Point-cloud Multiple Object Tracking (PMOT)

Software manual

PhD. Seongyong Koo

Lehrstuhl für
STEUERUNGS- und REGELUNGSTECHNIK
Technische Universität München

Dynamic Human-Robot-Interaction Group/Prof. Dongheui Lee

Beginn: 01.06.2014
Abgabe: 31.03.2015

Contents

Chapter 1

Introduction

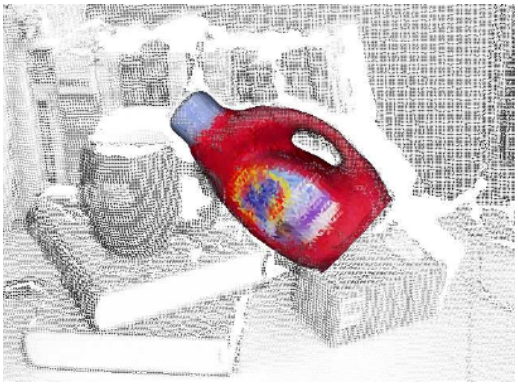
This document aims at introducing a new open-source software for tracking multiple unknown and random objects from point-cloud video that can be easily captured by low-cost and fast RGB-D camera [1]. The product named Point-cloud Multiple Object Tracking (PMOT)[2] features real-time, model-free, and standalone software by means of using highly trusted and actively developing technologies such as Robot Operating System (ROS) [3], point cloud library (PCL)[4], and Compute Unified Device Architecture (CUDA)[5]. Especially, this book is a generic user-guide for engineers and researchers who want to utilize the results of PMOT, which are point-level object indexes and object-level 3D poses. To make them easy to understand the software, the following chapters include each content of functionality, installation steps, execution steps, related data set, and reference for source files.
First of all, this chapter explains the main functionality of PMOT with brief summary of the related scientific papers. It will give you theoretical background to help your understanding of the software.

1.1 Background: Point-cloud object segmentation and tracking

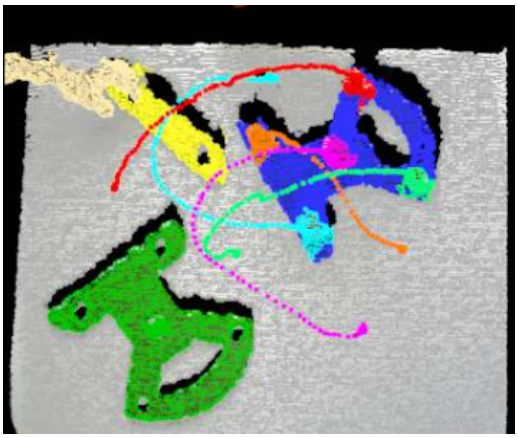
Point-cloud is a set of 6-dimensional points, each of which contains x,y,z,r,g,b information in the 3D space. In order to identify individual moving objects from a fast RGB-D camera, there have been two different approaches: model-based object tracking and model-free object segmentation. The first approach is mainly for 6D pose estimation of a known single object while the second one is extracting each point-cloud that is involved in each individual object among multiple and unknown

[1] e.g. ASUS Xtion Pro, Microsoft Kinect sensor, and Softkinectic iisu Pro series.
[2] http://www.hri.ei.tum.de/pmot/
[3] http://www.ros.org/
[4] http://pointclouds.org/
[5] https://developer.nvidia.com/cuda-zone

(a) Model-based 3D object tracking in [RC11]



(b) Model-based 3D object tracking in [CC13]



(c) Model-free multiple object segmentation and tracking in [PKAW13]

(d) Model-free multiple object segmentation and tracking in [KLK13b]

Figure 1.1: Existing point-cloud object tracking softwares

objects in the scene point-cloud. Considering each object segment a representative point, in the second approach, tracking is a problem of constructing temporal associations among multiple object segments between consecutive time frames, which is called multiple object tracking (MOT). Fig. 1.1 shows four existing methods for the two tracking approaches.

Although the two approaches have been significantly developed in the recent studies [RC11, CC13, PKAW13, KLK13a, KLK13b, KLK14a, KLK14b, LKLK15, PW15], they still have unreleased issues that are involved a trade-off with respect to the prior existence of corresponding object models. The model-based tracking is relatively faster and more precise than model-free tracking, but it requires constructing an accurate object model prior to tracking the specific target, which limits its application domain. On the other hand, the model-free approach allows to track multiple unknown objects without any prior knowledge. However, due to the modelless property, it performs lower tracking performance in not only accuracy but also

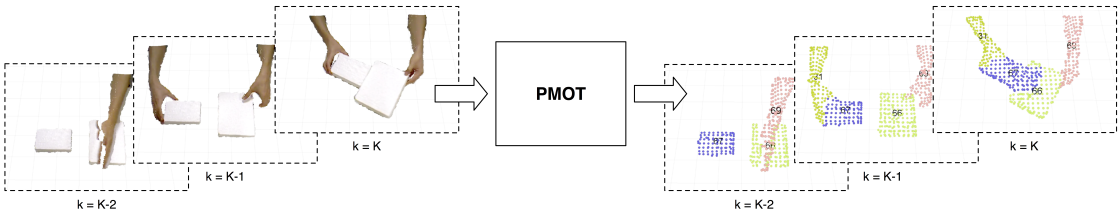


Figure 1.2: Input and output of PMOT

computation time.
These difficulties call for a new method that can have advantages of both approaches, which achieves *accurate* tracking performance for *multiple unknown objects* in *real-time without any prior knowledge*. With this goal in mind, PMOT has been realized as the result of a series of our prior works in [KLK13a, KLK13b, KLK14a, KLK14b, LKLK15]. The main function of PMOT is processing an input point-cloud and producing information of each individual object at each time step as shown in Fig. 1.2.

1.2 Simultaneous multiple object segmentation and tracking (SMOST)

To give you a principle idea behind how PMOT works, this section describes a formal definition of a main problem in PMOT: simultaneous multiple object segmentation and tracking (SMOST). With the input point-cloud at each time step t, $\mathbf{P}_t = \{\mathbf{p}_1, ..., \mathbf{p}_n\}$, each of which contains the 3-dimensional XYZ position and RGB color of a measured point on the object surface, SMOST estimates object indexes of the point cloud, $\mathbf{O}_t$, and 3D poses and shapes of multiple objects $\mathcal{X}_t$ at each time step.

$$SMOST(\mathbf{P}_t) \rightarrow \{\mathbf{I}_t, \mathcal{X}_t\} \tag{1.1}$$

Here, the object indexes, $\mathbf{I}_t = \{I(\mathbf{p}_1), ..., I(\mathbf{p}_n)\}$ is a set of index variables $I(\mathbf{p}_i) \in \mathbb{I}$ of each point $\mathbf{p}_i$, which indicates one of the k objects $\mathbb{I} = \{o_1, ..., o_k\}$, $o_i \in \mathbb{N}$ where the point is involved. The second output of SMOST is a set of k constructive object models, $\mathcal{X}_t = \{\mathcal{X}_t^{(1)}, ..., \mathcal{X}_t^{(k)}\}$. Constructive object model of an object i, $\mathcal{X}_t^{(i)} = \{\mathbf{M}_t^{(i)}, \mathbf{X}_t^{(i)}\}$, consists of a model point cloud $\mathbf{M}_t^{(i)}$, and 3D pose $\mathbf{X}_t^{(i)}$.
As you see in the Fig. 1.3, input point-cloud $\mathbf{P}_t$ is represented from the world coordinate W, while the model point-cloud $\mathbf{M}_t^{(i)}$ is represented from the moving coordinate at each time, O_t^i.

$$\mathbf{M}_t^{(i)} = \{\mathbf{p}_1, ..., \mathbf{p}_{n_i}\}_{|O_t^i} \tag{1.2}$$

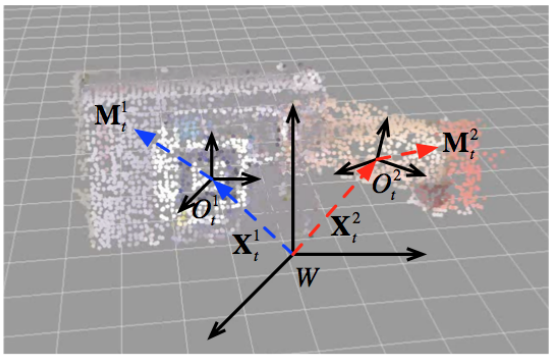


Figure 1.3: An example of constructive object models and their coordinates

The object pose in the 3D space $\mathbf{X}_t^{(i)}$ is represented as 6-DOF pose in $SE(3)$ group of the object coordinate O_t^i relative to the world coordinate W:

$$\mathbf{X}_t^{(i)} = \{x, y, z, roll, pitch, yaw\}_{O_t^i|W} \tag{1.3}$$

How does SMOST work? Fig. 1.4 shows a block diagram of the SMOST implementation. At each time step t, the only input is a point-cloud $\mathbf{P}_t$ from a camera. The first step is object segmentation of the point-cloud with previously constructed object models, $\mathcal{X}_{t-1}$. Without explicit object models that should be designed from the outside of the program, the object model $\mathcal{X}$ is internally constructed at every time step. With $\mathcal{X}_{t-1}$, the segmentation process is regarded as finding a index function, $I(\cdot)$, to match each point to one of the object indexes. As one of the possible solutions, the function can be defined to find the closest object around each point.

$$I_t(\mathbf{p}_t) = \arg\min_j \|\mathbf{p}_t - \mathcal{X}_{t-1}^{(j)}\| \tag{1.4}$$

After the segmentation step, the input-point cloud can be grouped according to the object indexes, $\{\mathbf{P}_t^{(1)}, ..., \mathbf{P}_t^{(k)}\}$, each of which becomes a new measurement point-cloud for each object. Now the multiple object tracking problem can be converted to a set of separated single object tracking problems. At each i object, the tracking problem consists of two parallel steps: pose estimation and model update. The pose estimation is to update the previous object pose $\mathbf{X}_{t-1}^{(i)}$ with the new measurement $\mathbf{P}_t^{(i)}$ and internal object model $\mathbf{M}_{t-1}^{(i)}$ by using a Bayesian Filtering technique [LKLK15]. In our PMOT, we used GPU-based particle filter which is similar to the method in [CC13]. The model update step is incrementally construct a model point-cloud $\mathbf{M}_t^{(i)}$ for each object i with the new measurement $\mathbf{P}_t^{(i)}$. In this software, the new model point-cloud is randomly sampled from the old model point-cloud, $\mathbf{M}_{t-1}^{(i)}$, and new measurement point-cloud, $\mathbf{P}_t^{(i)}$ with a fixed ratio parameter, which can be defined by a user. The updated object models, $\mathcal{X}_t$, are then feedback to the both segmentation and tracking steps in the next time step.

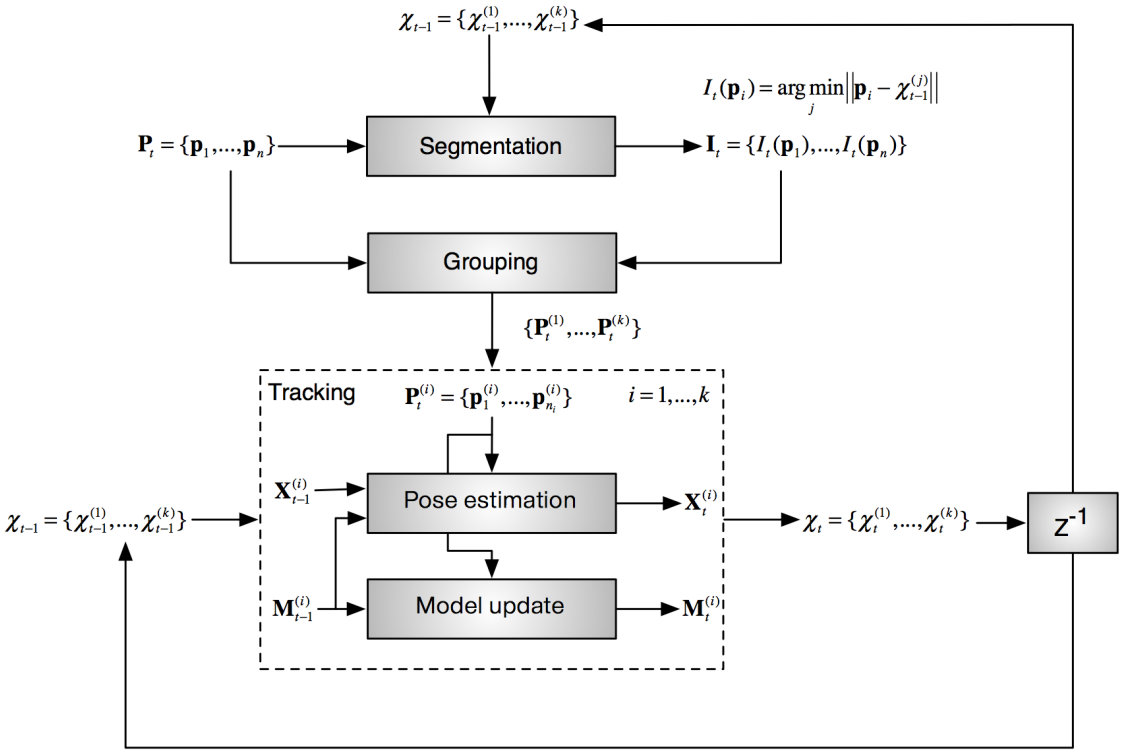


Figure 1.4: A conceptual block diagram of SMOST

Note that this is a conceptual diagram to represent an overview of data flows and main process blocks, which is not exactly same as source codes of PMOT. More details can be found in our research papers [LKLK15, KLK14b, KLK14a, KLK13b, KK13, KLK13a].
As outputs of PMOT, all information of point-level object indexes $\mathbf{I}_t$ and object-level state data $\mathcal{X}_t$ can be provided to the user by following the remaining chapters.

Chapter 2

Installation

This chapter explains how to install PMOT software and compile it on your computer. Because it needs some required 3rd party dependent libraries, please follow carefully step by step.

2.1 System requirements

In order to guarantee the performance reported in our papers, a system similar to or better than following specifications is required.

- CPU: Intel Core i7-4790K CPU @ 4.00GHz × 8
- Memory: 16GByte
- GPU: GeForce GTX 660
- OS: Ubuntu 12.04 (precise) 64-bit

2.2 Dependent libraries

In order to successfully compile PMOT, the following 3rd party software or libraries are required.

2.2.1 Robot Operating System (ROS)

PMOT is running as a standalone node on ROS, at least ROS Hydro. It should be sure that ROS is successfully installed on your system and operating with no problem.
To install ROS Hydro, do the following in a Shell[6]:

[6]More details can be found in http://wiki.ros.org/hydro/Installation/Ubuntu

```
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key
sudo apt-key add ros.key
sudo apt-get update
sudo apt-get install ros-hydro-desktop-full
```

After successful installation, you should make the ROS setup file be called once you open a new shell.
Open `.bashrc` file:

```
cd ~/
gedit .bashrc
```

Add this line at the bottom line of the .bashrc file:

```
source /opt/ros/hydro/setup.bash
```

2.2.2 CUDA

PMOT requires GPU processing, which is implemented by using CUDA library. Go to the CUDA download page and install the recent version of CUDA (higher than ver 6.5)[7]: https://developer.nvidia.com/cuda-downloads

2.2.3 Point cloud library (PCL)

One of the main dependent library of PMOT is the point cloud library (PCL)[8]. The current version of PMOT is using experimental version of PCL 1.8. Unfortunately the official release version is not available on the current time (13.03.2015), but you can download and compile it from source code as follows[9]:

```
git clone https://github.com/PointCloudLibrary/pcl pcl-trunk
cd pcl-trunk && mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
sudo make install
```

2.2.4 Qt4

Qt[9] is one of the most widely used and well structured cross-platform library especially for Graphic User Interface (GUI) system. Although PMOT does not have GUI system inside, another necessary ROS node that we privide, `camcalib`, should be built with Qt4. It will be explained in the next section.

[7]https://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/#ubuntu-installation
[8]http://pointclouds.org/
[9]http://pointclouds.org/downloads/source.html
[9]http://qt-project.org/

Ubuntu 12.04 originally includes Qt4 as a default system library. If you have it then skip this step.

```
sudo apt-get install qt4-dev-tools
```

2.2.5 VXL

VXL is C++ libraries for computer vision research and implementation[10], which includes libraries for numerics (vnl), imaging(vil), vgl(geometry), and so on. In PMOT, some code is written with vnl so that it could easily solve an optimization problem.
First download vxl-1.17 sorce code here: http://vxl.sourceforge.net/#download. Unzip the source file in your any project folder (let's call it `$VXLSRC`), then follows the next steps to install it using cmake[11].

```
cd $VXLSRC
mkdir build
cd build
ccmake ..
make
sudo make install
```

Note that, when you face to some errors to install whole packages in VXL, just carefully try to install vnl library successfully.

2.2.6 Lemon

Lemon[12] is one of the important libraries to implement graph-based algorithms in PMOT. Here is how to get and install it on your system.
Download lemon-1.3.1 source code here: http://lemon.cs.elte.hu/trac/lemon/wiki/Downloads. Installation steps[13] are almost similar to them of vxl above. Let's call you download and unzip source files in the project folder named `$LEMONSRC`). Then follow this.

```
cd $LEMONSRC
mkdir build
cd build
ccmake ..
make
sudo make install
```

[10] http://vxl.sourceforge.net/
[11] http://vxl.sourceforge.net/releases/install-release.html
[12] http://lemon.cs.elte.hu/trac/lemon
[13] http://lemon.cs.elte.hu/trac/lemon/wiki/InstallLinux

2.3 Compiling and building PMOT

After installing all dependencies as introduced above, now you are ready to compile and build PMOT software. Additionally in order to get point-cloud data from an RGB-D camera, the second section will introduce our camera calibration program (camcalib) to convert sensor data from the camera coordinate to the world coordinate.
When you get PMOT source code in our webpage[14], unzip the files in your project folder, called `$PMOTSRC` here. The subfolders in `$PMOTSRC` are listed as follows.

- `/pmot`: PMOT source files.
- `/camcalib`: Camera calibration (camcalib) source files.
- `/doc`: Latex source files of this document and class description document generated by doxygen.

Make sure that your project folder is included in the `ROS_PACKAGE_PATH` which is defined in the ubuntu environment variables. If not, run the following code in the current shell.

```
export ROS_PACKAGE_PATH=$PMOTSRC:$ROS_PACKAGE_PATH
```

The above command is valid only in the current shell. If you like to apply the setting every time you open a new shell, open `.bashrc` file as follows and add the command at the bottom line.

```
cd ~/
gedit .bashrc
```

Now you are ready to compile PMOT and camcalib. In order for PMOT source code to be compiled on cross platform policy (write on one system and compile on multiple systems.), it was written be using CMake[15]. You can use any CMake-available tools for compiling and building PMOT. Here, I recommend QtCreator and followings are how to install and compile PMOT with it.
To install QtCreator on ubuntu, run the command on the shell.

```
sudo apt-get install qtcreator
```

After installed, run QtCreator on a new shell that already includes ROS-related environment setup.

```
qtcreator
```

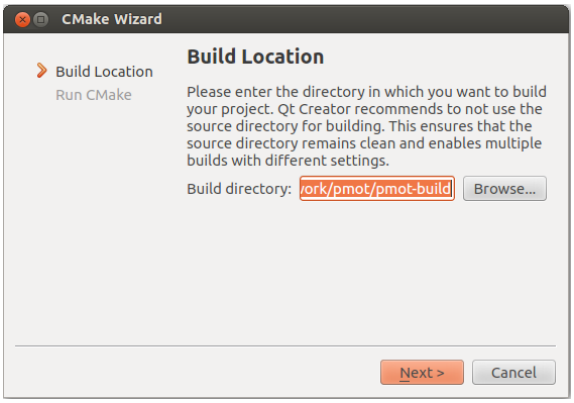


Figure 2.1: CMake Wizard

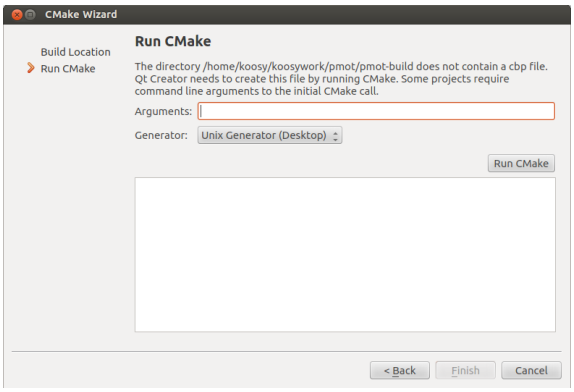


Figure 2.2: Run CMake

2.3.1 PMOT

To open PMOT project, follow the steps in QtCreator.

1. Go to the top menu and press `File` → `Open File or Project`.

2. Select `CMakeLists.txt` in the folder `$PMOTSRC/pmot`. Then, you will see a pop up window named CMake Wizard as Fig. 2.1.

3. Select the `build directory` where you want to make auxiliary files during the building process. These files are not necessary after building step so you can select any folder for this and delete it afterwards. I used `$PMOTSRC/pmot-build` for this example.

4. Click `Next` button. Then, you will see the window to run cmake as shown in Fig. 2.2

[14]http://www.hri.ei.tum.de/pmot
[15]http://www.cmake.org/

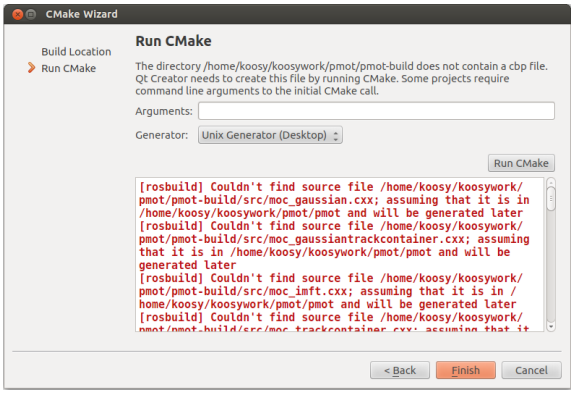


Figure 2.3: Finish CMake

5. Click `Run CMake` button.

6. If your project is successfully built by CMake, you can see the `Finish` button is turn to red. Then press the `Finish` button.

After finishing CMake, PMOT project is loaded in your QtCreator. You can check which files are included in your project in the left panel as shown in Fig. 2.4.

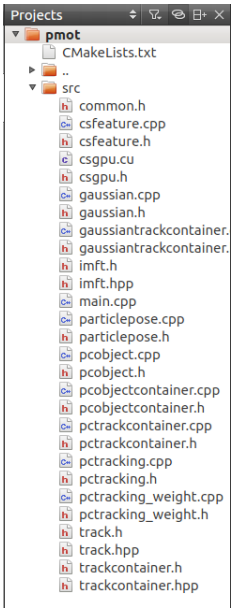


Figure 2.4: PMOT include files

Before compiling PMOT, let's check the content in CMakeLists.txt which describes all necessary settings for PMOT. Click CMakeLists.txt file on the left panel, then you can see the file in the main panel as follows.

```
cmake_minimum_required(VERSION 2.4.6)
include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
rosbuild_init()

set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)

## Include files
set( sources
src/main.cpp
src/pctracking.cpp
src/pctracking_weight.cpp
src/particlepose.cpp
src/csfeature.cpp

src/pcobject.cpp
src/pcobjectcontainer.cpp
src/track.hpp
src/trackcontainer.hpp
src/pctrackcontainer.cpp

src/imft.hpp
src/gaussian.cpp
src/gaussiantrackcontainer.cpp
)

set( headers
src/pctracking.h
src/pctracking_weight.h
src/particlepose.h
src/csfeature.h
src/csgpu.h

src/pcobject.h
src/pcobjectcontainer.h
src/track.h
src/trackcontainer.h
src/pctrackcontainer.h
src/imft.h
src/gaussian.h
src/gaussiantrackcontainer.h
)
```

```
## CUDA
FIND_PACKAGE(CUDA REQUIRED)
INCLUDE(FindCUDA)
CUDA_ADD_LIBRARY (csgpulib src/csgpu.cu src/csgpu.h)
CUDA_ADD_EXECUTABLE(pmot src/csgpu.cu)

## VXL
include_directories(
/usr/local/include/vxl/vcl
/usr/local/include/vxl/core
/usr/local/include/vxl/core/vnl/algo
/usr/local/include/vxl/v3p/netlib
)
FIND_PACKAGE(VXL)

## PCL 1.8
find_package(PCL 1.8 REQUIRED)
include_directories(BEFORE ${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})
if (NOT PCL_FOUND)
MESSAGE(FATAL_ERROR "PCL not found.\n")
endif (NOT PCL_FOUND)

## Qt
find_package(Qt4)
set(QT_USE_QTOPENGL TRUE)
include(${QT_USE_FILE})

## ROS build
rosbuild_add_executable(pmot ${sources} ${headers})
target_link_libraries(pmot vnl_algo vnl vcl QtGui ${PCL_LIBRARIES})
SET_TARGET_PROPERTIES(pmot PROPERTIES LINK_FLAGS -L${PCL_LIBRARY_DIRS})
```

Let's look at CMakeLists.txt inside by breaking down to the following separated parts.
The first part is a general step to use `rosbuild` tool in the project.

```
cmake_minimum_required(VERSION 2.4.6)
include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)
rosbuild_init()

set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
```

Next part defines C++ source and header files that are included in the project. For professional users, if you want to add another files in the project, please add the file names here.

```
## Includes
set( sources
src/main.cpp
src/pctracking.cpp
src/pctracking_weight.cpp
src/particlepose.cpp
src/csfeature.cpp

src/pcobject.cpp
src/pcobjectcontainer.cpp
src/track.hpp
src/trackcontainer.hpp
src/pctrackcontainer.cpp

src/imft.hpp
src/gaussian.cpp
src/gaussiantrackcontainer.cpp
)

set( headers
src/pctracking.h
src/pctracking_weight.h
src/particlepose.h
src/csfeature.h
src/csgpu.h

src/pcobject.h
src/pcobjectcontainer.h
src/track.h
src/trackcontainer.h
src/pctrackcontainer.h
src/imft.h
src/gaussian.h
src/gaussiantrackcontainer.h
)
```

This looks for the installed CUDA libraries and includes required header files.

```
FIND_PACKAGE(CUDA REQUIRED)
INCLUDE(FindCUDA)
```

These two lines are important to build GPU-related source codes in PMOT, which are `csgpu.cu` and `csgpu.h`. As a result of this setting, a new library file (`libcsgpulib.so`) will be generated in `$PMOTSRC\pmot\lib`.

```
CUDA_ADD_LIBRARY (csgpulib src/csgpu.cu src/csgpu.h)
CUDA_ADD_EXECUTABLE(pmot src/csgpu.cu)
```

Next three parts are for including other dependencies, VXL, PCL, and Qt, which are already installed in your system as introduced in the section 2.2.

```
## VXL
include_directories(
/usr/local/include/vxl/vcl
/usr/local/include/vxl/core
/usr/local/include/vxl/core/vnl/algo
/usr/local/include/vxl/v3p/netlib
)
FIND_PACKAGE(VXL)

## PCL 1.8
find_package(PCL 1.8 REQUIRED)
include_directories(BEFORE ${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})
if (NOT PCL_FOUND)
MESSAGE(FATAL_ERROR "PCL not found.\n")
endif (NOT PCL_FOUND)

## Qt
find_package(Qt4)
set(QT_USE_QTOPENGL TRUE)
include(${QT_USE_FILE})
```

The final part is to link all files and libraries to build ros node with the `rosbuild` tool.

```
## ROS build
rosbuild_add_executable(pmot ${sources} ${headers})
target_link_libraries(pmot vnl_algo vnl vcl QtGui ${PCL_LIBRARIES})
SET_TARGET_PROPERTIES(pmot PROPERTIES LINK_FLAGS -L${PCL_LIBRARY_DIRS})
```

To compile and build PMOT, press `crtl+b` in QtCreator. If it is successfully built it will show no errors, and the binary file, `pmot` will be found in `$PMOTSRC/pmot/bin`.

2.3.2 Camera calibration (camcalib)

To open camcalib project in QtCreator, follow the steps similar those of PMOT.

1. Go to the top menu and press `File` → `Open File or Project`.
2. Select `CMakeLists.txt` in the folder `$PMOTSRC/camcalib`.
3. Select the `build directory` where you want to make auxiliary files during the building process. I used `$PMOTSRC/camclib-build` for this example.
4. Click `Next` button.
5. Click `Run CMake` button. If your project is successfully built by CMake, you can see the `Finish` button is turn to red.
6. Press the `Finish` button.

Similar to the above PMOT, let's look at CMakeLists.txt of camcalib for better understanding of the source code structure. Here I will explain the only different part from the CMakeLists.txt file of PMOT.
The `##include` part shows C++ source and header files in camcalib. Note that this also needs to include `${CMAKE_CURRENT_BINARY_DIR}` to load the generated `ui_*` files in the build directory, which means that `$PMOTSRC/camclib-build` should not be deleted after building project.

```
## include
set( sources
src/calibrationOneCam.cpp
src/mainwindow.cpp
src/dialogcalibration.cpp
src/ckinectcalibration.cpp
src/ckinectthread.cpp
src/ckinecttfpublisher.cpp
)

set( headers
src/mainwindow.h
src/dialogcalibration.h
src/ckinectcalibration.h
src/ckinectthread.h
src/ckinecttfpublisher.h
)

include_directories(
${CMAKE_CURRENT_BINARY_DIR}
)
```

Same as PMOT, compile and build camcalib project by pressing `crtl+b`. The binary file `camcalib` will be generated in `$PMOTSRC/camcalib/bin`.

Chapter 3

Program Execution

After PMOT and camcalib are successfully compiled and built, now it is ready to execute those programs to get the results. This chapter will explain the execution steps first camcalib and second PMOT, because camcalib deals with pre-process of the sensor data to provide input data for PMOT.

3.1 Camera calibration (camcalib)

Our camera calibration probram (camcalib) changes sensor data nothing but to different coordinate system. Fig. 3.1 shows the transformed input point-cloud that is measured from `camera_link` frame to `origin` on the table. At each time step, camcalib transforms the input point-cloud data according to the predefined relative pose of the camera from the origin. Because it is not easy to find the relative pose values by users, camcalib provides GUI tool to adjust the pose values interactively and dynamically with ROS 3D visualization tool (RVIZ). In addition to the manual calibration, camcalib provides auto calibration with an arbitrary black-white checker board, specification of which can be defined.

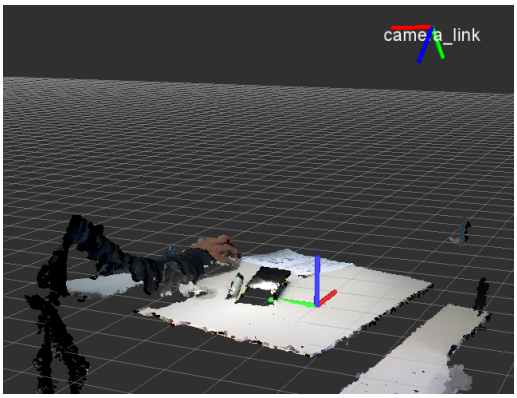


Figure 3.1: Camera calibration coordinates

3.1.1 Setting parameters

One of the benefit of using ROS is that it allows a parameter server to save and load several parameter variables outside of ROS nodes and dynamically share them for all nodes. Here, we use the server to define the relative pose so that transformation messages (TF) can be dynamically generated as a user changes its parameters in GUI.
There are two parameter files in `$PMOTSRC/camcalib`: `param_camera.yaml` and `param_checkboard.yaml`. As the names indicate, the first one defines the current camera pose from origin, and the second stores the size of checker board that will be used for auto calibration. The details of the use of those values will be explained in the next section.
Let's look at the first parameter `param_camera.yaml`.

```
/dhri/calibrationOneCam/camera/x: 0.593
/dhri/calibrationOneCam/camera/y: -0.041
/dhri/calibrationOneCam/camera/z: 0.584
/dhri/calibrationOneCam/camera/yaw: 82.15
/dhri/calibrationOneCam/camera/pitch: 1.12
/dhri/calibrationOneCam/camera/roll: -136.68
```

The numbers are examples, but you can see that there are six float-typed parameters to define the camera pose: x,y,z,yaw,pitch, and roll. The file can be changed not only by manual editing but also by the camcalib program. You can check how the values are changed after running camcalib.
The second file `param_checkboard.yaml` includes following parameters.

```
/dhri/calibrationOneCam/checkboard/nW: 8
/dhri/calibrationOneCam/checkboard/nH: 6
/dhri/calibrationOneCam/checkboard/size: 0.1083
```

`nW` means number of checkers in the width direction while `nH` indicates number of checkers in the height direction. `size` defines the size of each side of the checker in m. Fig. 3.2 shows an example of usable 5×5 checker pattern.

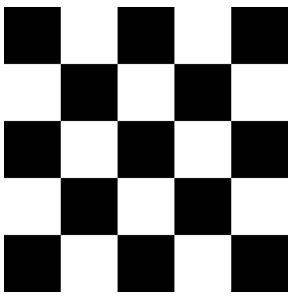


Figure 3.2: An example of usable checker pattern

3.1.2 Execution

Before running camcalib, check that `openni2` is already installed in your ROS. If not, run these two lines to install it.

```
sudo apt-get install ros-hydro-openni2-camera
sudo apt-get install ros-hydro-openni2-launch
```

camcalib needs to run a set of openni2 existing nodes and several commands to set parameters that are introduced above. Another useful ros tool, `roslaunch`, enables to run a set of ros packages and parameters only with a single command.
Here, to run camcalib, you only need to run the following single command at any location on your shell.

```
roslaunch camcalib.launch
```

Following list shows what the launch file automatically executes before runnning camcalib (only important parts are listed).

- rgbd_launch/manager.launch
- openni2_laundh/device.launch
- rgbd_launch/processing.launch
- rgbd_launch/kinect_frames.launch
- param_camera.yaml
- param_checkboard.yaml

When you launch camcalib, you will see a GUI window as shown in Fig. 3.3.
Each value in `camera parameters` is loaded from `param_camera.yaml` file that are already loaded in the ros parameter server.
To show the camera and origin coordinates with the sensor point-cloud data, open a new RVIZ in a new shell.

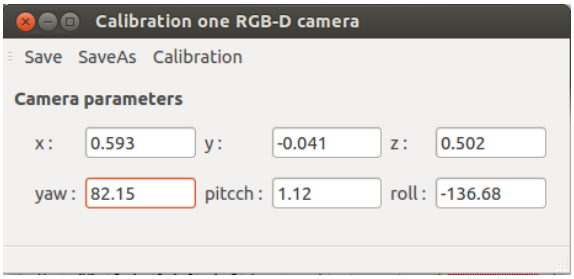


Figure 3.3: The first GUI of camcalib

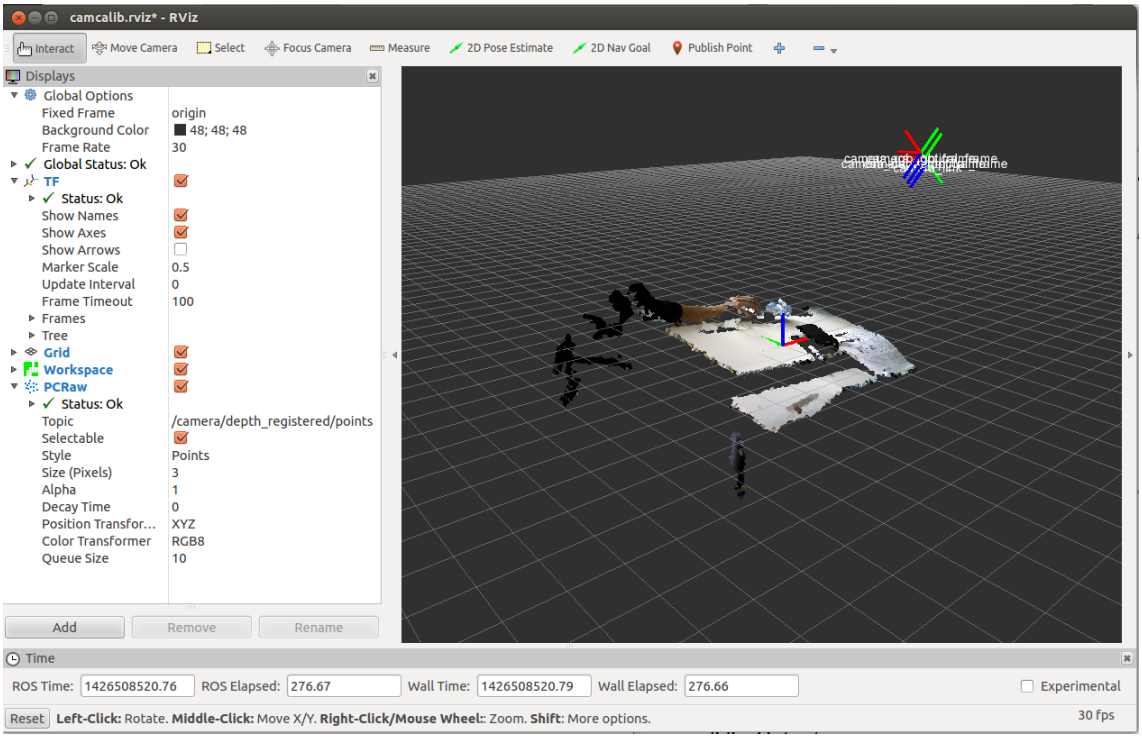


Figure 3.4: Transformed point-cloud and coordinates displayed in RVIZ

```
rosrun rviz rviz
```

To automatically configure RVIZ environment for camcalb, open a `camcalib.rviz` file in `$PMOTSRC/camcalib` (`File` → `Open Config`.).
As shown in Fig. 3.4, you can see the transformed point-cloud is displayed in RVIZ with its two coordinate systems (camera_* and origin). Note that the first global option `Fixed Frame` is assigned with `origin`.
Another important setting you have to notice is the name of point-cloud topic, `/camera/depth_registered/points`, which is the same as the name of raw point-cloud data before transformation. This fact says that camcalib does not produce another new point-cloud data after transformation. Rather than that, it only produces transformation (TF) message between `origin` and `camera` so that subscribers can use the data as additional information to the original point-cloud. In RVIZ, it is clearly shown that you can see original measured point-cloud by changing the fixed frame name to one of the camera frames.
There are two ways to calibrate camera poses: auto calibration with a checker board, and manual calibration with RVIZ. When you click `Calibration` button in the window of Fig. 3.3, you can see another pop-up window that has two taps: Auto and Manual. Unfortunately, the Auto calibration is not working in the current version. It will be updated soon.
When you click 'Manual' tap, you will see six horizontal sliding bars as shown in Fig. 3.5. By manipulating the bars, you can change the camera pose relative to the

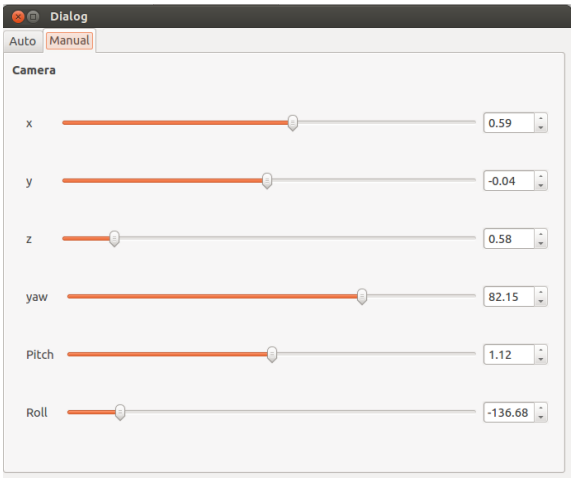


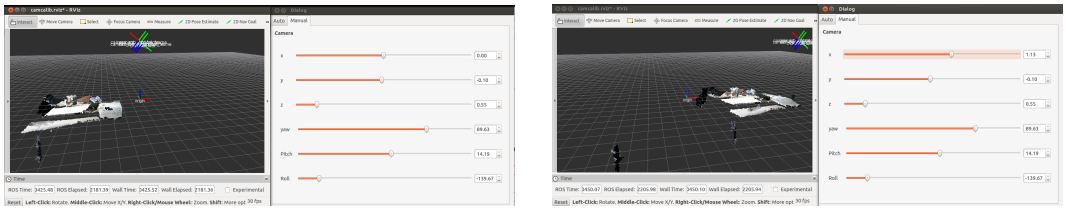
Figure 3.5: Transformed point-cloud and coordinates displayed in RVIZ

origin. Fig. 3.6 shows the two cases of changing x values. You can see the input poin-cloud is moving according to the camera poses. In this interactive way, you can adjust the camera pose exactly and delicately.

After calibrating the camera pose, do not forget to save the pose values in the `param_camera.yaml` file. This task can be done by pressing `save` button on the main window in Fig. 3.3. You can also save the parameters with different names to different locations by using `save as`. It is useful when you need to extract several different camera poses.

3.2 PMOT

With camcalib program as a running ros node, PMOT can process the input point-cloud and produce several outputs. This chapter explains how to get PMOT be ready by setting required parameters, and how to subscribe its outputs.



(a) x = 0 (b) x = 1.13

Figure 3.6: Existing point-cloud object tracking softwares

3.2.1 Setting parameters

As similar to camcalib above, PMOT also has several parameters to be set into the ros parameter server. Open the file `param.yaml` in `$PMOTSRC/pmot`.

```
/dhri/multipleObjectTracking/sub/topic: '/camera/depth_registered/points'
/dhri/multipleObjectTracking/frame/id: '/origin'
/dhri/multipleObjectTracking/pub/points/trackID: '/dhri/points/trackID'
/dhri/multipleObjectTracking/pub/points/filtered: '/dhri/points/filtered'
/dhri/multipleObjectTracking/pub/points/segmented: '/dhri/points/segmented'
/dhri/multipleObjectTracking/pub/markers/trackID: '/dhri/markers/trackID'
/dhri/multipleObjectTracking/pub/markers/gaussians: '/dhri/markers/gaussians'
/dhri/multipleObjectTracking/workspace/topic: '/dhri/workspace'
/dhri/multipleObjectTracking/workspace/x: 0.
/dhri/multipleObjectTracking/workspace/y: 0.
/dhri/multipleObjectTracking/workspace/z: 0.
/dhri/multipleObjectTracking/workspace/width: 0.5
/dhri/multipleObjectTracking/workspace/height: 0.5
/dhri/multipleObjectTracking/workspace/zheight: 1.
/dhri/multipleObjectTracking/param/samplingRatio: 5
/dhri/multipleObjectTracking/param/segmentTolerance: 0.05
/dhri/multipleObjectTracking/param/buffernum: 1
/dhri/multipleObjectTracking/gpu/usegpu:  true
/dhri/multipleObjectTracking/gpu/particlenumber: 500
/dhri/multipleObjectTracking/gpu/gridsize: 0.02
/dhri/multipleObjectTracking/gpu/newratio: 10.0
```

The first parameter is the topic name of the input point-cloud. As you saw in 3.4, the name `/camera/depth_registered/points` is same as the point-cloud topic from the openni2 launch file. The second parameter is the origin frame name that is defined in camcalib program. From the 3rd to 7th parameters define output ros topics that are published by PMOT. Each data will be explained in the next section. From the 8th to 14th parameters are involved in the 3D working space where the input point-cloud is filtered out. From 15th to final parameters are main setting values for PMOT, which are listed below.

- `samplingRatio`: This is downsampling distance in *mm* with which voxelgrid filter in pcl [RC11] is performed to the input point-cloud in the working space.
- segmentTolerance—: In order to segment the filtered point-cloud to individual objects, euclidean distance-based segmentation in [RC11] is performed initially. The parameter is a minimum distance between point-cloud segments that are separated.
- `buffernum`: This is the buffer size to subscribe input point-cloud. If it sets to 1, it means PMOT is operating unsynchronously, which only cares recently

arrived data. If you want to use PMOT as a real-time process without any delay, set this to 1. If you want to use PMOT in off-line but synchronous process, set this to enough value as much as your memory is available, e.g. 1000.

- `usegpu`: This bool-type variable (true/false) indicates whether GPU is available or not in your system. Note that, without GPU, real-time tracking (21fps) reported in [LKLK15] is not available.

- `particlenumber`: This is the number of particles in the particle filter for the pose estimation. Experimentally-obtained optimal number is 100 as reported in [LKLK15].

- `gridsize`: In [LKLK15], we defined Joing Color-Spatial Descriptor to represent an arbitrary object from a point-cloud. As shown in Fig. 3.7, the point-cloud is divided as a set of 3D grids, the `gridsize` defines the size of grid size r in m. The size depends on the target object size, but $2cm$ is the optimal value in our experiments.

- `newratio`: As explained in Fig. 1.4, the internal object model, $\mathbf{M}_t^{(i)}$, is updated from the previous object model, $\mathbf{M}_{t-1}^{(i)}$, and the newly measured point-cloud, $\mathbf{P}_{t-1}^{(i)}$, at every time step. The ratio is the sampling proportion in % of the new point-cloud $\mathbf{P}_{t-1}^{(i)}$. Accordingly, $100 - $ `newratio` is the sampling proportion of $\mathbf{M}_{t-1}^{(i)}$ accordingly. The value is sensitive to the dynamics of the target objects. If the target changes its shape fast, the value should be large. However, the bigger value allows the more errors, which possibly causes robustness problem. In our experiments, under the value of 10 is reasonable in general table-top object manipulation scenarios.

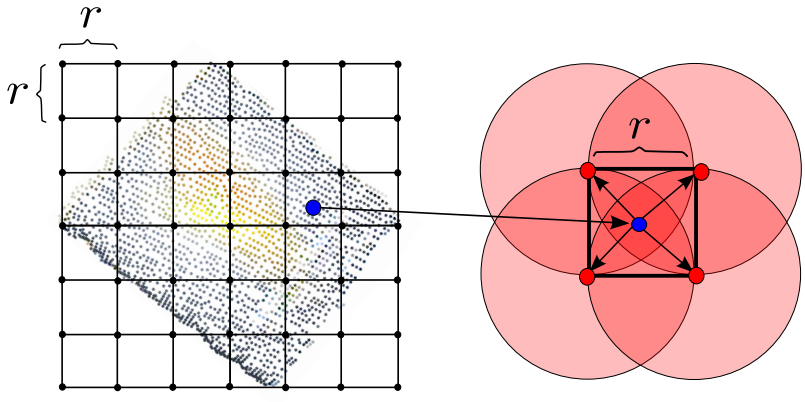


Figure 3.7: A set of grids to represent spatial distribution of an object.

3.2.2 Execution

Before executing PMOT, the parameters above should be loaded first. With the roslaunch tool, the two steps can be easily launched by executing only one command below at any location on your shell.

```
roslaunch pmot pmot.launch
```

The launch file `pmot.launch` contains following scripts.

```
<launch>
  <arg name="param" default="param.yaml" />
  <rosparam command="load" file="$(find pmot)/$(arg param)" />
  <node name="pmot" pkg="pmot" type="pmot" />
</launch>
```

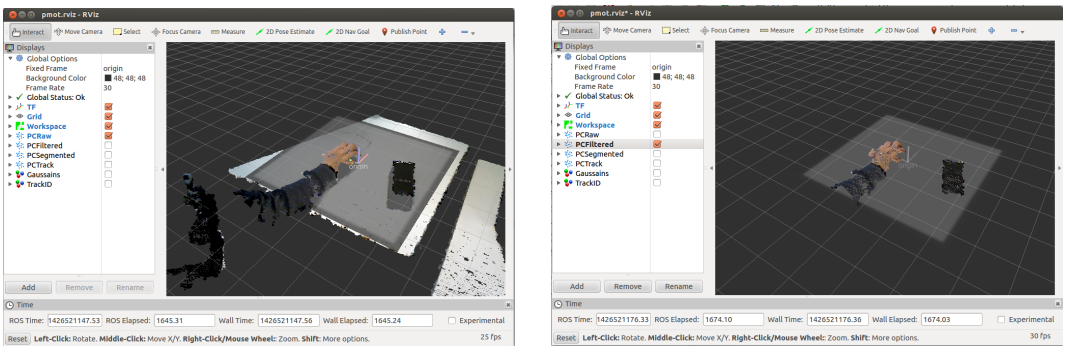
After launching PMOT and there is no error message on your shell, now the outputs of PMOT are ready to be subscribed. RVIZ is also an useful tool to test the PMOT outputs by subscribing them and visualizing the data in 3D space. Furthermore, multiple topics can be subscribed and visualized at the same time, or the other way around: multiple RVIZs can be executed at the same time, each of which subscribes one topic. Either way is good to examine and compare multiple outputs.
To run RVIZ, first follow the command:

```
rosrun rviz rviz
```

To automatically configure RVIZ environment for PMOT, open a `pmot.rviz` file in `$PMOTSRC/pmot` (`File` → `Open Config`.).

Filtered point-cloud

Let's check the first output, filtered point-cloud. The filtering is done by two steps: cutting out unnecessary part (outside of the working space) in the input point-cloud, and downsampling using the voxelgrid filter.



(a) Input point-cloud data (b) Filtered point-cloud in the work space

Figure 3.8: Existing point-cloud object tracking softwares

Fig. 3.8 shows the result of the filtered point-cloud comparing to the input point-cloud. You can see the point-cloud outside of the working space is cut out and the filtered one is less dense than the original one. The topic name of the filtered point-cloud is defined in the parameter file as follows:

```
/dhri/multipleObjectTracking/pub/points/filtered: '/dhri/points/filtered'
```

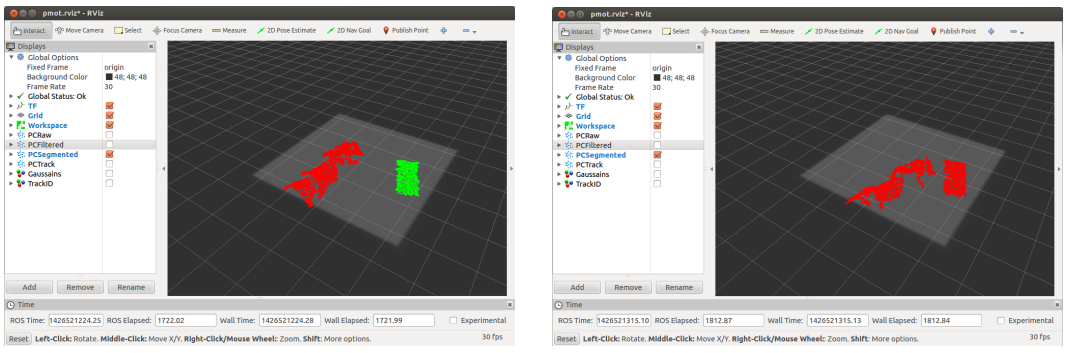
The working space is defined as a 3D box, which is specified in the parameter file.

```
/dhri/multipleObjectTracking/workspace/topic: '/dhri/workspace'
/dhri/multipleObjectTracking/workspace/x: 0.
/dhri/multipleObjectTracking/workspace/y: 0.
/dhri/multipleObjectTracking/workspace/z: 0.
/dhri/multipleObjectTracking/workspace/width: 0.5
/dhri/multipleObjectTracking/workspace/height: 0.5
/dhri/multipleObjectTracking/workspace/zheight: 1.
```

Here in RVIZ, only 2D surface is displayed but you can imagine the actual working space is $1m$ height.

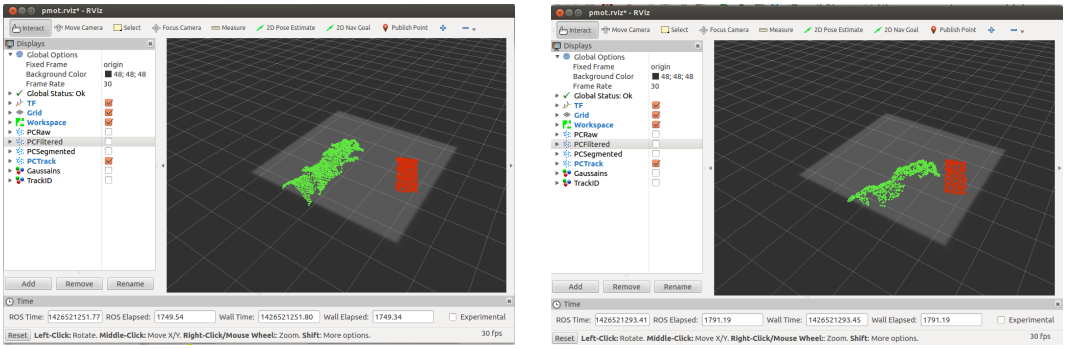
Segmented point-cloud

The second output is the initial segmented point-cloud. This is the result of euclidean clustering to segment each object individual only by location information. Even though this information is not final result of PMOT and looks noisy, it is good reference to show how model-free object segmentation is difficult to solve. Thus, it is useful to show the final results with the initial segmentation results. Fig. 3.9 shows the initial segmentation results in the two cases. When the two objects are separated as shown in Fig. 3.9a, the initial segmentation results in clear separations. However, as shown in Fig. 3.9b, when the two objects are contacted, the euclidean clustering cannot separate the two individual objects.



(a) The separation case (b) The contact case

Figure 3.9: Existing point-cloud object tracking softwares



(a) The separation case (b) The contact case

Figure 3.10: Existing point-cloud object tracking softwares

The topic name of the segmented point-cloud is defined in the parameter file as follows:

```
/dhri/multipleObjectTracking/pub/points/segmented: '/dhri/points/segmented'
```

Object indexed point-cloud

As a final result of PMOT, the object indexed point-cloud, $\mathbf{I}_t$ is generated. As you see in Fig. 3.10, the two individual objects are identified as two different colored point-clouds. Contrary to the segmentation results above, the indexes are robustly and correctly assigned even in the contact case as shown in Fig. 3.10b.
The topic name of the segmented point-cloud is defined in the parameter file as follows:

```
/dhri/multipleObjectTracking/pub/points/trackID: '/dhri/points/trackID'
```

List of Figures

Bibliography

[CC13] Changhyun Choi and Henrik I Christensen. Rgb-d object tracking: A particle filter approach on gpu. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1084–1091. IEEE, 2013.

[KK13] Seongyong Koo and Dong-Soo Kwon. Multiple people tracking from 2d depth data by deterministic spatiotemporal data association. In *RO-MAN, 2013 IEEE*, pages 656–661. IEEE, 2013.

[KLK13a] Seongyong Koo, Dongheui Lee, and Dong-Soo Kwon. Gmm-based 3d object representation and robust tracking in unconstructed dynamic environments. In *2013 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1106–1113. IEEE, 2013.

[KLK13b] Seongyong Koo, Dongheui Lee, and Dong-Soo Kwon. Multiple object tracking using an rgb-d camera by hierarchical spatiotemporal data association. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1113–1118. IEEE, 2013.

[KLK14a] Seongyong Koo, Dongheui Lee, and Dong-Soo Kwon. Incremental object learning and robust tracking of multiple objects from rgb-d point set data. *Journal of Visual Communication and Image Representation*, 25(1):108–121, 2014.

[KLK14b] Seongyong Koo, Dongheui Lee, and Dong-Soo Kwon. Unsupervised object individuation from rgb-d image sequences. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014)*, pages 4450–4457, 2014.

[LKLK15] Shile Li, Seongyong Koo, Dongheui Lee, and Dong-Soo Kwon. Real-time and model-free object tracking using particle filter with joint color-spatial descriptor. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2015), submitted*, 2015.

[PKAW13] Jeremie Papon, Tomas Kulvicius, Eren Erdal Aksoy, and Florentin Worgotter. Point cloud video object segmentation using a persistent super-

voxel world-model. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 3712–3718. IEEE, 2013.

[PW15] Jeremie Papon and Florentin Wörgötter. Spatially stratified correspondence sampling for real-time point cloud tracking. In *Applications of Computer Vision (WACV), 2015 IEEE Conference on (SUBMITTED)*, 2015.

[RC11] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.