Notes

1. The commands in this doc have been tested on Ubuntu 16.04 and 18.04.
2. The calibration is done using vicalib.The procedure follows this documentation.

Contents

Installing librealsense

Please refer to the official documentation, or run the following commands:

```
sudo apt-get install lsb-release software-properties-common
if [[ $(lsb_release -rs) == "16.04" ]]; then
  sudo apt-get install apt-transport-https
fi

sudo apt-key adv --keyserver keys.gnupg.net --recv-key
F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE || sudo apt-key adv --keyserver
hkp://keyserver.ubuntu.com:80 --recv-key F6E65AC044F831AC80A06380C8B3A55A6F3EFCDE

sudo add-apt-repository "deb https://librealsense.intel.com/Debian/apt-repo
$(lsb_release -cs) main" -u

sudo apt-get install librealsense2-dkms
sudo apt-get install librealsense2-utils
sudo apt-get install librealsense2-dev
```

Installing vicalib

Please refer to the official documentation, or run the following commands:

```
# Install dependencies.
sudo apt install \
  wget \
  cmake \
  build-essential \
  libeigen3-dev \
  libgoogle-glog-dev \
  libtinyxml2-dev \
  libopencv-dev \
  libprotobuf-dev \
  protobuf-compiler \
  libglew-dev \
  freeglut3-dev

# Create root directory.
mkdir -p $HOME/calibration

# Install ceres-solver-1.14.0.
cd $HOME/calibration
wget http://ceres-solver.org/ceres-solver-1.14.0.tar.gz
```

```
tar -zxvf ceres-solver-1.14.0.tar.gz
cd ceres-solver-1.14.0 && mkdir -p release && mkdir -p build && cd build
cmake .. \
  -DCMAKE_BUILD_TYPE=RELEASE \
  -DCMAKE_INSTALL_PREFIX=$HOME/calibration/ceres-solver-1.14.0/release
make -j8
make install

# Create ARPG install directory.
cd $HOME/calibration
mkdir -p arpg && cd arpg
mkdir -p releases && mkdir -p builds

# Update environment variables.
export PATH=$HOME/calibration/arpg/releases/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=$HOME/calibration/arpg/releases/lib${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
export LIBRARY_PATH=$HOME/calibration/arpg/releases/lib${LIBRARY_PATH:+:${LIBRARY_PATH}}
export C_INCLUDE_PATH=$HOME/calibration/arpg/releases/include${C_INCLUDE_PATH:+:${C_INCLUDE_PATH}}
export CPLUS_INCLUDE_PATH=$HOME/calibration/arpg/releases/include${CPLUS_INCLUDE_PATH:+:${CPLUS_INCLUDE_PATH}}

# For later use, also add the following lines to $HOME/.bashrc.
#   export PATH=$HOME/calibration/arpg/releases/bin${PATH:+:${PATH}}
#   export LD_LIBRARY_PATH=$HOME/calibration/arpg/releases/lib${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
#   export LIBRARY_PATH=$HOME/calibration/arpg/releases/lib${LIBRARY_PATH:+:${LIBRARY_PATH}}
#   export C_INCLUDE_PATH=$HOME/calibration/arpg/releases/include${C_INCLUDE_PATH:+:${C_INCLUDE_PATH}}
#   export CPLUS_INCLUDE_PATH=$HOME/calibration/arpg/releases/include${CPLUS_INCLUDE_PATH:+:${CPLUS_INCLUDE_PATH}}

# Install Sophus.
cd $HOME/calibration/arpg
git clone git@github.com:arpg/Sophus.git
mkdir -p builds/Sophus && cd builds/Sophus
cmake ../../Sophus \
  -DCMAKE_INSTALL_PREFIX=$HOME/calibration/arpg/releases
```

```
make -j8
make install

# Install Calibu.
cd $HOME/calibration/arpg
git clone git@github.com:arpg/Calibu.git
if [[ $(lsb_release -rs) == "16.04" ]]; then
  sed -i 's/std::cerr\ <<\ doc.ErrorStr()/\/\/\ std::cerr\ <<\ doc.ErrorStr()/g'
Calibu/src/cam/CameraXml.cpp
fi
mkdir -p builds/Calibu && cd builds/Calibu
cmake ../../Calibu \
  -DCMAKE_INSTALL_PREFIX=$HOME/calibration/arpg/releases
make -j8
make install

# Install CVars.
cd $HOME/calibration/arpg
git clone git@github.com:arpg/CVars.git
mkdir -p builds/CVars && cd builds/CVars
cmake ../../CVars \
  -DCMAKE_INSTALL_PREFIX=$HOME/calibration/arpg/releases
make -j8
make install

# Install HAL.
cd $HOME/calibration/arpg
git clone git@github.com:arpg/HAL.git
# If you are using D455, you may need to run the following commands here to avoid
the error "object doesn't support option":
#   sed -i 's/driver->SetExposure/\/\/ driver->SetExposure/g'
HAL/HAL/Camera/Drivers/RealSense2/RealSense2Factory.cpp
#   sed -i 's/driver->SetGain/\/\/ driver->SetGain/g'
HAL/HAL/Camera/Drivers/RealSense2/RealSense2Factory.cpp
mkdir -p builds/HAL && cd builds/HAL
cmake ../../HAL \
  -DCMAKE_INSTALL_PREFIX=$HOME/calibration/arpg/releases
make -j8
make install

# Make sure you see that HAL is building the 'RealSense2' camera driver when
running the cmake command:
#   -- HAL: building 'AutoExposure' camera driver
#   -- HAL: building 'Cleave' abstract camera driver.
#   -- HAL: building 'Convert' abstract camera driver (using libopencv).
#   -- HAL: building 'DC1394' (firewire) camera driver.
#   -- HAL: building 'Debayer' camera driver (using libdc1394).
```

```
#   -- HAL: building 'Deinterlace' camera driver (using libdc1394).
#   -- HAL: building 'FileReader' camera driver.
#   -- HAL: building 'Join' abstract camera driver.
#   -- HAL: building 'OpenCV' camera driver.
#   -- HAL: building 'ProtoReader' camera driver.
#   -- HAL: building 'RealSense2' camera driver.
#   -- HAL: building 'Rectify' abstract camera driver.
#   -- HAL: building 'Split' abstract camera driver.
#   -- HAL: building 'Undistort' camera driver.
#   -- HAL: building 'V4L' camera driver.
# If not, make sure you have installed librealsense.

# Install Pangolin.
cd $HOME/calibration/arpg
git clone git@github.com:arpg/Pangolin.git
mkdir -p builds/Pangolin && cd builds/Pangolin
cmake ../../Pangolin \
  -DCMAKE_INSTALL_PREFIX=$HOME/calibration/arpg/releases
make -j8
make install

# Install vicalib.
cd $HOME/calibration/arpg
git clone https://github.com/arpg/vicalib
if [[ $(lsb_release -rs) == "18.04" ]]; then
  sed -i 's/Eigen::VectorXd
params_(calibu::Rational6Camera<double>::NumParams);/Eigen::VectorXd
params_(10);/g' vicalib/src/vicalib-engine.cc && \
  sed -i 's/Eigen::VectorXd
params_(calibu::KannalaBrandtCamera<double>::NumParams);/Eigen::VectorXd
params_(8);/g' vicalib/src/vicalib-engine.cc && \
  sed -i 's/Eigen::VectorXd
params_(calibu::LinearCamera<double>::NumParams);/Eigen::VectorXd params_(4);/g'
vicalib/src/vicalib-engine.cc
fi
mkdir -p builds/vicalib && cd builds/vicalib
cmake ../../vicalib \
  -DCMAKE_INSTALL_PREFIX=$HOME/calibration/arpg/releases \
  -DCMAKE_PREFIX_PATH=$HOME/calibration/ceres-solver-1.14.0/release/lib/cmake/Ceres
make -j8
make install
```

Building Calibration Pattern

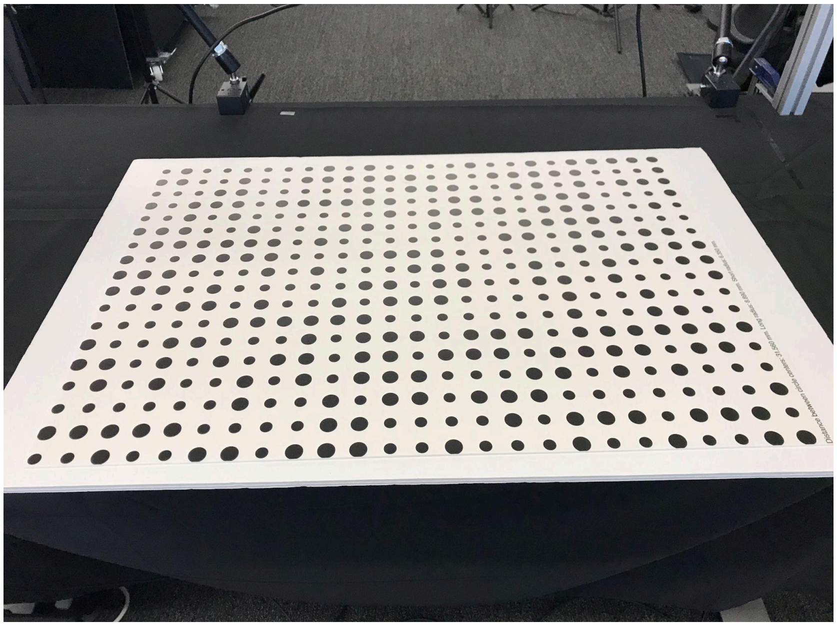
Please refer to the official documentation.

We used a “medium” pattern in all our calibration processes. To create a pattern in pdf, run:

```
$HOME/calibration/arpg/releases/bin/vicalib \
  -grid_preset medium \
  -output_pattern_file pattern-medium.svg
convert \
  -density 300 \
  pattern-medium.svg pattern-medium.pdf
```

You need to print out the pattern and attach it on a planar and rigid surface.

- Make sure you print the calibration pattern **in its actual size**.
- We printed the pattern on a PVC board through FedEx’s poster printing service, and mounted the PVC board on a bigger foam board.



Retrieving Intrinsics

We use vicalib to calibrate the extrinsics between two color cameras. When solving the extrinsics, we fix the intrinsics of each camera. To get the intrinsics of each camera, one option is to first use vicalib to calibrate each camera separately. However, we found that the manufacturer’s calibration for intrinsics is sufficiently good, so we used those directly.

To retrieve the manufacturer's calibration, first install pyrealsense2:

```
pip install pyrealsense2
```

Next, connect a realsense, and run the following Python commands to retrieve the intrinsics, provided the camera's serial ID:

```
import pyrealsense2 as rs
serial = '836212060125' # change this
w = 1280
h = 720
cfg = rs.config()
cfg.enable_device(serial)
cfg.enable_stream(rs.stream.depth, w, h, rs.format.z16, 30)
cfg.enable_stream(rs.stream.color, w, h, rs.format.rgb8, 30)
pipe = rs.pipeline()
selection = pipe.start(cfg)
depth_stream = selection.get_stream(rs.stream.depth).as_video_stream_profile()
color_stream = selection.get_stream(rs.stream.color).as_video_stream_profile()
id = depth_stream.get_intrinsics()
ic = color_stream.get_intrinsics()
e = depth_stream.get_extrinsics_to(color_stream)
print("{:s}".format(serial))
print("depth:")
print("  intrinsics:", id)
print("color")
print("  intrinsics:", ic)
print("extrinsics (depth to color):")
print("  rotation:", e.rotation)
print("  translation:", e.translation)
print("")
```

This will print out the intrinsics of color and depth cameras of the realsense, and also their extrinsics, for example:

```
color
  intrinsics: width: 1280, height: 720, ppx: 622.822, ppy: 372.785, fx: 924.961,
fy: 924.387, model: Brown Conrady, coeffs: [0, 0, 0, 0, 0]
```

Next, create a new file `$HOME/calibration/$ID.xml` (use your serial id) and copy the following lines into it. Make sure to change `serialno` and `params` according to your serial id and the retrieved intrinsics of the color camera.

```
<rig>
    <camera>
        <camera_model name="" index="0" serialno="836212060125"
type="calibu_fu_fv_u0_v0_k1_k2_k3" version="0">
```

```
            <width> 1280 </width>
            <height> 720 </height>
            <!-- Use RDF matrix, [right down forward], to define the coordinate frame convention -->
            <right> [ 1; 0; 0 ] </right>
            <down> [ 0; 1; 0 ] </down>
            <forward> [ 0; 0; 1 ] </forward>
            <!-- Camera parameters ordered as per type name. -->
            <params> [ 924.961; 924.387; 622.822; 372.785; 0.000; 0.000; 0.000 ] </params>
        </camera_model>
        <pose>
            <!-- Camera pose. World from Camera point transfer. 3x4 matrix, in the RDF frame convention defined above -->
            <T_wc> [ 1, 0, 0, 0; 0, 1, 0, 0; 0, 0, 1, 0 ] </T_wc>
        </pose>
    </camera>
</rig>
```

Finally, repeat this step to create the xml files for all your realsense cameras.

Calibrating Extrinsics of Two Cameras

Once you have all the xml files with intrinsics, we can do the extrinsics calibration now.

- In all our calibration processes, we calibrated a pair of realsenses each time (color camera to color camera).
- Note that vicalib allows calibrating more than two cameras at once, but we found that this will incur instability in optimization when the cameras viewpoints differ drastically.
- Since we have 8 cameras, positioned around the table, we do the calibration 8 times by doing a pair each time, i.e. cam1-cam2, cam2-cam3, …, cam8-cam1.

To calibrate a pair of cameras, change the serial ids accordingly and run the following script:

```
export ID0=836212060125 # change this
export ID1=839512060362 # change this
$HOME/calibration/arpg/releases/bin/vicalib \
  -grid_preset medium \
  -frame_skip 4 \
  -num_vicalib_frames 64 \
  -output $HOME/calibration/$ID0-$ID1.xml \
  -cam convert://realsense2:[id0=$ID0,id1=$ID1,size=1280x720,depth=0]// \
  -nocalibrate_intrinsics \
  -model_files $HOME/calibration/$ID0.xml,$HOME/calibration/$ID1.xml
```

This will launch vicalib and automatically capture 1 every 4 frames until capturing 64 frames, and it will automatically run the optimization to solve the extrinsics. The out extrinsics file `$HOME/calibration/$ID0-$ID1.xml` will look something like this:

```
<rig>
    <camera>
        <camera_model name="" index="0" serialno="836212060125"
type="calibu_fu_fv_u0_v0_k1_k2_k3" version="0">
            <width> 1280 </width>
            <height> 720 </height>
            <!-- Use RDF matrix, [right down forward], to define the coordinate
frame convention -->
            <right> [ 1; 0; 0 ] </right>
            <down> [ 0; 1; 0 ] </down>
            <forward> [ 0; 0; 1 ] </forward>
            <!-- Camera parameters ordered as per type name. -->
            <params> [ 924.961; 924.387; 622.822; 372.785; 0; 0; 0 ] </params>
        </camera_model>
        <pose>
            <!-- Camera pose. World from Camera point transfer. 3x4 matrix, in the
RDF frame convention defined above -->
            <T_wc> [ 1, 0, 0, 0; 0, 1, 0, 0; 0, 0, 1, 0 ] </T_wc>
        </pose>
    </camera>
    <camera>
        <camera_model name="" index="1" serialno="932122062010"
type="calibu_fu_fv_u0_v0_k1_k2_k3" version="0">
            <width> 1280 </width>
            <height> 720 </height>
            <!-- Use RDF matrix, [right down forward], to define the coordinate
frame convention -->
            <right> [ 1; 0; 0 ] </right>
            <down> [ 0; 1; 0 ] </down>
            <forward> [ 0; 0; 1 ] </forward>
            <!-- Camera parameters ordered as per type name. -->
            <params> [ 924.374; 923.815; 642.421; 367.242; 0; 0; 0 ] </params>
        </camera_model>
        <pose>
            <!-- Camera pose. World from Camera point transfer. 3x4 matrix, in the
RDF frame convention defined above -->
            <T_wc> [ 0.9943054, 0.09616721, 0.04592077, -0.1839901; -0.04086807,
0.7420496, -0.6690981, 0.5557837; -0.09842079, 0.6634111, 0.741754, 0.2430989 ]
</T_wc>
        </pose>
    </camera>
</rig>
```

The extrinsics is stored in `<T_wc>`. It is represented as a 3x4 transformation matrix, i.e. [R; t].

Here is a video of a calibration run.

We follow the guidelines below for calibration:

- Calibrate a pair of cameras each time.
- Make sure there are always some corners detected in each camera during the capture period.
- For both cameras, cover as much pattern board region as possible.
- For both cameras, observe as many corners on the board as possible.
- For both cameras, keep the angle of the board as less tilted as possible.
- For calibrating color cameras, re-calibrate the pair until the final MSE is below 0.15.

Finally, repeat this step to get the xml files and `<T_wc>` until you can compute a transformation from one camera to every other camera.

Computing Transformations from Master to Others

To verify the calibrated extrinsics, one way is to visualize the combined point cloud by transforming the point cloud from each camera to a canonical coordinate frame. If the extrincs is accurate, then the point clouds should stitch together in a nice way, like in this example.

First, you need to pick one camera and use its coordinate frame as the canonical frame. From now on let's refer to this camera as the "master". The next step is to find the transformation from the master to every other camera, using the extinsics you got previously.

Assuming we have 4 cameras (with serial IDs: `836212060125`, `839512060362`, `840412060917`, and `841412060263`), below is an example of the targeted output:

```
-------
836212060125:
  translation:
    [-0.0356385, -0.5543582,  1.0280061]
  rotation:
    [[-0.8911318, -0.0030943,  0.4537341],
     [ 0.4173740,  0.3866867,  0.8223578],
     [-0.1779976,  0.9222059, -0.3432974]]
  tf: -0.0356385 -0.5543582  1.0280061  2.7035728  0.1789511  1.9271598
-------
839512060362:
  translation:
    [-0.4701657, -0.0288896,  0.7349084]
  rotation:
    [[ 0.2392589, -0.0230820,  0.9706814],
```

```
     [-0.7915438, -0.5836228,  0.1812260],
     [ 0.5623287, -0.8116968, -0.1579074]]
  tf: -0.4701657 -0.0288896  0.7349084 -1.2772595 -0.5971993 -1.7629362
-------
840412060917:
  translation:
    [ 0.0000000,  0.0000000,  0.0000000]
  rotation:
    [[ 1.0000000,  0.0000000,  0.0000000],
     [ 0.0000000,  1.0000000,  0.0000000],
     [ 0.0000000,  0.0000000,  1.0000000]]
  tf:  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000
-------
841412060263:
  translation:
    [ 0.3648854, -0.8780225,  0.6363378]
  rotation:
    [[-0.8711858,  0.3966662, -0.2892943],
     [-0.2764904,  0.0905198,  0.9567441],
     [ 0.4056949,  0.9134890,  0.0308149]]
  tf:  0.3648854 -0.8780225  0.6363378 -2.8342750 -0.4177390  1.5370760
```

Note that the last line ("tf") for each camera prints the transformation in the ROS tf format (x, y, z, yaw, pitch, roll), and this is what we need for each camera.

You need to write a Python script to extract the tfs. In particular, for each camera other than the master (which has identity transformation), you need to do two things:

1. Find the total transformation (in a 4x4 transformation matrix). Recall that in extrinsics calibration we calibrate a pair of cameras each time and repeat in a circle, i.e. cam1-cam2, cam2-cam3, …, cam8-cam1. Let's say the master is cam1 and the target now is cam4, so you are trying to find the transformation of cam1-cam4. This is done by aggregating the pairwise transformation along the path from cam1 to cam4, i.e. cam1-cam2, cam2-cam3, cam3-cam4. So first write a function to read the 3x4 matrices `<T_wc>` from the extrinsics files, and then append the fourth row `[0, 0, 0, 1]` to each. Then multiply the transformation matrices:

```
T14 = T12.dot(T23).dot(T34)
```

   where `T12`, `T23`, `T34` are numpy arrays of shape 4x4. Note that if you need `T32` you can get it by the computing inverse of `T23`.

```
T32 = np.linalg.inv(T23)
```

2. Convert the transformation from a 4x4 matrix to the tf representation:

```
from scipy.spatial.transform import Rotation as Rot

t = T14[:3, 3]
R = T14[:3, :3]
e = Rot.from_dcm(R).as_euler('ZYX').astype(np.float32)

print('  tf: {:10.7f} {:10.7f} {:10.7f} {:10.7f} {:10.7f} {:10.7f}'.format(
    *t, *e))
```

You will need to scipy if you have not installed it:

```
pip install scipy
```

Once you have the tfs for all the cameras, you are now ready to visualize the point clouds using ROS.

Install realsense-ros

We use realsense-ros for visualizing point clouds and recording. To install, please refer to the official documentation, or run the following commands:

First, you need to install the ROS distribution:

```
sudo apt-get install lsb-release curl
if [[ $(lsb_release -rs) == "16.04" ]]; then
  sudo apt-get install gnupg2
fi

sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | apt-key
add -
sudo apt-get update
if [[ $(lsb_release -rs) == "16.04" ]]; then
  sudo apt-get install ros-kinetic-desktop-full
  echo "source /opt/ros/kinetic/setup.bash" >> $HOME/.bashrc
fi
if [[ $(lsb_release -rs) == "18.04" ]]; then
  sudo apt-get install ros-melodic-desktop-full
  echo "source /opt/ros/melodic/setup.bash" >> $HOME/.bashrc
fi
source $HOME/.bashrc
```

Next, you need to install librealsense if you have not.

Finally, install realsense-ros:

```
if [[ $(lsb_release -rs) == "16.04" ]]; then
  sudo apt-get install ros-kinetic-ddynamic-reconfigure
fi
if [[ $(lsb_release -rs) == "18.04" ]]; then
  sudo apt-get install ros-melodic-ddynamic-reconfigure
fi

mkdir -p $HOME/catkin_ws/realsense-ros/src
cd $HOME/catkin_ws/realsense-ros/src

git clone https://github.com/IntelRealSense/realsense-ros.git
cd realsense-ros
git checkout `git tag | sort -V | grep -P "^2.\d+\.\d+" | tail -1`
cd ..

catkin_init_workspace
cd ..
catkin_make clean
catkin_make -DCATKIN_ENABLE_TESTING=False -DCMAKE_BUILD_TYPE=Release
catkin_make install

echo "source $HOME/catkin_ws/realsense-ros/devel/setup.bash" >> $HOME/.bashrc
source $HOME/.bashrc
```

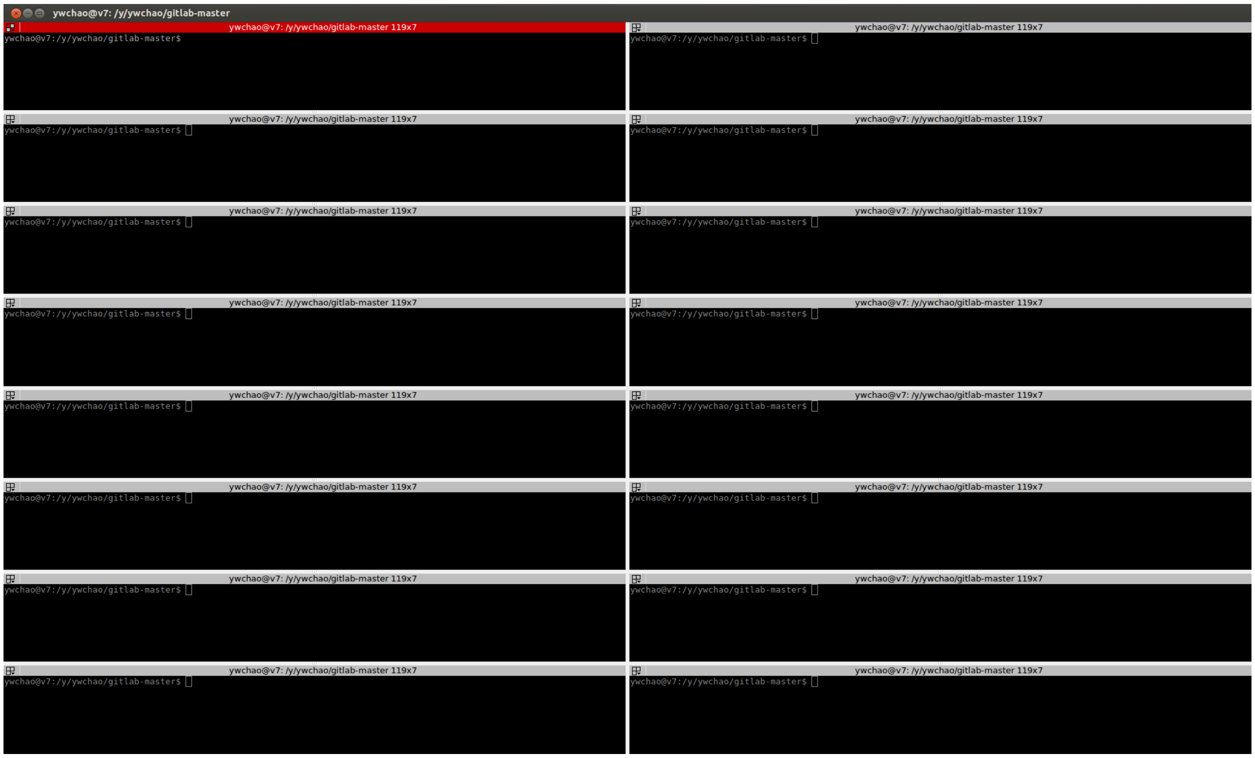
Visualizing Combined Point Cloud

A typical workflow with ROS will require launching multiple commands at once. You will need to open multiple terminals for that. We use a tool called terminator to manage multiple terminal windows. You can install it with:

```
sudo apt-get install terminator

# Installing terminator will change the default terminal. To change it back:
sudo update-alternatives --config x-terminal-emulator
# Select /usr/bin/gnome-terminal.wrapper
```

Once launched, you can create multiple terminal windows like below:



```
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
ywchao@v7:/y/ywchao/gitlab-master$
```

For now let’s assume you have 4 cameras. Extension to more than 4 cameras is straightforward. First, set the environment variables in each window for serial ids and master following the example below (after changing to your ids):

```
export ID1=836212060125 # change this
export ID2=839512060362 # change this
export ID3=840412060917 # change this
export ID4=841412060263 # change this
export MASTER=840412060917 # change this
```

Now, in **window 1**, launch roscore:

```
roscore
```

Next, in **window 2 to 5**, start the camera nodes for the realsenses.

window 2

```
roslaunch realsense2_camera rs_camera.launch serial_no:=$ID1 camera:=$ID1
depth_width:=640 depth_height:=480 color_width:=640 color_height:=480 depth_fps:=30
color_fps:=30 align_depth:=true
```

window 3

```
roslaunch realsense2_camera rs_camera.launch serial_no:=$ID2 camera:=$ID2
depth_width:=640 depth_height:=480 color_width:=640 color_height:=480 depth_fps:=30
color_fps:=30 align_depth:=true
```

window 4

```
roslaunch realsense2_camera rs_camera.launch serial_no:=$ID3 camera:=$ID3
depth_width:=640 depth_height:=480 color_width:=640 color_height:=480 depth_fps:=30
color_fps:=30 align_depth:=true
```

window 5

```
roslaunch realsense2_camera rs_camera.launch serial_no:=$ID4 camera:=$ID4
depth_width:=640 depth_height:=480 color_width:=640 color_height:=480 depth_fps:=30
color_fps:=30 align_depth:=true
```

Finally, you need to set the transformations between the camera frame of master to each camera using tf's static_trasnform_publisher. Below shows an example of the transformations obtained from this section:

```
-------
836212060125:
  tf: -0.0356385 -0.5543582  1.0280061  2.7035728  0.1789511  1.9271598
-------
839512060362:
  tf: -0.4701657 -0.0288896  0.7349084 -1.2772595 -0.5971993 -1.7629362
-------
840412060917:
  tf:  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000
-------
841412060263:
  tf:  0.3648854 -0.8780225  0.6363378 -2.8342750 -0.4177390  1.5370760
```

Now, in **window 6 to 9**, enter the following commands (after changing the transformations to yours):

window 6

```
rosrun tf static_transform_publisher -0.0356385 -0.5543582  1.0280061  2.7035728
0.1789511  1.9271598 ${MASTER}_color_optical_frame ${ID1}_color_optical_frame 30
```

window 7

```
rosrun tf static_transform_publisher -0.4701657 -0.0288896  0.7349084 -1.2772595
-0.5971993 -1.7629362 ${MASTER}_color_optical_frame ${ID2}_color_optical_frame 30
```

window 8

```
rosrun tf static_transform_publisher  0.0000000  0.0000000  0.0000000  0.0000000
0.0000000  0.0000000 ${MASTER}_color_optical_frame world 30
```

window 9

```
rosrun tf static_transform_publisher  0.3648854 -0.8780225  0.6363378 -2.8342750
-0.4177390  1.5370760 ${MASTER}_color_optical_frame ${ID4}_color_optical_frame 30
```

After launching all the above commands, open rviz:

```
rviz
```

and make the following changes in the Displays panel:

1. Change `Global Options->Fixed Frame` to `world`.
2. Click the `Add` button and add `rviz->DepthCloud`.
3. Expand `DepthCloud` in the Display panel.
4. Change `Depth Map Topic` to `/$ID1/aligned_depth_to_color/image_raw`.
5. Change `Color Image Topic` to `/$ID1/color/image_raw`.
6. Repeat 2.-5. for $ID2, $ID3, and $ID4.

You should now see the combined point cloud in the 3D view panel, as shown previously in this example.

As mentioned previously, the point clouds from each camera should stitch together in a meaningful way. If not, then there should be something wrong in the calibration pipeline.

Recording Data from All Cameras with rosbag

Besides calibration, we also show how you can record RGB-D data simultaneously from all the cameras. This can be done easily with ROS's rosbag.

First, make sure all the ROS processes are killed and start a new terminator.

Again, let's assume you have 4 cameras now. Extension to more than 4 cameras is again straightforward. First, set the environment variables in each window for serial ids and master following the example below (after changing to your ids):

```
export ID1=836212060125 # change this
export ID2=839512060362 # change this
export ID3=840412060917 # change this
```

```
export ID4=841412060263 # change this
```

Now, in **window 1**, launch roscore:

```
roscore
```

In our previous captures, we disabled auto exposure for rgb cameras. This can be done by entering the following commands in one window:

```
rosparam set /$ID1/rgb_camera/enable_auto_exposure false
rosparam set /$ID2/rgb_camera/enable_auto_exposure false
rosparam set /$ID3/rgb_camera/enable_auto_exposure false
rosparam set /$ID4/rgb_camera/enable_auto_exposure false
```

Now launch the 4 cameras in **window 2 to 5**. This is the same as in visualizing point clouds.

window 2

```
roslaunch realsense2_camera rs_camera.launch serial_no:=$ID1 camera:=$ID1
depth_width:=640 depth_height:=480 color_width:=640 color_height:=480 depth_fps:=30
color_fps:=30 align_depth:=true
```

window 3

```
roslaunch realsense2_camera rs_camera.launch serial_no:=$ID2 camera:=$ID2
depth_width:=640 depth_height:=480 color_width:=640 color_height:=480 depth_fps:=30
color_fps:=30 align_depth:=true
```

window 4

```
roslaunch realsense2_camera rs_camera.launch serial_no:=$ID3 camera:=$ID3
depth_width:=640 depth_height:=480 color_width:=640 color_height:=480 depth_fps:=30
color_fps:=30 align_depth:=true
```

window 5

```
roslaunch realsense2_camera rs_camera.launch serial_no:=$ID4 camera:=$ID4
depth_width:=640 depth_height:=480 color_width:=640 color_height:=480 depth_fps:=30
color_fps:=30 align_depth:=true
```

It is a good practice to check the frame rate of the color and depth streams for each camera before recording. This can be done by the example commands below:

color

```
rostopic hz /$ID1/color/image_raw
```

depth

```
rostopic hz /$ID1/aligned_depth_to_color/image_raw
```

Since we launched the cameras with 30 FPS, you should see the average rate close to 30. For example:

```
subscribed to [/836212060125/color/image_raw]
average rate: 29.661
	min: 0.030s max: 0.039s std dev: 0.00146s window: 30
average rate: 29.703
	min: 0.027s max: 0.039s std dev: 0.00197s window: 59
average rate: 29.724
	min: 0.027s max: 0.039s std dev: 0.00181s window: 89
```

If the frame rate is much lower than expected, then there might be something wrong with the cameras, or you are having bandwidth problems when streaming from all the cameras.

After verifying the frame rate, we can now do the recording. First, create a new directory for storing the recorded data:

```
RECORD_DIR=$HOME/record
mkdir -p $RECORD_DIR
```

You can then record with the following command:

```
sleep 0.5 && rosbag record \
  -a \
  -x "(.*)compressed(.*)|(.*)theora(.*)" \
  --duration=3 \
  -O $RECORD_DIR/$(date +'%Y%m%d_%H%M%S').bag
```

This will start a recording after a 0.5 second wait, and record for a duration of 3 seconds. The recorded data will be saved to a bag file named by the current datetime.

You can print the contents of a bag file with the following command:

```
rosbag info $BAG_FILE
```

You should see the color and depth topics from different cameras, for example:

```
topics:      /836212060125/aligned_depth_to_color/camera_info
82 msgs    : sensor_msgs/CameraInfo
             /836212060125/aligned_depth_to_color/image_raw
74 msgs    : sensor_msgs/Image
             /836212060125/color/camera_info
82 msgs    : sensor_msgs/CameraInfo
             /836212060125/color/image_raw
74 msgs    : sensor_msgs/Image
             /836212060125/depth/camera_info
78 msgs    : sensor_msgs/CameraInfo
             /836212060125/depth/image_rect_raw
74 msgs    : sensor_msgs/Image
.
.
.
             /840412060917/aligned_depth_to_color/camera_info
81 msgs    : sensor_msgs/CameraInfo
             /840412060917/aligned_depth_to_color/image_raw
74 msgs    : sensor_msgs/Image
             /840412060917/color/camera_info
82 msgs    : sensor_msgs/CameraInfo
             /840412060917/color/image_raw
79 msgs    : sensor_msgs/Image
             /840412060917/depth/camera_info
79 msgs    : sensor_msgs/CameraInfo
             /840412060917/depth/image_rect_raw
75 msgs    : sensor_msgs/Image
.
.
.
```

For a 3 second recording with 30 FPS, the number of messages for color and depth topics is typically between 70 and 85.

Extracting Color and Depth Images from rosbag

Once you have recorded the data into a bag file, the next step is to extract the color and depth images and save them into JPG and PNG. We also want to synchronize the images from different cameras. All of this can be done with ROS's Python API. Since the ROS distributions for Ubuntu 16.04 and 18.04 only support Python 2, you should use **Python 2** to run the python scripts below.

First, use rosbag's Python API to read a bag file named `bag_file`:

```
import rosbag
```

```
bag = rosbag.Bag(bag_file)
```

Next, we can retrieve a list of topics in the bag file. See rosbag Cookbook for more information. Note that we only need the topics for color and aligned depth images from each camera:

```
topics = bag.get_type_and_topic_info()
t = topics[1].keys()
t = [
    k for k in t if any([
        x in k
        for x in ('/color/image_raw', '/aligned_depth_to_color/image_raw')
    ])
]
serials = [x.split('/')[1] for x in t]
serials = sorted(list(set(serials)))
topic_list = [
    '/' + s + x
    for s in serials
    for x in ('/color/image_raw', '/aligned_depth_to_color/image_raw')
]
```

Given the list of topics, we want to synchronize their messages and save the messages of each topic to a variable. We use ROS's ApproximateTimeSynchronizer for synchronization:

```
import message_filters
synced_msgs = [[] for t in topic_list]
fs = [message_filters.SimpleFilter() for _ in topic_list]
ts = message_filters.ApproximateTimeSynchronizer(fs, queue_size=10, slop=0.1)

def callback(*msgs):
  for i, msg in enumerate(msgs):
    synced_msgs[i].append(msg)

ts.registerCallback(callback)
for topic, msg, t in bag.read_messages(topics=topic_list):
  fs[topic_list.index(topic)].signalMessage(msg)
```

Finally, we will create a new directory and save the synchronized image messages from each camera to JPG and PNG files. This is done with cv_bridge:

```
from cv_bridge import CvBridge, CvBridgeError
import os
import cv2
```

```
assert args.file[-4:] == ".bag"
save_root = args.file[:-4]
save_paths = [
    save_root + '/' + s + x
    for s in serials
    for x in ("/color_{:06d}.jpg", "/aligned_depth_to_color_{:06d}.png")
]
for s in serials:
  d = save_root + '/' + s
  if not os.path.exists(d):
    os.makedirs(d)

bridge = CvBridge()

for t, msgs in enumerate(synced_msgs):
  print("Saving topic: " + topic_list[t])
  for i, data in enumerate(msgs):
    print("{:06d}/{:06d}".format(i + 1, len(msgs)))
    save_file = os.path.join(save_paths[t].format(i))
    try:
      cv_image = bridge.imgmsg_to_cv2(data, data.encoding)
    except CvBridgeError as e:
      print(e)
    if data.encoding == 'rgb8':
      cv2.imwrite(save_file, cv_image[:, :, ::-1])
    elif data.encoding == '16UC1':
      cv2.imwrite(save_file, cv_image)
    else:
      assert 0
```

Once done, you will have a new directory in the same folder as the bag file with the following structure:

```
bag_file_name/
├── serial_id_1/
    ├── aligned_depth_to_color_000000.png
    ├── aligned_depth_to_color_000001.png
    ├── ...
    ├── color_000000.jpg
    ├── color_000001.jpg
    └── ...
├── serial_id_2/
└── ...
```

Reference: this script.

Installing apriltag_ros

Besides calibrating the extrinsics between the cameras, sometimes it is also helpful to get the transformation from the camera coordinate frame to a certain coordinate frame in the world, e.g. a corner of the table. AprilTag can be used for this. We use a ROS wrapper called apriltag_ros. To install, please refer to the official documentation, or run the following commands:

First, you need to install realsense-ros if you have not.

Next, install apriltag_ros:

```
mkdir -p $HOME/catkin_ws/apriltag_ros/src
cd $HOME/catkin_ws/apriltag_ros/src

git clone https://github.com/AprilRobotics/apriltag.git
git clone https://github.com/AprilRobotics/apriltag_ros.git

cd ..
catkin_make_isolated

echo "source $HOME/catkin_ws/apriltag_ros/devel/setup.bash" >> $HOME/.bashrc
source $HOME/.bashrc
```

To test AprilTag, you need to generate tags and print them out. To avoid generating tags yourself, you can also use some pre-generated tags online, such as from here. For now, let's test with a pre-generated tag. Download the pre-generated tags here and print out the first page (`april.tag.Tag36h11, id = 0`). Make sure you print the tag **in its actual size**.

In order to detect the tag, we need to set the type of the tag we printed out before launching apriltag_ros.

1. Open `src/apriltag_ros/apriltag_ros/config/settings.yaml` and make sure `tag_family` is set to `'tag36h11'` (this should be default after cloning):

   ```
   tag_family:        'tag36h11' # options: tagStandard52h13, tagStandard41h12,
   tag36h11, tag25h9, tag16h5, tagCustom48h12, tagCircle21h7, tagCircle49h12
   ```

2. Open `src/apriltag_ros/apriltag_ros/config/tags.yaml` and add `{id: 0, size: 0.172},` to `standalone_tags`:

   ```
   standalone_tags:
     [
       {id: 0, size: 0.172},
   ```

```
  ]
```

Now we are ready to demo tag detection with the camera feed from RealSense. Make sure all the ROS processes are killed and start a new terminator. And make sure you have one RealSense connected.

In **window 1**, launch roscore:

```
roscore
```

In **window 2**, launch the camera (after setting $ID1):

```
roslaunch realsense2_camera rs_camera.launch serial_no:=$ID1 camera:=$ID1
depth_width:=640 depth_height:=480 color_width:=640 color_height:=480 depth_fps:=30
color_fps:=30 align_depth:=true
```

In **window 3**, launch apriltag_ros using the feed of the color camera:

```
roslaunch apriltag_ros continuous_detection.launch camera_name:=/$ID1/color
image_topic:=image_raw camera_frame:=${ID1}_color_optical_frame
```

After launching all the above commands, open rviz:

```
rviz
```

and make the following changes in the Display panel:

1. Change `Global Options->Fixed Frame` from `map` to `${ID1}_link`.
2. Click the `Add` button and add `rviz->DepthCloud`.
3. Expand `DepthCloud` in the Display panel.
4. Change `Depth Map Topic` to `/$ID1/aligned_depth_to_color/image_raw`.
5. Change `Color Image Topic` to `/$ID1/color/image_raw`.
6. Click the `Add` button and add `rviz->TF`.

You should now see the point cloud from the camera and the tf transform tree in the 3D view panel. If you move the camera view to cover the tag, you should see the detected TF of the tag, like in this example.

You can also print out the detected tag pose in real time using the following command:

```
rostopic echo /tag_detections
```

This will print out messages like below:

```
header:
  seq: 19439
  stamp:
    secs: 1625269964
    nsecs: 607464552
  frame_id: "105322250873_color_optical_frame"
detections:
  -
    id: [0]
    size: [0.172]
    pose:
      header:
        seq: 29736
        stamp:
          secs: 1625269964
          nsecs: 607464552
        frame_id: "105322250873_color_optical_frame"
      pose:
        pose:
          position:
            x: -0.0240318327555
            y: -0.242946308493
            z: 0.576730116777
          orientation:
            x: 0.998853860836
            y: -0.034052332356
            z: 0.016474073164
            w: 0.0293258975643
        covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
---
.
.
.
```