Chapter 1
PCL: Toyota Code Sprint
Global Point Cloud Localization

Maurice F. Fallon and Hordur Johannsson

1.1 Project Overview

This report describes the development of new algorithms for real-time large-scale localization using an RGB-D sensor. During this project we have used a mix of Microsoft Kinect and Asus Xtion Pro Live sensors so as to emphasize the mobility of our proposed algorithms as well as explicitly excluding robot wheel odometry — which is a typical requirement of traditional LIDAR localization systems [1].

Some of the development of this research precedes this code sprint, however the PCL TOCS has given us the impetus to make our implementation widely available within the library. As such this report presents overlapping material with our upcoming International Conference on Robotics and Automation 2012 publication [Fallon et al., 2012]. Interesting overlapping material is represented here for a cohesive report.

1.2 System Overview

An overview of the system we are developing is illustrated in Figure 1.1. Sensor output in illustrated in lemon color, although only RGB-D data is considered in this project. Modules we have been actively developing are colored blue. They will each be discussed in the following section along with the Incremental Motion Estimation algorithm. This module utilizes a visual odometry algorithm developed independently of this project. Alternatives to this are discussed in Section 1.2.2.

Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA. mfallon,hordurj@mit.edu

[1] We also have at our disposal the SoftKinect DepthSense and Velodyne HDL-32E sensors (as well as the PR2's nodding Hokuyo LIDAR) if experimentation with these sensors is of interest.

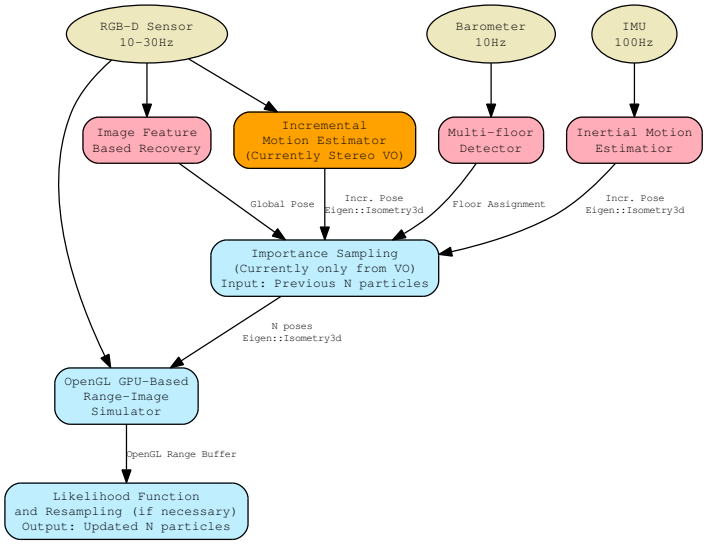


Fig. 1.1 *Global RGB-D Localization System.*

We have also illustrated in pink three modules which would naturally augment the point cloud-based localization system by providing redundancy and recovery methods. We have been developing these modules independently of PCL. They are illustrated here to demonstrate the flexibility of an SMC-based approach to localization. Additionally, if this system was integrated within a 2D wheeled robot, wheel odometry could easily be integrated also.

In particular the Image Feature-based recovery algorithm is of note. This Bag-of-Words algorithm heavily uses PCL's sister project OpenCV in its implementation. This algorithm uses the DBow library [Cadena et al., 2010] to build a visual map during map construction with each visual location corresponds to a location in the map. This visual bag-of-words application implements the hierarchical vocabulary tree described by [Nister and Stewenius, 2006].

An alternative recovery mechanism, avoiding RGB image features, could use 3D features — including the many implemented in PCL (Spin, VFH, RSD). We have not explored using these as yet.

1.2.1 Particle Filter

The core of the particle filter is a set of candidate poses (both velocity and position) which are weighted according to their relative likelihood (particle weight). This distribution, if accurately estimated represents a posterior likelihood of the unit of interest — in our case the RGB-D sensor.

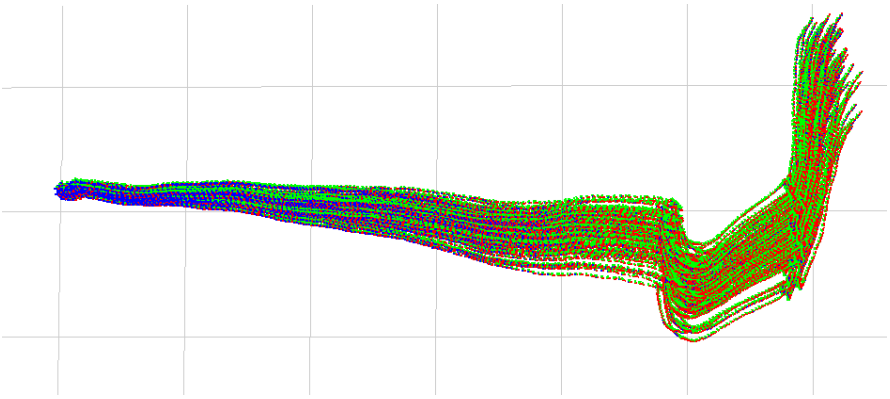


Fig. 1.2 *Indicative evolution of our particle cloud using Visual Odometry — growing due to increased uncertainty. The grid size is 5 meters.*

In the early part of this project we significantly overhauled our implementation of the particle filter to utilize the Eigen Linear Algebra Library (a PCL dependency) for geometry and to re-implement our particle propagation/sampling algorithm using a BOOST-based random number generator. We also extended our approach to geometry in 3D (where previously only the likelihood was computed in 3D).

1.2.2 Incremental Motion Estimation

As mentioned in Section 1.2, a crucial component to our localization algorithm is accurate particle pose propagation. While it is theoretically possible to estimate a particle filter distribution with an incremental motion estimate, without doing so would require a much larger number of particles for similar performance while also placing a much higher bar on the accuracy of the input map.

Currently our approach uses a visual odometry algorithm called FOVIS (Fast Odometry for Vision) [Huang et al., 2011] which is robust, accurate and lightweight. The FOVIS library produces accurate visual odometry at 30Hz using a single CPU core. While the full 30Hz frame rate results in the best accuracy, 10Hz has typically been sufficient for our purposes. FoVIS has recently been made open source and is available here:

http://code.google.com/p/fovis/

While this work has been developed independently of us, alternatives to traditional visual odometry have emerged. We have been implementing the dense odometry algorithm described in [Steinbrucker et al., 2011]. The authors suggest that it out performs tradition VO, as well as being suitable for lower light conditions.

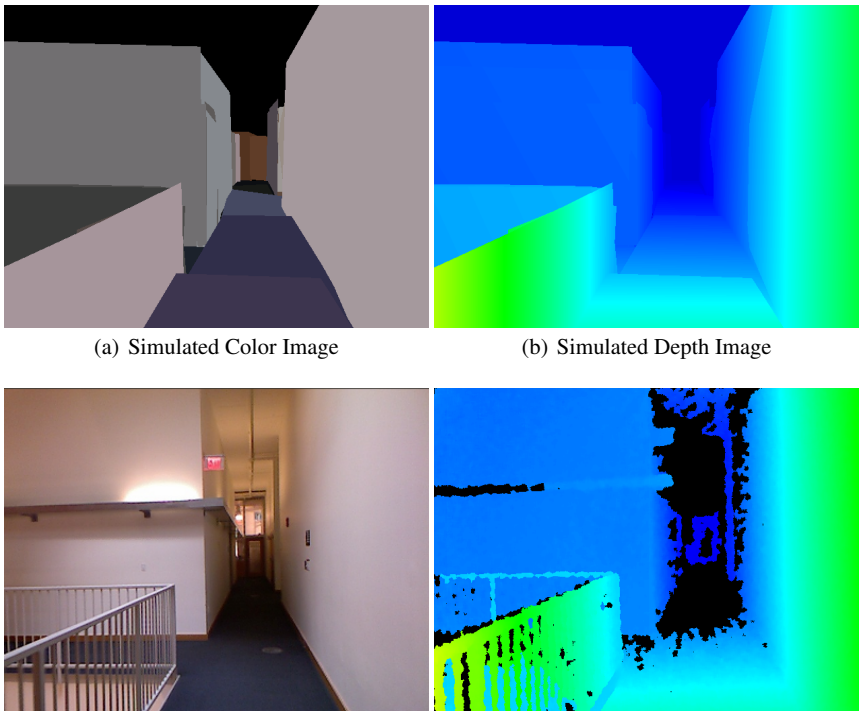
The output of the VO algorithm propagated using our particle filter's noise model is illustrated in Figure 1.2. The motion is from left to right. Starting with low uncertainty, the particle cloud spreads out as the user moved 45 meters along a corridor.

1.2.3 Range Image Simulation

The output poses of the particle propagation are then evaluated using a likelihood function. Our novel method uses an efficient OpenGL implementation (on the GPU) to propose candidate views. We match our sensor's calibration to that of the simulated range images. Example simulated images are demonstrated in Figure 1.3.

A likelihood is calculated by comparing each simulated depth point to each measured depth point. The likelihood updates the particle weights in the usual manner for a particle filter.

This model has been developed into a complete library for RGB-D simulation as discussed in Section 1.5.



(a) Simulated Color Image (b) Simulated Depth Image

(c) Measured Color Image (d) Measured Depth Image

Fig. 1.3 *Using a prior 3-D building model we efficiently generate simulated depth and color views (top) which are then compared to RGB-D camera data (bottom). Using a particle filter, hundreds of similar views are evaluated and used to estimate the sensor pose in 3-D.*

1.3 Producing a Prior Model

In this report we will not discuss the techniques required to produce a map sufficient for RGB-D localization in depth. The model will represent the location of large planes coarsely. This model is assumed to be the result of a typical SLAM method either in 3D (using vision) or in 2D (using LIDAR). We utilized the latter to date and also have access to building models for our place of work. While the output of 3D visual RGB-D SLAM cannot yet produce sufficiently accurate maps for our purposes, this area is of very active research, consider for example [Henry et al., 2010].

Given the set of SLAM sensor poses, the algorithms to the associated point clouds to our 3D plane model are relatively straightforward and the various algorithms required (RANSAC plane fitting, object segmentation) have been implemented in PCL some time ago.

1.4 Quantifying Results

To fully quantify the performance of our algorithm we carried out a series of Monte Carlo runs using the dataset collected with a robotic wheelchair. Varying the number of particles from 12 to 400, the algorithm was run on a 3 minute, 215 meter dataset for 20 independent runs. This represents a total of 15 hours of computation time.

The results of each run were compared to the LIDAR pose of the robot aligned with the particle filter pose trajectory. In Figure 1.5 we present our results. The results give an indication of the accuracy of the simulation algorithm as well as its stability for different particle numbers.

For each measurement we have compared the performance to a seperate CPU-based likelihood function which

In summary we observed roughly equivalent localization accuracy with equal numbers of particles — with the Euclidean likelihood function being slightly more accurate. The major difference between the two methods was in the computation requirements. For 100 particles and a frame-rate of 10 Hz, the generative method is real-time while the Euclidean method is 5 times slower than real-time. The slow pace of the Euclidean likelihood precluded us from testing with 200 and 400 particles (where the gap was even wider).

Stable real-time operation with 350 particles has been realized on a 4-core 2.53GHz Pentium Core2 powered laptop with an Nvidia Quadro 1700M with 32 Cores — utilizing one CPU core each for data capture and visual odometry and Monte Carlo localization split between each architecture and processing at an effective rate of 7–8Hz. In regions of low uncertainty as few as 10 particles are required for stable operation. Implementation of an adaptively resizing particle cloud would be useful in such circumstances [Fox, 2003].

This is essential to maintain a very large particle set to stable localization in challenging handheld circumstances locations. Approaches which attempt to agres-

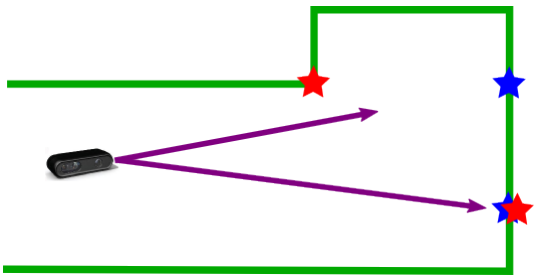


Fig. 1.4 *Illustrat.*

sively minimise the number of particles for real-time operation will catestrophically fail outside benign environments.

Finally we would like to reemphasize that only the RGB-D Kinect sensor was used in all of these experiments so as to demonstrate the robustness of Monte Carlo localization with such a low cost sensor. Additional sources of odometry such as wheel odometry or an IMU could of course have been used to improve the prediction model and to address locations in which the visual odometry fails, as mentioned above. We have avoided doing so for simplicity and generality.

1.4.1 Recent Optimizations

Recently we have implemented a series of optimizations which now allow for 1000 hypotheses to be simulated at 10Hz.

Evaluating Individual Measurement to Model Associations

This is the primary approach of this method. Essentially by using the Z-buffer and an assumption of a generative ray-based cost function, we can evaluate the likelihood along the ray rather than the distance in Euclidean space. This is as was mentioned in previous section, I just mention it here for context.

Computing Per-pixel Likelihood Functions on the GPU using a GLSL Shader

The initial implementation of the OpenGL range likelihood algorithm only used the GPU for rendering the scene. In addition the on-screen frame-buffer was used for the rendering which limits the number of particles that can be rendered in one pass. To improve on this we have looked into using more advanced features of the OpenGL library, including frame-buffer objects and the OpenGL Shading Language (GLSL).

The likelihood function we are currently evaluating is a normalized Gaussian with an added noise floor. For a given particle $\mathscr{A}_k^{(p)}$ at time step k, the inverse depth

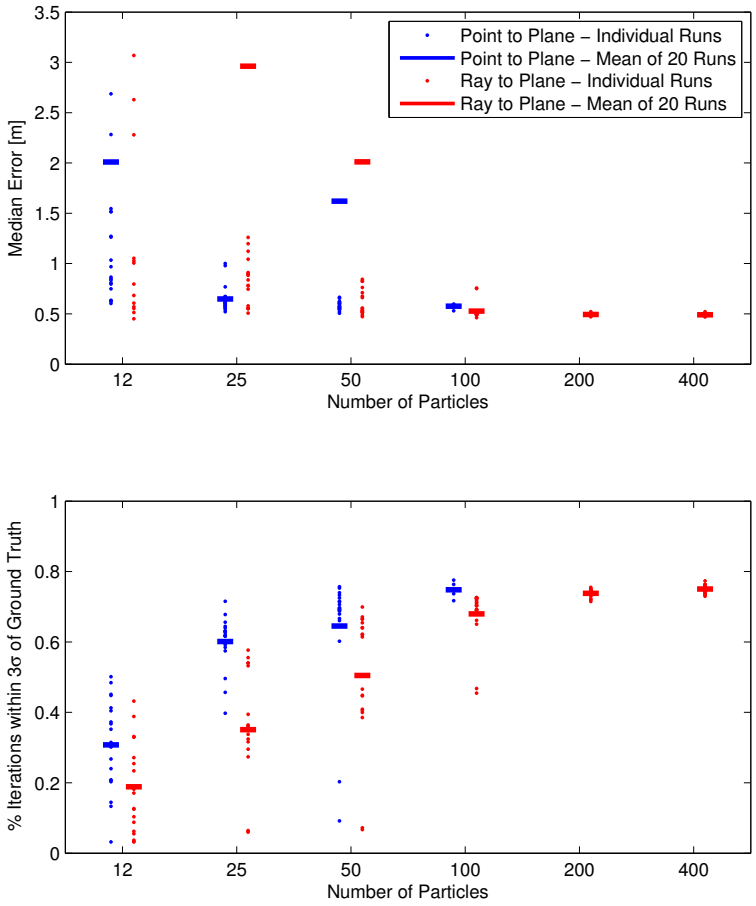


Fig. 1.5 *Performance metrics for both likelihood functions (averaged for 20 separate runs). Typical performance for 100 particles is of the order of 0.5 m median error and 78% of estimates within 3σ of the true location. Note that for some failed runs with low particle numbers the median error is greater than 3.5m.*

image, $Z^G_{(p)} = (z^0_{(p)}, \ldots, z^{N_i}_{(p)})$, containing N_i points is generated as described above. For each inverse depth pixel, z^i_k, in the measured image $Z^M_k = (z^0_k, \ldots, z^{N_i}_k)$, the likelihood is evaluated as follows

$$p(\mathbf{z}^i_k | \mathscr{A}^{(p)}_k) = \beta c_r \mathscr{N}(z^i_k; z^i_{(p)}, \sigma^2_d) + (1-\beta)\mathscr{U}(0,1) \tag{1.1}$$

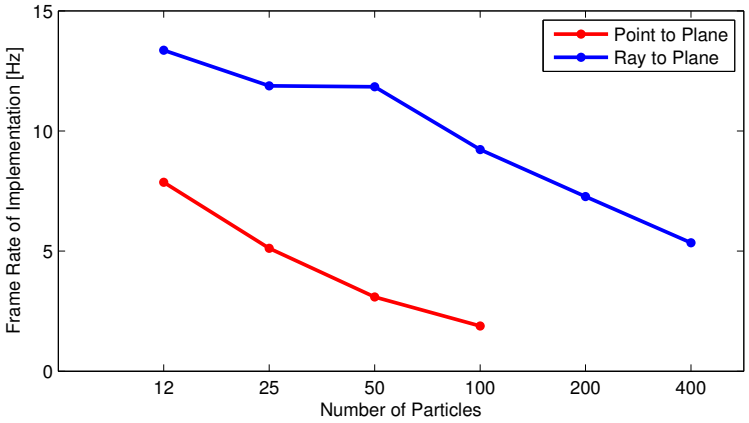


Fig. 1.6 *Timing statistics for both likelihood functions. In the case of the later, real time operation (at 10Hz) has be achieved with 100 particles and the frequency reduces linearly with the number of particles.*

where the inverse depth varies in the range $(0,1)$. An appropriate normalization constant, c_r, had been added for the truncated normal distribution and the inverse depth variance σ_d^2 was chosen to be $0.1m^{-1}$. The addition of the uniform distribution supports heavy tailed behavior and in doing so each point in the cloud has only a small effect on the overall likelihood function. The parameter $\beta = 0.01$ was found to give good experimental performance.

Using the frame-buffer objects and rendering to textures allows more flexibility in the size of the render targets. This also helps with minimizing the data transfer between the CPU and the GPU. The GPU can now directly render to the GPU memory. Then a shader program can use the texture that contains all the generated depths images, the current sensor depth image, and the cost function to compute the likelihood for each particle. This can then be summarized on the GPU and the only information that needs to be sent back is the score for each individual particle, instead of the depth for every pixel in all the particles.

Summing the Per-pixel Likelihood on the GPU

Having evaluted the log likelihood per pixel, the next step is to combine them into a single log likelihood per particle by log summation:

$$p(\mathbf{Z}_k|\mathscr{A}_k^{(p)}) = \prod_{i=1}^{N_i} p(\mathbf{z}_k^i|\mathscr{A}_k^{(p)}) \tag{1.2}$$

This can be optimized by parallel summation of the pixel images: e.g. from 32x32 to 16x16 and so on to 1x1 (the final particle likelihood). In addition there may be

some accuracy improvement by summing simularly sized values and avoiding floating point rounding errors: e.g. (a+b) + (c+d) instead of ((a+b)+c) +d

While this optimization is likely to be beneficial, the speed up is unlikely to be as substantial as the improvement in the previous section.

Single Transfer of Model to GPU

An addition to the above, previously we used the mixed polygon model. We have now transferred to using only a model made up of only triangles. This allows us to buffer the model on the GPU and instead we transmit the indices of the model triangles which should be rendered in a particular iteration. (Which is currently all the triangles).

Future work will look at determining the set of potentially visible polygons - perhaps using a voxel-based indice grid. This is more complex as it requires either implicit or explicit clustering of the particle set.

Putting it all together

Weve benchmarked the improvements using a single threaded application which carries out particle propogration (including Visual Odometry), renderering, likelihood scoring and resampling. The test log file was 94 seconds long at about 10Hz (about 950 frames in total). We are going to focus on the increased number of particles for real-time operation with all modules being fully computed.

The biggest improvement we found was using the triangle model. This resulted in a 4 times increase in processing speed.

Using the triangle model, we then tested with 100, 400 and 900 particles the effect of computing the likelihood on the GPU using the GLSL shader:

This results in a 20-40% improvement, although we would like to carry out more sample points to verify this. The result of this is that we can now achieve real-

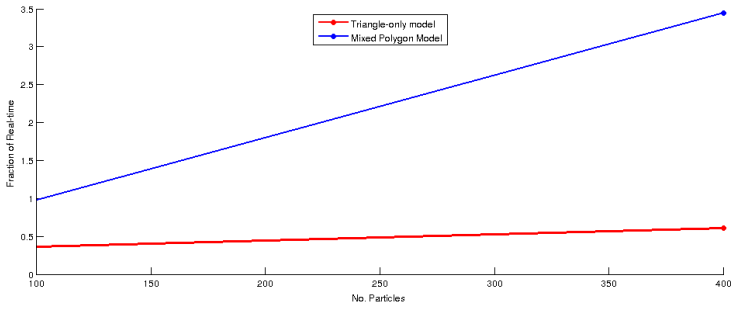


Fig. 1.7 *Comparison between triangle renderering and mixed polygon rendering.*

time performance with 1000 particles. For the log file we didnt observe significant improvement in accuracy beyond about 200 particles.

Finally, one variable we didn't explore was image decimation. Currently we decimate the measured depth image to 20x15 pixels and create the simulated views like those in Figure 1.3 - with only 300 pixels. The fact that localization works with 300 pixels is amazing - but we hope to look at varying this in future.

1.4.2 Overview of Computation

Finally, some analysis to determine the overall benefit of the improvements mentioned above. It will give a feel for the impact of various improvements in future also.

High Level Timing

First lets look at the high level computing usage. We did several tests and looked at elapsed time for different numbers of particles and also looked at (1) doing the likelihood evaluation on the GPU (using the shader) or (2) on the CPU (as previously). Figure 1.9 shows the performance of the major components of the entire localization system for different numbers of particles.

The data in this case was at 10Hz so real-time performance amounts to the sum of each set of bars being less than 0.1 seconds. For low number of particles, the visual odometry library, FoVis, represents a significant amount of computation - more than 50 percent for 100 particles. However as the number of particles increases the balance of computation shifts to the likelihood function.

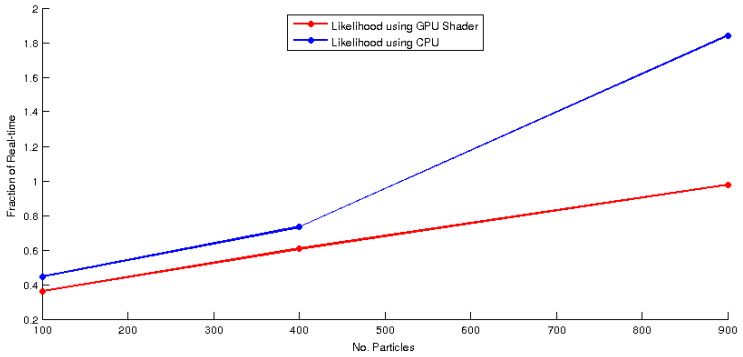


Fig. 1.8 *Comparison between computing likelihood values on the GPU (using a GLSL shader) and computing them on a CPU (traditionally).*

The visual odometry and likelihood function can be shifted to different threads and overlapped to avoid latency. We havent explored it yet. Other components such as particle propogation and resamping are very small fractions of the overall computation and are practically negiligible. In particular you can see that the likelihood increases much more quickly for the CPU evaluation.

Low Level Timing of Likelihood

The major elements in the likelihood are the rendering of the model view and the scoring of the simulated depths — again using the results of these same runs. Figure 1.10 shows the performance of the major components of the renderering for different numbers of particles.

Basically the cost of rendering is fixed - a direct function of the (building) model complexity and the number of particles. We insert the ENTIRE model into OpenGL for each iteration - so there will be a good saving to be had by determining the possibly visible set of polygons to optimize this. This requires clustering the particles and extra caching at launch time but could result in an order of magnitude improvement.

Its very interesting to see that the cost of doing the scoring is now significant here. The effect of it is that for large numbers of particles e.g. 900 scoring is only 5% of available time (5% of 0.1 seconds). Doing this on the CPU would be 30%. For the simplified model, this would be very important as the scoring will remain as is, but the render time will fall a lot.

NOTE: I think that some of these runs are slightly unreliable. For portions of operation there was a charasteristic chirp in computation during operation. I think this was the GPU being clocked down or perhaps my OpenGL-based viewer soaking up the GPU.

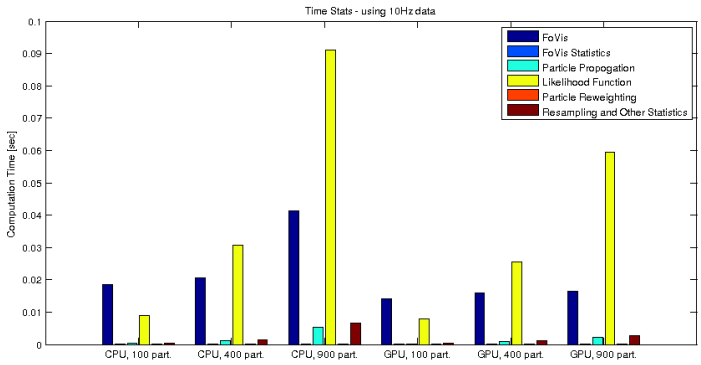


Fig. 1.9 *High level timing for the Kinect Localization algorithm.*

1.5 Standalone Use: RGB-D Simulation

The primary module we have added to PCL is a library for RGB-D data simulation. We have now integrated a complete RGB-D simulator which can simulate Kinect data at 30Hz. Our application takes as input a single .ply model and the user can use the GUI to save point clouds from that view point. The application is essentially a branch of the widely used pcl_pcd_viewer tool written by Radu Rusu.

Example simulated views are demonstrated in Figure 1.11, Figure 1.12 and Figure 1.13. This library is slated for addition to PCL Version 1.6.

1.5.1 Integration within KinFu

The PCL implementation of Kinect Fusion has be widely used by the PCL community. However as it requires (1) a high-end GPU (2) real-time capture of RGB-D data and is currently still experimental new users are unable to take use the implementation. Additionally the development would benefit from being able to vary noise levels, range, motion, configuration of the scene being renderer while also being able to perfectly ground truth the reconstruction.

We have also integrated our simulator to provide data to the KinFu. An example camera motion is illustrated in Figure 1.14, as well as an example reconstruction from KinFu. This implementation runs at the framerate of the KinFu algorithm — with simulation requiring only a small percentage of the required computation.

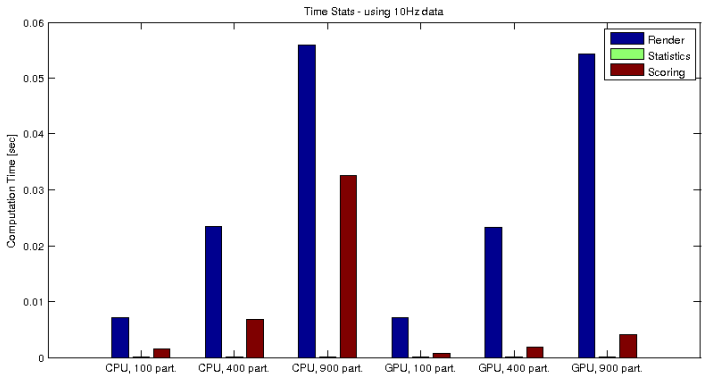
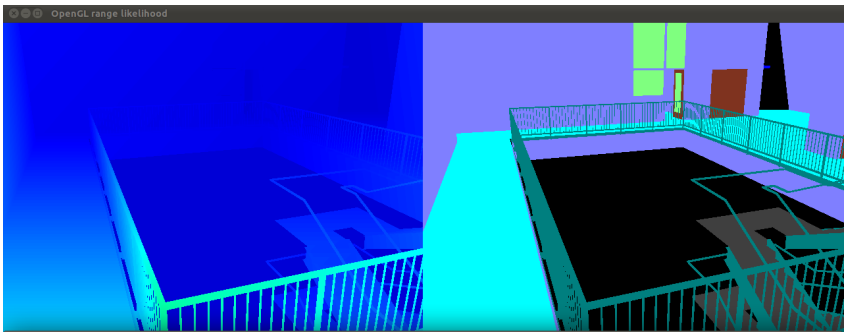


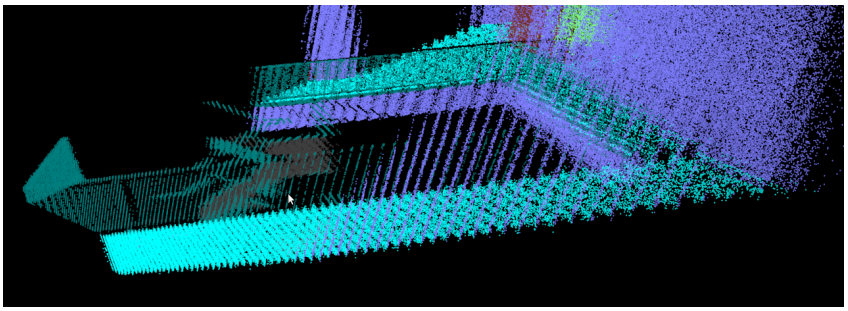
Fig. 1.10 *Low level timing plot — compared for different configurations of rendering and different numbers of particles.*

References

Cadena et al., 2010. Cadena, C., Gálvez, D., Ramos, F., Tardós, J., and Neira, J. (2010). Robust place recognition with stereo cameras. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*

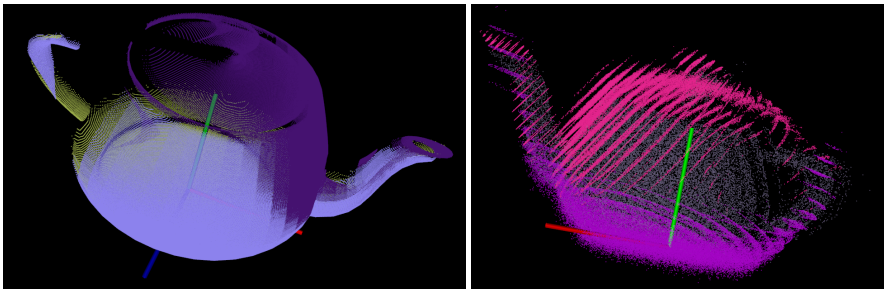


(a) View of Simulator GUI



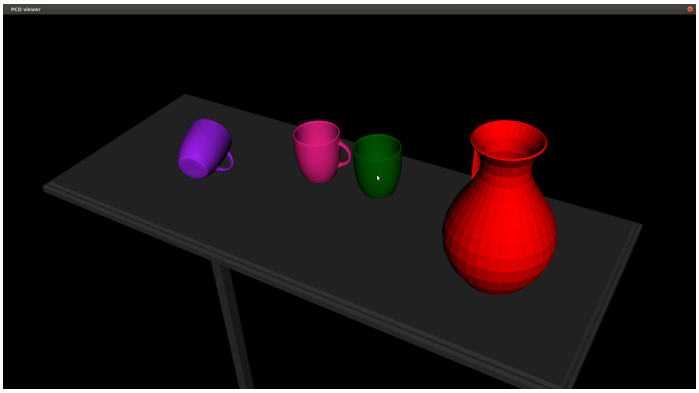
(b) Realistic Noise Added

Fig. 1.11 *Simulated views of MIT's Stata Center and the resultant point clouds.*

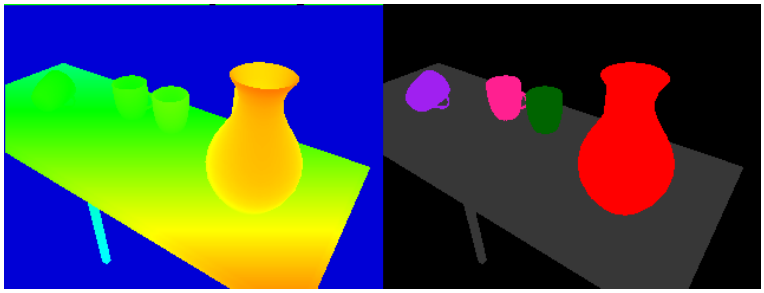


(a) No Noise Added (b) Realistic Noise Added

Fig. 1.12 *Simulated point clouds of a teapot using the method — registered afterwards. These figures illustrate 3 different simulated clouds.*



(a) View in pcl_pcd_viewer



(b) Simulated Data

Fig. 1.13 *Simulated point clouds of a table top. The top view represents what is seen by the user during capture. Bottom are the simulated range and RGB-D images.*

(IROS), Taipei, Taiwan.

Fallon et al., 2012. Fallon, M., Johannsson, H., and Leonard, J. (2012). Efficient scene simulation for robust Monte Carlo localization using an RGB-D camera. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, St. Paul, MN.

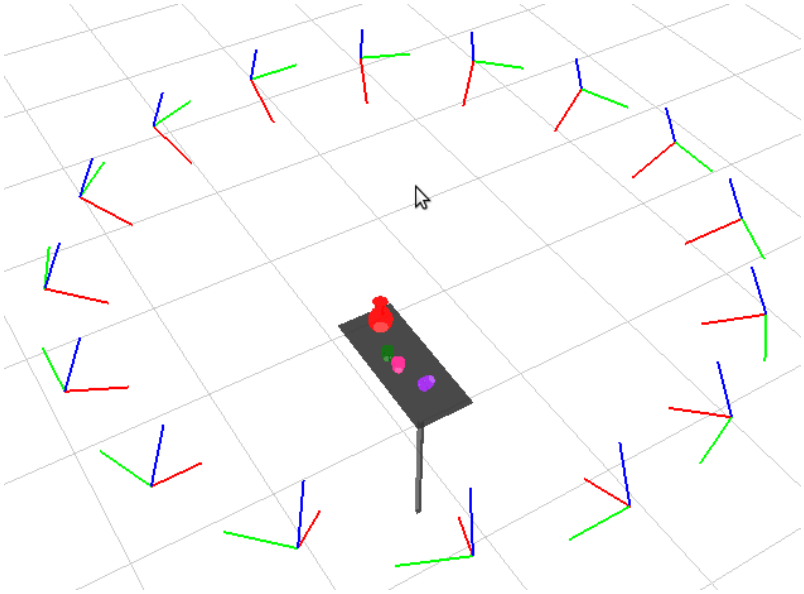
Fox, 2003. Fox, D. (2003). Adapting the sample size in particle filters through KLD-sampling. *Intl. J. of Robotics Research*, 22(12):985–1003.

Henry et al., 2010. Henry, P., Krainin, M., Herbst, E., Ren, X., and Fox, D. (2010). Rgbd mapping: Using depth cameras for dense 3D modeling of indoor environments. In *RGB-D: Advanced Reasoning with Depth Cameras Workshop in conjunction with RSS*, Zaragoza, Spain.

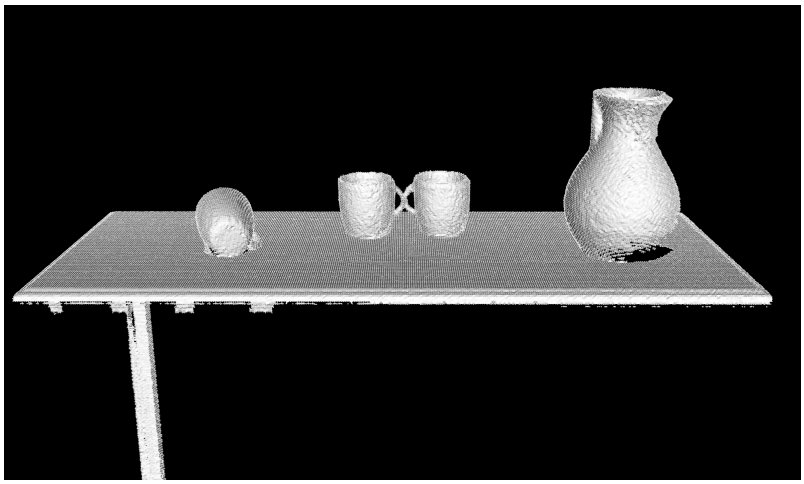
Huang et al., 2011. Huang, A., Bachrach, A., Henry, P., Krainin, M., Maturana, D., Fox, D., and Roy, N. (2011). Visual odometry and mapping for autonomous flight using an RGB-D camera. In *Proc. of the Intl. Symp. of Robotics Research (ISRR)*, Flagstaff, USA.

Nister and Stewenius, 2006. Nister, D. and Stewenius, H. (2006). Scalable recognition with a vocabulary tree. In *Proc. IEEE Int. Conf. Computer Vision and Pattern Recognition*, volume 2, pages 2161–2168.

Steinbrucker et al., 2011. Steinbrucker, F., Sturm, J., and Cremers, D. (2011). Real-time visual odometry from dense rgb-d images. In *Workshop on Live Dense Reconstruction with Moving Cameras at the Intl. Conf. on Computer Vision (ICCV)*.



(a) Motion of camera around table top



(b) Reconstruction using KinFu

Fig. 1.14 *Integration of simulation with KinFu. The top view represents a circle of poses of a simulated camera flying around a table top scene. Bottom is the reconstructed mesh from KinFu. Note that the right hand side of the vase has a hole - as it hasn't been sensed fully — this whole could be detected using a next best view algorithm.*