportMenu->setEnabled(true);
		exportSceneAction->setEnabled(true);
	}
	else
		centralWidget->setDisplay(tr("Error! Was not able to load
")+path);
}
void MainWindow::save(){}
void MainWindow::exportModel()
{
	QString path = QFileDialog::getSaveFileName(0,"Save model
as","/","Point Cloud Data (*.pcd);;Polygon File Format (*.ply)",
		new QString("Polygon File Format (*.ply)"));

	if (centralWidget->getPCL()-
>saveCloud("model",path.toStdString())==0)
	{
		centralWidget->setDisplay(tr("Model saved successfully to
")+path);
		centralWidget->getPCL()->_modelIsModified=false;
	}
	else
		centralWidget->setDisplay(tr("Error! Could not save model."));

}
void MainWindow::exportScene()
{
	QString path = QFileDialog::getSaveFileName(0,"Save scene
as","/","Point Cloud Data (*.pcd);;Polygon File Format (*.ply)",
		new QString("Polygon File Format (*.ply)"));

	if (centralWidget->getPCL()-
>saveCloud("scene",path.toStdString())==0)
	{
		centralWidget->setDisplay(tr("Scene saved successfully to
")+path);
		centralWidget->getPCL()->_sceneIsModified=false;
	}
	else
		centralWidget->setDisplay(tr("Error! Could not save scene."));
}
void MainWindow::exit()
{
	QApplication::quit();
}
void MainWindow::undo()
{
	centralWidget->getPCL()->undoLast();
	centralWidget->getPCL()->show(centralWidget->getChecked());
	centralWidget->setDisplay("Reverted to the previous cloud");
}
void MainWindow::screenshot(){}
```

```
void MainWindow::help(){}
void MainWindow::about(){}
```

B.4 centralwidget.h

```
#pragma once

#include "pcltools.h"
#include "button.h"
#include "panels.h"
#include <QtGui>
#include <pcl/io/pcd_io.h>
#include <pcl/visualization/cloud_viewer.h>
#include <pcl/visualization/pcl_visualizer.h>
#include <pcl/common/common.h>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <vtkRenderWindow.h>
#include <QWidget>
#include <QVTKWidget.h>
#include <QMainWindow>
#include <QFileDialog>
#include <QStackedWidget>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QRadioButton>
#include <QGroupBox>
#include <QMainWindow>
#include <QString>

class QLineEdit;
class Button;
class QVTKWidget;
class ResizePanel;
class SegmentPanel;
class FilterPanel;
class ToolsPanel;
class RecognitionPanel;
class ismRecognitionPanel;
class TransPanel;

class CWidget : public QWidget
{
	Q_OBJECT
public:
	CWidget(QWidget *parent = 0);
	PCLTools *getPCL();
	void setDisplay(QString text);
	void enableButtons(QString cloud);
	void refresh();
	void enablePanels(std::string cloud);
	void setUpperWidget(int widgetNum);
	std::string getChecked();
	void setChecked(std::string cloud);
private slots:
	void buttonClicked();
	void radioButtonClicked();
```

```
private:
	Button *createButton(const QString &text, const char *member);
	QRadioButton *createRadioButton(const QString &text, const char
*member);
	//Mainwindow Layout -------------------------
	QVBoxLayout *mainWindowLayout;
	//Upper Layout-------------------------------
	QWidget			*firstPage;
	QGridLayout		*firstPageLayout;
	QStackedWidget		*upperWidget;
	ResizePanel		*resizePanel;
	SegmentPanel		*segmentPanel;
	FilterPanel		*filterPanel;
	ToolsPanel		*toolsPanel;
	RecognitionPanel	*recognitionPanel;
	ismRecognitionPanel *ismRecPanel;
	TransPanel		*transPanel;

	Button *resizeButton;
	Button *segmentButton;
	Button *filterButton;
	Button *toolsButton;
	Button *recognitionButton;
	Button *transformButton;
	Button *ismRecButton;

	//Lower Layout-------------------------------
	QWidget			*lowerWidget;
	QHBoxLayout		*lowerLayout;

	QGroupBox		*rButtons;
	QHBoxLayout		*rbLayout;
	QRadioButton		*showScene;
	QRadioButton		*showModel;
	QRadioButton		*showBoth;
	QLineEdit		*display;

	//Central Layout-----------------------------
	QVTKWidget		*visWindow;
	boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer;
	vtkSmartPointer<vtkRenderWindow> renderWindow;
	//PCL Implementation
	PCLTools *pclTools;
};
```

B.5 centralwidget.cpp

```
#include "centralwidget.h"

CWidget::CWidget(QWidget *parent) : QWidget(parent)
{
      setMinimumSize(1100, 625);
      mainWindowLayout = new QVBoxLayout;

      //-----------------------------------------------------------
      //Upper Widget-----------------------------------------------
      //-----------------------------------------------------------
      upperWidget = new QStackedWidget;
      upperWidget->setMaximumHeight(70);

      firstPage = new QWidget;
      firstPageLayout = new QGridLayout;

      resizeButton = createButton(tr("Resize"), SLOT(buttonClicked()));
      transformButton = createButton(tr("Transform"),
SLOT(buttonClicked()));
      segmentButton = createButton(tr("Segmentation"),
SLOT(buttonClicked()));
      filterButton = createButton(tr("Filtration"), SLOT(buttonClicked()));
      toolsButton = createButton(tr("Tools"), SLOT(buttonClicked()));
      recognitionButton = createButton(tr("CG"), SLOT(buttonClicked()));
      ismRecButton = createButton(tr("ISM"), SLOT(buttonClicked()));

      firstPageLayout->setSizeConstraint(QLayout::SetFixedSize);

      //Add buttons to upper panel
      firstPageLayout->addWidget(resizeButton,       0, 0, 1, 1);
      firstPageLayout->addWidget(transformButton,          0, 1, 1, 1);
      firstPageLayout->addWidget(segmentButton,      0, 2, 1, 1);
      firstPageLayout->addWidget(filterButton,       0, 3, 1, 1);
      firstPageLayout->addWidget(toolsButton,              0, 4, 1, 1);
      firstPageLayout->addWidget(recognitionButton,  0, 5, 1, 1);
      firstPageLayout->addWidget(ismRecButton,       0, 6, 1, 1);

      firstPage->      setLayout(firstPageLayout);
      upperWidget->    addWidget(firstPage);

      resizePanel       =     new ResizePanel(this);
      transPanel        =     new   TransPanel(this);
      segmentPanel      =     new SegmentPanel(this);
      filterPanel       =     new FilterPanel(this);
      toolsPanel        =     new ToolsPanel(this);
      recognitionPanel= new RecognitionPanel(this);
      ismRecPanel       =   new ismRecognitionPanel(this);

      upperWidget->    addWidget(resizePanel);
      upperWidget->    addWidget(transPanel);
```

```
	upperWidget->	addWidget(segmentPanel);
	upperWidget->	addWidget(filterPanel);
	upperWidget->	addWidget(toolsPanel);
	upperWidget->	addWidget(recognitionPanel);
	upperWidget->	addWidget(ismRecPanel);

	//-------------------------------------------------------------
	//Visualizer--------------------------------------------------
	//-------------------------------------------------------------
	boost::shared_ptr<pcl::visualization::PCLVisualizer> v_ptr(new
pcl::visualization::PCLVisualizer("test_viz", false)); //default false
	viewer = v_ptr;
	visWindow = new QVTKWidget;
	visWindow->setMinimumSize(800,400);

	//-------------------------------------------------------------------
	// Following lines inspired by pcl forum user Lucas Walter,
	// first post http://www.pcl-users.org/QT-PCLVisualizer-mostly-
working-td3285187.html

	renderWindow = viewer->getRenderWindow ();
	visWindow->SetRenderWindow (renderWindow);
	viewer->setupInteractor (visWindow->GetInteractor (), visWindow-
>GetRenderWindow ());
	viewer->getInteractorStyle ()->setKeyboardModifier
(pcl::visualization::INTERACTOR_KB_MOD_SHIFT);
	viewer->addCoordinateSystem(1.0);
	viewer->setBackgroundColor(0.2, 0.2, 0.4);
	//---------------------------------------------------------------------


	//-------------------------------------------------------------
	//Lower Widget------------------------------------------------
	//-------------------------------------------------------------
	lowerWidget = new QWidget;
	lowerLayout = new QHBoxLayout;

	rButtons =  new QGroupBox;
	rbLayout =  new QHBoxLayout;
	showScene = createRadioButton(tr("Show
scene"),SLOT(radioButtonClicked()));
	showModel = createRadioButton(tr("Show
model"),SLOT(radioButtonClicked()));
	showBoth =  createRadioButton(tr("Show
both"),SLOT(radioButtonClicked()));

	showScene->setEnabled(false);
	showModel->setEnabled(false);
	showBoth->setEnabled(false);
	resizeButton->setEnabled(false);
	transformButton->setEnabled(false);
	segmentButton->setEnabled(false);
	filterButton->setEnabled(false);
	toolsButton->setEnabled(false);
	recognitionButton->setEnabled(false);
	ismRecButton->setEnabled(false);

	rbLayout->addWidget(showModel);
```

```
	rbLayout->addWidget(showScene);
	rbLayout->addWidget(showBoth);
	rButtons->setLayout(rbLayout);

	lowerLayout->addWidget(rButtons);

	display = new QLineEdit(" ");
	display->setReadOnly(true);
	display->setAlignment(Qt::AlignRight);
	lowerLayout->addWidget(display);
	lowerWidget->setLayout(lowerLayout);

	//-------------------------------------------------------------
	//Main Layout-------------------------------------------------
	//-------------------------------------------------------------
	mainWindowLayout->addWidget(upperWidget);
	mainWindowLayout->addWidget(visWindow);
	mainWindowLayout->addWidget(lowerWidget);
	setLayout(mainWindowLayout);

	pclTools = new PCLTools(viewer);

}

void CWidget::enablePanels(std::string cloud)
{
	resizePanel->enablePanel(cloud);
	segmentPanel->enablePanel();
	filterPanel->enablePanel();
	toolsPanel->enablePanel(cloud);
	transPanel->enablePanel(cloud);
	if(pclTools->_modelIsLoaded && pclTools->_sceneIsLoaded)
	{
		recognitionPanel->enablePanel();
		ismRecPanel->enablePanel();
	}

}
std::string CWidget::getChecked()
{
	if(showModel->isChecked())
		return "model";
	if(showScene->isChecked())
		return "scene";
	if(showBoth->isChecked())
		return "both";
	else
		return "none";
}

void CWidget::setChecked(std::string cloud)
{
	if(cloud.compare("model")==0)
		showModel->setChecked(true);
	if(cloud.compare("scene")==0)
		showScene->setChecked(true);
	if(cloud.compare("both")==0)
		showBoth->setChecked(true);
```

```
}

void CWidget::buttonClicked()
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();

	if(clicked.compare("Resize")==0)
	{
		upperWidget->setCurrentIndex(1);
		setDisplay(" ");
	}
	else if(clicked.compare("Transform")==0)
	{
		upperWidget->setCurrentIndex(2);
		setDisplay(" ");
	}
	else if(clicked.compare("Segmentation")==0)
	{
		upperWidget->setCurrentIndex(3);
		segmentPanel->setActivePanel(true);
		setDisplay(" ");
	}
	else if(clicked.compare("Filtration")==0)
	{
		upperWidget->setCurrentIndex(4);
		setDisplay(" ");
	}
	else if(clicked.compare("Tools")==0)
	{
		upperWidget->setCurrentIndex(5);
		setDisplay(" ");
	}
	else if(clicked.compare("CG")==0)
	{
		upperWidget->setCurrentIndex(6);
		setDisplay(" ");
	}
	else if(clicked.compare("ISM")==0)
	{
		upperWidget->setCurrentIndex(7);
		setDisplay("y");
	}
}

void CWidget::radioButtonClicked()
{
	QRadioButton *clickedButton = qobject_cast<QRadioButton *>(sender());
	QString clicked = clickedButton->text();

	if(clicked.compare(tr("Show both"))==0)
	{
		pclTools->show("both");
	}
	else if(clicked.compare(tr("Show model"))==0)
	{
		pclTools->show("model");
		segmentPanel->resetBoundingBox();
```

```
	}
	else if(clicked.compare(tr("Show scene"))==0)
	{
		pclTools->show("scene");
		segmentPanel->resetBoundingBox();
	}
	display->clear();
	display->setText(clicked);

	segmentPanel->disablePanel();
	segmentPanel->enablePanel();

	filterPanel->disablePanel();
	filterPanel->enablePanel();
}

Button *CWidget::createButton(const QString &text, const char *member)
{
	Button *button = new Button(text);
	connect(button, SIGNAL(clicked()), this, member);
	return button;
}

QRadioButton *CWidget::createRadioButton(const QString &text, const char
*member)
{
	QRadioButton *rbutton = new QRadioButton(text);
	connect(rbutton, SIGNAL(clicked()), this,member);
	return rbutton;
}

PCLTools *CWidget::getPCL()
{
	return pclTools;
}

void CWidget::setDisplay(QString text)
{
	display->clear();
	display->setText(text);
}

void CWidget::enableButtons(QString cloud)
{
	if((cloud.compare(tr("scene"))==0) && pclTools->_sceneIsLoaded)
	{
		showScene->setEnabled(true);
		showScene->setChecked(true);
		resizeButton->setEnabled(true);
		transformButton->setEnabled(true);
		segmentButton->setEnabled(true);
		filterButton->setEnabled(true);
		toolsButton->setEnabled(true);
	}
	else if((cloud.compare(tr("model"))==0) && pclTools->_modelIsLoaded)
	{
		resizeButton->setEnabled(true);
		transformButton->setEnabled(true);
```

```
            segmentButton->setEnabled(true);
            filterButton->setEnabled(true);
            toolsButton->setEnabled(true);
            showModel->setEnabled(true);
            showModel->setChecked(true);
      }
      if(pclTools->_modelIsLoaded && pclTools->_sceneIsLoaded)
      {
            showBoth->setEnabled(true);
            resizeButton->setEnabled(true);
            transformButton->setEnabled(true);
            segmentButton->setEnabled(true);
            filterButton->setEnabled(true);
            toolsButton->setEnabled(true);
            recognitionButton->setEnabled(true);
            ismRecButton->setEnabled(true);
      }
}

void CWidget::setUpperWidget(int widgetNum)
{
      upperWidget->setCurrentIndex(widgetNum);
}
```

B.6 panels.h

```
#pragma once

#include "centralwidget.h"
#include <iostream>
#include <QWidget>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QGroupBox>
#include <QLineEdit>
#include <QLabel>
#include <QDoubleValidator>
#include <QIntValidator>
#include <QRadioButton>
#include <QColor>

class Button;
class CWidget;

class ResizePanel : public QWidget
{
	Q_OBJECT
public:
	ResizePanel(CWidget *parent = 0);
	void enablePanel(std::string cloud);
	void disablePanel(std::string cloud);
private:
	Button *createButton(const QString &text, const char *member);
	CWidget *parentCWidget;
	double incrFactor;
	double decrFactor;
	QGridLayout *bLayout;
	QGridLayout *mLayout;
	QGridLayout *sLayout;

	Button *back;
	Button *mSet;
	Button *mPlus;
	Button *mMinus;
	Button *mrEqModel;
	QLineEdit *mFactor;

	Button *sSet;
	Button *sPlus;
	Button *sMinus;
	Button *mrEqScene;
	QLineEdit *sFactor;
```

```
private slots:
	void resizeClicked();
	void mPlusMinusClicked();
	void sPlusMinusClicked();
	void backClicked();
	void mEqualizeClicked();
	void sEqualizeClicked();
};

class SegmentPanel : public QWidget
{
	Q_OBJECT
public:
	SegmentPanel(CWidget *parent = 0);
	void enablePanel();
	void disablePanel();
	void setActivePanel(bool isActive);
	void resetBoundingBox();
private:
	std::string currentCloud;

	bool panelIsActive;
	bool isEnabled;
	pcl::PointXYZ centerPoint;
	float cloudMaxRadius;

	QGridLayout *rbLayout;
	QGridLayout *sizeLayout;
	QGridLayout *rotLayout;
	QGridLayout *bLayout;
	QGridLayout *sLayout;
	QHBoxLayout *panelLayout;

	CWidget *parentCWidget;
	Button *back;

	QLabel *choseAxis;
	QLabel *boxAdjLabel;
	QLabel *boxRotLabel;
	QLabel *currCloudLabel;

	Button *maxIncr;
	Button *maxDecr;
	Button *minIncr;
	Button *minDecr;
	Button *rotRight;
	Button *rotLeft;

	Button *segment;
	Button *resetBox;

	Button *remDomPlane;
	QLineEdit *ransacDistThres;

	QGroupBox *rButtons;
	QHBoxLayout *groupLayout;
	QRadioButton *rbX;
	QRadioButton *rbY;
	QRadioButton *rbZ;
```

```
        std::vector<float> boxParameters;
        double boxIncrement;

        double getBoxIncrement();
        char getCheckedAxis();
        void drawBoundingBox();
        void initBoundingBox();
        Button *createButton(const QString &text, const char *member);
        QRadioButton *createRadioButton(const QString &text, const char
*member);
private slots:
        void leftRightClicked();
        void incrDecrClicked();
        void backClicked();
        void segmentClicked();
        void remDomPlaneClicked();
        void resetClicked();
        void radioButtonClicked();
};

class FilterPanel : public QWidget
{
        Q_OBJECT
public:
        FilterPanel(CWidget *parent = 0);
        void enablePanel();
        void disablePanel();
private:
        QGridLayout *voxelLayout;
        QGridLayout *outlierLayout;

        QGridLayout *bLayout;
        QHBoxLayout *panelLayout;
      Button *createButton(const QString &text, const char *member);
        CWidget *parentCWidget;
        Button *back;
        Button *voxelFilter;
        Button *outlierFilter;
        QLineEdit *voxelLeaf;
        QLineEdit *outlierRad;
        QLineEdit *outlierNeigh;

private slots:
        void buttonClicked();
        void backClicked();
};

class ToolsPanel : public QWidget
{
        Q_OBJECT
public:
        ToolsPanel(CWidget *parent = 0);
        void enablePanel(std::string cloud);
        void disablePanel();
```

```
private:
	QGridLayout *tLayout;
	QGridLayout *bLayout;
	QHBoxLayout *panelLayout;
	Button *createButton(const QString &text, const char *member);
	CWidget *parentCWidget;
	Button *back;
	Button *modelDensity;
	Button *sceneDensity;
	Button *mergeButton;
	Button *fitnessButton;
private slots:
	void buttonClicked();
	void backClicked();
};

class RecognitionPanel : public QWidget
{
	Q_OBJECT
public:
	RecognitionPanel(CWidget *parent = 0);
	void enablePanel();
	void disablePanel();
	void enableCyclePanel(bool enableNext);
	void disableCyclePanel();
private:
	QGridLayout *icpLayout;
	QGridLayout *bLayout;
	QHBoxLayout *panelLayout;
	QLabel *icpItrLabel;
	Button *icpButton;
	QLineEdit *icpItr;

	QGridLayout *recLayout;
	Button *recognize;
	QLabel *model_ss_lab;
	QLabel *scene_ss_lab;
	QLabel *descr_rad_lab;
	QLabel *rf_rad_lab;
	QLabel *cg_size_lab;
	QLabel *cg_tresh_lab;
	QLineEdit *model_ss;
	QLineEdit *scene_ss;
	QLineEdit *descr_rad;
	QLineEdit *rf_rad;
	QLineEdit *cg_size;
	QLineEdit *cg_tresh;

	QGridLayout *cycleLayout;
	Button *nextButton;
	Button *useButton;
	QLabel *cycleLabel;
	QLabel *instLabel;

	QGridLayout *changeLayout;
	Button *changesButton;
	QLineEdit *octreeRes;
	QLabel *resLabel;
```

```
        Button *createButton(const QString &text, const char *member);
        CWidget *parentCWidget;
        Button *back;
private slots:
        void buttonClicked();
        void backClicked();
};


class ismRecognitionPanel : public QWidget
{
        Q_OBJECT
public:
        ismRecognitionPanel(CWidget *parent = 0);
        void enablePanel();
        void disablePanel();
private:
        QGridLayout *bLayout;
        QHBoxLayout *panelLayout;

        QGridLayout *ismLayout;
        Button *ismRecognize;
        QLabel *s_size_lab;
        QLabel *descr_rad_lab;
        QLabel *normal_rad_lab;
        QLabel *thresh_lab;

        QLineEdit *s_size;
        QLineEdit *descr_rad;
        QLineEdit *normal_rad;
        QLineEdit *thresh;

        QLabel *resLabel;

        Button *createButton(const QString &text, const char *member);
        CWidget *parentCWidget;
        Button *back;
private slots:
        void buttonClicked();
        void backClicked();
};


class TransPanel : public QWidget
{
        Q_OBJECT
public:
        TransPanel(CWidget *parent = 0);
        void enablePanel(std::string cloud);
        void disablePanel();
private:
        float moveStep;
        double rotateStep;

        QGridLayout *bLayout;
        QGroupBox   *axisSelGroup;
        QHBoxLayout *axisSelLayout;
```

```
        QHBoxLayout *modelTransLayout;
        QHBoxLayout *sceneTransLayout;

        QGridLayout *mTransLayout;
        QGridLayout *sTransLayout;

        QHBoxLayout *panelLayout;

        QRadioButton *xaxis;
        QRadioButton *yaxis;
        QRadioButton *zaxis;

        Button *mMoveRight;
        Button *mMoveLeft;
        Button *sMoveRight;
        Button *sMoveLeft;
        Button *mRotLeft;
        Button *mRotRight;
        Button *sRotLeft;
        Button *sRotRight;
        Button *mMatrix;
        Button *sMatrix;

        QRadioButton *createRadioButton(const QString &text, const char
*member);
        Button *createButton(const QString &text, const char *member);
        char getCheckedAxis();
        CWidget *parentCWidget;
        Button *back;
        Eigen::Matrix4f transMatrixBox();

private slots:
        void backClicked();
        void mMoveClicked();
        void mRotClicked();
        void sMoveClicked();
        void sRotClicked();
        void mMatrixClicked();
        void sMatrixClicked();
        void radioClicked();
};
```

B.7 panels.cpp

```
#include "panels.h"
#include "button.h"

//------------------------------------------------------------
//Resize Panel------------------------------------------------
//------------------------------------------------------------
ResizePanel::ResizePanel(CWidget *parent)
{

      incrFactor = 1.01;
      decrFactor = 0.99;
      parentCWidget = parent;
    QHBoxLayout *panelLayout = new QHBoxLayout;
      mLayout = new QGridLayout;
      sLayout = new QGridLayout;
      bLayout = new QGridLayout;
      mLayout->setSizeConstraint(QLayout::SetFixedSize);
      sLayout->setSizeConstraint(QLayout::SetFixedSize);
      mLayout->setSizeConstraint(QLayout::SetFixedSize);

      QWidget *spacer = new QWidget;

      QLabel *mLabel = new QLabel(tr("Model:"));
      QLabel *sLabel = new QLabel(tr("Scene:"));

      back = createButton(tr("Back"), SLOT(backClicked()));
      back->setMaximumWidth(60);

      QLabel *factorLabel1 = new QLabel(tr("Factor:"));
      QLabel *factorLabel2 = new QLabel(tr("Factor:"));

      QGroupBox *verticalLine = new QGroupBox;
      verticalLine->setFixedWidth(2);
      verticalLine->setFixedHeight(55);
      verticalLine->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);

      mSet = createButton(tr("Resize Model"), SLOT(resizeClicked()));
      mPlus = createButton(tr("+"), SLOT(mPlusMinusClicked()));
      mMinus = createButton(tr("-"), SLOT(mPlusMinusClicked()));
      mrEqModel = createButton(tr("Equalize scale"),
SLOT(mEqualizeClicked()));
      mSet->setMaximumWidth(80);
      mPlus->setMaximumHeight(15);
      mMinus->setMaximumHeight(15);
      mSet->setEnabled(false);
      mPlus->setEnabled(false);
      mMinus->setEnabled(false);
      mrEqModel->setEnabled(false);
```

```
        sSet = createButton(tr("Resize Scene"), SLOT(resizeClicked()));
        sPlus = createButton(tr("+"), SLOT(sPlusMinusClicked()));
        sMinus = createButton(tr("-"), SLOT(sPlusMinusClicked()));
        mrEqScene = createButton(tr("Equalize scale"),
SLOT(sEqualizeClicked()));
        sSet->setMaximumWidth(80);
        sPlus->setMaximumHeight(15);
        sMinus->setMaximumHeight(15);
        sSet->setEnabled(false);
        sPlus->setEnabled(false);
        sMinus->setEnabled(false);
        mrEqScene->setEnabled(false);

        mFactor = new QLineEdit("1");
        mFactor->setAlignment(Qt::AlignRight);
        mFactor->setMaximumWidth(40);
        mFactor->setMaxLength(4);
        mFactor->setValidator( new QDoubleValidator(0.01, 100,2) );
        mFactor->setEnabled(false);

        sFactor = new QLineEdit("1");
        sFactor->setAlignment(Qt::AlignRight);
        sFactor->setMaximumWidth(40);
        sFactor->setMaxLength(4);
        sFactor->setValidator( new QDoubleValidator(0.01, 100,2) );
        sFactor->setEnabled(false);

        bLayout->addWidget(back,0,0,2,2,Qt::AlignLeft);

        mLayout->addWidget(mLabel,          0,0,2,1,Qt::AlignLeft);
        mLayout->addWidget(factorLabel1,0,1,1,1,Qt::AlignBottom);
        mLayout->addWidget(mFactor,         1,1,1,1,Qt::AlignLeft);
        mLayout->addWidget(mSet,            0,2,2,1,Qt::AlignLeft);
        mLayout->addWidget(mPlus,           0,3,1,1,Qt::AlignLeft);
        mLayout->addWidget(mMinus,          1,3,1,1,Qt::AlignLeft);
        mLayout->addWidget(mrEqModel, 0,4,2,1,Qt::AlignLeft);

        sLayout->addWidget(sLabel,          0,0,2,1,Qt::AlignLeft);
        sLayout->addWidget(factorLabel2,0,1,1,1,Qt::AlignBottom);
        sLayout->addWidget(sFactor,         1,1,1,1,Qt::AlignLeft);
        sLayout->addWidget(sSet,            0,2,2,1,Qt::AlignLeft);
        sLayout->addWidget(sPlus,           0,3,1,1,Qt::AlignLeft);
        sLayout->addWidget(sMinus,          1,3,1,1,Qt::AlignLeft);
        sLayout->addWidget(mrEqScene, 0,4,2,1,Qt::AlignLeft);

        spacer->setMinimumWidth(20);


        panelLayout->setAlignment(Qt::AlignLeft);
        panelLayout->addLayout(bLayout);
        panelLayout->addLayout(mLayout);
        panelLayout->addWidget(spacer);
        panelLayout->addLayout(sLayout);
        setLayout(panelLayout);
}
```

```
void ResizePanel::backClicked()
{
	parentCWidget->setUpperWidget(0);
}

void ResizePanel::enablePanel(std::string cloud)
{
	if(cloud.compare("model")==0)
	{
		mSet->setEnabled(true);
		mPlus->setEnabled(true);
		mMinus->setEnabled(true);
		mFactor->setEnabled(true);
		if(parentCWidget->getPCL()->_sceneIsLoaded)
		{
			mrEqScene->setEnabled(true);
			mrEqModel->setEnabled(true);
		}
	}
	if(cloud.compare("scene")==0)
	{
		sSet->setEnabled(true);
		sPlus->setEnabled(true);
		sMinus->setEnabled(true);
		sFactor->setEnabled(true);
		if(parentCWidget->getPCL()->_modelIsLoaded)
		{
			mrEqScene->setEnabled(true);
			mrEqModel->setEnabled(true);
		}
	}
}

void ResizePanel::disablePanel(std::string cloud)
{
	if(cloud.compare("model")==0)
	{
		mSet->setEnabled(false);
		mPlus->setEnabled(false);
		mMinus->setEnabled(false);
		mFactor->setEnabled(false);
	}
	if(cloud.compare("scene")==0)
	{
		sSet->setEnabled(false);
		sPlus->setEnabled(false);
		sMinus->setEnabled(false);
		sFactor->setEnabled(false);
	}

}

void ResizePanel::mEqualizeClicked()
{
	QString factor = QString::number(parentCWidget->getPCL()-
>maxRadiusEqualization("model","scene"));
	parentCWidget->setDisplay(tr("Model resized with factor ")+factor);
```

```
        parentCWidget->getPCL()->show(parentCWidget->getChecked());
}
void ResizePanel::sEqualizeClicked()
{
        QString factor = QString::number(parentCWidget->getPCL()-
>maxRadiusEqualization("scene","model"));
        parentCWidget->setDisplay(tr("Model resized with factor ")+factor);
        parentCWidget->getPCL()->show(parentCWidget->getChecked());
}

void ResizePanel::resizeClicked()
{
        Button *clickedButton = qobject_cast<Button *>(sender());
        QString clicked = clickedButton->text();

        if(clicked.compare("Resize Model")==0)
        {

                double mF = mFactor->text().toDouble();
                if(parentCWidget->getPCL()->resizeCloud("model",mF)==0)
                {
                        parentCWidget->setDisplay(tr("Model resized with factor
")+mFactor->text());
                        parentCWidget->getPCL()->show(parentCWidget->getChecked());
                }
        }
        else if(clicked.compare("Resize Scene")==0)
        {
                double sF = sFactor->text().toDouble();
                if(parentCWidget->getPCL()->resizeCloud("scene",sF)==0)
                {
                        parentCWidget->setDisplay(tr("Scene resized with factor
")+sFactor->text());
                        parentCWidget->getPCL()->show(parentCWidget->getChecked());
                }
        }
}

void ResizePanel::mPlusMinusClicked()
{
        Button *clickedButton = qobject_cast<Button *>(sender());
        QString clicked = clickedButton->text();

        if(clicked.compare("+")==0)
        {
                QString factor = QString::number(incrFactor);
                if(parentCWidget->getPCL()->resizeCloud("model",incrFactor)==0)
                {
                        parentCWidget->setDisplay(tr("Model resized with factor
")+factor);
                        parentCWidget->getPCL()->show(parentCWidget->getChecked());
                }
        }
        else if(clicked.compare("-")==0)
        {
        QString factor = QString::number(decrFactor);
                if(parentCWidget->getPCL()->resizeCloud("model",decrFactor)==0)
```

```
            {
                  parentCWidget->setDisplay(tr("Model resized with factor
")+factor);
                  parentCWidget->getPCL()->show(parentCWidget->getChecked());
            }
      }
}

void ResizePanel::sPlusMinusClicked()
{
      Button *clickedButton = qobject_cast<Button *>(sender());
      QString clicked = clickedButton->text();

      if(clicked.compare("+")==0)
      {
            QString factor = QString::number(incrFactor);
            if(parentCWidget->getPCL()->resizeCloud("scene",incrFactor)==0)
            {
                  parentCWidget->setDisplay(tr("Scene resized with factor
")+factor);
                  parentCWidget->getPCL()->show(parentCWidget->getChecked());
            }
      }
      else if(clicked.compare("-")==0)
      {
            QString factor = QString::number(decrFactor);
            if(parentCWidget->getPCL()->resizeCloud("scene",decrFactor)==0)
            {
                  parentCWidget->setDisplay(tr("Scene resized with factor
")+factor);
                  parentCWidget->getPCL()->show(parentCWidget->getChecked());
            }
      }
}


Button *ResizePanel::createButton(const QString &text, const char *member)
{
      Button *button = new Button(text);
      connect(button, SIGNAL(clicked()), this, member);
      return button;
}

//-------------------------------------------------------------
//Segment Panel-----------------------------------------------
//-------------------------------------------------------------
SegmentPanel::SegmentPanel(CWidget *parent)
{
      parentCWidget = parent;
      panelLayout = new QHBoxLayout();
      bLayout = new QGridLayout();
      sizeLayout = new QGridLayout();
      rotLayout = new QGridLayout();
      sLayout= new QGridLayout();
      rbLayout = new QGridLayout();
      bLayout->setSizeConstraint(QLayout::SetFixedSize);
      sizeLayout->setSizeConstraint(QLayout::SetFixedSize);
```

```
	rotLayout->setSizeConstraint(QLayout::SetFixedSize);
	sLayout->setSizeConstraint(QLayout::SetFixedSize);

	QWidget *spacer = new QWidget();
	spacer->setMinimumWidth(5);
	QGroupBox *verticalLine = new QGroupBox;
	verticalLine->setFixedWidth(2);
	verticalLine->setFixedHeight(55);
	verticalLine->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);


	choseAxis=new QLabel(tr("Chose Axis"));
	boxAdjLabel=new QLabel(tr("Adjust Box"));
	boxRotLabel=new QLabel(tr("Rotate Cloud"));

	currCloudLabel=new QLabel(tr(" "));

	rButtons =  new QGroupBox;
	groupLayout =     new QHBoxLayout;

	rbX = createRadioButton(tr("X-axis"),SLOT(radioButtonClicked()));
	rbY = createRadioButton(tr("Y-axis"),SLOT(radioButtonClicked()));
	rbZ = createRadioButton(tr("Z-axis"),SLOT(radioButtonClicked()));
	rbX->setChecked(true);

	groupLayout->addWidget(rbX);
	groupLayout->addWidget(rbY);
	groupLayout->addWidget(rbZ);

	rButtons->setLayout(groupLayout);

	maxIncr           = createButton("Max +",SLOT(incrDecrClicked()));
	maxDecr           = createButton("Max -",SLOT(incrDecrClicked()));
	minIncr           = createButton("Min +",SLOT(incrDecrClicked()));
	minDecr           = createButton("Min -",SLOT(incrDecrClicked()));
	rotRight     = createButton("-->",SLOT(leftRightClicked()));
	rotLeft           = createButton("<--",SLOT(leftRightClicked()));
	segment           = createButton("Segment",SLOT(segmentClicked()));
	resetBox     = createButton("Reset Box",SLOT(resetClicked()));
	remDomPlane = createButton("Remove Dominant
Plane",SLOT(remDomPlaneClicked()));

	minDecr->setMaximumHeight(15);
	maxIncr->setMaximumHeight(15);
	maxDecr->setMaximumHeight(15);
	minIncr->setMaximumHeight(15);
	rotRight->setMaximumHeight(32);
	rotLeft->setMaximumHeight(32);

	QLabel *distThresLabel = new QLabel("Threshold:");
	QLabel *unit = new QLabel("[m]");
	ransacDistThres = new QLineEdit("0.1");
	ransacDistThres->setAlignment(Qt::AlignRight);
	ransacDistThres->setMaximumWidth(40);
	ransacDistThres->setMaxLength(4);
	ransacDistThres->setValidator( new QDoubleValidator(0.01, 100,2) );
```

```
	rbLayout->addWidget(choseAxis,		0,1,1,1,Qt::AlignLeft);
	rbLayout->addWidget(rButtons,	1,1,1,1,Qt::AlignLeft);

	sizeLayout->addWidget(boxAdjLabel,	0,0,1,2,Qt::AlignLeft);
	sizeLayout->addWidget(maxIncr,		1,1,1,1,Qt::AlignTop);
	sizeLayout->addWidget(minIncr,		1,2,1,1,Qt::AlignTop);
	sizeLayout->addWidget(maxDecr,		2,1,1,1,Qt::AlignTop);
	sizeLayout->addWidget(minDecr,		2,2,1,1,Qt::AlignTop);

	rotLayout->addWidget(boxRotLabel,	0,0,1,2,Qt::AlignLeft);
	rotLayout->addWidget(rotLeft,		1,0,1,1,Qt::AlignTop);
	rotLayout->addWidget(rotRight,		1,1,1,1,Qt::AlignTop);

	sLayout->addWidget(segment,		0,0,2,1,Qt::AlignLeft);
	sLayout->addWidget(resetBox,	0,1,2,1,Qt::AlignLeft);
	sLayout->addWidget(verticalLine,	0,2,2,1,Qt::AlignLeft);
	sLayout->addWidget(spacer,		0,3,2,1,Qt::AlignLeft);
	sLayout->addWidget(distThresLabel,	0,4,1,3,Qt::AlignLeft);
	sLayout->addWidget(ransacDistThres, 1,4,1,1,Qt::AlignTop);
	sLayout->addWidget(unit,		1,5,1,1,Qt::AlignTop);
	sLayout->addWidget(remDomPlane,		0,7,2,1,Qt::AlignLeft);

	back = createButton("Back",SLOT(backClicked()));

	bLayout->addWidget(back,		0,2,2,Qt::AlignLeft);

	panelLayout->setSizeConstraint(QLayout::SetFixedSize);
	panelLayout->setAlignment(Qt::AlignLeft);
	panelLayout->addLayout(bLayout);
	panelLayout->addLayout(rbLayout);
	panelLayout->addLayout(sizeLayout);
	panelLayout->addLayout(rotLayout);
	panelLayout->addLayout(sLayout);

	setLayout(panelLayout);

	minDecr->setEnabled(false);
	maxIncr->setEnabled(false);
	maxDecr->setEnabled(false);
	minIncr->setEnabled(false);
	rotRight->setEnabled(false);
	rotLeft->setEnabled(false);
	segment->setEnabled(false);
	remDomPlane->setEnabled(false);
	ransacDistThres->setEnabled(false);
	resetBox->setEnabled(false);
	rbX->setEnabled(false);
	rbY->setEnabled(false);
	rbZ->setEnabled(false);

	isEnabled=false;
	panelIsActive=false;
}

void SegmentPanel::enablePanel()
{
	currentCloud = parentCWidget->getChecked();
```

```
	if(currentCloud.compare("model")==0 || currentCloud.compare("scene")==0)
	{
		if(currentCloud.compare("model")==0)
		{
			currCloudLabel->setText(tr("Model:"));
		}
		if(currentCloud.compare("scene")==0)
		{
			currCloudLabel->setText(tr("Scene:"));
		}

		minDecr->setEnabled(true);
		maxIncr->setEnabled(true);
		maxDecr->setEnabled(true);
		minIncr->setEnabled(true);
		rotRight->setEnabled(true);
		rotLeft->setEnabled(true);
		segment->setEnabled(true);
		remDomPlane->setEnabled(true);
		ransacDistThres->setEnabled(true);
		resetBox->setEnabled(true);
		rbX->setEnabled(true);
		rbY->setEnabled(true);
		rbZ->setEnabled(true);

		isEnabled=true;

	}
}

void SegmentPanel::setActivePanel(bool isActive)
{
	panelIsActive = isActive;
	if(panelIsActive&&isEnabled)
	{
		initBoundingBox();
		drawBoundingBox();
	}
	else
	{
		parentCWidget->getPCL()->removeBoundingBox();
	}

}

void SegmentPanel::resetBoundingBox()
{
	if(panelIsActive)
	{
		currentCloud = parentCWidget->getChecked();
		parentCWidget->getPCL()->removeBoundingBox();
		boxParameters = parentCWidget->getPCL()-
>getInitBoundingBox(currentCloud);
		drawBoundingBox();
	}
}
```

```
void SegmentPanel::initBoundingBox()
{
	currentCloud = parentCWidget->getChecked();
	boxParameters = parentCWidget->getPCL()-
>getInitBoundingBox(currentCloud);
}

void SegmentPanel::drawBoundingBox()
{
	parentCWidget->getPCL()->showBoundingBox(boxParameters);
}

void SegmentPanel::disablePanel()
{
	minDecr->setEnabled(false);
	maxIncr->setEnabled(false);
	maxDecr->setEnabled(false);
	minIncr->setEnabled(false);
	rotRight->setEnabled(false);
	rotLeft->setEnabled(false);
	segment->setEnabled(false);
	remDomPlane->setEnabled(false);
	ransacDistThres->setEnabled(false);
	resetBox->setEnabled(false);
	rbX->setEnabled(false);
	rbY->setEnabled(false);
	rbZ->setEnabled(false);

	isEnabled=false;
}
char SegmentPanel::getCheckedAxis()
{
	if(rbX->isChecked())
		return 'x';
	else if (rbY->isChecked())
		return 'y';
	else if (rbZ->isChecked())
		return 'z';
}

double SegmentPanel::getBoxIncrement()
{
	double stepFactor = 0.05;
	double xLength = boxParameters[1]-boxParameters[0];
	double yLength = boxParameters[3]-boxParameters[2];
	double zLength = boxParameters[5]-boxParameters[4];

	if (xLength<yLength && xLength<zLength)
		return xLength*stepFactor;
	if (yLength<zLength)
		return yLength*stepFactor;
	else
		return zLength*stepFactor;

}

Button *SegmentPanel::createButton(const QString &text, const char *member)
```

```
{
	Button *button = new Button(text);
	connect(button, SIGNAL(clicked()), this, member);
	return button;
}
QRadioButton *SegmentPanel::createRadioButton(const QString &text, const char
*member)
{
	QRadioButton *rbutton = new QRadioButton(text);
	connect(rbutton, SIGNAL(clicked()), this,member);
	return rbutton;
}
void SegmentPanel::backClicked()
{
	setActivePanel(false);
	parentCWidget->setUpperWidget(0);
	//remove bounding box
}

void SegmentPanel::segmentClicked()
{
	currentCloud = parentCWidget->getChecked();
	int sizeBefore(parentCWidget->getPCL()->getCloudSize(currentCloud));

	parentCWidget->getPCL()->boxSegmentation(currentCloud, boxParameters);
	resetBoundingBox();

	int sizeAfter(parentCWidget->getPCL()->getCloudSize(currentCloud));
	int pointsRemoved = sizeBefore-sizeAfter;
	stringstream toDisplay;
	toDisplay << pointsRemoved << " points removed from " << currentCloud
<<"-cloud";
	parentCWidget->setDisplay(QString::fromStdString(toDisplay.str()));
}

void SegmentPanel::remDomPlaneClicked()
{
	currentCloud = parentCWidget->getChecked();
	int sizeBefore(parentCWidget->getPCL()->getCloudSize(currentCloud));

	double distanceThreshold = ransacDistThres->text().toDouble();
	parentCWidget->getPCL()->planeFitting(currentCloud,distanceThreshold);
	resetBoundingBox();

	int sizeAfter(parentCWidget->getPCL()->getCloudSize(currentCloud));
	int pointsRemoved = sizeBefore-sizeAfter;
	stringstream toDisplay;
	toDisplay << pointsRemoved << " points removed from " << currentCloud
<<"-cloud";
	parentCWidget->setDisplay(QString::fromStdString(toDisplay.str()));
}

void SegmentPanel::resetClicked()
{
	resetBoundingBox();
}
```

```
void SegmentPanel::leftRightClicked()
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();
	double angleStep = 5.0;

	char axis = getCheckedAxis();

	if(clicked.compare("-->")==0)
	{
		parentCWidget->getPCL()->rotateCloud(currentCloud,axis,angleStep);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
	}
	else if(clicked.compare("<--")==0)
	{
		parentCWidget->getPCL()->rotateCloud(currentCloud,axis,-
angleStep);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
	}
	resetBoundingBox();

}
void SegmentPanel::incrDecrClicked()
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();

	boxIncrement = getBoxIncrement();

	char axis = getCheckedAxis();
	if(axis=='x')
	{
		if(clicked.compare("Min +")==0)
			boxParameters[0]-=boxIncrement;
		else if(clicked.compare("Max +")==0)
			boxParameters[1]+=boxIncrement;
		else if(clicked.compare("Min -")==0)
			boxParameters[0]+=boxIncrement;
		else if(clicked.compare("Max -")==0)
			boxParameters[1]-=boxIncrement;
	}
	else if(axis=='y')
	{
		if(clicked.compare("Min +")==0)
			boxParameters[2]-=boxIncrement;
		else if(clicked.compare("Max +")==0)
			boxParameters[3]+=boxIncrement;
		else if(clicked.compare("Min -")==0)
			boxParameters[2]+=boxIncrement;
		else if(clicked.compare("Max -")==0)
			boxParameters[3]-=boxIncrement;
	}
	else if(axis=='z')
	{
		if(clicked.compare("Min +")==0)
			boxParameters[4]-=boxIncrement;
		else if(clicked.compare("Max +")==0)
```

```
                boxParameters[5]+=boxIncrement;
            else if(clicked.compare("Min -")==0)
                boxParameters[4]+=boxIncrement;
            else if(clicked.compare("Max -")==0)
                boxParameters[5]-=boxIncrement;
        }
        drawBoundingBox();
}
void SegmentPanel::radioButtonClicked(){}

//-------------------------------------------------------------
//Filter Panel------------------------------------------------
//-------------------------------------------------------------
FilterPanel::FilterPanel(CWidget *parent)
{
        parentCWidget = parent;
        panelLayout = new QHBoxLayout();
        bLayout = new QGridLayout();

        voxelLeaf = new QLineEdit("0.03");
        voxelLeaf->setAlignment(Qt::AlignRight);
        voxelLeaf->setMaximumWidth(40);
        voxelLeaf->setMaxLength(5);
        voxelLeaf->setValidator( new QDoubleValidator(0.001, 100,2) );
        voxelLeaf->setEnabled(false);

        QLabel *voxelLabel = new QLabel(tr("Leaf size[m]:"));
        voxelFilter = createButton(tr("Voxel Filter"),SLOT(buttonClicked()));
        voxelFilter->setEnabled(false);

        voxelLayout = new QGridLayout();
        voxelLayout->setAlignment(Qt::AlignLeft);
        voxelLayout->addWidget(voxelLabel,      0,1,1,1,Qt::AlignLeft);
        voxelLayout->addWidget(voxelLeaf,       1,1,1,1,Qt::AlignLeft);
        voxelLayout->addWidget(voxelFilter,     0,2,2,1,Qt::AlignLeft);

        QWidget *space1 = new QWidget();
        space1->setFixedWidth(10);
        QWidget *space2 = new QWidget();
        space2->setFixedWidth(10);
        QWidget *space3 = new QWidget();
        space3->setFixedWidth(10);

        outlierRad = new QLineEdit("0.5");
        outlierRad->setAlignment(Qt::AlignRight);
        outlierRad->setMaximumWidth(40);
        outlierRad->setMaxLength(5);
        outlierRad->setValidator( new QDoubleValidator(0.001, 100,2) );
        outlierRad->setEnabled(false);

        outlierNeigh = new QLineEdit("500");
        outlierNeigh->setAlignment(Qt::AlignRight);
        outlierNeigh->setMaximumWidth(40);
        outlierNeigh->setMaxLength(5);
        outlierNeigh->setValidator( new QIntValidator(1, 99999) );
        outlierNeigh->setEnabled(false);
```

```
	QLabel *radLabel = new QLabel(tr("Radius:"));
	QLabel *neighLabel = new QLabel(tr("Neighbors:"));
	outlierFilter = createButton(tr("Outlier 
Removal"),SLOT(buttonClicked()));
	outlierFilter-> setEnabled(false);

	outlierLayout = new QGridLayout();
	outlierLayout->setAlignment(Qt::AlignLeft);
	outlierLayout->addWidget(radLabel,		0,1,1,1,Qt::AlignLeft);
	outlierLayout->addWidget(outlierRad,		1,1,1,1,Qt::AlignLeft);
	outlierLayout->addWidget(neighLabel,		0,2,1,1,Qt::AlignLeft);
	outlierLayout->addWidget(outlierNeigh,		1,2,1,1,Qt::AlignLeft);
	outlierLayout->addWidget(outlierFilter,		0,3,2,1,Qt::AlignLeft);

	back = createButton("Back",SLOT(backClicked()));
	bLayout->addWidget(back,0,0,2,2,Qt::AlignLeft);

	panelLayout->setAlignment(Qt::AlignLeft);
	panelLayout->addLayout(bLayout);
	panelLayout->addWidget(space1);
	panelLayout->addLayout(voxelLayout);
	panelLayout->addWidget(space2);
	panelLayout->addLayout(outlierLayout);

	setLayout(panelLayout);
}

void FilterPanel::enablePanel()
{
	std::string currentCloud = parentCWidget->getChecked();
	if(currentCloud.compare("model")==0 || currentCloud.compare("scene")==0)
	{
		voxelLeaf->setEnabled(true);
		voxelFilter->setEnabled(true);
		outlierRad->setEnabled(true);
		outlierFilter->setEnabled(true);
		outlierNeigh->setEnabled(true);
	}
}

void FilterPanel::disablePanel()
{
	voxelFilter->setEnabled(false);
	voxelLeaf->setEnabled(false);
	outlierRad->setEnabled(false);
	outlierFilter->setEnabled(false);
	outlierNeigh->setEnabled(false);
}

Button *FilterPanel::createButton(const QString &text, const char *member)
{
	Button *button = new Button(text);
	connect(button, SIGNAL(clicked()), this, member);
	return button;
}

void FilterPanel::buttonClicked()
```

```
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();

	if(clicked.compare("Voxel Filter")==0)
	{
		parentCWidget->getPCL()->voxelGrid(parentCWidget-
>getChecked(),voxelLeaf->text().toFloat());
	}
	if(clicked.compare("Outlier Removal")==0)
	{
		std::string cloud = parentCWidget->getChecked();
		double radius = outlierRad->text().toDouble();
		int neighbors = outlierNeigh->text().toInt();
		parentCWidget->getPCL()-
>radOutlierRemoval(cloud,radius,neighbors);
	}
}

void FilterPanel::backClicked()
{
	parentCWidget->setUpperWidget(0);
}

//------------------------------------------------------------
//Tools Panel-------------------------------------------------
//------------------------------------------------------------

ToolsPanel::ToolsPanel(CWidget *parent)
{
	parentCWidget = parent;
	panelLayout = new QHBoxLayout();
	bLayout = new QGridLayout();
	tLayout = new QGridLayout();

	back = createButton("Back",SLOT(backClicked()));
	bLayout->addWidget(back,0,0,2,2,Qt::AlignLeft);

	modelDensity = createButton("Model Density", SLOT(buttonClicked()));
	sceneDensity = createButton("Scene Density", SLOT(buttonClicked()));
	mergeButton = createButton("Merge Clouds", SLOT(buttonClicked()));
	fitnessButton = createButton("Fitness Score", SLOT(buttonClicked()));

	modelDensity->setEnabled(false);
	sceneDensity->setEnabled(false);
	mergeButton->setEnabled(false);
	fitnessButton->setEnabled(false);

	tLayout->addWidget(modelDensity,0,0,1,1,Qt::AlignLeft);
	tLayout->addWidget(sceneDensity,0,1,1,1,Qt::AlignLeft);
	tLayout->addWidget(mergeButton,    0,2,1,1,Qt::AlignLeft);
	tLayout->addWidget(fitnessButton,0,3,1,1,Qt::AlignLeft);

	panelLayout->setAlignment(Qt::AlignLeft);
	panelLayout->addLayout(bLayout);
	panelLayout->addLayout(tLayout);
```

```
	setLayout(panelLayout);
}

void ToolsPanel::enablePanel(std::string cloud)
{
	if(cloud.compare("model")==0)
		modelDensity->setEnabled(true);
	if(cloud.compare("scene")==0)
		sceneDensity->setEnabled(true);
	if(parentCWidget->getPCL()->_sceneIsLoaded && parentCWidget->getPCL()-
>_modelIsLoaded)
	{
		mergeButton->setEnabled(true);
		fitnessButton->setEnabled(false);
	}
}

void ToolsPanel::disablePanel()
{
	modelDensity->setEnabled(false);
	sceneDensity->setEnabled(false);
	mergeButton->setEnabled(false);
	fitnessButton->setEnabled(false);
}

void ToolsPanel::buttonClicked()
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();

	if(clicked.compare("Model Density")==0)
	{
		float dens = parentCWidget->getPCL()->getDensity("model");
		parentCWidget->setDisplay(tr("Model: Average distance between
points are ")+QString::number(dens));

	}
	if(clicked.compare("Scene Density")==0)
	{
		float dens = parentCWidget->getPCL()->getDensity("scene");
		parentCWidget->setDisplay(tr("Scene: Average distance between
points are ")+QString::number(dens));
	}
	if(clicked.compare("Merge Clouds")==0)
	{
		parentCWidget->getPCL()->mergeClouds();
	}
}

Button *ToolsPanel::createButton(const QString &text, const char *member)
{
	Button *button = new Button(text);
	connect(button, SIGNAL(clicked()), this, member);
	return button;
}
```

```
void ToolsPanel::backClicked()
{
      parentCWidget->setUpperWidget(0);
}

//-----------------------------------------------------------
//Recognition Panel------------------------------------------
//-----------------------------------------------------------

RecognitionPanel::RecognitionPanel(CWidget *parent)
{
      parentCWidget = parent;
      panelLayout = new QHBoxLayout();
      bLayout = new QGridLayout();
      icpLayout = new QGridLayout();
      icpButton = createButton(tr("Iterative Closest
Point"),SLOT(buttonClicked()));

      QWidget *spacer1 = new QWidget();
      spacer1->setFixedWidth(10);

      icpItrLabel = new QLabel("Max iterations:");

      icpItr = new QLineEdit("1");
      icpItr->setAlignment(Qt::AlignRight);
      icpItr->setMaximumWidth(40);
      icpItr->setMaxLength(3);
      icpItr->setValidator( new QIntValidator(0, 999) );

      icpLayout->addWidget(spacer1,       0,0,1,1,Qt::AlignLeft);
      icpLayout->addWidget(icpItrLabel,   0,1,1,1,Qt::AlignLeft);
      icpLayout->addWidget(icpItr,        1,1,1,1,Qt::AlignLeft);
      icpLayout->addWidget(icpButton,           0,2,2,1,Qt::AlignLeft);

      recognize = createButton("Recognize",SLOT(buttonClicked()));
      model_ss_lab = new QLabel("Model sr:");
      scene_ss_lab = new QLabel("Scene sr:");
      descr_rad_lab = new QLabel("Descr. r:");
      rf_rad_lab = new QLabel("Ref.frame r:");
      cg_size_lab = new QLabel("Cluster size:");
      cg_tresh_lab = new QLabel("Cluster thres:");

      model_ss =  new QLineEdit("0.05");
      model_ss->setAlignment(Qt::AlignRight);
      model_ss->setMaximumWidth(40);
      model_ss->setMaxLength(5);
      model_ss->setValidator( new QDoubleValidator(0, 1, 3) );

      scene_ss =  new QLineEdit("0.05");
      scene_ss->setAlignment(Qt::AlignRight);
      scene_ss->setMaximumWidth(40);
      scene_ss->setMaxLength(5);
      scene_ss->setValidator( new QDoubleValidator(0, 1, 3) );


      descr_rad = new QLineEdit("0.1");
      descr_rad->setAlignment(Qt::AlignRight);
```

```
descr_rad->setMaximumWidth(40);
descr_rad->setMaxLength(5);
descr_rad->setValidator( new QDoubleValidator(0, 1, 3) );

rf_rad =    new QLineEdit("0.1");
rf_rad->setAlignment(Qt::AlignRight);
rf_rad->setMaximumWidth(40);
rf_rad->setMaxLength(5);
rf_rad->setValidator( new QDoubleValidator(0, 1, 3) );

cg_size =   new QLineEdit("0.07");
cg_size->setAlignment(Qt::AlignRight);
cg_size->setMaximumWidth(40);
cg_size->setMaxLength(5);
cg_size->setValidator( new QDoubleValidator(0, 1, 3) );

cg_tresh =  new QLineEdit("5");
cg_tresh->setAlignment(Qt::AlignRight);
cg_tresh->setMaximumWidth(40);
cg_tresh->setMaxLength(5);
cg_tresh->setValidator( new QIntValidator(0, 99999) );

recLayout = new QGridLayout();

QWidget *spacer2 = new QWidget();
spacer2->setFixedWidth(10);

recLayout->addWidget(spacer2,      0,0,2,1,Qt::AlignLeft);
recLayout->addWidget(model_ss_lab, 0,1,1,1,Qt::AlignLeft);
recLayout->addWidget(model_ss,           1,1,1,1,Qt::AlignLeft);
recLayout->addWidget(scene_ss_lab, 0,2,1,1,Qt::AlignLeft);
recLayout->addWidget(scene_ss,           1,2,1,1,Qt::AlignLeft);
recLayout->addWidget(rf_rad_lab,   0,3,1,1,Qt::AlignLeft);
recLayout->addWidget(rf_rad,       1,3,1,1,Qt::AlignLeft);
recLayout->addWidget(descr_rad_lab, 0,4,1,1,Qt::AlignLeft);
recLayout->addWidget(descr_rad,          1,4,1,1,Qt::AlignLeft);
recLayout->addWidget(cg_size_lab,  0,5,1,1,Qt::AlignLeft);
recLayout->addWidget(cg_size,      1,5,1,1,Qt::AlignLeft);
recLayout->addWidget(cg_tresh_lab, 0,6,1,1,Qt::AlignLeft);
recLayout->addWidget(cg_tresh,           1,6,1,1,Qt::AlignLeft);
recLayout->addWidget(recognize,          0,7,2,1,Qt::AlignLeft);

cycleLayout = new QGridLayout();
nextButton = createButton("Next",SLOT(buttonClicked()));
useButton = createButton("Use",SLOT(buttonClicked()));
cycleLabel = new QLabel("Recognized transformations");
instLabel = new QLabel();

QWidget *spacer3 = new QWidget();
spacer3->setFixedWidth(10);
cycleLayout->addWidget(spacer3,          0,0,2,1,Qt::AlignLeft);
cycleLayout->addWidget(cycleLabel, 0,1,1,2,Qt::AlignLeft);
cycleLayout->addWidget(nextButton, 1,1,1,1,Qt::AlignLeft);
cycleLayout->addWidget(useButton,  1,2,1,1,Qt::AlignLeft);
cycleLayout->addWidget(instLabel,  0,3,2,1,Qt::AlignLeft);
nextButton->setEnabled(false);
useButton->setEnabled(false);
```

```
	QWidget *spacer4 = new QWidget();
	spacer4->setFixedWidth(10);

	changeLayout = new QGridLayout();
	changesButton = createButton("Show differences",SLOT(buttonClicked()));
	octreeRes = new QLineEdit("0.03");
	octreeRes->setAlignment(Qt::AlignRight);
	octreeRes->setMaximumWidth(40);
	octreeRes->setMaxLength(5);
	octreeRes->setValidator( new QDoubleValidator(0, 1, 3) );
	resLabel = new QLabel("Octree Resolution:");

	changeLayout->addWidget(spacer4,         0,0,2,1,Qt::AlignLeft);
	changeLayout->addWidget(resLabel,        0,1,1,1,Qt::AlignLeft);
	changeLayout->addWidget(octreeRes,       1,1,1,1,Qt::AlignLeft);
	changeLayout->addWidget(changesButton,   0,2,2,1,Qt::AlignLeft);

	back = createButton("Back",SLOT(backClicked()));
	bLayout->addWidget(back,0,0,2,2,Qt::AlignLeft);

	panelLayout->setAlignment(Qt::AlignLeft);
	panelLayout->addLayout(bLayout);
	panelLayout->addLayout(recLayout);
	panelLayout->addLayout(cycleLayout);
	panelLayout->addLayout(icpLayout);
	panelLayout->addLayout(changeLayout);

	setLayout(panelLayout);
}

void RecognitionPanel::enablePanel()
{}

void RecognitionPanel::disablePanel()
{}

void RecognitionPanel::enableCyclePanel(bool enableNext)
{
	nextButton->setEnabled(enableNext);
	useButton->setEnabled(true);
}

void RecognitionPanel::disableCyclePanel()
{
	nextButton->setEnabled(false);
	useButton->setEnabled(false);
}

Button *RecognitionPanel::createButton(const QString &text, const char
*member)
{
	Button *button = new Button(text);
	connect(button, SIGNAL(clicked()), this, member);
	return button;
}
```

```
void RecognitionPanel::backClicked()
{
	parentCWidget->setUpperWidget(0);
}

void RecognitionPanel::buttonClicked()
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();

	if(clicked.compare("Iterative Closest Point")==0)
	{
		int itr = icpItr->text().toInt();
		parentCWidget->getPCL()->icpTransformation(itr);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
	}
	if(clicked.compare("Recognize")==0)
	{
		enableCyclePanel(false);
		parentCWidget->setChecked("both");
		parentCWidget->getPCL()->recognize( model_ss->  text().toFloat(),
		                                                            scene_ss->
	text().toFloat(),
		                                                            rf_rad->
	text().toFloat(),
		                                                            descr_rad->
	text().toFloat(),
		                                                            cg_size->
	text().toFloat(),
		                                                            cg_tresh->
	text().toInt());
		if(parentCWidget->getPCL()->getTransCount()==0)
		{
			disableCyclePanel();
			instLabel->setText("0/0");

		}
		if(parentCWidget->getPCL()->getTransCount()==1)
		{
			enableCyclePanel(false);
			instLabel->setText("1/1");
		}

		else
		{
			int counter = parentCWidget->getPCL()->getTransCount();
			enableCyclePanel(true);
			instLabel->setText("1/"+QString::number(counter));
		}
	}
	if(clicked.compare("Next")==0)
	{
		parentCWidget->setChecked("both");
		parentCWidget->getPCL()->visNextTransformation();

		int current = parentCWidget->getPCL()->getCurrentTransCount();
		int total = parentCWidget->getPCL()->getTransCount();
```

```
            instLabel-
>setText(QString::number(current)+"/"+QString::number(total));

      }
      if(clicked.compare("Use")==0)
      {
            parentCWidget->getPCL()->setTransformation();
            //pcl->save total transformation
      }
      if(clicked.compare("Show differences")==0)
      {
            parentCWidget->setChecked("both");
            parentCWidget->getPCL()->spatialChange(octreeRes-
>text().toFloat());
      }
}

//--------------------------------------------------------------
//Transformation Panel---------------------------------------
//--------------------------------------------------------------
TransPanel::TransPanel(CWidget *parent)
{

      moveStep = 0.05f; //5cm
      rotateStep = 1.0f;      //1 degrees

      parentCWidget = parent;
      panelLayout = new QHBoxLayout();
      bLayout = new QGridLayout();
      axisSelLayout = new QHBoxLayout();
      axisSelGroup  = new QGroupBox();
      mTransLayout = new QGridLayout();
      sTransLayout = new QGridLayout();

      back = createButton("Back",SLOT(backClicked()));
      bLayout->addWidget(back,0,0,2,2,Qt::AlignLeft);

      xaxis = createRadioButton(tr("X-axis"),SLOT(radioClicked()));
      yaxis = createRadioButton(tr("Y-axis"),SLOT(radioClicked()));
      zaxis = createRadioButton(tr("Z-axis"),SLOT(radioClicked()));

      QWidget *spacer0 = new QWidget();
      spacer0->setFixedWidth(10);
      axisSelLayout->addWidget(spacer0);
      axisSelLayout->addWidget(xaxis);
      axisSelLayout->addWidget(yaxis);
      axisSelLayout->addWidget(zaxis);

      axisSelGroup->setLayout(axisSelLayout);

      QLabel *modelLabel = new QLabel("Model");
      QLabel *modelRotLabel = new QLabel("Rotation:");
      QLabel *modelTransLabel = new QLabel("Translation:");
      QLabel *emptyLabel1 = new QLabel(" ");
      QWidget *spacer1 = new QWidget();
      spacer1->setFixedWidth(10);
      mMoveLeft = createButton(tr("<--"),SLOT(mMoveClicked()));
```

```
mMoveRight = createButton(tr("-->"),SLOT(mMoveClicked()));
mRotLeft = createButton(tr("<--"),SLOT(mRotClicked()));
mRotRight = createButton(tr("-->"),SLOT(mRotClicked()));
mMatrix = createButton(tr("Matrix"),SLOT(mMatrixClicked()));

mTransLayout->addWidget(spacer1,         0,0,1,1,Qt::AlignLeft);
mTransLayout->addWidget(modelLabel,      0,1,2,1,Qt::AlignLeft);
mTransLayout->addWidget(modelTransLabel,0,2,1,2,Qt::AlignLeft);
mTransLayout->addWidget(mMoveLeft,       1,2,1,1,Qt::AlignLeft);
mTransLayout->addWidget(mMoveRight,      1,3,1,1,Qt::AlignLeft);
mTransLayout->addWidget(modelRotLabel,   0,4,1,2,Qt::AlignLeft);
mTransLayout->addWidget(mRotLeft,        1,4,1,1,Qt::AlignLeft);
mTransLayout->addWidget(mRotRight,       1,5,1,1,Qt::AlignLeft);
mTransLayout->addWidget(emptyLabel1,     0,6,1,1,Qt::AlignLeft);
mTransLayout->addWidget(mMatrix,         1,6,1,1,Qt::AlignLeft);

QLabel *sceneLabel = new QLabel("Scene");
QLabel *sceneRotLabel = new QLabel("Rotation:");
QLabel *sceneTransLabel = new QLabel("Translation:");
QLabel *emptyLabel2 = new QLabel(" ");
QWidget *spacer2 = new QWidget();
spacer2->setFixedWidth(10);
sMoveLeft = createButton(tr("<--"),SLOT(sMoveClicked()));
sMoveRight = createButton(tr("-->"),SLOT(sMoveClicked()));
sRotLeft = createButton(tr("<--"),SLOT(sRotClicked()));
sRotRight = createButton(tr("-->"),SLOT(sRotClicked()));
sMatrix = createButton(tr("Matrix"),SLOT(sMatrixClicked()));

sTransLayout->addWidget(spacer2,         0,0,1,1,Qt::AlignLeft);
sTransLayout->addWidget(sceneLabel,      0,1,2,1,Qt::AlignLeft);
sTransLayout->addWidget(sceneTransLabel,0,2,1,2,Qt::AlignLeft);
sTransLayout->addWidget(sMoveLeft,       1,2,1,1,Qt::AlignLeft);
sTransLayout->addWidget(sMoveRight,      1,3,1,1,Qt::AlignLeft);
sTransLayout->addWidget(sceneRotLabel,   0,4,1,2,Qt::AlignLeft);
sTransLayout->addWidget(sRotLeft,        1,4,1,1,Qt::AlignLeft);
sTransLayout->addWidget(sRotRight,       1,5,1,1,Qt::AlignLeft);
sTransLayout->addWidget(emptyLabel2,     0,6,1,1,Qt::AlignLeft);
sTransLayout->addWidget(sMatrix,         1,6,1,1,Qt::AlignLeft);

panelLayout->setAlignment(Qt::AlignLeft);
panelLayout->addLayout(bLayout);
panelLayout->addWidget(axisSelGroup);
panelLayout->addLayout(mTransLayout);
panelLayout->addLayout(sTransLayout);

setLayout(panelLayout);

xaxis->setEnabled(false);
xaxis->setChecked(true);
yaxis->setEnabled(false);
zaxis->setEnabled(false);

mMoveRight->setEnabled(false);
mMoveLeft->setEnabled(false);
sMoveRight->setEnabled(false);
sMoveLeft->setEnabled(false);
mRotLeft->setEnabled(false);
```

```
      mRotRight->setEnabled(false);
      sRotLeft->setEnabled(false);
      sRotRight->setEnabled(false);
      mMatrix->setEnabled(false);
      sMatrix->setEnabled(false);
}

void TransPanel::enablePanel(std::string cloud)
{
      if(cloud.compare("model")==0)
      {
            xaxis->setEnabled(true);
            yaxis->setEnabled(true);
            zaxis->setEnabled(true);

            mMoveRight->setEnabled(true);
            mMoveLeft->setEnabled(true);
            mRotLeft->setEnabled(true);
            mRotRight->setEnabled(true);
            mMatrix->setEnabled(true);

      }
      if(cloud.compare("scene")==0)
      {
            xaxis->setEnabled(true);
            yaxis->setEnabled(true);
            zaxis->setEnabled(true);

            sMoveRight->setEnabled(true);
            sMoveLeft->setEnabled(true);
            sRotLeft->setEnabled(true);
            sRotRight->setEnabled(true);
            sMatrix->setEnabled(true);
      }
}

void TransPanel::disablePanel()
{
      xaxis->setEnabled(false);
      yaxis->setEnabled(false);
      zaxis->setEnabled(false);

      mMoveRight->setEnabled(false);
      mMoveLeft->setEnabled(false);
      sMoveRight->setEnabled(false);
      sMoveLeft->setEnabled(false);
      mRotLeft->setEnabled(false);
      mRotRight->setEnabled(false);
      sRotLeft->setEnabled(false);
      sRotRight->setEnabled(false);
      mMatrix->setEnabled(false);
      sMatrix->setEnabled(false);
}

Button *TransPanel::createButton(const QString &text, const char *member)
{
      Button *button = new Button(text);
```

```
	connect(button, SIGNAL(clicked()), this, member);
	return button;
}

QRadioButton *TransPanel::createRadioButton(const QString &text, const char
*member)
{
	QRadioButton *rbutton = new QRadioButton(text);
	connect(rbutton, SIGNAL(clicked()), this,member);
	return rbutton;
}

char TransPanel::getCheckedAxis()
{
	if(xaxis->isChecked())
		return 'x';
	else if (yaxis->isChecked())
		return 'y';
	else if (zaxis->isChecked())
		return 'z';
}

void TransPanel::mMoveClicked()
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();

	if(clicked.compare("-->")==0)
	{
		parentCWidget->getPCL()-
>translateCloud("model",getCheckedAxis(),moveStep);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
	}
	if(clicked.compare("<--")==0)
	{
		parentCWidget->getPCL()->translateCloud("model",getCheckedAxis(),-
moveStep);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
	}
}

void TransPanel::sMoveClicked()
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();

	if(clicked.compare("-->")==0)
	{
		parentCWidget->getPCL()-
>translateCloud("scene",getCheckedAxis(),moveStep);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
	}
	if(clicked.compare("<--")==0)
	{
		parentCWidget->getPCL()->translateCloud("scene",getCheckedAxis(),-
moveStep);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
```

```
	}
}

void TransPanel::mRotClicked()
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();

	if(clicked.compare("-->")==0)
	{
		parentCWidget->getPCL()-
>rotateCloud("model",getCheckedAxis(),rotateStep);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
	}
	if(clicked.compare("<--")==0)
	{
		parentCWidget->getPCL()->rotateCloud("model",getCheckedAxis(),-
rotateStep);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
	}
}

void TransPanel::sRotClicked()
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();

	if(clicked.compare("-->")==0)
	{
		parentCWidget->getPCL()-
>rotateCloud("scene",getCheckedAxis(),rotateStep);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
	}
	if(clicked.compare("<--")==0)
	{
		parentCWidget->getPCL()->rotateCloud("scene",getCheckedAxis(),-
rotateStep);
		parentCWidget->getPCL()->show(parentCWidget->getChecked());
	}
}

void TransPanel::radioClicked()
{}

void TransPanel::mMatrixClicked()
{
	parentCWidget->getPCL()->transformCloud("model",transMatrixBox());
	parentCWidget->getPCL()->show(parentCWidget->getChecked());
}

void TransPanel::sMatrixClicked()
{
	parentCWidget->getPCL()->transformCloud("scene",transMatrixBox());
	parentCWidget->getPCL()->show(parentCWidget->getChecked());
}

void TransPanel::backClicked()
```

```
{
	parentCWidget->setUpperWidget(0);
}

Eigen::Matrix4f TransPanel::transMatrixBox()
{
	QLabel *rmatrix = new QLabel("Rotation-matrix:");
	QLabel *tmatrix = new QLabel("Translation-vector:");
	QHBoxLayout *rLayout;

	QGridLayout *tGLayout = new QGridLayout();
	QGridLayout *rGLayout = new QGridLayout();
	QGridLayout *totalLayout = new QGridLayout();

	std::vector<QLineEdit*> column1(3);
	std::vector<QLineEdit*> column2(3);
	std::vector<QLineEdit*> column3(3);

	for(int i=0; i!=3; ++i)
	{
		column1[i] = new QLineEdit("0");
		if(i==0)
			column1[i]->setText("1");
		column1[i]->setAlignment(Qt::AlignRight);
		column1[i]->setMaximumWidth(40);
		column1[i]->setMaxLength(6);
		column1[i]->setValidator( new QDoubleValidator(-9999.9, 9999.9, 3)
);
		rGLayout->addWidget(column1[i],i,0,1,1,Qt::AlignLeft);
	}
	for(int i=0; i!=3; ++i)
	{
		column2[i] = new QLineEdit("0");
		if(i==1)
			column2[i]->setText("1");
		column2[i]->setAlignment(Qt::AlignRight);
		column2[i]->setMaximumWidth(40);
		column2[i]->setMaxLength(6);
		column2[i]->setValidator( new QDoubleValidator(-9999.9, 9999.9, 3)
);
		rGLayout->addWidget(column2[i],i,1,1,1,Qt::AlignLeft);
	}
	for(int i=0; i!=3; ++i)
	{
		column3[i] = new QLineEdit("0");
		if(i==2)
			column3[i]->setText("1");
		column3[i]->setAlignment(Qt::AlignRight);
		column3[i]->setMaximumWidth(40);
		column3[i]->setMaxLength(6);
		column3[i]->setValidator( new QDoubleValidator(-9999.9, 9999.9, 3)
);
		rGLayout->addWidget(column3[i],i,2,1,1,Qt::AlignLeft);
	}

	std::vector<QLineEdit*> tColumn(3);
	for(int i=0; i!=3; ++i)
```

```
	{
		tColumn[i] = new QLineEdit("0");
		tColumn[i]->setAlignment(Qt::AlignRight);
		tColumn[i]->setMaximumWidth(40);
		tColumn[i]->setMaxLength(6);
		tColumn[i]->setValidator( new QDoubleValidator(-9999.9, 9999.9, 3)
);
		tGLayout->addWidget(tColumn[i],i,0,1,1,Qt::AlignLeft);
	}

	totalLayout->addWidget(rmatrix,		0,1,1,1,Qt::AlignLeft);
	totalLayout->addLayout(rGLayout,	1,1,1,1,Qt::AlignLeft);
	totalLayout->addWidget(tmatrix,		0,2,1,1,Qt::AlignLeft);
	totalLayout->addLayout(tGLayout,	1,2,1,1,Qt::AlignRight);

	QWidget *content = new QWidget();
	content->setLayout(totalLayout);

	QMessageBox msgBox;
	msgBox.setText("Transformation Matrix");
	msgBox.layout()->addWidget(content);
	msgBox.layout()->setAlignment(Qt::AlignRight);
	msgBox.setStandardButtons(QMessageBox::Ok | QMessageBox::Cancel);
	msgBox.setDefaultButton(QMessageBox::Ok);
	int answer = msgBox.exec();

	Eigen::Matrix4f matrix = Eigen::Matrix4f::Identity();
	if (answer==QMessageBox::Ok)
	{
		matrix(0,0)=column1[0]->text().toFloat();
		matrix(1,0)=column1[1]->text().toFloat();
		matrix(2,0)=column1[2]->text().toFloat();
		matrix(0,1)=column2[0]->text().toFloat();
		matrix(1,1)=column2[1]->text().toFloat();
		matrix(2,1)=column2[2]->text().toFloat();
		matrix(0,2)=column3[0]->text().toFloat();
		matrix(1,2)=column3[1]->text().toFloat();
		matrix(2,2)=column3[2]->text().toFloat();
		matrix(0,3)=tColumn[0]->text().toFloat();
		matrix(1,3)=tColumn[1]->text().toFloat();
		matrix(2,3)=tColumn[2]->text().toFloat();
	}
	return matrix;
}

//--------------------------------------------------------------
//ISM-recognition Panel-----------------------------------------
//--------------------------------------------------------------
ismRecognitionPanel::ismRecognitionPanel(CWidget *parent)
{
	parentCWidget = parent;
	panelLayout = new QHBoxLayout();
	bLayout = new QGridLayout();

	back = createButton("Back",SLOT(backClicked()));
	bLayout->addWidget(back,0,0,2,2,Qt::AlignLeft);
```

```
	ismRecognize = createButton(tr("Recognize"), SLOT(buttonClicked()));

	s_size_lab = new QLabel("Sampling s");
	descr_rad_lab = new QLabel("Descr. r");
	normal_rad_lab = new QLabel("Norm. r");
	thresh_lab = new QLabel("Thresh.");
	resLabel = new QLabel(" ");

	s_size =    new QLineEdit("0.05");
	s_size->setAlignment(Qt::AlignRight);
	s_size->setMaximumWidth(40);
	s_size->setMaxLength(5);
	s_size->setValidator( new QDoubleValidator(0, 1, 3) );

	descr_rad = new QLineEdit("0.25");
	descr_rad->setAlignment(Qt::AlignRight);
	descr_rad->setMaximumWidth(40);
	descr_rad->setMaxLength(5);
	descr_rad->setValidator( new QDoubleValidator(0, 1, 3) );

	normal_rad =      new QLineEdit("0.25");
	normal_rad->setAlignment(Qt::AlignRight);
	normal_rad->setMaximumWidth(40);
	normal_rad->setMaxLength(5);
	normal_rad->setValidator( new QDoubleValidator(0, 1, 3) );

	thresh =    new QLineEdit("0.02");
	thresh->setAlignment(Qt::AlignRight);
	thresh->setMaximumWidth(40);
	thresh->setMaxLength(5);
	thresh->setValidator( new QDoubleValidator(0, 1, 3) );

	ismLayout = new QGridLayout();
	ismLayout->addWidget(s_size_lab,		0,0,1,1,Qt::AlignLeft);
	ismLayout->addWidget(s_size,		1,0,1,1,Qt::AlignLeft);
	ismLayout->addWidget(normal_rad_lab,	0,1,1,1,Qt::AlignLeft);
	ismLayout->addWidget(normal_rad,		1,1,1,1,Qt::AlignLeft);
	ismLayout->addWidget(descr_rad_lab,	0,2,1,1,Qt::AlignLeft);
	ismLayout->addWidget(descr_rad,			1,2,1,1,Qt::AlignLeft);
	ismLayout->addWidget(thresh_lab,		0,3,1,1,Qt::AlignLeft);
	ismLayout->addWidget(thresh,		1,3,1,1,Qt::AlignLeft);
	ismLayout->addWidget(ismRecognize,	0,4,2,1,Qt::AlignLeft);
	ismLayout->addWidget(resLabel,			0,5,2,1,Qt::AlignLeft);

	panelLayout->setAlignment(Qt::AlignLeft);
	panelLayout->addLayout(bLayout);
	panelLayout->addLayout(ismLayout);

	setLayout(panelLayout);
}

void ismRecognitionPanel::enablePanel()
{}

void ismRecognitionPanel::disablePanel()
{}
```

```
Button *ismRecognitionPanel::createButton(const QString &text, const char
*member)
{
	Button *button = new Button(text);
	connect(button, SIGNAL(clicked()), this, member);
	return button;
}

void ismRecognitionPanel::backClicked()
{
	parentCWidget->setUpperWidget(0);
}

void ismRecognitionPanel::buttonClicked()
{
	Button *clickedButton = qobject_cast<Button *>(sender());
	QString clicked = clickedButton->text();

	if(clicked.compare("Recognize")==0)
	{
		float ss = s_size->text().toFloat();
		float descrr = descr_rad->text().toFloat();
		float normrr = normal_rad->text().toFloat();
		float tr = thresh->text().toFloat();
		parentCWidget->getPCL()->ismRecognition(ss,descrr,normrr,tr);
	}
}
```

B.8 pcltools.h

```
#pragma once

#include <pcl/io/pcd_io.h>
#include <pcl/io/ply_io.h>
#include <pcl/visualization/pcl_visualizer.h>
#include <pcl/visualization/point_cloud_handlers.h>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/common/common.h>
#include <pcl/common/transforms.h>
#include <pcl/common/common_headers.h>
#include <pcl/common/distances.h>
#include <boost/shared_ptr.hpp>
#include <Eigen/Core>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/kdtree/impl/kdtree_flann.hpp>
#include <pcl/filters/extract_indices.h>
#include <pcl/sample_consensus/ransac.h>
#include <pcl/sample_consensus/sac_model_plane.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/conditional_removal.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/octree/octree.h>
#include <pcl/features/normal_3d.h>
#include <pcl/surface/impl/mls.hpp>
#include <pcl/filters/radius_outlier_removal.h>
#include <pcl/registration/icp.h>

#include "recognition.h"
#include "recognitionISM.h"

using namespace pcl;
using namespace std;

class PCLTools
{
public:
      PCLTools(boost::shared_ptr<pcl::visualization::PCLVisualizer>
viewer);
      bool _modelLastEdited;
      bool _sceneLastEdited;
      bool _modelIsLoaded;
      bool _sceneIsLoaded;
      bool _modelIsModified;
      bool _sceneIsModified;
      std::string _modelString;
      std::string _sceneString;
```

```
        void viewerText(std::string text);
        int  saveCloud(std::string cloud, std::string path);
        int  loadCloud(std::string path, bool isModel);
        void initialShow(std::string cloud);
        void show(std::string cloud);
        void undoLast();
        int resizeCloud (std::string cloud, double factor);
        void updateString(std::string cloud);
        std::vector<float> getInitBoundingBox(std::string cloud);
        void showBoundingBox(std::vector<float> par);
        void removeBoundingBox();
        void planeFitting(std::string cloud, double distThres);
        int getCloudSize(std::string cloud);
        void boxSegmentation(std::string cloud, std::vector<float>
boxParameters);
        void rotateCloud(std::string cloud, char axis, double angle);
        float maxRadiusEqualization(std::string cloud, std::string refCloud);
        float getDensity(std::string cloud);
        void voxelGrid(std::string cloud, float leaf_size);
        void radOutlierRemoval(std::string cloud, double radius, int
minNeighbors);
        void transformCloud(std::string cloud, Eigen::Matrix4f
transformation_matrix);
        void translateCloud(std::string cloud, char axis, float step);
        void icpTransformation(int maxIterations);
        void recognize(float model_ss, float scene_ss, float rf_rad, float
descr_rad, float cg_size, int cg_tresh );
        void setTransformation();
        void visNextTransformation();
        void visTransformation(int i);
        int getTransCount();
        int getCurrentTransCount();
        void mergeClouds();
        void spatialChange(float res);
        void setShowCorresp(bool showcp);
        void ismRecognition(float ss, float descrr, float normr, float
thresh);

private:

        boost::shared_ptr<pcl::visualization::PCLVisualizer> _viewer;
        std::string _modelFileName;
        std::string _sceneFileName;
        Eigen::Vector3d _modelColor;
        Eigen::Vector3d _sceneColor;
        pcl::PointCloud<pcl::PointXYZ>::Ptr _modelCloud;
        pcl::PointCloud<pcl::PointXYZ>::Ptr _sceneCloud;
        pcl::PointCloud<pcl::PointXYZ>::Ptr _prevModelCloud;
        pcl::PointCloud<pcl::PointXYZ>::Ptr _prevSceneCloud;
        boost::shared_ptr<CGRecognition> rec;
        bool _show_correspondences;
```

```
      void cloudAdder(pcl::PointCloud<pcl::PointXYZ>::Ptr incloud,
std::string cloudTag, double r, double g,double b);
      void cloudAdder(pcl::PointCloud<pcl::PointXYZRGB>::Ptr incloud,
std::string cloudTag);
      void setLastEdited(std::string cloud);
      void setPrevCloud(std::string cloud);
      int cloudSize(std::string cloud);
      double computeCloudResolution (std::string cloud);
      float maxCloudDistance(std::string cloud);
      float getCloudMaxRadius(std::string cloud);
      pcl::PointXYZ getCenterPoint(std::string cloud);
      std::vector<float> getMinXYZ(std::string cloud);
      std::vector<float> getMaxXYZ(std::string cloud);
      pcl::PointCloud<pcl::PointXYZ>::Ptr strToCloud(std::string cloud);
      void PCLTools::visualizeCenterPoints(std::vector<pcl::PointXYZ>
centerpoints, std::vector<pcl::ISMPeak,
Eigen::aligned_allocator<pcl::ISMPeak>> strongest_peaks, float radius,
float thresh);
      int findMaxPeakIdx(std::vector<pcl::ISMPeak,
Eigen::aligned_allocator<pcl::ISMPeak>> strongest_peaks);
      std::vector<Eigen::Matrix4f,
Eigen::aligned_allocator<Eigen::Matrix4f> > transformations;
      std::vector<pcl::Correspondences> correspondences;
      int transCounter;
};
```

B.9 pcltools.cpp

```
#include "pcltools.h"

PCLTools::PCLTools(boost::shared_ptr<pcl::visualization::PCLVisualizer>
viewer)
{
	_modelLastEdited=false;
	_sceneLastEdited=false;
	_modelIsLoaded=false;
	_sceneIsLoaded=false;
	_modelFileName=" ";
	_sceneFileName=" ";
	_modelString="No info";
	_sceneString="No info";
	_viewer = viewer;
	_modelCloud.reset(new pcl::PointCloud<pcl::PointXYZ>());
	_sceneCloud.reset(new pcl::PointCloud<pcl::PointXYZ>());
	_prevModelCloud.reset(new pcl::PointCloud<pcl::PointXYZ>());
	_prevSceneCloud.reset(new pcl::PointCloud<pcl::PointXYZ>());
	_modelColor<<255,0,50;
	_sceneColor<<0,255,50;
	_viewer->resetCamera();
	_modelIsModified = false;
	_sceneIsModified = false;
	_show_correspondences = true;

}

//--------------------------------------------
// Visualization-----------------------------
//--------------------------------------------
void PCLTools::initialShow(std::string cloud)
{
//Input: string - Name of the cloud that is going to be visualized
//This function visualizes the specified cloud, and resets the camera.
	show(cloud);
	_viewer->resetCamera();
	_viewer->getInteractorStyle()->Spin();//Force the visualizer to
update...
}

void PCLTools::show(std::string cloud)
{
//Input: string - Name of the cloud that is going to be visualized
//This function updates the viewer with the specified cloud.

	if (cloud.compare("clear")==0)
	{
		_viewer->removeAllPointClouds(0);
		_viewer->removeAllShapes();
```

```
	}
	if(cloud.compare("both")==0)
	{
		_viewer->removeAllShapes();
		if(_modelIsLoaded || _sceneIsLoaded)
		{
			_viewer->removeAllPointClouds(0);
		}

	cloudAdder(_modelCloud,_modelFileName+"m",_modelColor(0),_modelColor(1),
_modelColor(2));

	cloudAdder(_sceneCloud,_sceneFileName+"s",_sceneColor(0),_sceneColor(1),
_sceneColor(2));
		viewerText("both");
	}
	else if(cloud.compare("model")==0)
	{
		_viewer->removeAllShapes();
		if(_modelIsLoaded)
		{
			_viewer->removeAllPointClouds(0);
		}
		viewerText("model");

	cloudAdder(_modelCloud,_modelFileName,_modelColor(0),_modelColor(1),_mod
elColor(2));

	}
	else if(cloud.compare("scene")==0)
	{
		_viewer->removeAllShapes();
		if(_sceneIsLoaded)
		{
			_viewer->removeAllPointClouds(0);
		}
		viewerText("scene");

	cloudAdder(_sceneCloud,_sceneFileName,_sceneColor(0),_sceneColor(1),_sce
neColor(2));

	}
	_viewer->getInteractorStyle()->Spin();//Force the visualizer to
update...
}


void PCLTools::updateString(std::string cloud)
{
//Input: string - Name of the cloud which cloud-string is going to be updated
//This function updates the string containing information about a cloud:
//Size, Radius

	int cSize = cloudSize(cloud);
	float maxRadius = getCloudMaxRadius(cloud);

	if(cloud.compare("model")==0)
```

```
	{
		stringstream stream;
		stream<<"Model Cloud: Size: "<<cSize<<" Max Radius: "<< maxRadius;
		_modelString=stream.str();
	}
	if(cloud.compare("scene")==0)
	{
		stringstream stream;
		stream<<"Scene Cloud: Size: "<<cSize<<" Max Radius: "<< maxRadius;
		_sceneString=stream.str();
	}
}

void PCLTools::viewerText(std::string cloud)
{
//Input: string - Name of the cloud in the visualizer.
//Removes old text from viewer and updates viewer with new text.

	_viewer->removeShape("filename",0);
	_viewer->removeShape("fileinfo",0);
	_viewer->removeShape("xaxis",0);
	_viewer->removeShape("yaxis",0);
	_viewer->removeShape("zaxis",0);
	_viewer->addText("x-axis",1,65,10,1.0,0,0,"xaxis",0);
	_viewer->addText("y-axis",1,55,10,0,1.0,0,"yaxis",0);
	_viewer->addText("z-axis",1,48,10,0,0,1.0,"zaxis",0);

	if(cloud.compare("model")==0)
	{
		_viewer->addText(_modelFileName,1,25,20,0.1,0.4,0.3,"filename");
		_viewer->addText(_modelString,1,10,10,0.5,0.0,0.3,"fileinfo");
	}
	else if(cloud.compare("scene")==0)
	{
		_viewer->addText(_sceneFileName,1,25,20,0.1,0.4,0.3,"filename");
		_viewer->addText(_sceneString,1,10,10,0.5,0.0,0.3,"fileinfo");
	}
	else if(cloud.compare("both")==0)
	{
		stringstream stream;
		stringstream info;
		if(_modelIsLoaded && _sceneIsLoaded)
		{
			info<<_modelString<<" "<<_sceneString;
			stream<<_modelFileName<<" and "<<_sceneFileName;
			_viewer-
>addText(stream.str(),1,25,20,0.1,0.4,0.3,"filename");
			_viewer->addText(info.str(),1,10,10,0.5,0.0,0.3,"fileinfo");

		}
	}

}

void PCLTools::cloudAdder(pcl::PointCloud<pcl::PointXYZ>::Ptr incloud,
std::string cloudTag, double r, double g,double b)
{
```

```
      // Input: cloud_pointer - pointer to a pointcloud.
      // Input: string - cloudtag
      // Input: double: r,g,b - visualization colors.
      // Function visualizes the input cloud with the input colors.
      pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
rgb(incloud,r,g,b);
      _viewer->addPointCloud<pcl::PointXYZ> (incloud, rgb, cloudTag);
}


void PCLTools::cloudAdder(pcl::PointCloud<pcl::PointXYZRGB>::Ptr incloud,
std::string cloudTag)
{
      // Input: colored_cloud_pointer - pointer to a colored pointcloud.
      // Input: string - cloudtag
      // Function visualizes the input cloud.
      pcl::visualization::PointCloudColorHandlerRGBField<pcl::PointXYZRGB>
rgb(incloud);
      _viewer->addPointCloud<pcl::PointXYZRGB> (incloud, rgb, cloudTag);
}
void PCLTools::showBoundingBox(std::vector<float> par)
{
      // Input: vector - parameters for bounding box
[xmin,xmax,ymin,ymax,zmin,zmax]
      // Function creates a boundingox in the visualization.
      _viewer->removeShape("mycube",0);
      _viewer-
>addCube(par[0],par[1],par[2],par[3],par[4],par[5],1.0,1.0,1.0,"mycube",0);
      _viewer->getInteractorStyle()->Spin();//Force the visualizer to
update...
}

void PCLTools::removeBoundingBox()
{
      // Function creates removes boundingox in the visualization.
      _viewer->removeShape("mycube",0);
      _viewer->getInteractorStyle()->Spin();//Force the visualizer to
update...
}

//---------------------------------------------
// IO-----------------------------------------
//---------------------------------------------
int  PCLTools::saveCloud(std::string cloud, std::string path)
{
//Input: string - Name of the cloud that is going to be saved.
//Input: string - savepath
//The function saves a cloud in ply or pcd-format

      pcl::PointCloud<pcl::PointXYZ>::Ptr saveCloud = strToCloud(cloud);

      unsigned int found = path.find_last_of(".");
      std::string filetype = path.substr(found+1,path.length());
      if(filetype.compare("pcd")==0)
      {
            if (io::savePCDFile(path, *saveCloud, true) == 0)
            {
```

```
				return 0;
			}
			return -1;
		}
		else if(filetype.compare("ply")==0)
		{
			if (io::savePLYFileASCII(path, *saveCloud) == 0)
			{
				return 0;
			}
			return -1;
		}
		else
			return -1;
}

int  PCLTools::loadCloud(std::string path, bool isModel)
{
		//Input: string - Name of the cloud that is going to be loaded.
		//Input: string - loadpath
		//The function loads a cloud in ply or pcd-format
		unsigned found = path.find_last_of(".");
		std::string filetype = path.substr(found+1,path.length());

		if(filetype.compare("pcd")!=1)
		{
			std::cout << "Loading pcd file " << path << ".\n";
			if(isModel)
			{
				if(pcl::io::loadPCDFile<pcl::PointXYZ>(path,
*_modelCloud)==0)
				{
					int slashpos = path.find_last_of("/");
					_modelFileName =
path.substr(slashpos+1,path.length());
					_modelIsLoaded=true;
					updateString("model");
					cout<<"Successfully loaded file at "<< path;
					return 0;
				}
			}
			else if(!isModel)
			{
				if(pcl::io::loadPCDFile<pcl::PointXYZ>(path,
*_sceneCloud)==0)
				{
					int slashpos = path.find_last_of("/");
					_sceneFileName =
path.substr(slashpos+1,path.length());
					_sceneIsLoaded=true;
					updateString("scene");
					cout<<"Successfully loaded file at "<< path;
					return 0;
				}
			}
			cerr<<"ERROR: Was not able to load "<<path;
			return -1;
```

```
	}
	else if(filetype.compare("ply")!=1)
	{
		Eigen::Vector4f origin = Eigen::Vector4f::Zero ();
		Eigen::Quaternionf orientation = Eigen::Quaternionf::Identity ();

		std::cout << "Loading ply file " << path << ".\n";

		pcl::PLYReader read;
		pcl::PCLPointCloud2 sky;

		int plyversion = 1;
		if(read.read(path,sky,origin,orientation,plyversion)==0)
		{
			if(isModel)
			{
				int slashpos = path.find_last_of("/");
				_modelFileName =
path.substr(slashpos+1,path.length());
				_modelIsLoaded=true;
				pcl::fromPCLPointCloud2(sky,*_modelCloud);
				updateString("model");
				cout<<"Successfully loaded file at "<< path;
				return 0;
			}
			else if(!isModel)
			{
				int slashpos = path.find_last_of("/");
				_sceneFileName =
path.substr(slashpos+1,path.length());
				_sceneIsLoaded=true;
				pcl::fromPCLPointCloud2(sky,*_sceneCloud);
				updateString("scene");
				cout<<"Successfully loaded file at "<< path;
				return 0;
			}
			cerr<<"ERROR: Was not able to load "<<path;
			return -1;
		}
	}
	else
	{
		cerr<<"ERROR: Was not able to load "<<path;
		return -1;
	}
}

//----------------------------------------------
// Undo-functions-----------------------------
//----------------------------------------------
void PCLTools::setPrevCloud(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Function saves this cloud as the previous cloud for the undo
function.
	if(cloud.compare("model")==0)
	{
```

```
		setLastEdited("model");

	pcl::copyPointCloud<PointXYZ,PointXYZ>(*_modelCloud,*_prevModelCloud);
	}
	else if(cloud.compare("scene")==0)
	{
		setLastEdited("scene");

	pcl::copyPointCloud<PointXYZ,PointXYZ>(*_sceneCloud,*_prevSceneCloud);
	}
}

void PCLTools::setLastEdited(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Function sets a bool variable that tells which cloud was the last
edited.
	if(cloud.compare("model")==0)
	{
		_modelLastEdited=true;
		_sceneLastEdited=false;
		_modelIsModified=true;
	}
	else if(cloud.compare("scene")==0)
	{
		_modelLastEdited=false;
		_sceneLastEdited=true;
		_sceneIsModified=true;
	}
}

void PCLTools::undoLast()
{
	// Input: string - name of the specified cloud.
	// Copy the previous cloud backup to the actual cloud.
	if(_modelLastEdited)
	{

	pcl::copyPointCloud<PointXYZ,PointXYZ>(*_prevModelCloud,*_modelCloud);
		_modelLastEdited=false;

		//Run show externally to update viewer
	}
	if(_sceneLastEdited)
	{

	pcl::copyPointCloud<PointXYZ,PointXYZ>(*_prevSceneCloud,*_sceneCloud);
		_sceneLastEdited=false;

		//Run show externally to update viewer
	}
}
```

```
//--------------------------------------------
// Cloud operations---------------------------
//--------------------------------------------
int PCLTools::resizeCloud(std::string cloud, double factor)
{
	// Input: string - name of the specified cloud.
	// Input: double - resize factor.
	// Function resizes the specified cloud.
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud;
	if(cloud.compare("model")==0)
	{
		setPrevCloud("model");
		incloud = _modelCloud;
	}
	else if(cloud.compare("scene")==0)
	{
		setPrevCloud("scene");
		incloud = _sceneCloud;
	}
	else
	{
		std::cerr << "ERROR!: Function input not recognized. Input must be
\"model\" or \"scene\".";
		return -1;
	}
	std::cout<<"\nResizing "<<cloud<<" with scale "<<factor<<"...\n";
	std::vector<pcl::PointXYZ, Eigen::aligned_allocator<pcl::PointXYZ>>
cloud_points = incloud->points;

	for(std::vector<pcl::PointXYZ,
Eigen::aligned_allocator<pcl::PointXYZ>>::iterator i = cloud_points.begin();
i!=cloud_points.end(); ++i)
	{
		i->x = i->x*factor;
		i->y = i->y*factor;
		i->z = i->z*factor;
	}

	incloud->points = cloud_points;
	std::cout<<"\nDone!\n";
	updateString(cloud);
	return 0;
}

int PCLTools::getCloudSize(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Out: cloud size
	// Function returns number of points in the specified cloud.
	return strToCloud(cloud)->size();
}

int PCLTools::cloudSize(std::string cloud)
{
	// Input: string - name of the specified cloud.
```

```
	// Out: cloud size
	// Function returns number of points in the specified cloud.
	PointCloud<PointXYZ>::Ptr cloud_ptr(new PointCloud<PointXYZ>());
	if(cloud.compare("model")==0)
	{
		cloud_ptr = _modelCloud;
	}
	else if(cloud.compare("scene")==0)
	{
		cloud_ptr = _sceneCloud;
	}
	else
		return -1;
	return cloud_ptr->points.size();
}

float PCLTools::maxCloudDistance(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Out: float - max euclidean length in the cloud.
	// Function returns the maximum euclidean length in the cloud.
	PointCloud<PointXYZ>::Ptr cloud_ptr(new PointCloud<PointXYZ>());
	if(cloud.compare("model")==0)
	{
		cloud_ptr = _modelCloud;
	}
	else if(cloud.compare("scene")==0)
	{
		cloud_ptr = _sceneCloud;
	}
	else
		return -1;
	PointXYZ pmax;
	PointXYZ pmin;
	std::vector<int> indices;

	pcl::getMaxSegment<pcl::PointXYZ>(*cloud_ptr,pmin,pmax);
	return pcl::euclideanDistance<pcl::PointXYZ>(pmin,pmax);
}

float PCLTools::getCloudMaxRadius(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Output: float - length of the longest radius in the cloud.
	// Function computes the radius from the centerpoint to the most extreme
point in the cloud.
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud;
	if (cloud.compare("model")==0)
	{
		incloud=_modelCloud;
	}
	else if (cloud.compare("scene")==0)
	{
		incloud=_sceneCloud;
	}
	pcl::PointXYZ centrepoint(getCenterPoint(cloud));
	pcl::PointXYZ edgepoint;
```

```
	float radius(0);
	float temp(0);

	std::vector<pcl::PointXYZ, Eigen::aligned_allocator<pcl::PointXYZ>> data
= incloud->points;

	for (std::vector<pcl::PointXYZ,
Eigen::aligned_allocator<pcl::PointXYZ>>::iterator i = data.begin();
i!=data.end(); ++i)
	{

		temp = std::sqrt(pow((i->x - centrepoint.x),2)+
			pow((i->y - centrepoint.y),2)+
			pow((i->z - centrepoint.z),2));
		if(temp>radius)
		{
			radius=temp;
			edgepoint = *i;
		}
	}
	//_viewer->addLine(centrepoint,edgepoint,"line");
	return radius;

}

pcl::PointXYZ PCLTools::getCenterPoint(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Output: PointXYZ - centroid of the cloud.
	// Function computes the centroid of the cloud.
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud;
	if (cloud.compare("model")==0)
	{
		incloud=_modelCloud;
	}
	else if (cloud.compare("scene")==0)
	{
		incloud=_sceneCloud;
	}
	Eigen::Vector4f cent;
	pcl::compute3DCentroid(*incloud, cent);
	pcl::PointXYZ centerpoint;
	centerpoint.x = cent(0);
	centerpoint.y = cent(1);
	centerpoint.z = cent(2);
	return centerpoint;

}

std::vector<float> PCLTools::getInitBoundingBox(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Output: vector - boundingbox parameters
[xmin,xmax,ymin,ymax,zmin,zmax]
	// Function computes parameters for a boundingbox that encapsulates the
specified cloud.
```

```
	std::vector<float> min(getMinXYZ(cloud));
	std::vector<float> max(getMaxXYZ(cloud));

	std::vector<float> boxParameters;
	boxParameters.push_back(min[0]); //xmin
	boxParameters.push_back(max[0]); //xmax
	boxParameters.push_back(min[1]); //ymin
	boxParameters.push_back(max[1]); //ymax
	boxParameters.push_back(min[2]); //zmin
	boxParameters.push_back(max[2]); //zmax
	return boxParameters;
}

std::vector<float> PCLTools::getMinXYZ(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Output: vector - boundingbox parameters [xmin,ymin,zmin]
	// Function computes the minimum point values of x, y and z in a cloud.
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud;
	if (cloud.compare("model")==0)
	{
		incloud=_modelCloud;
	}
	else if (cloud.compare("scene")==0)
	{
		incloud=_sceneCloud;
	}

	std::vector<float> min;
	float minX(0),minY(0),minZ(0);
	float tempX(0),tempY(0),tempZ(0);

	std::vector<pcl::PointXYZ, Eigen::aligned_allocator<pcl::PointXYZ>> data
= incloud->points;

	for (std::vector<pcl::PointXYZ,
Eigen::aligned_allocator<pcl::PointXYZ>>::iterator i = data.begin();
i!=data.end(); ++i)
	{
		if(i==data.begin())
		{
			minX=i->x;
			minY=i->y;
			minZ=i->z;
		}

		tempX =i->x;
		if (tempX<minX)
			minX = tempX;

		tempY = i->y;
		if (tempY<minY)
			minY = tempY;

		tempZ = i->z;
		if (tempZ<minZ)
			minZ = tempZ;
```

```
	}

	min.push_back(minX);
	min.push_back(minY);
	min.push_back(minZ);

	return min;
}

std::vector<float> PCLTools::getMaxXYZ(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Output: vector - boundingbox parameters [xmax,ymax,zmax]
	// Function computes the maximum point values of x, y and z in a cloud.
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud;
	if (cloud.compare("model")==0)
	{
		incloud=_modelCloud;
	}
	else if (cloud.compare("scene")==0)
	{
		incloud=_sceneCloud;
	}

	std::vector<float> max;
	float maxX(0),maxY(0),maxZ(0);
	float tempX(0),tempY(0),tempZ(0);

	std::vector<pcl::PointXYZ, Eigen::aligned_allocator<pcl::PointXYZ>> data
= incloud->points;

	for (std::vector<pcl::PointXYZ,
Eigen::aligned_allocator<pcl::PointXYZ>>::iterator i = data.begin();
i!=data.end(); ++i)
	{
		if(i==data.begin())
		{
			maxX=i->x;
			maxY=i->y;
			maxZ=i->z;
		}

		tempX = i->x;
		if (tempX>maxX)
			maxX = tempX;

		tempY = i->y;
		if (tempY>maxY)
			maxY = tempY;

		tempZ = i->z;
		if (tempZ>maxZ)
			maxZ = tempZ;
	}

	max.push_back(maxX);
	max.push_back(maxY);
```

```
	max.push_back(maxZ);

	return max;
}

void PCLTools::planeFitting(std::string cloud,double distThres)
{
	// Input: string - name of the specified cloud.
	// Input: double - distance threshold
	// Function removes the largest plane in the cloud. Distance threshold
is the thickness of the removed plane
	// Some code from by
http://pointclouds.org/documentation/tutorials/random_sample_consensus.php
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud;
	incloud=strToCloud(cloud);

	std::vector<int> inliers;
	pcl::PointIndices indices_removed;
	pcl::PointCloud<pcl::PointXYZ>::Ptr plane_out(new
pcl::PointCloud<pcl::PointXYZ>());
	pcl::PointCloud<pcl::PointXYZ>::Ptr outcloud(new
pcl::PointCloud<pcl::PointXYZ>());
	pcl::SampleConsensusModelPlane<pcl::PointXYZ>::Ptr model_p (new
pcl::SampleConsensusModelPlane<pcl::PointXYZ> (incloud));
	pcl::RandomSampleConsensus<pcl::PointXYZ> ransac(model_p);

	ransac.setDistanceThreshold (distThres);
	ransac.computeModel();
	ransac.getInliers(inliers);


	pcl::copyPointCloud<pcl::PointXYZ>(*incloud, inliers, *plane_out);
	pcl::ExtractIndices<pcl::PointXYZ> eifilter (true); // Initializing with
true will allow us to extract the removed indices
	eifilter.setInputCloud (incloud);
	pcl::IndicesPtr planeIndicesPtr =
boost::make_shared<std::vector<int>>(inliers);
	eifilter.setIndices(planeIndicesPtr);
	eifilter.filter (*plane_out); // Dette er planet
	eifilter.getRemovedIndices (indices_removed);
	copyPointCloud(*incloud, indices_removed, *outcloud); //rest er resten
av skyen uten planet

	setPrevCloud(cloud);

	if (cloud.compare("model")==0)
	{
		_modelCloud=outcloud;
	}
	else if (cloud.compare("scene")==0)
	{
		_sceneCloud=outcloud;
	}
	updateString(cloud);
	show(cloud);

}
```

```
pcl::PointCloud<pcl::PointXYZ>::Ptr PCLTools::strToCloud(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Output: cloud_ptr - ptr to the cloud with the specified name
	// Function is a helper function. Returns a cloudpointer to the cloud
with the specified name.
	if (cloud.compare("model")==0)
	{
		return _modelCloud;
	}
	else if (cloud.compare("scene")==0)
	{
		return _sceneCloud;
	}
}

void PCLTools::boxSegmentation(std::string cloud, std::vector<float>
boxParameters)
{
	// Input: string - name of the specified cloud.
	// Input: vector<float> - parameters of a bounding box.
	// Function removes points outside of the bounding box.
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud = strToCloud(cloud);
	pcl::PointCloud<pcl::PointXYZ>::Ptr outcloud(new
pcl::PointCloud<pcl::PointXYZ>);
	pcl::ConditionAnd<pcl::PointXYZ>::Ptr range_cond (new
pcl::ConditionAnd<pcl::PointXYZ> ());

	range_cond->addComparison (pcl::FieldComparison<pcl::PointXYZ>::ConstPtr
(new pcl::FieldComparison<pcl::PointXYZ> ("x", pcl::ComparisonOps::LT,
boxParameters[1])));
	range_cond->addComparison (pcl::FieldComparison<pcl::PointXYZ>::ConstPtr
(new pcl::FieldComparison<pcl::PointXYZ> ("x", pcl::ComparisonOps::GT,
boxParameters[0])));
	range_cond->addComparison (pcl::FieldComparison<pcl::PointXYZ>::ConstPtr
(new pcl::FieldComparison<pcl::PointXYZ> ("y", pcl::ComparisonOps::LT,
boxParameters[3])));
	range_cond->addComparison (pcl::FieldComparison<pcl::PointXYZ>::ConstPtr
(new pcl::FieldComparison<pcl::PointXYZ> ("y", pcl::ComparisonOps::GT,
boxParameters[2])));
	range_cond->addComparison (pcl::FieldComparison<pcl::PointXYZ>::ConstPtr
(new pcl::FieldComparison<pcl::PointXYZ> ("z", pcl::ComparisonOps::LT,
boxParameters[5])));
	range_cond->addComparison (pcl::FieldComparison<pcl::PointXYZ>::ConstPtr
(new pcl::FieldComparison<pcl::PointXYZ> ("z", pcl::ComparisonOps::GT,
boxParameters[4])));
	pcl::ConditionalRemoval<pcl::PointXYZ> condrem (range_cond);

	condrem.setInputCloud (incloud);
	condrem.filter (*outcloud);

	setPrevCloud(cloud);

	if (cloud.compare("model")==0)
	{
		_modelCloud=outcloud;
```

```
	}
	else if (cloud.compare("scene")==0)
	{
		_sceneCloud=outcloud;
	}
	updateString(cloud);
	show(cloud);

}

void PCLTools::transformCloud(std::string cloud, Eigen::Matrix4f
transformation_matrix)
{
	// Input: string - name of the specified cloud.
	// Input: Matrix4f - 4x4 transformationmatrix.
	// Function transforms a cloud with a transformation matrix
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud = strToCloud(cloud);
	pcl::PointCloud<pcl::PointXYZ>::Ptr outcloud (new
pcl::PointCloud<pcl::PointXYZ> ());

	pcl::transformPointCloud (*incloud, *outcloud, transformation_matrix);

	setPrevCloud(cloud);

	if (cloud.compare("model")==0)
	{
		_modelCloud=outcloud;
	}
	else if (cloud.compare("scene")==0)
	{
		_sceneCloud=outcloud;
	}
	// Update viewer externally
}

void PCLTools::translateCloud(std::string cloud, char axis, float step)
{
	// Input: string - name of the specified cloud.
	// Input: char - axis.
	// Input: float - stepsize.
	// Function moves cloud along the specified axis with the specified
stepsize.
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud = strToCloud(cloud);
	pcl::PointCloud<pcl::PointXYZ>::Ptr outcloud (new
pcl::PointCloud<pcl::PointXYZ> ());
	Eigen::Matrix4f transformation_matrix = Eigen::Matrix4f::Identity();
	if(axis=='x')
	{
		transformation_matrix(0,3)=step;
	}
	else if(axis=='y')
	{
		transformation_matrix(1,3)=step;
	}
	else if(axis=='z')
	{
		transformation_matrix(2,3)=step;
```

```
	}

	pcl::transformPointCloud (*incloud, *outcloud, transformation_matrix);

	setPrevCloud(cloud);

	if (cloud.compare("model")==0)
	{
		_modelCloud=outcloud;
	}
	else if (cloud.compare("scene")==0)
	{
		_sceneCloud=outcloud;
	}
	// Update viewer externally
}

void PCLTools::rotateCloud(std::string cloud, char axis, double angle)
{
	// Input: string - name of the specified cloud.
	// Input: char - axis.
	// Input: float - angle.
	// Function rotates cloud around the specified axis with the specified
angle.
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud = strToCloud(cloud);

	Eigen::Matrix4f rotate_transformation_matrix =
Eigen::Matrix4f::Identity();
	Eigen::Matrix4f to_zero_transformation_matrix =
Eigen::Matrix4f::Identity();
	Eigen::Matrix4f from_zero_transformation_matrix =
Eigen::Matrix4f::Identity();
	float theta = (M_PI*angle)/180.0;

	pcl::PointXYZ centrePoint = getCenterPoint(cloud);


	to_zero_transformation_matrix (0,3)=-centrePoint.x;
	to_zero_transformation_matrix (1,3)=-centrePoint.y;
	to_zero_transformation_matrix (2,3)=-centrePoint.z;
	from_zero_transformation_matrix (0,3)=centrePoint.x;
	from_zero_transformation_matrix (1,3)=centrePoint.y;
	from_zero_transformation_matrix (2,3)=centrePoint.z;

	if(axis=='x')
	{
		//	|	1		0		0		0	|
		//	|	0		cos(th)-sin(th)	0	|
		//	|	0		sin(th) cos(th)	0	|
		//	|	0		0		0		1	|
		rotate_transformation_matrix (1,1) = cos(theta);
		rotate_transformation_matrix (1,2) = -sin(theta);
		rotate_transformation_matrix (2,1) = sin(theta);
		rotate_transformation_matrix (2,2) = cos(theta);
	}
	else if(axis=='y')
	{
```

```
		//	|	cos(th)		0		sin(th)		0	|
		//	|	0		1		0		0	|
		//	|   -sin(th) 0		cos(th)		0	|
		//	|	0		0		0		1	|
		rotate_transformation_matrix (0,0) = cos(theta);
		rotate_transformation_matrix (0,2) = sin(theta);
		rotate_transformation_matrix (2,0) = -sin(theta);
		rotate_transformation_matrix (2,2) = cos(theta);
	}
	else if(axis=='z'){
		//	|	cos(th)-sin(th)	0		0	|
		//	|	sin(th)		cos(th)		0		0	|
		//	|	0		0		1		0	|
		//	|	0		0		0		1	|
		rotate_transformation_matrix (0,0) = cos(theta);
		rotate_transformation_matrix (0,1) = -sin(theta);
		rotate_transformation_matrix (1,0) = sin(theta);
		rotate_transformation_matrix (1,1) = cos(theta);
	}
	pcl::PointCloud<pcl::PointXYZ>::Ptr outcloud (new
pcl::PointCloud<pcl::PointXYZ> ());
	pcl::PointCloud<pcl::PointXYZ>::Ptr tempcloud1 (new
pcl::PointCloud<pcl::PointXYZ> ());
	pcl::PointCloud<pcl::PointXYZ>::Ptr tempcloud2 (new
pcl::PointCloud<pcl::PointXYZ> ());

	pcl::transformPointCloud (*incloud, *tempcloud1,
to_zero_transformation_matrix);
	pcl::transformPointCloud (*tempcloud1, *tempcloud2,
rotate_transformation_matrix);
	pcl::transformPointCloud (*tempcloud2, *outcloud,
from_zero_transformation_matrix);

	setPrevCloud(cloud);

	if (cloud.compare("model")==0)
	{
		_modelCloud=outcloud;
	}
	else if (cloud.compare("scene")==0)
	{
		_sceneCloud=outcloud;
	}
	// Update viewer externally
}

float PCLTools::maxRadiusEqualization(std::string cloud, std::string refCloud)
{
	// Input: string - name of the specified cloud.
	// Input: string - name of the second specified cloud.
	// Function perform resize on the refCloud to equalize the maxRadius of
the two clouds.
	float cloudMaxRadius = getCloudMaxRadius(cloud);
	float refCloudMaxRadius = getCloudMaxRadius(refCloud);
	float factor = refCloudMaxRadius/cloudMaxRadius;
	resizeCloud(cloud,factor);
	return factor;
```

```
}

float PCLTools::getDensity(std::string cloud)
{
	// Input: string - name of the specified cloud.
	// Function calculates the average density of the pointcloud.
	pcl::PointCloud<PointXYZ>::Ptr cloud_ptr;
	if(cloud.compare("model")==0)
	{
		cloud_ptr = _modelCloud;
	}
	else if(cloud.compare("scene")==0)
	{
		cloud_ptr = _sceneCloud;
	}
	else
		return -1;

	double tempRes = -1.0;
	double res = 0.0;
	int n_points = 0;
	int nres(0);
	std::vector<int> indices(2);
	std::vector<float> sqr_distances(2);
	pcl::KdTreeFLANN<pcl::PointXYZ> tree;
	tree.setInputCloud (cloud_ptr);
	int size = cloud_ptr->size();

	for(int i = 0; i!=size; ++i)
	{
		if (!pcl_isfinite(cloud_ptr->points[i].x))
		{
			continue;
		}

		int prog = static_cast<double>(i)/static_cast<double>(size);
		if(n_points!=0)
		{
			tempRes =
static_cast<double>(res)/static_cast<double>(n_points);
		}
		std::cout<<i<<"    "<<cloud_ptr->size()<<"\n";
		//Considering the second neighbor since the first is the point
itself.
		nres = tree.nearestKSearch (i, 2, indices, sqr_distances);
		cout<<nres<<"\n";
		if (nres == 2)
		{
			res += sqrt (sqr_distances[1]);
			++n_points;
		}
	}
	if (n_points != 0)
	{
		res /= n_points;
	}
	return res;
```

```
}

void PCLTools::voxelGrid(std::string cloud, float leaf_size)
{
	// Input: string - name of the specified cloud.
	// Input: float - size of the leaf used for downsampling
	// Function performs downsampling on a cloud.
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud = strToCloud(cloud);
	pcl::PointCloud<pcl::PointXYZ>::Ptr outcloud(new
pcl::PointCloud<pcl::PointXYZ>);

	std::cout<<"Starting downsampling. Current cloud size are "<< incloud-
>points.size()<<".\n";
	pcl::VoxelGrid<pcl::PointXYZ> filter;
	filter.setInputCloud (incloud);
	filter.setLeafSize (leaf_size, leaf_size, leaf_size);
	filter.filter (*outcloud);
	std::cout<<"Downsampling complete! Current cloud size are "<< incloud-
>points.size()<<".\n";

	setPrevCloud(cloud);

	if (cloud.compare("model")==0)
	{
		_modelCloud=outcloud;
	}
	else if (cloud.compare("scene")==0)
	{
		_sceneCloud=outcloud;
	}
	show(cloud);
}

void PCLTools::radOutlierRemoval(std::string cloud, double radius, int
minNeighbors)
{
	// Input: string - name of the specified cloud.
	// Input: double - radius
	// Input: int - minimum allowed neighbors.
	// Function checks neighbourhoods with specified radius and removes
small neighbourhoods.
	// Using code from
http://pointclouds.org/documentation/tutorials/statistical_outlier.php#statist
ical-outlier-removal
	pcl::PointCloud<pcl::PointXYZ>::Ptr incloud = strToCloud(cloud);
	pcl::PointCloud<pcl::PointXYZ>::Ptr outcloud(new
pcl::PointCloud<pcl::PointXYZ>);
	pcl::RadiusOutlierRemoval<pcl::PointXYZ> ror;
	ror.setInputCloud (incloud);
	ror.setRadiusSearch (radius);
	ror.setMinNeighborsInRadius (minNeighbors);
	ror.filter (*outcloud);
	setPrevCloud(cloud);

	if (cloud.compare("model")==0)
	{
		_modelCloud=outcloud;
```

```
	}
	else if (cloud.compare("scene")==0)
	{
		_sceneCloud=outcloud;
	}
	show(cloud);

}

void PCLTools::icpTransformation(int maxIterations)
{
	// Input: int - max iterations
	// Function tries to align the modelcloud to the scenecloud using ICP.
	// Using code from
http://pointclouds.org/documentation/tutorials/iterative_closest_point.php#ite
rative-closest-point
	pcl::PointCloud<pcl::PointXYZ>::Ptr transformed(new
pcl::PointCloud<pcl::PointXYZ>);
	pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;
	icp.setInputCloud(_modelCloud);
	icp.setInputTarget(_sceneCloud);
	if(maxIterations!=0)
	{
		icp.setMaximumIterations(maxIterations);
	}
	icp.align(*transformed);
	std::cout<<"Model has converged: "<<icp.hasConverged()<<std::endl;
	std::cout<< "Fitness score: "<< icp.getFitnessScore()<< std::endl;
	std::cout<<"Transformation matrix: "<< std::endl;


	Eigen::Matrix4f transform = icp.getFinalTransformation();

	// Print the rotation matrix and translation vector
	// Code for printing matrix from
http://pointclouds.org/documentation/tutorials/correspondence_grouping.php#cor
respondence-grouping
	// Downloaded at 22.05.2014

	Eigen::Matrix3f rotation = transform.block<3,3>(0, 0);
	Eigen::Vector3f translation = transform.block<3,1>(0, 3);

	printf ("\n");
	printf ("            | %6.3f %6.3f %6.3f | \n", rotation (0,0), rotation
(0,1), rotation (0,2));
	printf ("        R = | %6.3f %6.3f %6.3f | \n", rotation (1,0), rotation
(1,1), rotation (1,2));
	printf ("            | %6.3f %6.3f %6.3f | \n", rotation (2,0), rotation
(2,1), rotation (2,2));
	printf ("\n");
	printf ("        t = < %0.3f, %0.3f, %0.3f >\n", translation (0),
translation (1), translation (2));

	std::string cloud;
	cloud="model";
	setPrevCloud(cloud);
	_modelCloud = transformed;
```

```
}

void PCLTools::spatialChange(float res)
{
      // Input: float - resolution of the smallest octree cube.
      // Function calculates the differences between the modelcloud and the
scenecloud.
      // Code based on
http://pointclouds.org/documentation/tutorials/octree_change.php#octree-
change-detection
      pcl::octree::OctreePointCloudChangeDetector<pcl::PointXYZ> octree (res);

      octree.setInputCloud (_sceneCloud);
      octree.addPointsFromInputCloud ();
      octree.switchBuffers ();

      octree.setInputCloud (_modelCloud);
      octree.addPointsFromInputCloud ();

      std::vector<int> newPointIdxVector;
      octree.getPointIndicesFromNewVoxels (newPointIdxVector);

      pcl::PointCloud<pcl::PointXYZ>::Ptr changes(new
pcl::PointCloud<pcl::PointXYZ>);
      pcl::copyPointCloud<pcl::PointXYZ,pcl::PointXYZ>(*_modelCloud,newPointId
xVector,*changes);

      show("clear");
      cloudAdder(changes,"changes", 100, 100, 140);
      _viewer->getInteractorStyle()->Spin();//Force the visualizer to
update...
}

void PCLTools::mergeClouds()
{
      // Function merges modelcloud with scenecloud.
      if(_modelIsLoaded && _sceneIsLoaded)
            *_sceneCloud+=*_modelCloud;
}

//--------------------------------------------
// CGRecognition------------------------------
//--------------------------------------------
void PCLTools::recognize(float model_ss, float scene_ss, float rf_rad, float
descr_rad, float cg_size, int cg_tresh )
{
      // Input: float - model_ss - model sampling size
      // Input: float - scene_ss - scene sampling size
      // Input: float - rf_rad - radius for reference frame
      // Input: float - descr_rad - radius for descriptor
      // Input: float - cg_size - bin size of voting algorithm.
      // Input: float - cg_thresh - threshold for correspondences.
      // Function starts recognition based on Correspondence Grouping
      rec.reset(new CGRecognition());
      rec->setInputModel(_modelCloud);
      rec->setInputScene(_sceneCloud);
```

```
	rec->setParameters(model_ss, scene_ss, rf_rad, descr_rad, cg_size, 
cg_tresh);
	rec->compute();

	transformations = rec->getTransformations();
	correspondences = rec->getCorrespondences();

	cout<<"\nRec sucsess!\n";

	if(transformations.size()>0)
	{
		transCounter = 0;
		visTransformation(transCounter);
	}
}

void PCLTools::setShowCorresp(bool showcp)
{
	//Function sets boolean to decide if correspondences should be shown.
	_show_correspondences = showcp;
}

void PCLTools::visTransformation(int i)
{
	// Input: int - index of the transformation.
	// Function visualizes a transformation and prints it in the command 
window.
	// Using code from 
http://pointclouds.org/documentation/tutorials/correspondence_grouping.php#cor
respondence-grouping

	std::cout << "\n    Showing transformation " << transCounter << "of" << 
transformations.size()<<":" << std::endl;
	std::cout << "        Correspondences belonging to this instance: " << 
correspondences[i].size () << std::endl;


	Eigen::Matrix3f rotation = transformations[i].block<3,3>(0, 0);
	Eigen::Vector3f translation = transformations[i].block<3,1>(0, 3);

	printf ("\n");
	printf ("            | %6.3f %6.3f %6.3f | \n", rotation (0,0), rotation 
(0,1), rotation (0,2));
	printf ("        R = | %6.3f %6.3f %6.3f | \n", rotation (1,0), rotation 
(1,1), rotation (1,2));
	printf ("            | %6.3f %6.3f %6.3f | \n", rotation (2,0), rotation 
(2,1), rotation (2,2));
	printf ("\n");
	printf ("        t = < %0.3f, %0.3f, %0.3f >\n", translation (0), 
translation (1), translation (2));

	pcl::PointCloud<PointXYZ>::Ptr rotated_model (new 
pcl::PointCloud<PointXYZ> ());
	pcl::transformPointCloud (*_modelCloud, *rotated_model, 
transformations[i]);

	std::stringstream ss_cloud;
```

```
	ss_cloud << "instance" << i;


	show("both");
	cloudAdder(rotated_model,ss_cloud.str(), 230, 0, 140);

	if (_show_correspondences)
	{
		_viewer->removeAllShapes();
		viewerText("both");
		for (size_t j = 0; j < correspondences[i].size (); ++j)
		{
			std::stringstream ss_line;
			ss_line << "correspondence_line" << i << "_" << j;
			pcl::PointXYZ& model_point = rec->getModelKeypoints()->at
(correspondences[i][j].index_query);
			pcl::PointXYZ& scene_point = rec->getSceneKeypoints()->at
(correspondences[i][j].index_match);
			_viewer->addLine<pcl::PointXYZ, pcl::PointXYZ> (model_point,
scene_point, 0, 255, 80, ss_line.str ());
		}
	}

	_viewer->getInteractorStyle()->Spin();//Force the visualizer to
update...

}

void PCLTools::visNextTransformation()
{
	// Function visualizes the next transformation from the CGRecognition.
	transCounter+=1;
	if(transCounter>transformations.size()-1)
		transCounter=0;
	visTransformation(transCounter);
}

void PCLTools::setTransformation()
{
	// Function moves cloud to a detected transformation given by index
transCounter.
	setPrevCloud("model");
	pcl::transformPointCloud (*_modelCloud, *_modelCloud,
transformations[transCounter]);
	show("both");
}

int PCLTools::getTransCount()
{
	// Function returns number of detexted transformations
	return transformations.size();
}

int PCLTools::getCurrentTransCount()
{
	// Function returns current transformation number.
	return (transCounter+1);
```

```
}

//--------------------------------------------
// ISMRecognition-----------------------------
//--------------------------------------------
void PCLTools::ismRecognition(float ss, float descrr, float normr, float
thresh)
{
	// Input: float - ss - model sampling size
	// Input: float - descrr - radius for descriptor
	// Input: float - normr - bin size of voting algorithm.
	// Input: float - thresh - threshold of the visualization.
	// Function starts recognition based on Implicit shape model.
	ISMRecognition ismrec;
	ismrec.setInputModel(_modelCloud);
	ismrec.setInputScene(_sceneCloud);
	ismrec.setParameters(ss,descrr,normr);
	ismrec.compute();
	std::vector<pcl::ISMPeak, Eigen::aligned_allocator<pcl::ISMPeak>>
strongest_peaks;
	std::vector<pcl::PointXYZ> centerpoints;
	strongest_peaks=ismrec.getStrongestPeaks();
	centerpoints=ismrec.getCenterPoints();
	if (centerpoints.size()>0)
	{
		float radius = 0.05*getCloudMaxRadius("model");

		visualizeCenterPoints(centerpoints, strongest_peaks, radius,
thresh);
	}
	else
		std::cout<<"\nNo centerpoints detected!\n";

}

void PCLTools::visualizeCenterPoints(std::vector<pcl::PointXYZ> centerpoints,
std::vector<pcl::ISMPeak, Eigen::aligned_allocator<pcl::ISMPeak>>
strongest_peaks, float radius,  float thresh)
{
	// Input: vector<pcl::PointXYZ> - detected centerpoints.
	// Input: vector<pcl::ISMPeak, Eigen::aligned_allocator<pcl::ISMPeak>> -
strongest peaks from ISM.
	// Input: float - radius for the spheres
	// Input: float - threshold for the visualuzation
	// Function visualizes the detexted centerpoints from ISM, with spheres.
	// strongest peak is painted blue, peaks with density above threshold is
painted yellow. Rest is painted in red.
	pcl::PointCloud <pcl::PointXYZ>::Ptr sphere = (new pcl::PointCloud
<pcl::PointXYZ>)->makeShared();
	int strongest_peak_idx = findMaxPeakIdx(strongest_peaks);
	int sphereCount = 0;
	pcl::PointXYZ basic_point;
	float stepsize = 5.0;
	show("both");
	for (size_t p_num = 0; p_num < centerpoints.size(); p_num++)
	{
```

```
            float z(0), radius2(0);
            for (float angle1(0.0); angle1 <= 180.0; angle1 += stepsize)
            {
                  z = radius * cosf (pcl::deg2rad(angle1)) +
centerpoints[p_num].z;
                  radius2= radius * sinf(pcl::deg2rad(angle1));
                  for (float angle2(0.0); angle2 <= 360.0; angle2 += stepsize)
                  {
                        pcl::PointXYZ basic_point;
                        basic_point.x = radius2 * cosf
(pcl::deg2rad(angle2))+centerpoints[p_num].x;
                        basic_point.y = radius2 * sinf
(pcl::deg2rad(angle2))+centerpoints[p_num].y;
                        basic_point.z = z;
                        sphere->points.push_back(basic_point);
                  }
            }
            sphere-> width = (int) sphere->points.size ();
            sphere-> height = 1;
            if(p_num==strongest_peak_idx)
            {
                  sphereCount++;
                  std::stringstream s;
                  s<<"peak"<<sphereCount;
                  cloudAdder(sphere,s.str(),20,0,230);
            }
            else if(strongest_peaks.at(p_num).density>thresh)
            {
                  sphereCount++;
                  std::stringstream s;
                  s<<"peak"<<sphereCount;
                  cloudAdder(sphere,s.str(),200,200,0);
            }
            else
            {
                  sphereCount++;
                  std::stringstream s;
                  s<<"peak"<<sphereCount;
                  cloudAdder(sphere,s.str(),200,50,0);
            }
            sphere->clear();
      }

      _viewer->getInteractorStyle()->Spin();//Force the visualizer to
update...
}

int PCLTools::findMaxPeakIdx(std::vector<pcl::ISMPeak,
Eigen::aligned_allocator<pcl::ISMPeak>> strongest_peaks)
{
      // Input: vector<pcl::ISMPeak, Eigen::aligned_allocator<pcl::ISMPeak>> -
strongest peaks from ISM.
      // Function returns the index of the strongest peak in the list.
      int max_peak_idx(0);
      double temp(0);
      std::cout<<std::endl;
      for(int i=0; i<strongest_peaks.size(); i++)
```

```
	{
		std::cout<<"Peak number " << i << " density: 
"<<strongest_peaks.at(i).density<<std::endl;
		if(strongest_peaks.at(i).density>temp)
		{
			max_peak_idx=i;
			temp=strongest_peaks.at(i).density;
		}
	}
	return max_peak_idx;
}
```

B.10 recognition.h

```
// Class based on code from:
//
http://pointclouds.org/documentation/tutorials/correspondence_grouping.php#cor
respondence-grouping
#pragma once

#include <pcl/io/pcd_io.h>
#include <pcl/point_cloud.h>
#include <pcl/correspondence.h>
#include <pcl/features/normal_3d_omp.h>
#include <pcl/features/shot_omp.h>
#include <pcl/features/board.h>
#include <pcl/keypoints/uniform_sampling.h>
#include <pcl/recognition/cg/hough_3d.h>
#include <pcl/recognition/cg/geometric_consistency.h>
#include <pcl/visualization/pcl_visualizer.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/kdtree/impl/kdtree_flann.hpp>
#include <pcl/common/transforms.h>
#include <pcl/console/parse.h>
#include <pcl/common/time_trigger.h>
#include <pcl/common/time.h>

using namespace pcl;
using namespace std;

typedef pcl::PointXYZ PointType;
typedef pcl::Normal NormalType;
typedef pcl::ReferenceFrame RFType;
typedef pcl::SHOT352 DescriptorType;

class CGRecognition
{
public:
	CGRecognition();
	void setParameters(float model_ss, float scene_ss, float rf_rad, float
descr_rad, float cg_size, int ch_tresh);
	void setInputModel(PointCloud<PointXYZ>::Ptr inmodel);
	void setInputScene(PointCloud<PointXYZ>::Ptr inscene);
	void compute();
	std::vector<Eigen::Matrix4f, Eigen::aligned_allocator<Eigen::Matrix4f> >
getTransformations();
	std::vector<pcl::Correspondences> getCorrespondences();
	pcl::PointCloud<PointType>::Ptr getModelKeypoints();
	pcl::PointCloud<PointType>::Ptr getSceneKeypoints();
```

```
private:
      pcl::PointCloud<PointType>::Ptr model;
      pcl::PointCloud<PointType>::Ptr model_keypoints;
      pcl::PointCloud<PointType>::Ptr scene;
      pcl::PointCloud<PointType>::Ptr scene_keypoints;
      pcl::PointCloud<NormalType>::Ptr model_normals;
      pcl::PointCloud<NormalType>::Ptr scene_normals;
      pcl::PointCloud<DescriptorType>::Ptr model_descriptors;
      pcl::PointCloud<DescriptorType>::Ptr scene_descriptors;
      std::vector<Eigen::Matrix4f, Eigen::aligned_allocator<Eigen::Matrix4f> >
rototranslations;
      std::vector<pcl::Correspondences> clustered_corrs;
      float model_ss_;
      float scene_ss_;
      float rf_rad_;
      float descr_rad_;
      float cg_size_;
      float cg_thresh_;
};
```

B.11 recognition.cpp

```
// Class based on code from:
//
http://pointclouds.org/documentation/tutorials/correspondence_grouping.php#cor
respondence-grouping
#include "recognition.h"

CGRecognition::CGRecognition()
{
	model.reset(new pcl::PointCloud<PointType> ());
	model_keypoints.reset (new pcl::PointCloud<PointType> ());
	scene.reset (new pcl::PointCloud<PointType> ());
	scene_keypoints.reset (new pcl::PointCloud<PointType> ());
	model_normals.reset (new pcl::PointCloud<NormalType> ());
	scene_normals.reset (new pcl::PointCloud<NormalType> ());
	model_descriptors.reset (new pcl::PointCloud<DescriptorType> ());
	scene_descriptors.reset (new pcl::PointCloud<DescriptorType> ());
}

void CGRecognition::setParameters(float model_ss, float scene_ss, float
rf_rad, float descr_rad, float cg_size, int ch_tresh)
{
	model_ss_=model_ss;
	scene_ss_=scene_ss;
	rf_rad_=rf_rad;
	descr_rad_=descr_rad;
	cg_size_=cg_size;
	cg_thresh_=ch_tresh;
}
void CGRecognition::setInputModel(PointCloud<PointXYZ>::Ptr inmodel)
{
	model=inmodel;
}
void CGRecognition::setInputScene(PointCloud<PointXYZ>::Ptr inscene)
{
	scene = inscene;
}

std::vector<Eigen::Matrix4f, Eigen::aligned_allocator<Eigen::Matrix4f> >
CGRecognition::getTransformations()
{
	return rototranslations;
}

std::vector<pcl::Correspondences> CGRecognition::getCorrespondences()
{
	return clustered_corrs;
}

pcl::PointCloud<PointType>::Ptr CGRecognition::getModelKeypoints()
```

```
{
	return model_keypoints;
}
pcl::PointCloud<PointType>::Ptr CGRecognition::getSceneKeypoints()
{
	return scene_keypoints;
}

void CGRecognition::compute()
{
	//Start Timer
	double start_rec = pcl::getTime ();

	//Compute Normals
	pcl::NormalEstimationOMP<PointType, NormalType> norm_est;
	norm_est.setKSearch (10);
	norm_est.setInputCloud (model);
	norm_est.compute (*model_normals);

	norm_est.setInputCloud (scene);
	norm_est.compute (*scene_normals);

	//Downsample to extract keypoints
	pcl::PointCloud<int> sampled_indices;

	pcl::UniformSampling<PointType> uniform_sampling;
	uniform_sampling.setInputCloud (model);
	uniform_sampling.setRadiusSearch (model_ss_);
	uniform_sampling.compute (sampled_indices);
	pcl::copyPointCloud (*model, sampled_indices.points, *model_keypoints);
	std::cout << "Model total points: " << model->size () << "; Selected
Keypoints: " << model_keypoints->size () << std::endl;

	uniform_sampling.setInputCloud (scene);
	uniform_sampling.setRadiusSearch (scene_ss_);
	uniform_sampling.compute (sampled_indices);
	pcl::copyPointCloud (*scene, sampled_indices.points, *scene_keypoints);
	std::cout << "Scene total points: " << scene->size () << "; Selected
Keypoints: " << scene_keypoints->size () << std::endl;

	//Compute Descriptors
	pcl::SHOTEstimationOMP<PointType, NormalType, DescriptorType> descr_est;
	descr_est.setRadiusSearch (descr_rad_);

	descr_est.setInputCloud (model_keypoints);
	descr_est.setInputNormals (model_normals);
	descr_est.setSearchSurface (model);
	descr_est.compute (*model_descriptors);

	descr_est.setInputCloud (scene_keypoints);
	descr_est.setInputNormals (scene_normals);
	descr_est.setSearchSurface (scene);
	descr_est.compute (*scene_descriptors);

	//Find correspondances with kdtree
	pcl::CorrespondencesPtr model_scene_corrs (new pcl::Correspondences ());
```

```
      pcl::KdTreeFLANN<DescriptorType> match_search;
      match_search.setInputCloud (model_descriptors);

      //  For each scene keypoint descriptor, find nearest neighbor into the
model keypoints descriptor cloud and add it to the correspondences vector.
      for (size_t i = 0; i < scene_descriptors->size (); ++i)
      {
            std::vector<int> neigh_indices (1);
            std::vector<float> neigh_sqr_dists (1);
            if (!pcl_isfinite (scene_descriptors->at (i).descriptor[0]))
//skipping NaNs
            {
                  continue;
            }
            int found_neighs = match_search.nearestKSearch (scene_descriptors-
>at (i), 1, neigh_indices, neigh_sqr_dists);
            if(found_neighs == 1 && neigh_sqr_dists[0] < 0.25f) //  add match
only if the squared descriptor distance is less than 0.25 (SHOT descriptor
distances are between 0 and 1 by design)
            {
                  pcl::Correspondence corr (neigh_indices[0], static_cast<int>
(i), neigh_sqr_dists[0]);
                  model_scene_corrs->push_back (corr);
            }
      }
      std::cout << "Correspondences found: " << model_scene_corrs->size () <<
std::endl;

      pcl::PointCloud<RFType>::Ptr model_rf (new pcl::PointCloud<RFType> ());
      pcl::PointCloud<RFType>::Ptr scene_rf (new pcl::PointCloud<RFType> ());

      pcl::BOARDLocalReferenceFrameEstimation<PointType, NormalType, RFType>
rf_est;
      rf_est.setFindHoles (true);
      rf_est.setRadiusSearch (rf_rad_);

      rf_est.setInputCloud (model_keypoints);
      rf_est.setInputNormals (model_normals);
      rf_est.setSearchSurface (model);
      rf_est.compute (*model_rf);

      rf_est.setInputCloud (scene_keypoints);
      rf_est.setInputNormals (scene_normals);
      rf_est.setSearchSurface (scene);
      rf_est.compute (*scene_rf);

      //  Clustering
      pcl::Hough3DGrouping<PointType, PointType, RFType, RFType> clusterer;
      clusterer.setHoughBinSize (cg_size_);
      clusterer.setHoughThreshold (cg_thresh_);
      clusterer.setUseInterpolation (true);
      clusterer.setUseDistanceWeight (false);

      clusterer.setInputCloud (model_keypoints);
      clusterer.setInputRf (model_rf);
      clusterer.setSceneCloud (scene_keypoints);
      clusterer.setSceneRf (scene_rf);
```

```
    clusterer.setModelSceneCorrespondences (model_scene_corrs);

    //clusterer.cluster (clustered_corrs);
    clusterer.recognize (rototranslations, clustered_corrs);

    //stop timer
    double stop_rec = pcl::getTime () - start_rec;

    std::cout << "Recognition completed in " << stop_rec << " seconds." <<
std::endl;
    std::cout << "Model instances found: " << rototranslations.size () <<
std::endl;
}
```

B.12 recognitionISM.cpp

```
// Class based on code from:
// http://pointclouds.org/documentation/tutorials/implicit_shape_model.php
#pragma once

#include <iostream>
#include <pcl/point_cloud.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/feature.h>
#include <pcl/features/fpfh.h>
#include <pcl/features/impl/fpfh.hpp>
#include <pcl/recognition/implicit_shape_model.h>
#include <pcl/common/time_trigger.h>
#include <pcl/common/time.h>

using namespace pcl;
using namespace std;

class ISMRecognition
{
public:
	ISMRecognition();
	void setParameters(float ss, float descr_rad, float norm_rad );
	void setInputModel(pcl::PointCloud<pcl::PointXYZ>::Ptr inmodel);
	void setInputScene(pcl::PointCloud<pcl::PointXYZ>::Ptr inscene);
	void compute();
	std::vector<pcl::PointXYZ> getCenterPoints();
	std::vector<pcl::ISMPeak, Eigen::aligned_allocator<pcl::ISMPeak>>
getStrongestPeaks();
	float ss_;
	float norm_rad_;
	float descr_rad_;
private:
	pcl::PointCloud<pcl::PointXYZ>::Ptr model;
	pcl::PointCloud<pcl::PointXYZ>::Ptr scene;
	std::vector<pcl::PointXYZ> detected_centerpoints;
	std::vector<pcl::ISMPeak, Eigen::aligned_allocator<pcl::ISMPeak> >
strongest_peaks;
};
```

B.13 recognitionISM.cpp

```
// Class based on code from:
// http://pointclouds.org/documentation/tutorials/implicit_shape_model.php
#include "recognitionISM.h"
ISMRecognition::ISMRecognition()
{
	model.reset(new pcl::PointCloud<pcl::PointXYZ> ());
	scene.reset(new pcl::PointCloud<pcl::PointXYZ> ());
}
void ISMRecognition::setParameters(float ss, float descr_rad, float
norm_rad )
{
	ss_= ss;
	norm_rad_ = norm_rad;
	descr_rad_ = descr_rad;
}
void ISMRecognition::setInputModel(pcl::PointCloud<pcl::PointXYZ>::Ptr
inmodel)
{
	model = inmodel;
}
void ISMRecognition::setInputScene(pcl::PointCloud<pcl::PointXYZ>::Ptr
inscene)
{
	scene = inscene;
}
void ISMRecognition::compute()
{
	double start_training = pcl::getTime ();
	pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> normal_estimator;
	normal_estimator.setRadiusSearch (norm_rad_);

	std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> training_clouds;
	std::vector<pcl::PointCloud<pcl::Normal>::Ptr> training_normals;
	std::vector<unsigned int> training_classes;

	pcl::PointCloud<pcl::Normal>::Ptr tr_normals = (new
pcl::PointCloud<pcl::Normal>)->makeShared ();
	normal_estimator.setInputCloud (model);
	normal_estimator.compute (*tr_normals);

	training_clouds.push_back (model);
	training_normals.push_back (tr_normals);
	training_classes.push_back (0);

	pcl::FPFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::Histogram<153>
>::Ptr fpfh(new pcl::FPFHEstimation<pcl::PointXYZ, pcl::Normal,
pcl::Histogram<153> >);
	fpfh->setRadiusSearch (descr_rad_);
	pcl::Feature< pcl::PointXYZ, pcl::Histogram<153> >::Ptr
feature_estimator(fpfh);
```

```
pcl::ism::ImplicitShapeModelEstimation<153, pcl::PointXYZ, pcl::Normal>
ism;
      ism.setFeatureEstimator(feature_estimator);
      ism.setTrainingClouds (training_clouds);
      ism.setTrainingNormals (training_normals);
      ism.setTrainingClasses (training_classes);
      ism.setSamplingSize (ss_);

      pcl::ism::ImplicitShapeModelEstimation<153, pcl::PointXYZ,
pcl::Normal>::ISMModelPtr ismModel =
boost::shared_ptr<pcl::features::ISMModel>
            (new pcl::features::ISMModel);
      ism.trainISM (ismModel);
      // END TRAINING
      //ismModel->loadModelFromfile (file);
      double end_training = pcl::getTime () - start_training;
      std::cout<<"Training completed in " << end_training<<"
seconds"<<std::endl;
      //ism.setSamplingSize (scene_ss_);
      double start_testing = pcl::getTime ();

      unsigned int testing_class = 0;

      pcl::PointCloud<pcl::Normal>::Ptr testing_normals = (new
pcl::PointCloud<pcl::Normal>)->makeShared ();
      normal_estimator.setInputCloud (scene);
      normal_estimator.compute (*testing_normals);

      boost::shared_ptr<pcl::features::ISMVoteList<pcl::PointXYZ> >
vote_list = ism.findObjects (
            ismModel,
            scene,
            testing_normals,
            testing_class);

      double radius = ismModel->sigmas_[testing_class] * 10.0;
      double sigma = ismModel->sigmas_[testing_class];
      vote_list->findStrongestPeaks (strongest_peaks, testing_class,
radius, sigma);

      double end_testing = pcl::getTime () - start_testing;
      std::cout<<"Testing completed in " << end_testing<<"
seconds"<<std::endl;

      pcl::PointXYZ point;

      for (size_t i_vote = 0; i_vote < strongest_peaks.size (); i_vote++)
      {
            point.x = strongest_peaks[i_vote].x;
            point.y = strongest_peaks[i_vote].y;
            point.z = strongest_peaks[i_vote].z;
            detected_centerpoints.push_back(point);
      }

}
std::vector<pcl::PointXYZ> ISMRecognition::getCenterPoints()
{
      return detected_centerpoints;
}
```

```
std::vector<pcl::ISMPeak, Eigen::aligned_allocator<pcl::ISMPeak>>
ISMRecognition::getStrongestPeaks()
{
      return strongest_peaks;
}
```

B.14 button.h

```
/****************************************************************************
**
** Copyright (C) 2014 Digia Plc and/or its subsidiary(-ies).
** Contact: http://www.qt-project.org/legal
**
** This file is part of the examples of the Qt Toolkit.
**
** $QT_BEGIN_LICENSE:BSD$
** You may use this file under the terms of the BSD license as follows:
**
** "Redistribution and use in source and binary forms, with or without
** modification, are permitted provided that the following conditions are
** met:
**   * Redistributions of source code must retain the above copyright
**     notice, this list of conditions and the following disclaimer.
**   * Redistributions 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.
**   * Neither the name of Digia Plc and its Subsidiary(-ies) nor the names
**     of its contributors may be used to endorse or promote products derived
**     from this software without specific prior written permission.
**
**
** 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 COPYRIGHT
** OWNER 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."
**
** $QT_END_LICENSE$
**
****************************************************************************/
#pragma once
#include <QToolButton>
// from: http://qt-project.org/doc/qt-4.8/widgets-calculator-button-h.html
class Button : public QToolButton
{
      Q_OBJECT

public:
      Button(const QString &text, QWidget *parent = 0);
      QSize sizeHint() const;
};
```

B.15 button.cpp

```
/***************************************************************************
**
 **
 ** Copyright (C) 2014 Digia Plc and/or its subsidiary(-ies).
 ** Contact: http://www.qt-project.org/legal
 **
 ** This file is part of the examples of the Qt Toolkit.
 **
 ** $QT_BEGIN_LICENSE:BSD$
 ** You may use this file under the terms of the BSD license as follows:
 **
 ** "Redistribution and use in source and binary forms, with or without
 ** modification, are permitted provided that the following conditions are
 ** met:
 **   * Redistributions of source code must retain the above copyright
 **     notice, this list of conditions and the following disclaimer.
 **   * Redistributions 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.
 **   * Neither the name of Digia Plc and its Subsidiary(-ies) nor the
names
 **     of its contributors may be used to endorse or promote products
derived
 **     from this software without specific prior written permission.
 **
 **
 ** 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 COPYRIGHT
 ** OWNER 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."
 **
 ** $QT_END_LICENSE$
 **

***************************************************************************
*/
#include "button.h"
#include <QtGui>
// From http://qt-project.org/doc/qt-4.8/widgets-calculator-button-cpp.html
Button::Button(const QString &text, QWidget *parent)
: QToolButton(parent)
{
      setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Preferred);
      setText(text);
      setMaximumHeight(40);
}
```

```
QSize Button::sizeHint() const
{
      QSize size = QToolButton::sizeHint();
      size.rheight() += 20;
      size.rwidth() = qMax(size.width(), size.height());
      return size;
}
```

B.16 CMakeLists.txt

```
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)
project(PCRTool)

set(MY_PCL_DIR "D:/1_Masteroppgave_Prosjektmappe/libs/pcl-1.7.1")
#set(PCL_DIR "${MY_PCL_DIR}/cmake/PCLConfig.cmake")
set(PCL_ROOT "${MY_PCL_DIR}")
#set(FLANN_ROOT "E:/Upload/pcl-1.7.1/")
#set(EIGEN_ROOT "")
#set(Boost_ROOT "")
set( CMAKE_PREFIX_PATH "${CMAKE_PREFIX_PATH};${MY_PCL_DIR}/cmake")

find_package(PCL 1.7 REQUIRED HINTS "${MY_PCL_DIR}/cmake"
NO_SYSTEM_ENVIRONMENT_PATH NO_DEFAULT_PATH)
find_package( Qt4 REQUIRED )
find_package (QVTK)

set( QT_USE_QTOPENGL TRUE )
set( QT_USE_QTMAIN TRUE )
set( QT_USE_QTGUI TRUE )
set( QT_USE_QTSCRIPT TRUE )
set( QT_USE_QTSVG TRUE )
set( QT_USE_QTDECLARATIVE TRUE )
include( ${QT_USE_FILE} )

set( MINE_HEADER_FILER
      button.h
      mainwindow.h
      pcltools.h
      recognition.h
      recognitionISM.h
      centralwidget.h
      panels.h
)

set( MINE_SOURCE_FILER
      PCRTool.cpp
      button.cpp
      mainwindow.cpp
      pcltools.cpp
      recognition.cpp
      recognitionISM.cpp
      centralwidget.cpp
      panels.cpp
)

# HEADERE SOM BRUKER QT
set( MOC_HEADERS
      button.h
      mainwindow.h
      centralwidget.h
      panels.h
)
```

```
# Qt Pre-processing
if ( MOC_HEADERS )
  qt_wrap_cpp( pcl_test MOC_SOURCES ${MOC_HEADERS} )
  set( MINE_SOURCE_FILER ${MOC_SOURCES} ${MINE_SOURCE_FILER} )
endif()


include_directories(
      ${PCL_INCLUDE_DIRS}
      ${QVTK_INCLUDE_DIR}
)
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})

add_executable(PCRTool
      ${MINE_SOURCE_FILER}
      ${MINE_HEADER_FILER}
)

target_link_libraries(PCRTool
      ${PCL_LIBRARIES}
      ${QT_LIBRARIES}
      ${QVTK_LIBRARY}
)
```