Koç University

Point Cloud Normal Estimation with CUDDA

Final Project Report for Parallel Programming Course

Doğancan Kebüde
25.12.2016

Contents

1. Introduction

Simultaneous Localization and Mapping (SLAM) is one of the biggest problems in Robotics. To address it, several sensors has been developed. However, none of them are as efficient as time-of-flight based sensors such as Microsoft's Kinect. Other sensors are slowly becoming obsolete: GPS only works outdoors and not with high resolution, LIDAR and Acoustic Ranging Devices (e.g. SONAR) only works in 2D, Stereo Cameras are expensive and it requires comparably high workload to reconstruct depth information from them. Time-of-flight based sensors, on the other hand, can gather information fast and provide both color and depth information in RGB-D and Point Cloud format. They provide color 3D information which is very similar to the human eye (figure 1).

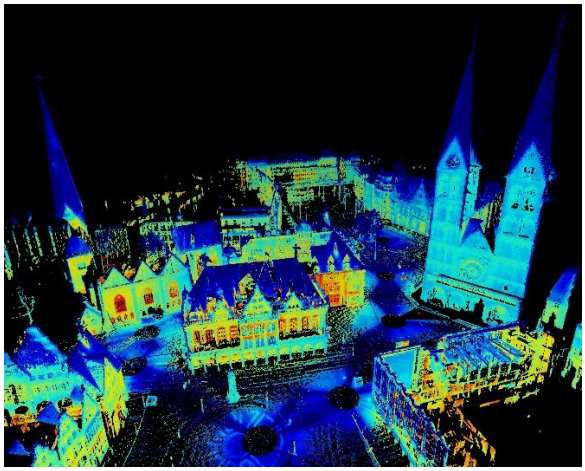

Figure 1. Point Cloud image of Bremen City Center, gathered by Microsoft Kinect [1].

Using Kinect's capabilities, a robot can be constructed such that it can "see" the world. Its "understanding" of that world, however, is another problem. A robot can gather point clouds using Kinect, with color and depth information; but it cannot understand how the surfaces/planes are oriented. To provide a robot with that capability, surface normals should be calculated. Having surface normal information, a robot can "understand" how surfaces are oriented and can try to avoid/interact with them.

For a robot to act fast, this processing should be done fast (near real-time, i.e. less than 100 ms). A point cloud gathered with Kinect can have up to 307,200 points in a cloud (640 x 480 resolution). This is a huge information to process, generally taking more than a second to estimate point normals. With a robot in a work environment, 1 second delay with each sampling cannot be tolerated. Therefore, normal estimation requires a huge performance increase, which brings us to the goal of this project. *The aim of this project is to estimate point cloud normals in less than 100 ms. To achieve this, OpenMP and CUDA parallelization will be investigated.*

The remainder of this report is structured as follows: Chapter 2 will introduce Point Cloud Library (PCL), Chapter 3 will describe the Normal Estimation Methodology that was applied in this work, Chapter 4 will explain different serial and parallel implementations of the methodology, Chapter 5 will talk about some challenges encountered along the way, Chapter 6 will measure performance and Chapter 7 will conclude the report and talk about the results.

2. Point Cloud Library (PCL)

PCL has started as a Ph.D. dissertation thesis by Radu Bogdan Rusu in 2009 [2] and it was then supported by Willow Garage in 2010 [3]. PCL was released as an open source project in 2011 [4]. Now this library has a huge community including companies such as Open Perception, Google Open Source, NVidia, Intel, Urban Robotics, Ocular Robotics, Toyota etc. and several universities worldwide.

PCL is a C++ library and it is the best way to work with point clouds since it is providing different classes for I/O, search, feature extraction, pattern recognition, point cloud image segmentation, visualization etc. It has also started supporting parallelization with OpenMP and CUDA; but these capabilities are still in development. Another important contribution of PCL is the *.pcd data format in which the point clouds are stored.

During this project, the implementation of normal estimation utilized some of the PCL capabilities such as I/O [5], Search [6], Normal Estimation without [7] and with OpenMP [8] and lastly, Normal Estimation with CUDA [9].

3. Normal Estimation Methodology

Normal Estimation methodology was implemented following the footsteps in [10] and [11]. PCL community also implemented their Normal Estimation class using these papers, therefore PCL's implementation of normal estimation is very similar. Steps are as follows:

1. For each point $\boldsymbol{p}$ in the point cloud, k nearest neighbors will be gathered through Kd-tree search (if unorganized point cloud) or organized cloud search (if organized point cloud).
2. For that k nearest neighbors, calculate the 3D centroid $\overline{\boldsymbol{p}}$.
3. Using the centroid, calculate a covariance matrix $\boldsymbol{C}$ utilizing the following equation:

$$C = \frac{1}{k}\sum_{i=1}^{k}(\boldsymbol{p}_i - \overline{\boldsymbol{p}})^T * (\boldsymbol{p}_i - \overline{\boldsymbol{p}})^T$$

   where k is the number of points in k-nearest neighborhood, $\boldsymbol{p}_i$ is the point of interest and $\overline{\boldsymbol{p}}$ is the 3D centroid of the nearest neighbors.
4. Apply Principal Component Analysis (PCA) to get the principal eigenvector of the neighborhood. This eigenvector will serve as our normal.
5. Check if the normal is oriented towards the viewpoint, if not, flip it. Orientation check will be done using the following inequality:

$$\overrightarrow{\boldsymbol{n}_i} \cdot (\boldsymbol{v}_p - \boldsymbol{p}_i) > 0$$

   where $\boldsymbol{v}_p$ is the viewpoint and $\overrightarrow{\boldsymbol{n}_i}$ is the surface normal for point $\boldsymbol{p}_i$.

Since this methodology does not have any data dependencies parallelization will be embarrassingly parallel. However, there are bottlenecks at kNN search. These can be handled by assuming and organized point cloud and working with 9-point stencils since data from Kinect always provides an organized point cloud. By utilizing this method, very good performance can be achieved in which the only constraint will be communication due to huge size of point clouds.

4. Implementation

The methodology explained above was implemented in six different ways: (1) Serial implementation with PCL, (2) Serial implementation without PCL, (3) OpenMP implementation with PCL, (4) OpenMP implementation without PCL, (5) CUDA implementation with PCL and (6) CUDA implementation without PCL. There are many aspects that are changing with each implementation and those will be explained in the following sub-chapters. Performance comparison between these different methods will be investigated in chapter 6.

It is important to note that each one of these implementations utilize PCL's I/O capabilities to read from and write to *.pcd files. Also, all implementations use "eigen" class to apply PCA.

4.1. Serial Implementation with PCL

This implementation is the most straight-forward version to be implemented. Search and normal estimation are both applied through PCL capabilities. This implementation reads the data, applies the normal estimation methodology explained in chapter 3 and writes the point cloud that has the normal information back to the *.pcd file.

4.2. Serial Implementation without PCL

This implementation reads data from the *.pcd, then takes the 9-point stencil for each point in the point cloud instead of applying a search method. However, taking the 9-point stencil requires assumption of an organized point cloud. This means, this version would not work with an unorganized point cloud. This is okay, since information gathered through Kinect is already organized if the user does not explicitly stores it as an unorganized data.

In this version, computation of centroids and covariance matrix elements are written explicitly, without utilizing PCL capabilities. PCA is achieved through "eigen" class' eigen33 function. The gathered eigenvector orientation is checked through the inequality provided in chapter 3 and it is flipped if the inequality does not hold. Lastly, the gathered normal information is written back to the point cloud and then it is stored back to a *.pcd file.

4.3. OpenMP implementation with PCL

This implementation uses *NormalEstimationOMP* class of PCL instead of the *NormalEstimation* class that was used in 4.1. The only difference between these two classes is that, NormalEstimationOMP writes an OpenMP pragma at the beginning of the for loop that computes the normals.

4.4. OpenMP implementation without PCL

This implementation takes the version in 4.2. and places a pragma OpenMP parallel for statement at the beginning of the compute loop. This requires identification of private data for each thread; but other than that, the application is embarrassingly parallel. The performance comparison between two versions will be further explained in chapter 6.

4.5. CUDA implementation with PCL

This version utilizes PCL's CUDA capabilities. Again, no kernel was implemented, PCL's own kernel was used. PCL implemented "upload" and "download" functions that can achieve cudaMalloc and cudaMemcpy regarding point clouds very efficient. The only thing that is left to the programmer is to specify which point cloud will be copied to and from the device.

However, there is a problem with PCL's CUDA implementation: the block size that the kernel is called with is built-in and cannot be changed if the library is not edited. Since library is placed in the HPC and requires sudo privileges to edit the code, the code could not be changed to check with different thread block sizes. Also, the kernel works with 1D blocks and block size is specified to be 256 threads.

4.6. CUDA implementation without PCL

In this version, a new CUDA kernel is written, utilizing the same approach explained in 4.2. However, in 4.2. the data was accessed several times by the CPU, now the code is optimized to take advantage of data reuse by using temporary variables and keeping them in registers so that the GPU does not have to grab the same data several times.

There are two files in this version: (1) estimate_normals_cuda.cu which implements the kernel and the kernel wrapper and (2) estimate_normals_cuda.cpp which reads from and writes to *.pcd file, processes point cloud data into a form that can be passed to the GPU device and calls the kernel wrapper.

The problem with this version is that there is not an efficient way to copy point cloud data to the device without PCL capabilities as there was in version 4.5., so the code becomes communication bound. However, the estimation of normals is incredibly fast. If the two versions are combined, the normal estimation can become very fast. The comparison will be explained in chapter 6.

5. Challenges

There has been several challenges along the way to finish this project. These and their solutions are explained below:

- Due to PCL's dependencies and CUDA GPU compatibility problems (PCL requires a post-Tesla architecture to work with CUDA), it was hard to find a PCL+CUDA compatible stand-alone device to run the code. A stand-alone device was required since this project will be run with a Kinect (and not *.pcd files) in the long run.
  ***Solution**:* To work on Somon cluster which has compute nodes that can work with PCL+CUDA.
- HPC Support had problems while installing PCL on Somon cluster. At first CUDA support of PCL was not built.
  ***Solution**:* A more detailed explanation for installation was provided to HPC Support staff.
- Boost and Eigen libraries had very complex structs, they did not work in CUDA code.
  ***Solution**:* PCL developed support for Eigen in their CUDA class, that support was utilized for solving problems with Eigen library. On the other hand, Boost required more complex solutions, at first the libraries were tried to be rewritten; but that did not solve the problem since point cloud processing in general requires Boost library. Therefore, the code is separated into two: a *.cu file that has the kernel and kernel wrapper in it and a *.cpp file that processes point cloud information using the Boost library.
- Linker errors due to CUDA support in Somon.
  ***Solution**:* The problem turned out to be related to Somon headnode not having some of the CUDA-7.5 libraries. Instead, one of the compute nodes of Somon (biyofiz-4-3) was utilized to build and run the code.

6. Performance Comparison

The performance comparison was done with demo0_kf0.pcd for each version (provided in the supplements file delivered in the final project package). That file has a point cloud with 307,200 points in it, has 640 x 480 resolution and it stores 12 MBs of data. The verification of results were done according to 4.1.'s results since that is the base version. Also, the performance comparison does not take whole running time of the code, since reading/writing *.pcd data is not in the scope of this project. In the long run, Kinect data will be read. Thus, the important performance criteria here is the time related to processing of the point cloud and estimation of normals. Finally, all versions were run on biyofiz-4-3 compute node of Somon.

With these in mind, the following chapters compare (1) serial versions, (2) OpenMP versions and (3) CUDA versions.

6.1. Comparison of Serial Versions

Serial implementation with PCL is much more efficient compared to the serial implementation without PCL. The PCL version can process point cloud data and estimate normals three times faster than the version without PCL. This means PCL's classes are much more efficient compared to the naïve approach.

PCL version can process point cloud data and estimate normals in only 1.62 seconds, while the version that does not utilize PCL's capabilities needs 4.53 seconds to achieve same results. Neither of these running times work for our need of near real-time normal estimation. A running time of less than 100 ms is needed. This run times make it obvious why we need parallelization for this problem.

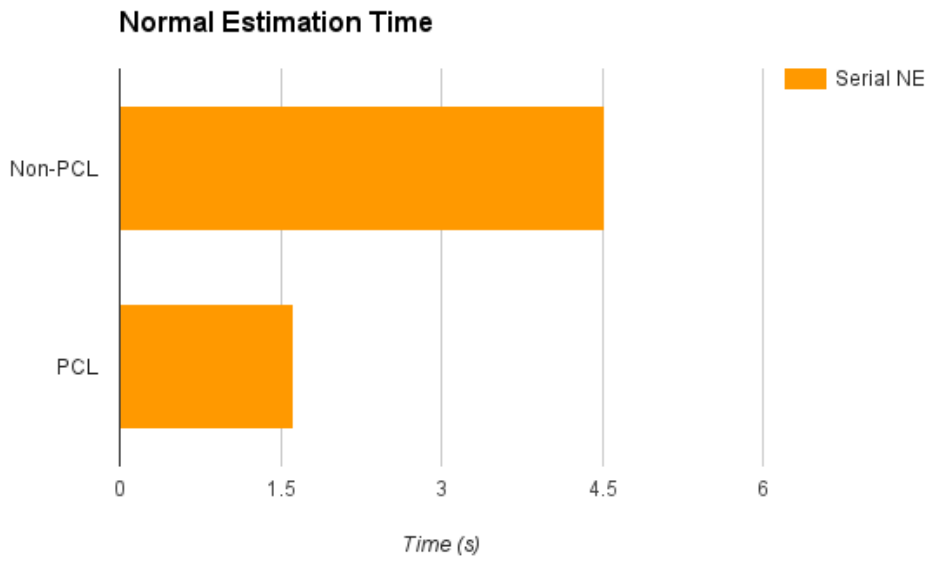

Figure 2. Comparison of serial versions.

6.2. Comparison of OpenMP Versions

The OpenMP versions are developed directly from serial versions, therefore it would be expected for the more efficient version (PCL version) to scale better. However, non-PCL version scales better compared to the PCL version with OpenMP parallelization. Still, PCL version is faster than the non-PCL version in the end. Best speedups are 9.4x for non-PCL version and 7x for PCL version with 64 threads. However highest parallelization efficiencies are 0.93 for non-PCL version and 1.04 for PCL version with 2 threads. That means PCL achieves superlinear speedup with 2 threads. That is enough to explain how efficient PCL functions are written.

	Single Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads	64 Threads
	NE	NE	NE	NE	NE	NE	NE
Non-PCL	4.53	2.44	1.83	1.07	0.81	0.55	0.48
PCL	1.62	0.78	0.94	0.5	0.31	0.35	0.23

Table 1. Time comparisons with different thread counts for OpenMP versions.

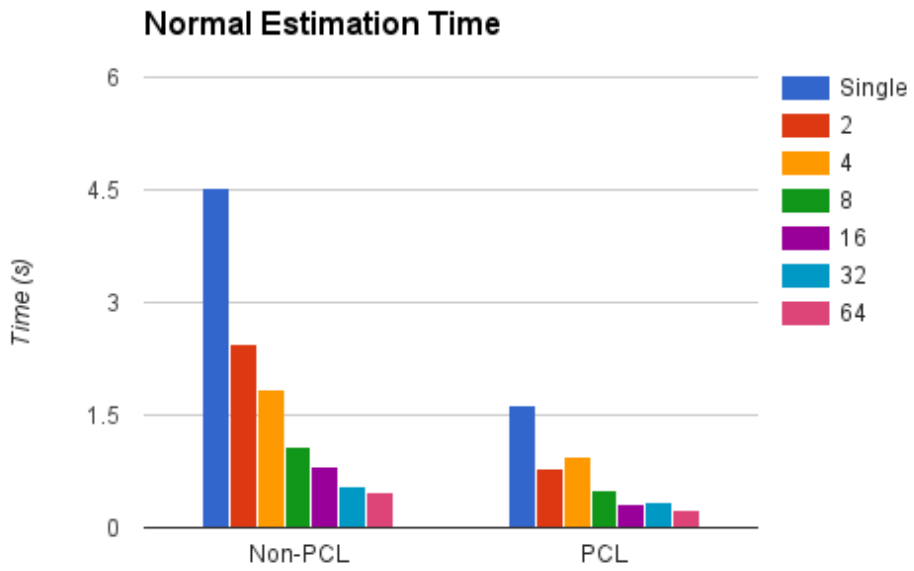

Figure 3. Comparison of OpenMP versions.

Figure 3 shows a comparison of two versions. In figure 3, it can easily be seen that Non-PCL version scales better. However, it can only achieve what PCL version can achieve with 8 threads using 64 threads. This means the code is inefficient for OpenMP parallelization and PCL version is still more efficient.

Looking at table 1, the fastest run time is 0.23 s which is still over 100 ms threshold for near real-time normal estimation. Further parallelization is required. This is where CUDA comes in.

6.3. Comparison of CUDA versions

The CUDA versions prove to provide incredible speedup that can surpass 100 ms threshold with ease. PCL community did not yet implement an efficient kernel wrapper with adjustable block size, however, they implemented methods that can upload data to and download data from GPU device very efficiently. However, non-PCL version implements a very efficient kernel wrapper as well as a very efficient and optimized kernel. Still, non-PCL version lacks the ability to efficiently copy data and takes very long to copy data to and from device. Non-PCL version therefore becomes communication bound.

In this chapter we will investigate performance two ways: (1) Without communication (estimating normals alone) and (2) with communication (copying data + estimating normals).

6.3.1. Comparison of CUDA versions without communication

Since PCL's kernel wrapper can only work with a pre-defined 1D block size of 256, there is no way to check scaling on par with block size. It takes 0.037 seconds to estimate normals. On the other hand, non-PCL version was investigated with different 2D block sizes of 1 to 32 (scaling with a factor of 2). There is a linear scaling in non-PCL version's speedup and it can achieve sub-millisecond timing (0.1 ms at most). This means non-PCL version implements a more efficient way to compute normals with a GPU device.

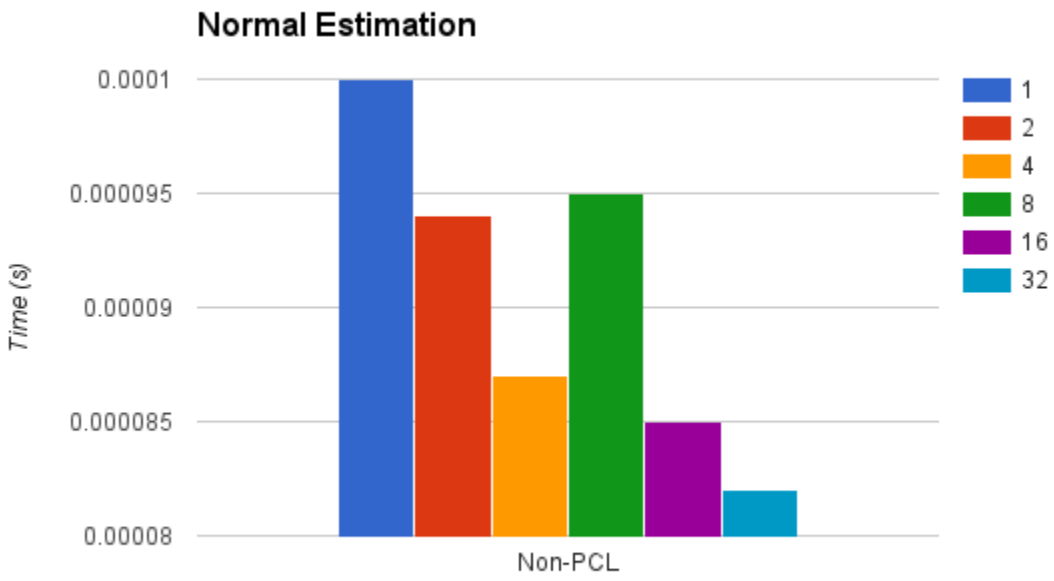

Figure 4. Non-PCL version scaling with different 2D block sizes.

6.3.2. Comparison of CUDA versions with communication

If we add communication on top of normal estimation, the performance comparison changes drastically. PCL version can upload data to and download data from the GPU device in 8 ms (total). Therefore, its communication + computation time is 0.045 s. On the other hand, non-PCL version does not implement an efficient method to copy data to and from GPU device. It takes a very long time to migrate data with the non-PCL version. Since non-PCL version's computation time is negligible compared to

communication time and since communication time does not scale with block size, non-PCL's version's communication + computation time is also stable and it is between 1.3~1.5 range.

A hybrid methodology was not implemented; because this work's scope was to compare PCL vs. non-PCL methods. However, it is seen that with PCL's communication capabilities and non-PCL version's computation capabilities, normal estimation can be achieved under 10 ms which is 10 times faster than what was needed in the first place.

7. Results, Conclusion and Future Work

The results gathered with normal estimation can help with most robotics research. This project helped improve normal estimation speed and the end-product code will help improve robotics research. Point clouds without and with normals are shown in figures 5, 6 and 7.

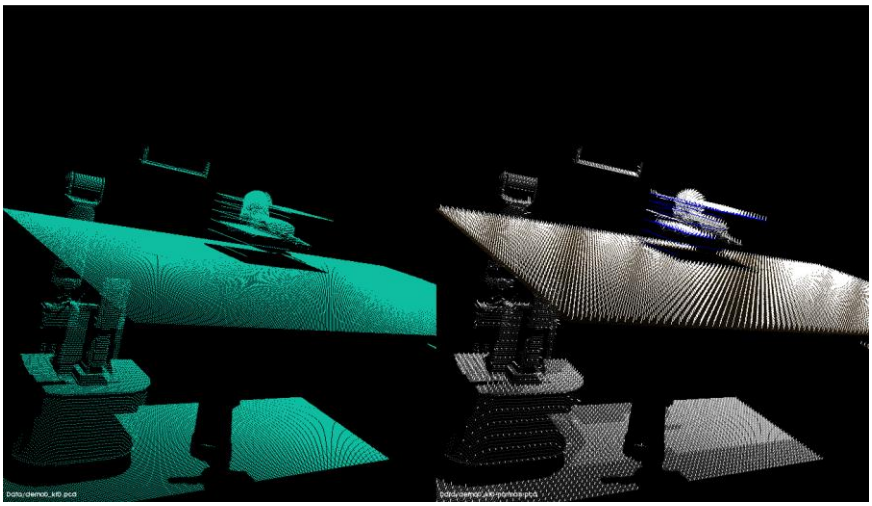

Figure 5. Contents of demo0_kf0.pcd without normals on the left and with estimated normals on the right.

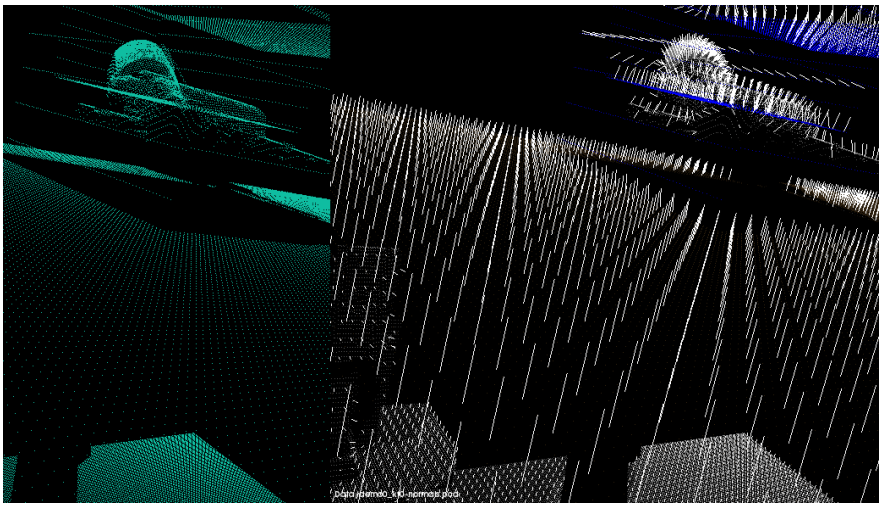

Figure 6. Close up of figure 5.

Figure 7. partial_cup_model_new.pcd with estimated normals.

It was seen during this project that, PCL capabilities are generally more efficient than to write new code, aside from advanced and more complex parallelization methods such as CUDA kernels. Combination of PCL and efficient, more optimal CUDA kernels was proven to provide the most efficient results.

With these in mind, the future work includes estimation of normals directly from data gathered through Kinect simultaneously and also combining PCL's host-to-device and device-to-host communication capabilities with the newly implemented and efficient CUDA kernel.

References

[1] Nüchter, A. (n.d.). Point Cloud Image of Bremen City Center. *Robotic 3D Scan Repository.* Jacobs University Bremen, Bremen, Germany.

[2] Rusu, R. B. (2009, 09 18). Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments. München, Germany: Technische Universität München.

[3] Anonymous. (2011, 03 28). *PointClouds.org: A new home for Point Cloud Library (PCL)*. Retrieved from Willow Garage: http://www.willowgarage.com/blog/2011/03/27/point-cloud-library-pcl-moved-pointcloudsorg

[4] Rusu, R. B. (2011, 05 12). *PCL 1.0!* Retrieved from PCL: http://www.pointclouds.org/news/2011/05/12/pcl-1.0/

[5] PCL. (2017, 01 08). *pcl::io Namespace Reference.* Retrieved from Point Cloud Library (PCL): http://docs.pointclouds.org/trunk/namespacepcl_1_1io.html

[6] PCL. (2017, 01 08). *pcl::search Namespace Reference.* Retrieved from Point Cloud Library (PCL): http://docs.pointclouds.org/trunk/namespacepcl_1_1search.html

[7] PCL. (2017, 01 08). *pcl::NormalEstimation Class Reference.* Retrieved from Point Cloud Library (PCL): http://docs.pointclouds.org/trunk/classpcl_1_1_normal_estimation.html

[8] PCL. (2017, 01 08). *pcl::NormalEstimationOMP Class Reference.* Retrieved from Point Cloud Library (PCL): http://docs.pointclouds.org/trunk/classpcl_1_1_normal_estimation_o_m_p.html

[9] PCL. (2017, 01 08). *pcl::gpu::NormalEstimation Class Reference.* Retrieved from Point Cloud Library (PCL): http://docs.pointclouds.org/trunk/classpcl_1_1gpu_1_1_normal_estimation.html

[10] Hoppe, H. D. (1992). Surface Reconstruction from Unorganized Points. *Computer Graphics. SIGGRAPH 1992 Proceedings*, (pp. 71-78).

[11] Wendland, H. (2017, 01 08). *Computation of Normal Vectors.* Retrieved from University of Bayreuth: http://www.staff.uni-bayreuth.de/~bt300425/research/old/reconhtml/node3.html