Lab Course: Robot Vision WS 2013/2014

Philipp Heise, Brian Jensen

Assignment 3 - Due: 04.12.2013

In this assignment sheet, you are going to work with stereo data and RGBD data. The goal is to create a motion estimation system referred to as Visual Odometry, for use with calibrated stereo or RGBD data. You will be using a class of techniques typically known as Structure from Motion (SfM) in the computer vision community or Visual Simultaneous Localization and Mapping (VSLAM) in the robotics community. You will be working with the KITTI dataset for stereo (`http://www.cvlibs.net/datasets/kitti/`) and the TUM-RGBD datasets (`http://vision.in.tum.de/data/datasets/rgbd-dataset/download`) for RGBD input.
We packed the necessary content of the KITTI dataset into bag-files for usage within ROS, which can be downloaded here: `http://www6.in.tum.de/~jensen/rvc/stereo_data/`

ATTENTION: there was a typo in the bag file generation, so the topic name for the stereo frame unfortunately is **kitty** instead of `kitti`

NOTE: All the Kitti datasets contain rectified images. If you use other datasets you will likely need to perform stereo rectification prior to any of the processing steps in this sheet.

Exercise 1 Sparse Stereo Matching

The first step in the your visual motion estimation process is to estimate the 3D position for a sparse set of robustly matched points between the left and right views of a calibrated stereo system.

1. *(Visual Odometry Package)*
   - Create a new package named `<GroupName>_vo` within your repository. You are going to reuse your feature detectors and descriptors from the last sheet, so you should make your new package depend on your previous packages.
2. *(Synchronized Stereo Subscription)* Subscribe to the two image topics together with the associated `camera_info` topics in a synchronized manner, similar to the last sheet.
   - Use `http://www.ros.org/wiki/message_filters` to achieve time synchronous subscription.
   - There is also a special and useful class for stereo calibration data (`StereoCameraModel`) in the `image_geomerty` package: `http://www.ros.org/wiki/image_geometry`. This class has some handy methods for converting between disparity and depth values given the `camera_info` calibration data.
3. *(Feature Detection)*
   - Use your Feature detector as well as your feature descriptor implementation from the previous assignment sheets to extract features from both input images at each time step.

- All important parameters governing your implementation should again be available over dynamic_reconfigure (e.g. by reusing the your nodes from the previous sheets where appropriate).

4. *(Feature Matching and Disparity Computation)* In order to compute disparity values, you need to find correspondences in the stereo frames. We'll do this by matching features from the left frame to features in the right frame:
   - Match the extracted features from the left image to the right image from the same time step. Implement a stereo-matching strategy, which makes use of the additional constraint that feature matches must be in the same horizontal line in both images (epipolar constraint in rectified stereo).
   - Disparity should be within a useful range. Range thresholds should configurable using dynamic reconfigure as minimum depth and maximum depth in meters.
   - Experiment with different techniques to speed up and improve your implementation, for example:
     – Create a row lookup table for your features.
     – Preemptively exclude features matches that are outside of valid disparity range.
     – Reciprocal Matching: Perform consistency checking: match from left to right and from right to left, only accepting consistent matches.
     – Soften the requirement that features matches must be in the same line. (e.g. +/- 1 line)

     Choose the parameters and implementation that deliver the best results.
   - Create 3D points from your matches using the left matching point's x, y and disparity value (You can use functions of the `StereoCameraModel`).

5. *(PointCloud publishing)* To check the result of your "triangulation", publish a pointcloud and visualize it in `rviz`.
   - Publish a `PointCloud2` message from your sparse set of matched feature points: http://docs.ros.org/api/sensor_msgs/html/msg/PointCloud2.html.
   - Make sure to set your frame ids correctly (set it to an existing `tf` frame, e.g. `start` or the frame id of the left camera message)

Exercise 2 Temporal Feature Matching

The next phase of your 3D motion estimation implementation involves taking the robustly matched 3D points gathered at each time step and finding correspondences between the left-right feature matches at the previous time step with left-right feature matches at the current one.

1. *(Windowed Matching)* A useful assumption for tracking features throughout images is that their positions will only change gradually. Therefore a typical approach is to constrain the matching between features to a certain window around a predicted (or the previous) position. You will implement a matching strategy, that finds the closest match for a given left-to-right feature correspondence from the last frame within a user specified radius. Experiment with different matching strategies, for example:
   - Use descriptors from left and right image while enforcing the epipolar constraint.
   - Use a reciprocal matching (see stereo description)

   At the end of this stage, you should have a set of 3D-3D Point correspondences between two time steps.

2. *(Visualization of the 3D Matches)* To visualize your results, draw your matches side by side into an image augmented with the appropriate views. For the temporal matching you can e.g. draw the movement of the features as small lines (flow). For stereo you can e.g. also draw lines between the matched positions of the two images.

Exercise 3 3D Pose from Correspondences

1. *(3D pose estimation)* Use the method of **Umeyama** to compute the 3D pose between consecutive frames. A nice overview is presented Szeliski "Computer Vision: Algorithms and Applications" chapter 6.1.5. A more detailed explanation can be found in the original publication "Least-squares estimation of transformation parameters between two point patterns" (http://cis.jhu.edu/software/lddmm-similitude/umeyama.pdf).
   - Use the method of Umeyama to compute the euclidean transform that maps from the previous time step to the current time step

$$T_{step-1}^{step} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$

   - Maintain a a cumulative transform that maps from the world coordinate system to the camera coordinate T_{world}^{camera}.
   - Make sure to initialize the cumulative transform with an appropriate value for the data set (for the KITTI data sets this would be the identity).

2. *(Robust estimation)* In order to cope with possible outliers primarily due to incorrect 3D-3D correspondences and prevent incorrect point cloud estimates a naive version of RANSAC is helpful. A user specified number of iterations with random minimal subsets of the 3D-3D point correspondences are performed and in each iteration the set of all 3D-3D correspondences is partitioned into subsets consisting of inliers and outliers according to some distance metric. Correspondence distances below a user specified distance threshold are then considered inliers and all others outliers. After completion the largest subset of inliers is then used to calculate the final 3D pose estimate.
   - Perform this n times:
     - use the matched 3D point correspondences from the previous exercise
     - select a minimum subset of matches randomly (3)
     - compute the pose using the the subset
     - calculate the number of correspondence inliers that are below the distance threshold
     - if this result has more inliers than current best, remember these inliers
   - recalculate the pose from the largest subset of inliers.

3. *(Pose Publishing)*
   - Use the `tf` or `tf2` ROS package to publish your current camera pose estimate T_{world}^{camera}.
   - Make sure to use the correct stamp when publishing your camera pose estimate.
   - Compare the movement of your `tf` frame with the one published within the bag files (e.g. by observing it in rViz)

Exercise 4 RGBD Visual Odometry (Optional)

Now you are going to modify your visual odometry pipeline from the previous exercises to work with data from RGBD cameras such as the Kinect that have become quite popular in

the robotics community in the last years. Essentially you simply skip disparity computation step since RGBD sensors directly estimate a depth value for almost all pixels in the image and perform only the temporal feature matching and 3D pose estimation.

1. *(RGBD Visual Odometry Pipeline)*
   - Create a new executable for performing RGBD visual odometry.
   - Since the depth and color images are not published synchronously with the Kinect you should use the `Synchronizer` and `ApproximateTimePolicy` classes from the `message_filters` package to achieve approximate time synchronisation.
   - Use the `PinholeCameraModel` class from the `image_geometry` package for handling the image calibration parameters necessary for obtaining 3D points.
   - Extract key points from the color images taking care to filter out key points where there is no depth information available.
   - Reuse your temporal matching and pose estimation from the previous exercises.
   - Test your implementation with some of the TUM-RGBD benchmark datasets.