Institut für Parallele und Verteilte Systeme

Maschinelles Lernen und Robotik

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 3660

Curvature based Analysis of Point Clouds

Lars-Alexander Albrecht

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Marc Toussaint
Betreuer:	M. Sc. Dmitry Zarubin
begonnen am:	20.05.2014
beendet am:	19.11.2014
CR-Klassifikation:	B.2.1,B.2.2,D.1.7

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Steinheim den 17.11.2014

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

Signature:

Steinheim 17.11.2014

Table of Contents

Abstract. Analyzing of large data and visual information is becoming more and more crucial in today's world. Faster and more accurate analysis for visual data is vital for industries, which depend on visual information, such as robotics and automated drive systems. No matter how the information is obtained, it must be analyzed in order to draw conclusions and create hypothesis. One of these factors, which is to be analyzed are critical points. With the help of critical point analysis, borders of objects in a scene can be found. Here robots could calculate grasping points of the object and use this information for further tasks. In this thesis a pipeline for computing critical area curvature on point clouds is created. The entire information of the processing background will be explained and it will be shown how these interact to create the pipeline. Computations of different normal estimations and curvatures are going to be introduced and compared. Later these will be implemented to a pipeline, which computes critical areas of the scanned point clouds. The thesis concludes with the comparison for real and synthetic point clouds.

1 Introduction

With the Kinect sensor many different applications were created, which would not have been possible a few years ago. One of these is the analysis of point clouds. Point clouds of the environment can be obtained very fast and easy. These point clouds can have different attributes. Whether to cross the point clouds with color, to receive a RGB point cloud or using filters to capture certain object, point clouds allow all of these operations without the need of many adjustments. The Point Cloud Library and the Robot Operating System are two different open source projects, which supply some different applications for working with point clouds. When working with point clouds, different analytical methods can be applied in order to be able to create some hypothesis about the cloud. This is needed, since the point clouds do not have all the information, which a real scene has. Compared to a real scene image point clouds of the same scene can be computed a lot faster. One of the analysis, which has to be made, is the critical point analysis of point clouds. In this thesis the different open source projects are combined, in order to create a visualization pipeline for critical area analysis. The Point Cloud Library and the Robot Operating System are both used, to enable a fast analysis of a scanned point cloud for critical areas. The goal of this thesis is first to compare different methods for normal estimations. With the gained knowledge curvature computations are to be evaluated based on the normal estimations. Finally the curvatures are visualized. All these computations are implemented into a visualization pipeline, which is to be used on point clouds. The point clouds were obtained using the Kinect. The thesis starts with a general introduction to how the Kinect functions and how point clouds are obtained. The Robot Operating System and the Point Cloud Library will be introduced and the functionality behind these technologies will be explained shortly. After the basis has been laid out, the comparison of the different normal estimations will be done. All compared estimation methods, are provided inside the PCL.

2 Introduction to the functionality of the Kinect

The most important component of the entire thesis is the Kinect Sensor from Microsoft. The Kinect Sensor allows the real-time scanning of point clouds either as direct infrared data or combined with the RGB (red, green, blue) camera for colored point clouds. The Kinect has an infrared laser projector and a infrared sensor, which, when combined, can create a 3D point cloud scan of the scene[wik14d]. Scanning point clouds with the laser creates real-time images with approximately 30 fps (frames per second) at a resolution of 300000 points per frame [Kho11]. The range for scanning objects with the Kinect through the infrared laser is between 0.5m and 5m. Within this distance, the point clouds will have the best properties for further calculations and estimations. Points, which are closer than 0.5m, can not be scanned without loss of focus on the other area. Points, which are further away than 5m, are not optimal for the analysis since the depth of a point, which is further away, cannot be computed very accurately. Every point image has a resolution of 640x480 pixels, which results in a decrease of density, if the scanned object is further away. The point density can be computed as follows:

$$p \approx \frac{1}{Z^2} \tag{1}$$

where p is the resulting point density and Z is the distance form the sensor [Kho11]. All the data, which was collected for the thesis, was scanned with a Kinect for Xbox 360 from 0.5m to 4.5m distance. In order to focus on a certain object or area, which is to be scanned, a simple implementation using axis filters was used. Points inside a point cloud carry depth, width and height information of the point. This information is not a global value of the scanned area, since this data is related to the position of the Kinect. The points are vectors with x,y and z values. The input data is first filtered at the x-axis. The filtered point cloud is then passed to the y-axis filter. Now the resulting point cloud is filtered at the z-axis. This cuts all points away, that are larger than the given threshold. The Kinect 360 and the Kinect for Windows are apparently 100 percent identical in hardware, but a few drivers may differ. This limits the use of the Kinect for Windows in Ubuntu. Since the software, that was used in this project, runs stable on Ubuntu, a Kinect 360 was used.

The figure 1 shows the IR (infrared) sensor and the IR projector. The projector and the sensor work together to capture the 3-D depth information. This is done through triangulation. The sensor and the projector both take the same point on the infrared point cluster, which is emitted from the projector. Then the inner angles of the projector and the sensor are computed. Since the space between the sensor and the projector is known, the distance from the Kinect to the object can be computed. Figure 2 underlines this. The red and green angles from the sensor have a direct relationship between the red and green angels from the projector. The gray area shows, how objects can be lost in the depth image. When the projector or the sensor scan different objects, these shadows can be created. In the depth image this is shown as a black surrounding of the object, which lies in front of the other objects in the scene.

Image 3 shows a camera scan of depth information. The first picture shows the original data. Here you can see a person standing in front of a scene. The outline of the person

Fig. 1: shows the Kinect sensor . The positions of the IR Projector and IR Sensor are shown, as well as the position of the RGB Camera. The distance between the IR Sensor and the IR Projector are needed for the depth triangulation. [b]

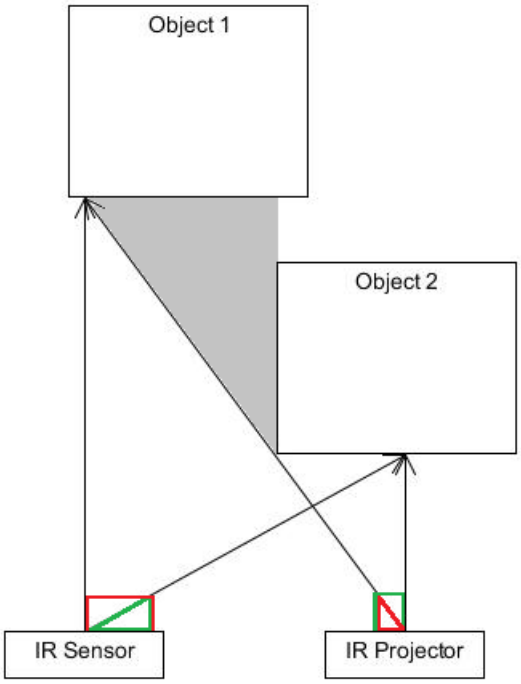

Fig. 2: shows the theoretical capturing of depth information. The objects are at different distances from the Kinect. With the help of triangulation, the distances for each point in the cloud from the Kinect can be computed. When objects block a part of another object, a shadow is shown inside the point cloud. This effect is shown through the grey area inside the figure.

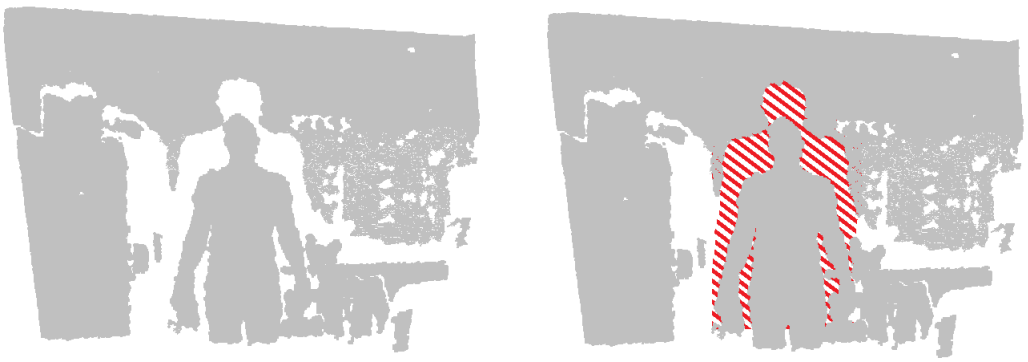

Fig. 3: shows a real point cloud, which was captured with the Kinect. In the left picture you can see a person standing in front of a scene. The person seems to have a shadow, which surrounds him. The red area inside the right picture shows the points, which can not be captured from the objects, which are behind the person.

in the figure can also be seen. The second image shows the area, which can not be scanned and where data loss occurs. This is due to the fact that the object, in this case the person, is to far away from the other objects. Since the infrared scan only emits from one point, the information, which lies directly behind and around the person, is lost. When scanning point clouds with the Kinect this has to be taken into account. This is often critical, because in small scale objects may be lost, when the scene is scanned from the wrong angle.

3 The Robot Operating System

The Robot Operating System (ROS) is an open source project, which provides a set of tools to create software for different robotic applications [ROS14b]. The framework is based on publish and subscribe, which allows the modularization of the point cloud analysis. In the following figure 4

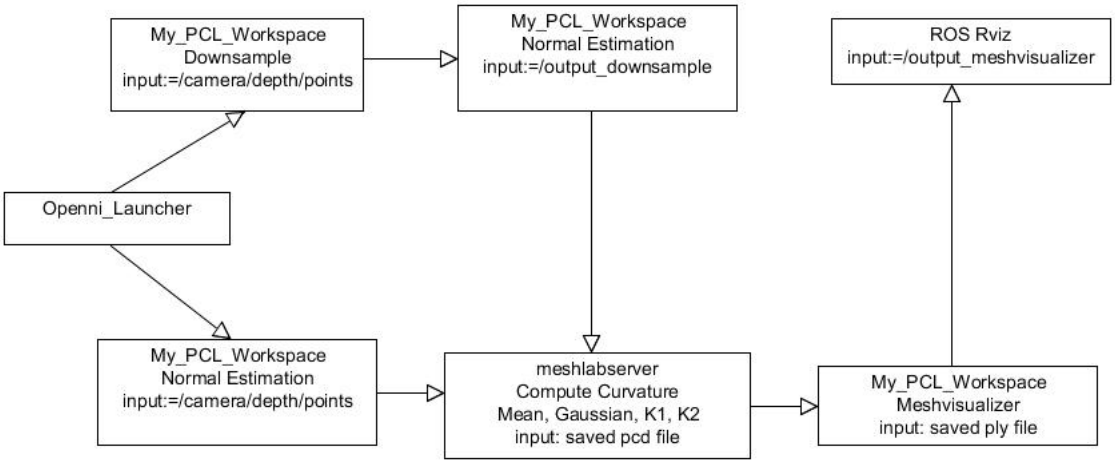

Fig. 4: shows the pipeline with the different topics, from which the information for the points is passed. Starting with the Openni_Launcher, which captures the data from the Kinect. This data is passed either to get down sampled or to have its normals estimated. The input here is the output of the previous topic. In this case it is the Openni_Launcher. The rest of the pipeline shows the topics for the computation of the curvature and the computation of the radii for the critical area analysis.

one can see, how the different nodes communicate with each other and how the information is passed using different topics. Each nodes input, which is not a pcd or ply file, is the topic, which the node listens for. The topic is passed from the previous node. All the data in the thesis were obtained using ROS-Groovy. ROS has its own implementation of the PCL (Point Cloud Library), which in some cases may cause transformation errors when handling different point cloud data-types.

4 Point Clouds and PCL

But first of all: What is a point cloud? According to the following source [wik14e] a point cloud is a certain amount of points within a vector space, which show a certain unorganized structure. A point cloud is hereby described through the contained points, which are given through their space coordinates. The point cloud library "is a standalone, large-scale, open project for 2D/3D image point cloud processing" [PCL14b]. It provides different libraries for different calculations and manipulations on point clouds. All the data in the thesis were obtained using PCL-1.6. Mainly filters and normal estimations were implemented to gain data for the different experimental comparisons. In the following sections the different data-types for point clouds will be looked at briefly. Figure 5 below is a scan of a point cloud with an already reduced point cloud density. Even though the points are relatively far away from each other, the object can still be identified as a armchair with a ball on it.

Fig. 5: The Point Cloud consists of 2658 points. This image was obtained through down sampling a point cloud with 307200 points

4.1 Different Point Cloud Data Types

The PCL library uses different point types for different computations. At first this caused some difficulties implementing the different test classes, which were needed for the comparison. The main problem was the Kinect input data type, which needed to be converted to do the needed computation and then reconverting the resulting data to the Kinect data-type. In some cases this was not achievable and therefore no live data could be used. This resulted from the fact, that the live data was not compatible with the functions, which were needed to compute normals, curvature etc. When saving the live data to a pcd (point cloud data) file format and the file was loaded, the different

computations were possible. The following enlist the different point cloud data types, which were used in the thesis:

- PointCloud2
- PointXYZ
- PointNormal
- PointCloudXYZRGB ...

The **PointCloud2** data-type is the ROS (Robot Operating System) specific point cloud. All information about the point is stored inside this data structure. This data-type contains information such as: point coordinates stored in a point field, height and width of the point cloud, length of a point in bytes, whether the cloud contains valid and invalid points, offset from the starting position, normal orientation and color information. Of course not all of this information is present from the start. However within ROS this data-type is used to communicate with different nodes and topics.

PointXYZ is a standard data-type used in PCL. The PointXYZ contains only the float values for the x,y and z coordinates. This data-type is needed for the simple normal estimation method included in the PCL. It is also the data-type, which was used for reading in most saved pcd files in the implementation.

The **PointNormal** data-type is a special type of point in the PCL. The PointNormal stores the x,y,z coordinates of the point as well as a 3-dimensional vector. This 3-dimensional vector is the computed normal vector of the point. In this thesis the PointNormal is used to combine the original point cloud with the computed normals. Through this possibility the original cloud can be shown with the normals in each point without loss of points.

The last data-type, which was used in the thesis, was the **PointCloudXYZRGB**. This special point cloud was used in the colorization process. Besides the X,Y and Z coordinates, the RGB values from 0 to 255 are stored for each point.

5 Introduction of Normal Estimation Methods

To be able to calculate the areas of interest, whether using the Gauss or the Mean Curvature algorithms, the normals of the point clouds have to be computed in advance. When the normals are precomputed with the PCL normal estimation methods, the run-time of the colorization is sped up. In the following sections the different normal-estimation methods, which were used with the Kinect point clouds, will be introduced.

5.1 Simple Normal Estimation

For the comparison of normal estimation methods the most basic should not be skipped. The normal estimation used computes the surface normals directly from the point cloud. The basic idea behind the estimation of the normal here is, that a normal is estimated on the tangent plane of the underlying surface [PCL14c]. Figure 6 shows a tangent plane on a curved surface with the normal N and the tangent vectors v and u.

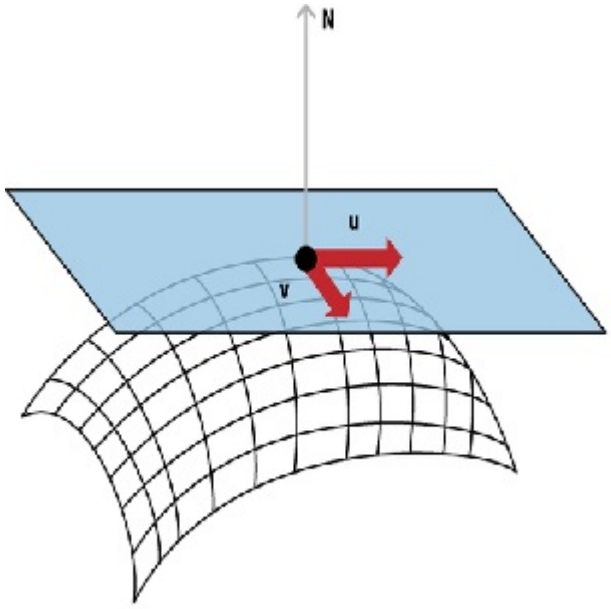

Fig. 6: The tangent plane (blue) lies on a curved surface with the normal of N and the different tangent vectors v and u source [c]

A tangent plane is defined as follows: Let (x_0, y_0) be any point of a surface function

$$z = f(x, y). \tag{2}$$

Then the surface has a non tangent plane at (x_O, y_O) with the equation:

$$Z = f(x_O, y_O) + f_x(x_O, y_O)(x - x_O) + f_y(x_O, y_O)(y - y_O) \tag{3}$$

[SHN09] shows, how the normal estimation can hereby be broken down into a covariance matrix analysis of its eigenvectors and values, considering the nearest neighbors. The resulting covariance matrix is:

$$C = \frac{1}{k} \sum_{i=1}^{k} (p_i - \bar{p}) * (p_i - \bar{p})^T \tag{4}$$

where

$$C * \overrightarrow{v_j} = \lambda_j * \overrightarrow{v_j} : j \in \{0, 1, 2\} \tag{5}$$

k are the number of neighbors, which are in the neighborhood of p_i. The centroid p is the centroid, which is formed around the area of the nearest neighbors. λ_j is the j-th eigenvalue of the covariance matrix vj of the h-th eigenvalue. In order to allow fully understand what will be done, the definitions of the eigenvector and eigenvalue will be introduced.

If V is a vectorspace over scalar K and a linear transformation $f : V \to V$ exists, then an eigenvector $v \neq 0$ of f exists, if there is a scalar λ in K such that $f(v) = \lambda * v$. The equation is called the "eigenvalue equation" of f and λ is the eigenvalue of f corresponding to the eigenvector v [wik14a].

Searching for the nearest neighbors is the last thing, that is needed to compute the normals. The search for the nearest neighbor is done using a K-D Tree.

A K-D tree is a k-dimensional tree, which organizes points in k-dimensional space [KDT]. The basic search for a nearest neighbor is done as follows: Since the K-D Tree is based on the binary space partitioning tree, it has multiple branches, which will be visited in a recursive manner. Starting at the root node the algorithm moves down to the branch to the first leave node. That leave node then is saved as the momentary nearest neighbor. Then the parent node is visited. If the parent node is closer than the current nearest neighbor, it is saved. This is done until the root node is reached. The tree is build up on splitting planes and the algorithm checks, if there are any other points, that are closer than the current nearest neighbor on the other side of the plane. Finding the nearest neighbor has been optimized, such that the time complexity is in $\mathcal{O}(logN)$, where N is the number of points in the point cloud. Considering all of the above one final step has to be taken, if the normals are to be oriented in a consistent way. If the viewpoint is known, the normals can be oriented towards the viewpoint. If no viewpoint is known, the default point (0,0,0) will be used as the viewpoint.

The resulting algorithm does the following;

Listing 1: Simple Normal Estimation

```
In{Point Cloud}
Out{Point Cloud with computed normals}
   For{each point p in cloud P}{

         1. Calculate the nearest neighbors of p.

         2. Compute the surface normal of p.

         3. Check if the normal is consistently
               oriented toward the viewpoint.
}
```

For the algorithms the runtime was calculated with regards to the input cloud size and the radius. The radius is computed from the pre-created K-D tree and is passed to the normal computation. This information is available in cm. The scanned Object consists of 307200 points. Table 1 shows these different runtimes. These runtimes are extremely high in comparison to the cloud size. As one can easily notice, the runtimes increase tremendously with the radius of the nearest neighbor search. To emphasize these runtimes, the three figures 7,8,9 show the average, min and max runtimes for their respective radius. As one can see, the difference in the max and min values seem to be about the same percent for all radii.

Table 1: Simple Normal Estimation Runtimes [s]

Specification	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Avg Runtime
1	1,4880	1,4892	1,5067	1,5319	1,5028	1,5037
3	8,7910	8,6918	8,5493	8,5049	8,5514	8,6177
5	22,2429	22,4472	22,4941	22,2436	22,1929	22,3242

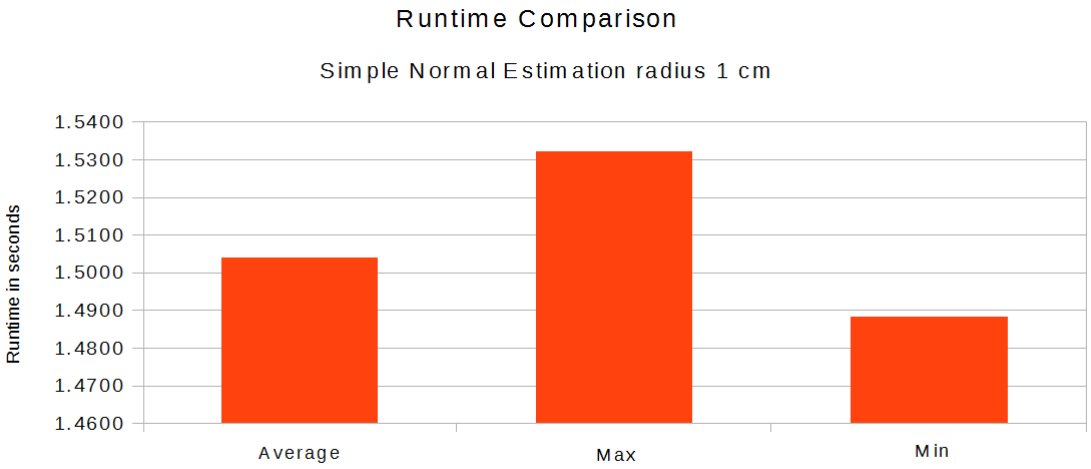

Fig. 7: Simple Normal Estimation runtimes with 1 cm radius

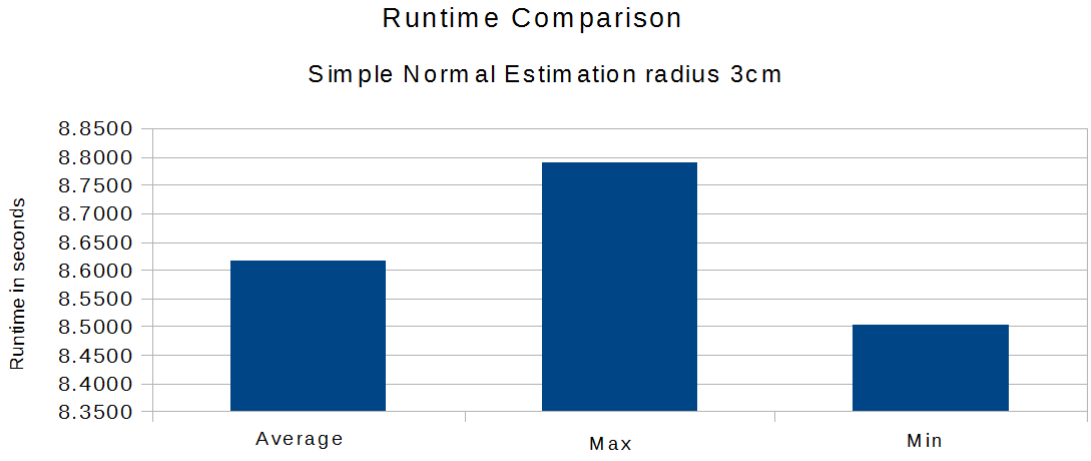

Fig. 8: Simple Normal Estimation runtimes with 3 cm radius

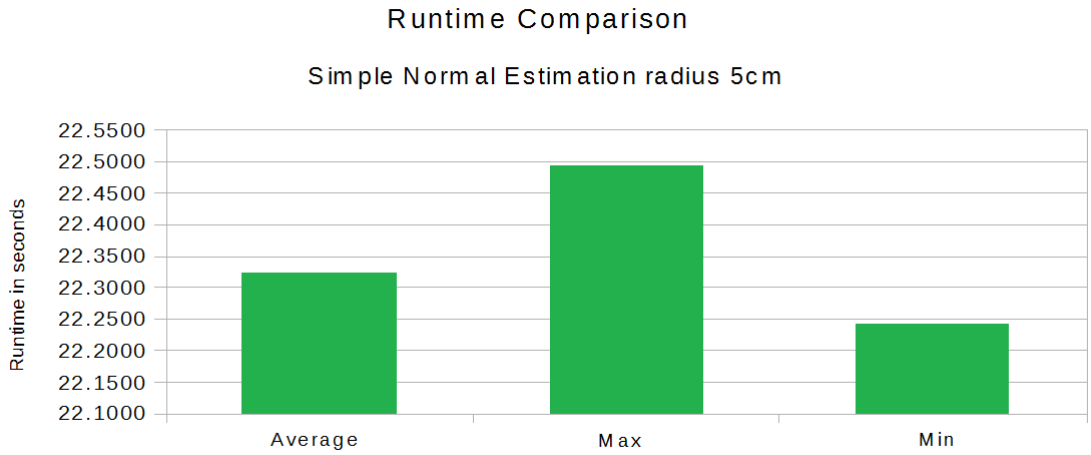

Fig. 9: Simple Normal Estimation runtimes with 5 cm radius

5.2 Parallelized Normal Estimation

Considering the results in table 1 a faster computation for normals must be found. To speed up the normal estimation, the first idea is to use more computing power through parallelization. With OpenMP (Open Multi-Processing) this is achievable. OpenMP is an API (application programming interface) for shared memory processing. The parallelization is based on multi-threading and parallel message passing [ope14]. PCL provides a OpenMP implementation for the simple normal estimation algorithm. The implementation apparently is able to speed up the normal estimation with a standard set up of eight cores about six to 8 times [PCL14a]. However as table 2 shows, the expected faster computation times were not nearly achieved. The object, on which the parallelized implementation was tested, was the same as in the simple normal estimation. The specification shows a tuple (a, b), where a resembles the amount of threads and b the radius (in cm) of the nearest neighbor search. The max, min and the average values, have the same percentual difference as for the simple normal estimation.

Table 2 shows some aspects, which could not be accurate, since more threads should result in faster computation times. This is due to the fact, that the input point clouds are not nearly large enough for the multi-threading to reach its potential. The largest point cloud, which was used, had 307200 points. This is the highest single image resolution, which could be achieved through the used Kinect. It should be mentioned, that all experimental data is taken directly form the camera and not manipulated before computation.

Table 2: Paralellized Simple Normal Estimation Runtimes [s]

Specification	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Avg Runtime
2,1	0,6787	0,6563	0,6621	0,6572	0,6550	0,6619
2,3	3,1321	3,1252	3,1374	3,1314	3,1372	3,1327
2,5	7,8661	7,8719	7,8750	7,880	7,9459	7,8878
4,1	0,6394	,63905	0,6392	0,6492	0,6455	0,6425
4,3	3,1017	3,1013	3,0955	3,0930	3,1010	3,0985
4,5	7,8679	7,8774	7,8814	7,8810	7,8739	7,8763
6,1	0,6365	0,6444	0,6394	0,6415	0,6370	0,6397
6,3	3,0991	3,0967	3,0943	3,0954	3,0975	3,0967
6,5	7,8747	7,8724	7,8855	7,8971	7,8772	7,8814
8,1	0,6426	0,6354	0,6381	0,6425	0,6376	0,6392
8,3	3,1032	3,0835	3,0957	3,0936	3,0934	3,0939
8,5	7,8746	7,8840	7,8700	7,8816	7,8807	7,8782

5.3 Normal Estimation Using Integral Images

The simple normal estimation and the normal estimation using OMP are not able to perform the normal computation in a manner, which could be referred to as "realtime". Considering the runtimes, which were achieved, even faster methods are needed. In the following section 5.3.1 the normal estimation method "Normal Estimation Using Integral Images" are introduced. In order to use any of the integral image estimation methods, the input cloud may not have any non-assigned values. These are values, which are created when filtering the point cloud. To show how these values affect the computation of an integral image, the basic structure of such has to be explained.

5.3.1 Introduction Integral Images An integral image (IO) of an image O is the sum of all values in a certain region of O [SHN09] [Der07]. An integral image is the same as a summed area table. The integral image is defined as

$$I_O(m,n) = \sum_{i=0}^{m} \sum_{j=0}^{N} O(i,j) \tag{6}$$

where $O(0,0)$ and $O(m,n)$ create a rectangular area. In order to find any area inside the integral image all that needs to be done, is to compute the corners of the rectangular region of IO. This results in a runtime complexity of $\mathcal{O}(1)$ [Sou14].

The next step after computing the integral image is to apply a filter to the input cloud to reduce noise. Noise inside a point cloud are points, which are superfluous. The process for reducing noise in a point cloud is called smoothing [NJM03]. When using integral images the duration for smoothing is not dependent on the size of the area, which is being filtered. This is due to the fact, that no mater how large the area is, the same amount of memory is needed to compute the integral image. Since no access memory is needed for larger areas, different indicators may be used to manipulate the area of smoothing. One of the first ideas for indicators, that can be used for smoothing, is the depth of a point. Choosing the depth as a smoothing factor is a logical choice, since close objects have a better signal to noise ratio than objects, which are further

away. The signal to noise ratio is a scientific measurement, which compares a signal strength to the amount of background noise [wik14f].

The depth itself however is not sufficient enough. If one would only filter according to the depth values of a cloud, depth changes, which occur at object borders, would also be filtered. This in turn would result in bad normal estimations, since the surfaces would be blurred. To fix this issue, Holzer, Rusu and Dixon propose to set the size of the smoothing area in dependence on the depth changes, which occur in the area. They then combine both, the depth and the depth change indicator, to produce a "Smoothing Area Map". Holzer, Rusu and Dixon approximated the following function for their depth change factor with d being the depth value.

$$f_{DC}(d) = \alpha * d^2 \tag{7}$$

In their results they proposed, that α should be approximately 0.0028. The smoothing area map results as

$$\beta(m,n) = \beta * f_{DC}(D(m,n)) \tag{8}$$

β is used to control the smoothing area size. $D(m,n)$ is the resulting depth value. Now a depth change indicator map could be computed with a threshold from the previously computed depth map. A problem, which arises now, is that the threshold would only hold to a specific depth. In order to have a dynamic threshold, which applies to different depths, Holzer, Rusu and Dixon create a "Binary Depth Change Indication Map", which uses a "depth change detection threshold". The threshold is dependent on the distance and is computed through the following:

$$t_{DC}(d) = \gamma * f_{DC}(d) \tag{9}$$

γ here is a factor, which defines the sensitivity of the depth change. Later the depth change indication map is computed. Combining the depth change indication map C and the smoothing area map B they compute a final smoothing area map R.

$$R(m,n) = min(B(m,n), \frac{\tau(m,n)}{\sqrt{(2)}} \tag{10}$$

Finally the normals are estimated with the help of the smoothed depth changes. The normal vector is computed through the vectors of the left and right neighbors and the vectors from the neighbors above and below the point. Calculating the cross product of these two vectors results in the normal vector of the point. The vectors are smoothed before the computation using the smoothing area map. This reduces the noise, but keeps the boundaries of the objects in place. With the smoothed vectors the normals are computed using the cross product like in the original normal computation, see Holzer, Rusu and Dixon for more details.

5.3.2 Average 3D Gradient The Average 3D Gradient is another normal estimation method based on integral images. In this case 6 integral images are needed. Three

images are needed to compute the horizontal smoothed versions of the normals and the other three images are needed to compute the vertical smoothed normals. The computation is again the cross product between these two.

5.3.3 Average Depth Change The normal computation, using the Average Depth change, creates only a single integral image. The normals are then computed from the average depth changes in the neighborhood of the individual points.

5.3.4 Covariance Matrix To compute normals with the help of a covariance matrix, the eigenvector of the matrix Cp of a point p have to be calculated. In order to compute the covariance matrix a nearest neighbor search has to be done in prior to calculating the eigenvectors. The neighborhood of a point p is found by computing its nearest neighbors. This operation is expensive. Instead it is possible to compute covariance matrices through integral images. The algorithm, which is described in the work of Holzer, Rusu and Dixon, does the following: Nine integral images are needed; I_{P_X}, I_{P_Y}, I_{P_Z}, $I_{P_X X}$, $I_{P_Y Y}$, $I_{P_Z Z}$, $I_{P_X Y}$, $I_{P_X Z}$, $I_{P_Y Y}$, $I_{P_Y Z}$ and $I_{P_Z Z}$ "Here $I_{P_A B}$ is the element-wise multiplication of I_{P_A} and I_{P_B}" [SHN09]. The resulting covariance matrix results into:

$$C_p = \begin{pmatrix} cxx\ cxy\ cxz \\ cyx\ cyy\ cyz \\ czx\ czy\ czz \end{pmatrix} - \begin{pmatrix} cx \\ cy \\ cz \end{pmatrix} * \begin{pmatrix} cx \\ cy \\ cz \end{pmatrix}$$

where

$$cxx = s(IPxx, m, n, R(mn)) \tag{11}$$

$$cxy, cyx = s(IPxy, m, n, R(mn)) \tag{12}$$

$$... \tag{13}$$

$$czz = s(IPzz, m, n, R(mn)) \tag{14}$$

Then the eigenvector of the matrix is computed. The resulting eigenvector is the normal in point p.

6 Description of different Curvature Estimation Methods based on MLS

The next step after computing the normals of a point cloud is to compute the curvature of the cloud. The curvature allows us to view critical points in the point cloud. Different types of curvatures will show different aspects of the point cloud and enable different estimations on the same data set. The curvature is shown through different color values, which are implemented inside the curvature computation algorithms. The colorization in the implementation and the resulting experiments are obtained using the MLS (Moving-Least-Squares) curvature estimation algorithm within Meshlab.

6.1 MLS

For computing the Moving-Least-Squares of a point cloud, the input data must be precomputed. This means, that the normals in each point have to be calculated prior to running the MLS-Algorithm [YQ07]. Normals are needed, so that the orientation of the underlying surface can be estimated. Colorization is dependent on the orientation of the normals, since the different colors determine how the curvature, in relation to the object, is oriented. To get an estimation of the surface, the different compositions need to be calculated. To calculate these compositions, the eigenvectors of the covariance matrix of a point are analyzed. In a 3-D point cloud a centroid is used to determine the covariance matrix of a certain point. With the resulting three dimensional covariance matrix the eigenvalues and eigenvectors can be obtained [Koc78].

$$C = \frac{1}{k}\sum_{1}^{K}(p_i - \bar{p}) * (p_i - \bar{p})^T, p_i \in N_p \tag{15}$$

C is symmetrical and positive. Due to these facts, all eigenvalues are real. To obtain the eigenvector the following formula has to be examined.

$$C * vl = \lambda * vl, l \in 0, 1, 2 \tag{16}$$

Hereby λ is the eigenvalue of the eigenvector vl. Given that we compute the formula for three values, three λ values will be obtained. Estimating the surface variations of a point p in a neighborhood, n can be described by looking at the deviation from a point p of the underlying tangent plane [PGK02].

$$C * vl = \lambda * vl, l \in 0, 1, 2 \tag{17}$$

$$\sigma_n(p) = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2} \tag{18}$$

All that the MLS has done till here is estimate the surface normals with the help of an underlying surface, which was projected with the help of the covariance matrix and its eigenvalues. The visualization of the curvature can now be calculated.

6.1.1 Simple Curvature, K1 and K2 The simplest idea to measure the curvature of a point is to measure, how the tangent line changes, when the point is moved toward the neighboring points. This however is only applicable in R^2. When computing curves in R^3 the normal curvature rely on a plane O. This plane has a surface normal U and the tangent plane tp in a certain point p. Both the tangent tp and p have to reside inside O. Now assuming that p is in R^3, every tangent direction is assigned to the normal curvature. The tangent direction is the direction, which the tangent vector can have at a given point p [wik14b]. This is considered the normal curvature. The minimum and maximum values of the normal curvature are the principal curvatures $k1$ and $k2$. These can also be obtained using the "Weingartenmap". The "Weingartenmap" is based on the inner product space. This is a vectorspace with an inner product. The inner product is the scalar product over the vectorspace and hereby provides all computations for a pair of vectors in the given space. With this in mind, the shape operator(Weingartenmap) Sx can be defined [wik14g].

$$(S_x v, w) = (\delta f(v), w) \tag{19}$$

v and w are here the tangent vectors of point p.

6.1.2 Gaussian Curvature The eigenvalues of Sx result in the principal curvature. After computing $k1$ and $k2$, the mean and Gaussian curvatures can be computed. The Gaussian curvature K is the product of the principal curvatures $K = k1 * k2$. With the help of the Gaussian curvature some analysis of the underlying object can be made. If $k1 * k2$ are greater than 0, the resulting Gaussian curvature is positive. If this is the case, the underlying surface is elliptic. If $k1 * k2$ are smaller than zero, the resulting Gaussian curvature is negative. The underlying surface is hyperbolic. The last case has a curvature of 0. This results when $k1 * k2 = 0$. This can occur either at a parabolic surface or when the surface is planar [DM99] [JP03] [Jia13]. Figure 10 below shows three of the four different cases.

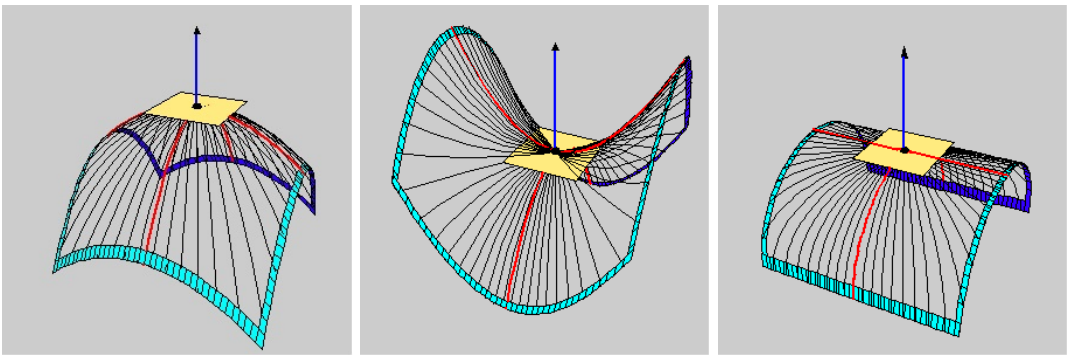

Fig. 10: The first image shows a elliptic curvature, the second image shows a hyperbolic curvature and the third image shows a parabolic curvature [a]

6.1.3 Mean Curvature Computing the mean curvature also allows assumptions about the underlying object. The mean curvature can be used to identify sudden changes in the curvature of a surface or object.

Hereby the mean curvature H is calculated as follows:

$$H = 1/2 * (k1 + k2) \tag{20}$$

7 Experimental Setup for the Pipeline

7.1 Installation Issues

In this thesis multiple examples where implemented and tested in order to find a suitable workflow, which finally allows the analysis of point clouds. As mentioned earlier, the experiments were all executed on a laptop with an i7 processor under Ubuntu 12.04. To get the point cloud input, a Kinect 360 with the corresponding openni drivers was used. As one might have already guessed, the experimental workflow is based on the following pattern as shown in figure 11.

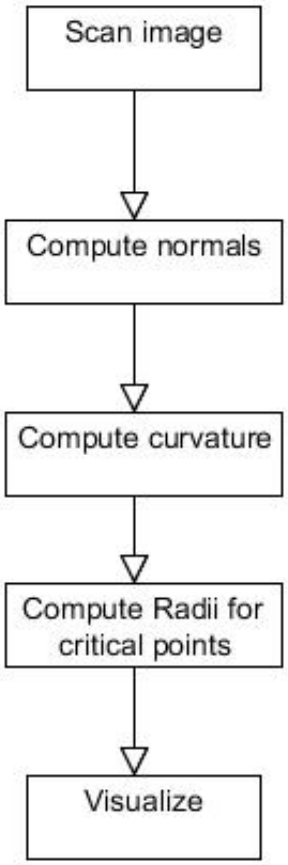

Fig. 11: The image shows the general workflow of the pipeline. Starting with the scanning of the image, computing the normals, computing the curvature, computing the radii and finally visualizing.

The first and major problem was installing and setting up of the environment, in order to get some input data. The problems ran from wrong hardware, to difficult to solve software dependency issues, up to not functioning drivers. The systems, which were tested, were Ubuntu 14.04 LTS, 13.10, 12.04 LTS, and Windows 7. Apart from the

different systems, three different input drivers were installed and checked for compatibility. These included the openni driver, the freenect driver and the primesense driver for Windows and Ubuntu. Furthermore two different Kinects, the Kinect 360 and the Microsoft Kinect, were used. With the different operating systems, different versions of the ROS were also tested, as well as different versions of the PCL. The final working setup, which was used, was: Ubuntu 12.04 LTS, Kinect 360, ROS-Groovy, PCL-version 1.6 and Openni. With the setup at hand, the first idea was to use live data for all the experiments. The problem with live data is the compatibility and conversions between all the different components, which were used. Live data was not possible to scan and compute within ROS itself, since ROS has a close interaction with PCL and uses a lot of its functionality. So live data was not an option and it was decided, that it would be the best to run the experiments from saved data. The saved data was obtained by scanning in and saving the point cloud directly to the pcd file format. Then the pcd files were used for the different experiments.

7.2 Different Normal Estimations

The first experiment was the simple scanning and resizing of the point clouds. The standardized point cloud has a size of 300000 points, which was to big to do the computations on. This resulted from the limited amount of RAM capacity on the experiment laptop. For the experiments the cloud was resized to 56850 and 811 points. The objects were between scanned 0.5m and 4.5m away from the Kinect. The scanned objects were saved for the next step. From here different normal estimation methods were implemented and tested on the saved point clouds. For the normalization the basic setup was the same. At first was the loading of the pcd file into the cloud. A K-D tree was implemented for calculating the surrounding neighbors and then the different normalization methods followed. First of all was the simple normal estimation. The different runtimes are shown in the table 3. Here the object, which was used for the normal estimation, consisted of 307000 points. The numbers in the specification show the radius in cm of the nearest neighbor search. As one can easily spot, there are large

Table 3: Simple Normal Estimation Runtimes [s]

Specification	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Avg Runtime
1	1,4880	1,4892	1,5067	1,5319	1,5028	1,5037
3	8,7910	8,6918	8,5493	8,5049	8,5514	8,6177
5	22,2429	22,4472	22,4941	22,2436	22,1929	22,3242

differences in the used radii. The runtime is dependent on the amount of neighbors, which are used in the normal computation. The resulting normals are shown in pictures 12, 13 and 14. The best and most spread normals are achieved using the higher radii. The lower the radii are, the more uneven the normals seem. This effect occurs especially at the borders of the planes.

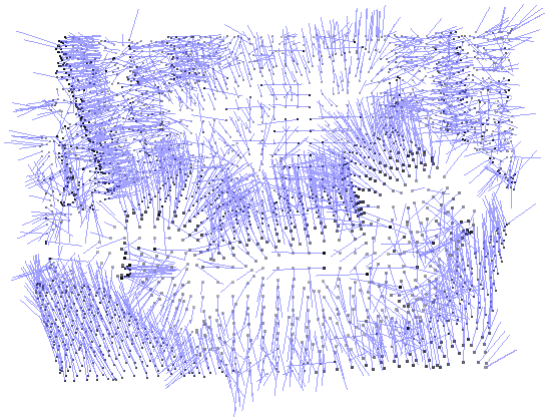

Fig. 12: An armchair, which was down sampled to 2568 points. The image shows the normals computed from a radius of 5cm.

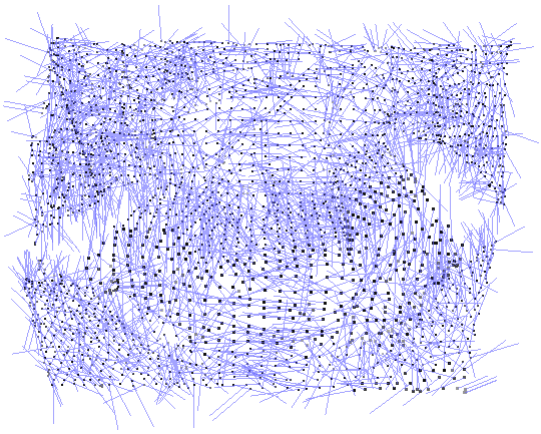

Fig. 13: An armchair, which was down sampled to 2568 points. The image shows the normals computed from a radius of 3cm.

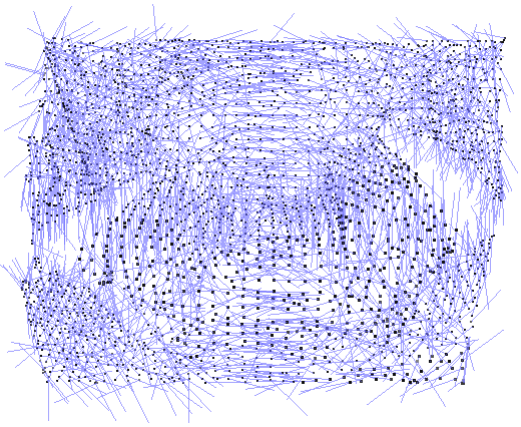

Fig. 14: An armchair, which was down sampled to 2568 points. The image shows the normals computed from a radius of 1cm.

The second tested normal estimation method was the parallel implementation with the OMP framework. Here the normal estimation is sped up according to the amount of threads specified in the program. The amount of threads, which were used, was dependent on the hardware, it was run on. This limits the amount of possible threads, which in this case was 8. For the usage of the comparison only even numbers of threads were used. Table 4 shows the resulting runtimes. What one might notice immediately is, that the runtimes do not seem to be faster the more cores are participating in the computation. This results from the size of the used point cloud. The largest point cloud, that was analyzed, had 307200 points. To efficiently use the parallel computation, the point cloud size would have to be much larger. For parallelization the multi-threading has to go through the K-D tree for each point and assign each point normal computation to a thread. The passage through the K-D tree takes more time in comparison to the normal computation, when the point cloud size is relatively small. The table 4 should be read as follows: Specification shows a tuple (a, b), where a resembles the amount of threads and b the radius (in cm) of the nearest neighbor search.

Table 4: Multithreaded Normal Estimation Runtimes [s]

Specification	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Avg Runtime
2,1	0,6787	0,6563	0,6621	0,6572	0,6550	0,6619
2,3	3,1321	3,1252	3,1374	3,1314	3,1372	3,1327
2,5	7,8661	7,8719	7,8750	7,880	7,9459	7,8878
4,1	0,6394	,63905	0,6392	0,6492	0,6455	0,6425
4,3	3,1017	3,1013	3,0955	3,0930	3,1010	3,0985
4,5	7,8679	7,8774	7,8814	7,8810	7,8739	7,8763
6,1	0,6365	0,6444	0,6394	0,6415	0,6370	0,6397
6,3	3,0991	3,0967	3,0943	3,0954	3,0975	3,0967
6,5	7,8747	7,8724	7,8855	7,8971	7,8772	7,8814
8,1	0,6426	0,6354	0,6381	0,6425	0,6376	0,6392
8,3	3,1032	3,0835	3,0957	3,0936	3,0934	3,0939
8,5	7,8746	7,8840	7,8700	7,8816	7,8807	7,8782

What immediately comes to mind, when taking a look at table 4, is, that the computation times for larger radii take more than one second. In order to create a pipeline, which allows the analysis of the point cloud, faster computation times for the normal estimations are needed. For the experiments the next step was the down sampling of the point clouds. This was done through a voxel grid filter. The voxel grid filter is part of the PCL and works as follows: The voxel grid consists of small 3-D boxes, which are placed on the point cloud. These boxes have a certain size, which can be specified inside the implementation. All the points, which are inside one of the voxel grid elements, are merged into one at the center of the box. This reduces the amount of points inside the point cloud. This is sufficient for the simple normal estimation. With the voxel grid two smaller input point clouds were created. The first is the CameraExampleScan.pcd, which consists of 56850 Points, and the second is the CameraExampleScan1000.pcd, which consists only of 811 Points. To receive an idea of the difference in the point cloud, figure 15 shows the three different sizes in comparison from left to right, with left being the densest and right being the loosest point cloud.

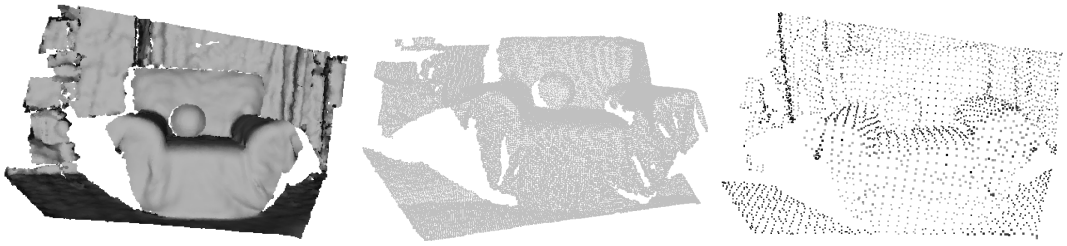

Fig. 15: From left to right: Cloud with 307200 points, the same cloud down sampled to 43937 points and the cloud down sampled to 2658 points

The runtime comparison was done with these down sampled point clouds as well. First the simple normal estimation was tested to see, if the improvements were sufficient enough. The table 5 shows the runtime of the simple normal estimation on two different down sampled clouds. CES.pcd consists of 56850 points, while CES100 consists of only 811 points.

Table 5: Simple Normal Estimation Runtimes [s] Comparison With Down Sampled Cloud

Object	Specification	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Avg Runtime
CES.pcd	1 cm	0,14383	0,1428	0,1444	0,1422	0,1413	0,1429
CES.pcd	3 cm	0,3376	0,3360	0,3353	0,3398	0,3470	0,3391
CES.pcd	5 cm	0,7497	0,7497	0,7571	0,7756	0,7840	0,7632
CES1000.pcd	1 cm	0,0015	0,0015	0,0015	0,0015	0,0016	0,0015
CES1000.pcd	3 cm	0,0015	0,0015	0,0018	0,0015	0,0016	0,0016
CES1000.pcd	5 cm	0,0015	0,0018	0,0019	0,0016	0,0016	0,0017

With the down sampled point clouds the runtimes were sped up, so that the resulting normal estimations took less than 1 second. However the larger radii still took too much time to be used for "Realtime". The logical next step was to try the multi-threaded approach on the down sampled point clouds. The tables 6 and 7 show the results of the multi-threaded approaches on both down sampled point clouds. The initial acceleration in speed, is increased just like in the none down sampled case. But the increase off multiple processors did not change the results, since the point clouds contain even less points, and the multithreading can not efficiently use its potential. The following tables 6 and 7 show the runtimes for both down sampled point clouds with different amounts of threads. Table 6 shows the resulting runtimes from a point cloud with 56800 points. Table 7 shows the runtimes from a point cloud with 811 points. The specification shows a tuple (a, b), where a resembles the amount of threads and b the radius (in cm) of the nearest neighbor search.

The tables 6 abd 7 only show the runtimes, but to make a serious statement, the resulting normals have to be analyzed. To compare the normal estimations, the pictures have to be analyzed. Image 18 shows the normals computed with the radius of 5 cm. The error rate within this picture is low enough, so that, if it would be used for further

Table 6: Multithreaded Runtimes [s] On Down Sampled Cloud With 56800 Points

Specification	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Avg Runtime
2,1	0,0869	0,0856	0,0887	0,0885	0,0771	0,0854
2,3	0,1504	0,1522	0,1544	0,1577	0,1566	0,1543
2,5	0,2941	0,2890	0,2872	0,2855	0,2907	0,2893
4,1	0,0795	0,0824	0,0787	0,0784	0,0770	0,0792
4,3	0,1525	0,1570	0,1611	0,1531	0,1605	0,1568
4,5	0,2944	0,2877	0,2927	0,2912	0,2930	0,2918
6,1	0,0773	0,0819	0,0766	0,0772	0,0803	0,0787
6,3	0,1529	0,1500	0,1508	0,1505	0,1479	0,1504
6,5	0,2809	0,2805	0,2782	0,2770	0,2829	0,2799
8,1	0,0877	0,0771	0,0845	0,0736	0,0764	0,0798
8,3	0,1512	0,1547	0,1504	0,1507	0,1495	0,1513
8,5	0,2829	0,2880	0,2806	0,2821	0,2825	0,2832

Table 7: Multi-threaded Runtimes [s] On Down Sampled Cloud With 811 Points

Specification	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Avg Runtime
2,1 cm Radius	0,0161	0,0151	0,0161	0,014108775	0,0060	0,0135
2,3 cm Radius	0,0153	0,0145	0,0077	0,018163271	0,0131	0,0138
2,5 cm Radius	0,0083	0,0173	0,0105	0,013787366	0,0144	0,0128
4,1 cm Radius	0,0166	0,0158	0,0183	0,016355505	0,0175	0,0169
4,3 cm Radius	0,0131	0,0131	0,0106	0,014974114	0,0126	0,0129
4,5 cm Radius	0,0099	0,0077	0,0141	0,02062239	0,0087	0,0122
6,1 cm Radius	0,0149	0,0136	0,0126	0,012617526	0,0125	0,0132
6,3 cm Radius	0,0110	0,0136	0,0155	0,009134126	0,0146	0,0128
6,5 cm Radius	0,0137	0,0150	0,0154	0,013340522	0,0147	0,0144
8,1 cm Radius	0,0075	0,0101	0,0119	0,008896383	0,0150	0,0107
8,3 cm Radius	0,0189	0,0112	0,0110	0,015349163	0,0160	0,0145
8,5 cm Radius	0,0112	0,0072	0,0136	0,015719037	0,0138	0,0123

computation, the results would be satisfying. Image 16 shows the normals computed with the radius of 3 cm. Here one can already see, that there are a few areas, where the normals are not consistently oriented. This is due to the fact, that the error occurs in a region, where there are not enough points inside the neighborhood (3cm). Hence resulting in false computations. Image 17 shows a chaotic normal distribution. The points in the image are to far apart, so that the chosen radius finds only few neighbors to compute the surface on. Due to the point scarcity in comparison to the radius, the normal estimation with this radius can not be used for further computation. Images 16 and 18 show normal computations for the point cloud, which were computed with larger radii.

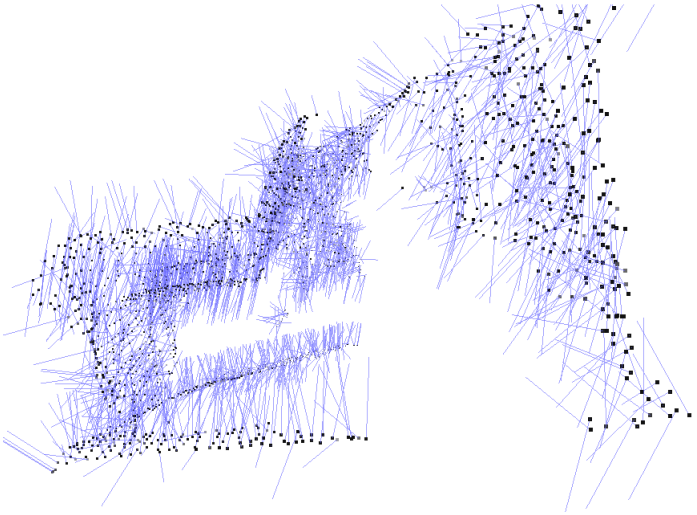

Fig. 16: The sideview of an armchair, which was down sampled to 2568 points. The image shows the normals computed from a radius of 3cm. The orientation here is not that well structured in low point density areas.

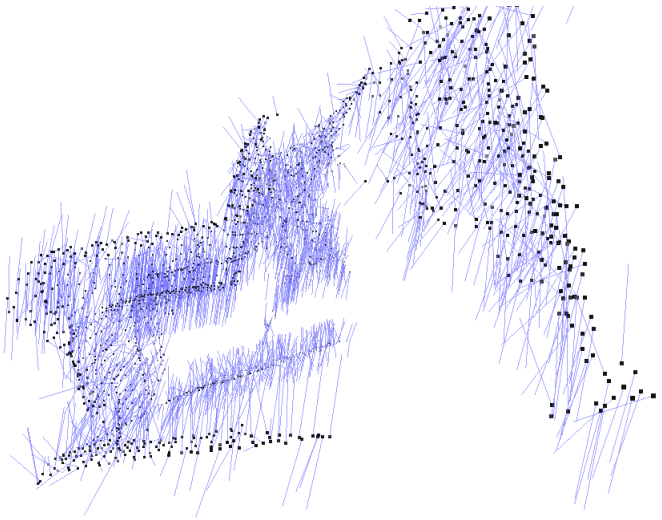

Fig. 17: The sideview of an armchair, which was down sampled to 2568 points. The image shows the normals computed from a radius of 1cm. The orientation of the normals does not allow further computations, due to the chaotic distribution of the different normals.

What can be seen here is, if one was to use only normals with a small radius, the spread and accuracy of the normals in critical areas would not be good enough to conduct further computations. However the next step in the pipeline is the computation of the MLS. If the MLS should compute an accurate curvature, the precomputed normals can not have to many wrong vertex normals. Due to this fact other normal estimation methods have to be used. Another normal estimation method, which is provided within the PCL, are the methods, which are all based on integral images. Integral images are described in section 5.3. The integral image can only be computed from point clouds, where the values are ordered. This means, that no down sampling can take place. Down sampling reduces the three-dimensional array of the point cloud into a two-dimensional array. The integrity of the points is lost, when down sampled, thus not allowing the normal computation with integral images on down sampled clouds.

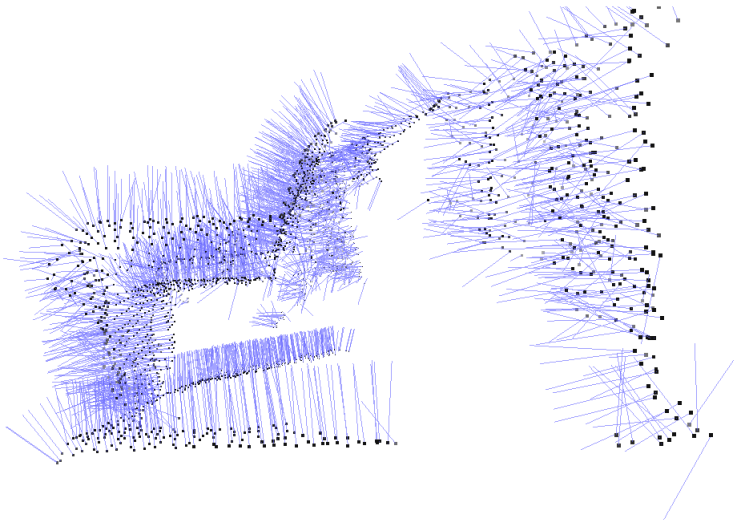

Fig. 18: The sideview of an armchair, which was down sampled to 2568 points. The image shows the normals computed from a radius of 5cm. The orientation of the normals here are structured and can be used for further computation.

The table 8 shows the normal computation using integral images with different specifications.

Table 8: Normal Estimation Runtimes [s] Using Integral Images

Specification	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Avg Runtime
Average 3d Gradient	0,0497	0,0490	0,0484	0,0486	0,0492	0,0490
Simple 3d Gradient	0,0535	0,0534	0,0532	0,0538	0,0536	0,0535
Average Depth Change	0,0679	0,0679	0,0683	0,0676	0,0716	0,0686
Covariance Matrix	0,1029	0,1035	0,1030	0,1027	0,1031	0,1030

When comparing these to the simple normal estimation you can see a tremendous lowering of the computation time. Especially, when looking at the larger radii, the computation with the integral images is superior to the normal and even the multi-threaded normal estimation. The normal estimation, which uses the covariance matrix, takes twice as long. However the covariance matrix provides better results than the other estimation methods. Even though the runtime was higher, it was still in an area, where a possible ”realtime” computation could be achieved. Figure 19 shows the the average, max and min runtimes of the different normal estimation methods based on integral images. As you can see the difference between the covariance method, and the other methods is close to double. However the speed up in comparison to the simple normal estimation is significant.

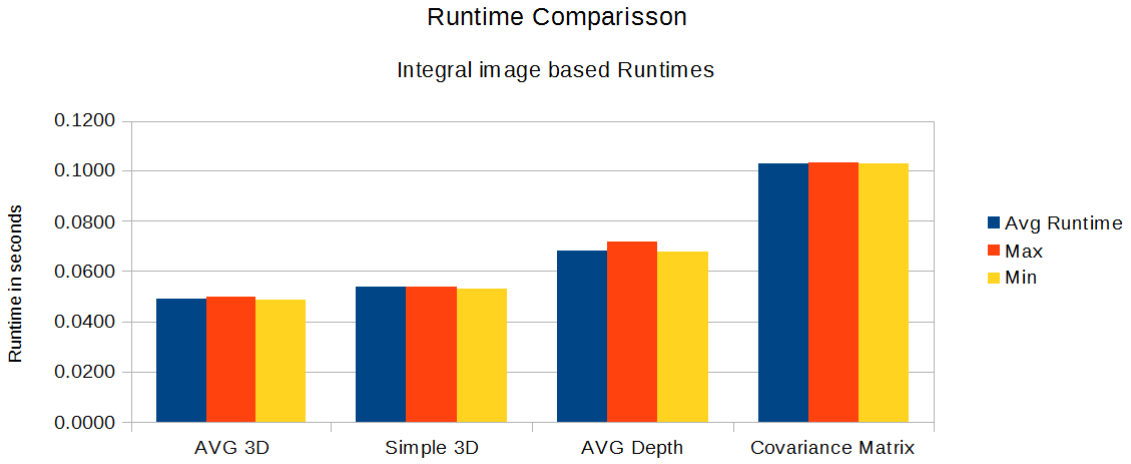

Fig. 19: The image shows a table of the different average, max and min runtimes of the normal estimations based on integral images.

The image 20 shows the synthetic bunny. The different normal estimations were also used on this mesh. In figure 20 the first picture shows the down sampled bunny point cloud. Here you might notice, that the bunny seems to be black. This results from the scale of the bunny. The normals are computed on the point set, but the bunny is to small, so the normals are all inverted. The second picture shows the inverted normals of the bunny. The distribution here shows, that the algorithms also work at very small scale. Since the inverted normals were used here to demonstrate the distribution of

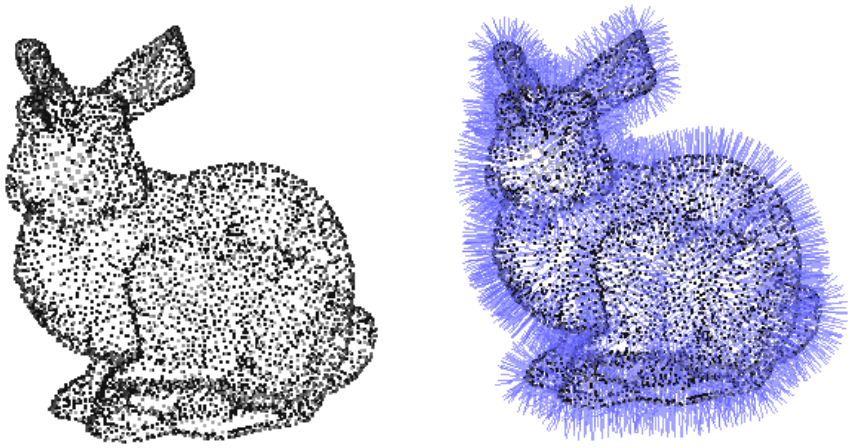

Fig. 20: The figure shows a bunny point cloud. The bunny to the left, shows the bunny with its inverted normals.

the normals on synthetic objects, the original file should not be forgotten. Figure 21

is a magnified picture of the bunny. The normals here are all facing inwards, however the distribution is very accurate and allows further calculations.

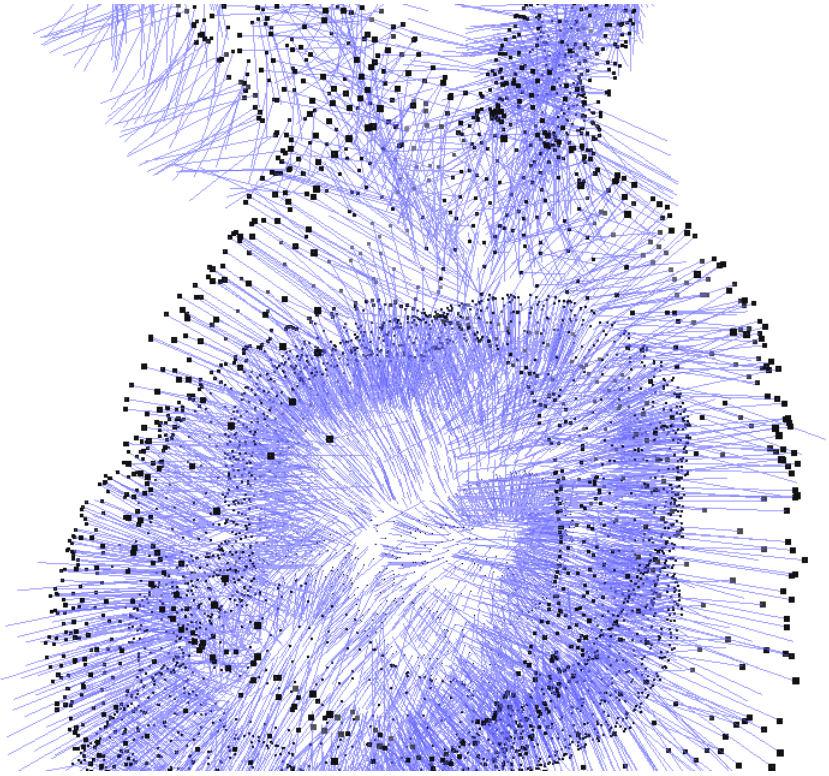

Fig. 21: The figure shows a cut through the bunny. The normals which can be seen here, are the normals directly after computation. These normals are facing inward due to the fact, that the bunny is to small to use a normal estimation method based on the distance between the points. The normals have to be inverted to be facing away and outward from the bunny.

7.3 Conversion and Colorization

Having found a normal estimation, which satisfied the needs for the pipeline, the next step was computing the curvature of the point clouds. In order to do so, we used the MLS algorithm, which was introduced earlier in section 6.1[GGG08]. The point clouds with normals were provided as a pcd file. Since it was decided, that the MLS from Meshlab could be used, the conversion between the pcd and ply file was needed. This was achieved by calling the conversion inside the main pipeline and did not create a problem. After the conversion, a mlx script was used to access the meshlabserver to and compute the curvature. For our experiments it was sufficient enough to use the standardized parameters for all curvature estimations. The point cloud with the curvature colors was saved as a ply file and had to be converted back to a pcd file for further computation. This computation was needed, because further computations were done using ROS. After calling two conversions and colorizations the workflow can proceed to the next step. Figure 22 shows two different curvatures, which were calculated from the normals. The input cloud here was the down sampled armchair with already precalculated normals. This can be seen very well in the first image. The second image shows the mean curvature, which was calculated from the normals. Third is the Gaussian curvature. The different types of curvature can be used inside the pipeline, enabling different critical area analysis.

Fig. 22: From left to right: image with the calculated normals, image with the mean curvature, image with the Gaussian curvature

7.4 Computing Areas of Interest and Visualization

The standardized visualizer, which comes with ROS, is Rviz. Rviz is able to directly listen on topics independent of the underlying file format of the cloud. This made it extremely practical for first scanning and then analyzing the point clouds. When scanning the point clouds, Rviz can subscribe directly to the input topic to visualize the depth data from the Kinect. This allows the user to efficiently scan, what is needed. Resulting from this are numerous implementations for filtering, which were tested to receive the optimal results for the input cloud data. One of the implementations uses the axis-filtering to create an area of focus. The input data is filtered at the x-axis and passed to the next topics, where it is filtered at the y- and z-axis. The result is a cubic area, where all points, which lie within the area, are saved and the rest are discarded. Such filtering allows the user to create areas of focus for further analysis. The main goal of the experiments however was to analyze and visualize the points for critical areas. These areas of course are computed with the curvature, but are of no use without the proper visualization. With Rviz there was already a very powerful tool to visualize the incoming data. The incoming data however was not very useful, since one is not able to analyze the critical points. For this a re computation of the points has to be done. This re computation and refitting was done in a separate topic, which Rviz could subscribe to. From the color of the curvature different spheres were computed in order to show the critical areas. Since the different point clouds vary in size and their neighbors are at different lengths, an integral shift function is included, so that the visualization of the points is independent of the actual size of the point cloud. Hence allowing different point cloud sizes to be evaluated without having to change the area of focus. Even though the mlx script allows multiple methods for colorizing curvature, the evaluation of the points only required one algorithm.

Listing 2: Point Size Calculation

```
In{RGB Colors of a Point}
Out{radius}

Step 1: Compute min and max color values of the input point.

Step 2: Convert RGB color to HSV color.

Step 3: Adapt radius based on the resulting HSV color.

Step 4: Integral shift to create radii whitin a certain size
threshold.

Step 5: Return radius
```

The algorithm reads the color of each point from the input cloud. In order to compute the radius, the HSV model is used [wik14c]. The HSV is a color space, where the color is only 1 value. The HSV model is based on the ”hue”, the saturation and the value of

the color. Since only the value of the color is necessary for the computation, the other values are not needed. First the min and max values of the RGB have to be computed. One of the preconditions for computing max and min values is that $RGB \in \{0, 1\}$.

$$MAX = max(R, G, B) \tag{21}$$

and

$$MIN = min(R, G, B) \tag{22}$$

After computing MIN and MAX four different values must be calculated. If MIN and MAX are the same, the resulting value is 0. This case is the simplest to test. The next cases test for each max value. In the normal HSV model one starts with R and then goes to G and B. In the first implementation of the colorization, the R and B values were switched. This is due to the fact, that R would have the smallest hue and B would have had the largest hue. The first visualization experiment was to show large values for R. The resulting cases were implemented using the following functions:

$$max = B \tag{23}$$

$$hue = 60 * (0 + \frac{G - R}{max - min}) \tag{24}$$

$$max = G \tag{25}$$

$$hue = 60 * (2 + \frac{R - B}{max - min}) \tag{26}$$

$$max = R \tag{27}$$

$$hue = 60 * (4 + \frac{G - B}{max - min}) \tag{28}$$

After the hue has been computed, the hue has to be changed from its values. Inside the HSV model, the hue can range from 0°to 360°. The next step is to remove any hue, that have a larger value than 240. When all hues have been calculated, they can be normalized. Normalizing the hues reduces them form $hue \in \{0..240\}$ to $hue \in \{0..1\}$. The reduced hues could now be used to calculate the radii, but the smaller values would not be noticeable in Rviz. To counter the loss of visual information, an integral shift is used. An integral shift creates a new field, in which the value can possibly lie. The standard formula for an integral shift works as follows:

Say the integral should be between the values a and b, then the resulting interval would be written as $[a, b]$. Since the hue should only be able to take these integers as values, the resulting hue would be calculated as:

$$hue = ((b - a) * hue) + a \tag{29}$$

The standardized hue is then passed to the point as its input radius. Rviz however has a different method to visualize points. The method Rviz uses are called marker arrays. "The Markers display allows programmatic addition of various primitive shapes to the 3D view by sending a visualization_msgsMarker or visualization_msgsMarkerArray message" [ROS14a] The marker array basically creates small primitive shapes at the position of the points, which it receives from its subscribed topic. The topic in our pipeline has loaded the colorized pcd file. For each point inside the cloud, a marker is created. The marker receives the color and the position directly from the loaded point cloud. The marker shape is set to be a simple sphere. Thus only needing the radius and a point for it to be calculateable. When all the information is passed to the marker, it is written inside the marker array, which is displayed in Rviz. The resulting image can be seen in figure 23. In this case the blue values are too small

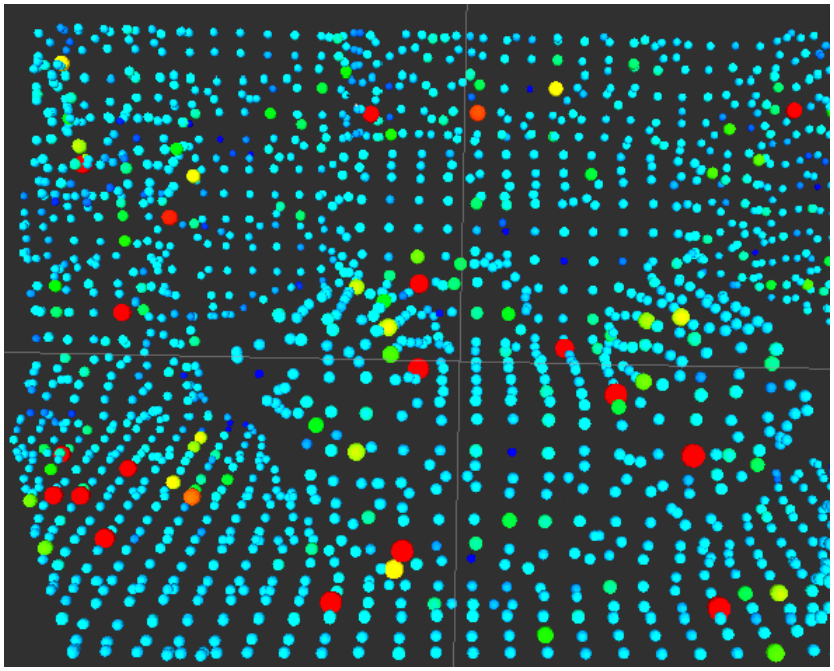

Fig. 23: Armchair with crital points and radii computed with integral shifting

in relation to their importance. Recalling the Gaussian curvature from 6.1.2, red and blue coloring show the critical points. Considering that the critical points are the focus of the implementation, their size has to be adapted. This results in a slightly changed implementation, where the hue is recalculated. The hue is still normalized, but is ordered according to the different values of RGB. The radii are then set accordingly. The results can be seen in image 24. The largest radii are set for the blue and red values. These are the critical points inside the scanned point cloud. The large yellow colored points, are points, where the color values are able to occur on both sides of the threshold, hence resulting in large and small yellow points.

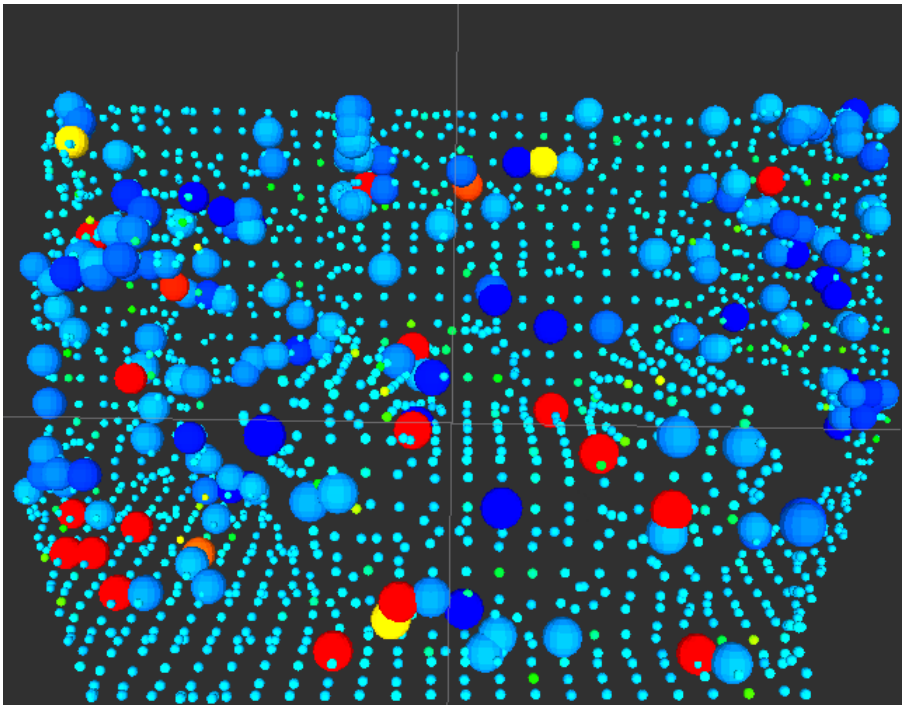

Fig. 24: Armchair with crital points with set radii for certain color values

8 Comparison

To show, that the resulting pipeline works with real and synthetic point clouds, two images will be shown in comparison. One being the down sampled armchair with a ball, and the other being a synthetic hippo. The armchair consists of about 3000 points and the hippo of 8127. Figure 25 shows both point clouds. Considering the pipeline,

Fig. 25: Hippo point cloud and armchair point cloud

the normals in both point clouds have to be computed next. The resulting normal estimations are shown in figure 26, Even though the hippos normals are pointing outward in this figure, they are inverted. When examining the hippo closely, the hippo

seems to have black points. This is due to the fact, that the back of the points are on the outside. The non inverted normals would be harder to compare, since they would be facing inward. With the computed normals both pictures are passed to the

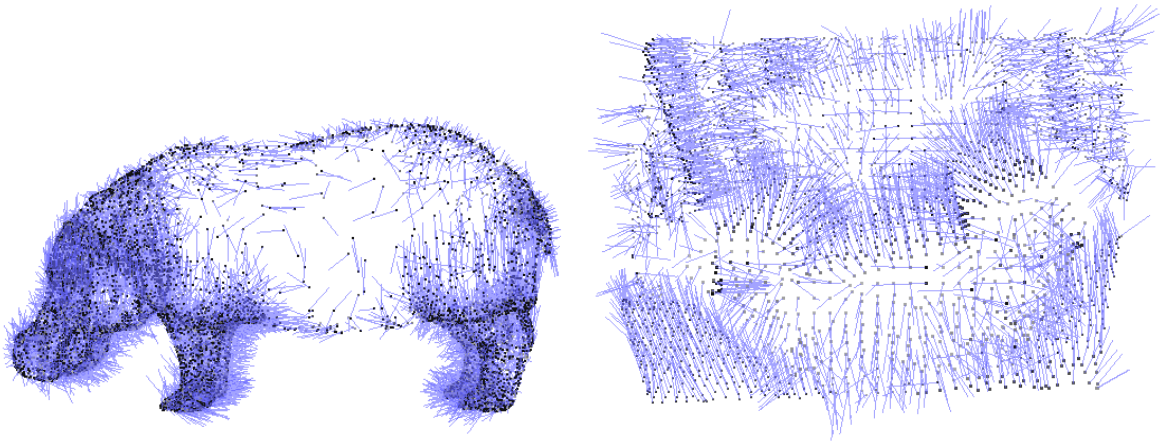

Fig. 26: Hippo point cloud with normals and armchair point cloud with normals

meshlabserver, which computes the curvature from the precomputed normals. Figure 27 shows the Gaussian curvature for both images. These pictures show the colorized

Fig. 27: Hippo point cloud and armchair point cloud with the computed Gaussian curvature for both.

point clouds. The colorized pictures are then converted back into the pcd file format, where they are passed to a ROS topic. The ROS topic computes the marker arrays from the computed curvature and passes it to the ROS visualizer Rviz. The following figure 28 shows both the armchair and the hippo as marker arrays. Since the hippo is relatively large, it is hard to check, if the algorithm worked. To show that the pipeline works, a closeup of the a hippos ear is shown in figure 29. As seen here, the point cloud is analyzed with the help of the marker arrays. These now show a detailed visualization

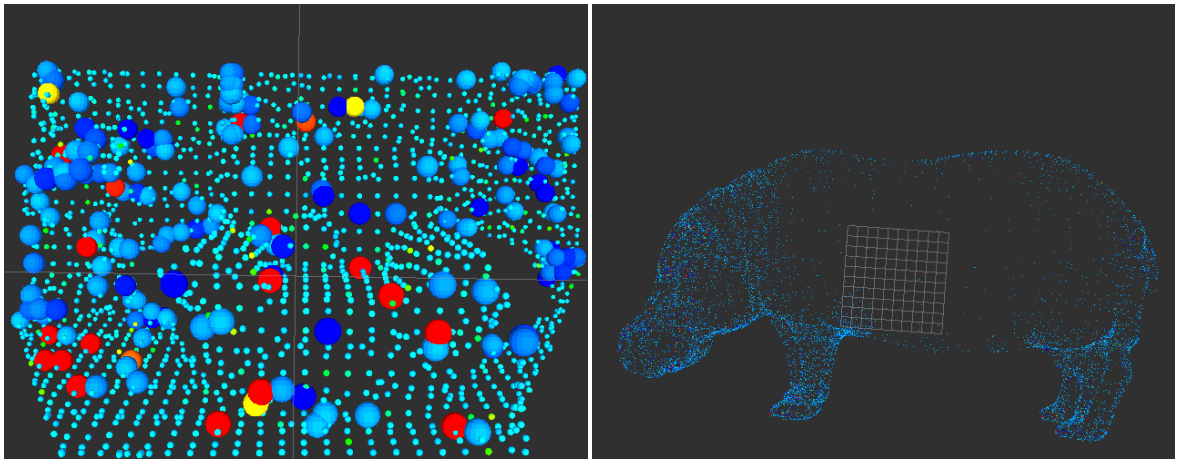

Fig. 28: Armchair and hippo with their critical points, with set radii for certain color values

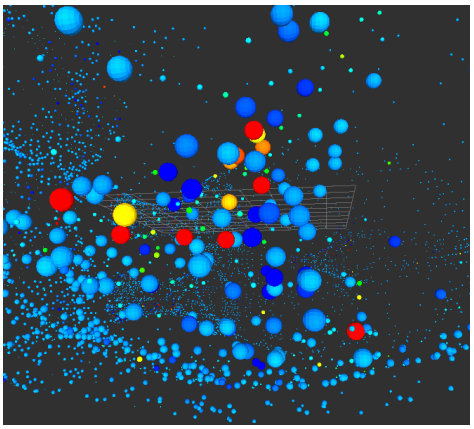

Fig. 29: This figure shows the ear of the hippo with the adapted radii. These seem small in comparison to the armchair because the hippo is much larger. However this shows, that the pipeline works.

of the critical points in both point clouds. With these critical points different structures in the surfaces of the object can be detected. The borders of an object can be estimated without having to save every detail about the object in advance.

9 Future Work

The here implemented pipeline for critical point analysis in point clouds shows, that it is possible to compute critical areas from a scanned point cloud with a Kinect. In this thesis the computation of the critical areas was not done in realtime since a few factors were missing: The first being the different point types, which need to be converted. In the pipeline more than 5 conversions had to be made. This results from the different ROS and PCL versions, but also from the different point types in PCL itself. The second factor was the time, which was needed to compute the normals. The normal computation, based on the integral images, turned out to be fast enough for point clouds with a maximum size of 307200 points. However these might not be sufficient enough, when the point cloud size increases. The integral images also have disadvantages, which might not appear directly. Integral images can only be computed on non resized clouds. Any point cloud, that has unorganized points, can not be computed with the integral image approach. If realtime data is to be used in the future, faster normal estimation methods are needed, which also allow the computation of normals on down sampled point clouds. Down sampling is another factor, where the calculation speed has to be increased. The larger the point clouds are, the longer the calculations need. However the algorithms, which can be used on down sampled point clouds, still have to be found. When larger point clouds have to be computed, the colorization speed has to be increased in order to compute the curvature in realtime. When the camera is moved and we accept real data, the curvature computation has to be fast enough to show it in the visualizer software. When these items are solved, it should be possible to do the realtime curvature computation.

10 Summary

In this thesis a pipeline for curvature based analysis of point clouds was introduced and implemented. To receive an overall understanding of the technology, all the used techniques were introduced and explained. Starting with the Kinect and how the point clouds are scanned, over to the different libraries and integrations, such as PCL and ROS. With in this integration different problems with the compatibility have been detected. These problems were driver and compatibility problems, which were solved, so that the pipeline can be used on any system with the same software and hardware. Following the integration the different normal estimations and the curvature computation with the means of MLS were shown. After the introduction the different normal estimation methods were compared and weighted against each other, to find the one, that was most suitable for the pipeline. Finally the pipeline was implemented and tested. In the comparison it was shown, how well the pipeline works with real and synthetic data. Hereby it should again be mentioned, that the real size of the point clouds matter to analyze these efficiently. The hippo, that was used, was so large, that an efficient view on a certain area was not easy to achieve. With the help of curvature computation, areas of interest can be integrated. This provides a better and more robust method to compute areas of interest.

11 Appendix

In order to use and understand some of the algorithms, that were mentioned in this thesis, a few code fragments will be shown in order to deepen the understanding of the pipeline. The first code fragment, which will be looked at briefly, is the computation of the max and min values for the colors. This is needed since the HSV color space is used for calculating the radii. The HSV model allows the computation of a color only with the value of the color. This enables the use of the color as indicators for the radii. The code receives the color values of the input point. Then these values are normalized, so that $R, G, B \in 0..1$ After the normalization the max and min values are calculated.

Listing 3: Computation of max and min values for the HSV model transformation

```
float rm = cloud->points[i].r;
float gm = cloud->points[i].g;
float bm = cloud->points[i].b;
float R = (rm / 255.0);
float G = (gm / 255.0);
float B = (bm / 255.0);

float min = fmin(R, fmin(G,B));
float max = fmax(R, fmax(G,B));
```

After calculating the min and max values for the HSV model, the radii can be computed.

Listing 4: Computation of the radii

```
float hue = 0.0;

if (min == max){
}
else if(max == B){
        hue = 60 * (0+((G-R)/(max-min)));
}
else if(max == G){
        hue = 60 * (2+((R-B)/(max-min)));
}
else{
        hue = 60 * (4+((B-G)/(max-min)));
}
if(hue < 0.0){
        hue = hue + 360;
}
if(hue > 240){
        hue = hue;
}
```

One might notice, that the hue computation in listing 4 is not the same as for the standard HSV model. This is due to the fact, that the critical areas, which we want to examine here, are red. If the original HSV transformation would be used, red would be assigned the smallest values. This is, why in the code the red and blue cases are switched. After the hue is calculated, all that is left, is the normalization of the hue. This is done in listing 5

Listing 5: Computation of the radii

```
hue = hue / 360.0;
```

Now the two different cases for the radii computation will be shown. The first three lines of code in listing 6 show the first colorization of Rviz marker arrays, which were tested. Here the hue was transformed, so that the values were between 0.02 and 0.07. This was done with the help of the integralshift function, which is called below. The second case shows a case differentiation, where the hue is changed according to different hue values, which were obtained earlier.

Listing 6: Computation of the radii

```
        float a = 0.02;
        float b = 0.07;
        hue = (((b-a)* hue) + a);

And

        if((hue<0.1)||(hue>0.5)){
                hue = 0.1;
        }
        else{
                hue = 0.025;
        }
```

These code fragments showed the math, which was behind the colorization and allow a better understanding of what happens inside the visualization pipeline, aside from the saving and transmorphing of point clouds.

References

a. : *Curvatures*. http://www.grad.hr/itproject_math/Links/sonja/gausseng/ehpp/ehpp.html. Version: - -, Abruf: 1.11.2014

b. : *kinect*. http://www.codeproject.com/Articles/317974/KinectDepthSmoothing. Version: - -, Abruf: 1.11.2014

c. : *TangentPlane*. http://programming4.us/image/012011/Surface%20Normals_1.jpg. Version: - -, Abruf: 1.11.2014

Der07. Derpanis, Konstantinos G.: *Integral image-based representations*. 2007

DM99. D.S. Meek, D.J. W.: On surface normal and Gaussian curvature approximations given data sampled from a smooth surface. In: *Computer Aided Geometric Design 17*, 1999, S. 521–543

GGG08. Guennebaud, Gaël ; Germann, Marcel ; Gross, Markus: *(Guest Editors) Dynamic Sampling and Rendering of Algebraic Point Set Surfaces*. 2008

Jia13. Jia, Yan-Bin: *Gaussian Curvature*. 2013

JP03. Jingliang Peng, C.-C. Jay Kuo Manli Z. Qing Li L. Qing Li: *Estimating Gaussian Curvatures from 3D Meshes*. 2003

KDT. Diplomarbeit

Kho11. Khoshelham, K.: *Accuracy Analysis of Kinect Depth Data*. 2011

Koc78. Koch, Matthias: *Simplification of Point-based Geometry*, University of Konstanz, Informatics and Information Technologie, Diplomarbeit, 1978

NJM03. Niloy J. Mitra, An N.: *Estimating Surface Normals in Noise Point Cloud Data*. 2003

ope14. openMP: *OpenMP*. http://openmp.org. Version: 2014. – [Online; accessed 1-Nov-2014]

PCL14a. PCL: *OMP Normal Estimation*. http://pointclouds.org/documentation/tutorials/norma_estimation.php. Version: 2014. – [Online; accessed 1-Nov-2014]

PCL14b. PCL: *Point Cloud Library*. http://pointclouds.org/. Version: 2014. – [Online; accessed 1-Nov-2014]

PCL14c. PCL: *Simple Normal Estimation*. http://pointclouds.org/documentation/tutorials/normal_estimation.php. Version: 2014. – [Online; accessed 1-Nov-2014]

PGK02. Pauly, M. ; Gross, Markus ; Kobbelt, L.P.: Efficient simplification of point-sampled surfaces. In: *Visualization, 2002. VIS 2002. IEEE*, 2002, S. 163–170

ROS14a. ROS: *Marker Array*. http://wiki.ros.org/rviz/DisplayTypes/Marker. Version: 2014. – [Online; accessed 1-Nov-2014]

ROS14b. ROS: *Robot Operating System*. http://www.ros.org/about-ros/. Version: 2014. – [Online; accessed 1-Nov-2014]

SHN09. S. Holzer, M. Dixon S. G. R. B. Rusu R. R. B. Rusu ; Navab, N.: Adaptive Neighborhood Selection for Real-Time Surface Normal Estimation from Organized Point Cloud Data Using Integral Images. In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009, S. 2684–2689

Sou14. Source, Computer S.: *Integral Images*. http://computersciencesource.wordpress.com/2010/09/03/computer-vision-the-integral-image/. Version: 2014. – [Online; accessed 1-Nov-2014]

wik14a. wiki: *Eigenvalues and Eigenvectors*. http://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors. Version: 2014. – [Online; accessed 1-Nov-2014]

wik14b. wiki: *Hauptkruemmung*. http://de.wikipedia.org/wiki/Hauptkr%C3%BCmmung. Version: 2014. – [Online; accessed 1-Nov-2014]

wik14c. wiki: *HSV*. http://de.wikipedia.org/wiki/HSV-Farbraum. Version: 2014. – [Online; accessed 1-Nov-2014]

wik14d. wiki: *KinectTechnology*. http://en.wikipedia.org/wiki/KinectTechnology. Version: 2014. – [Online; accessed 1-Nov-2014]

wik14e. wiki: *Punktwolke*. http://de.wikipedia.org/wiki/Punktwolke. Version: 2014. – [Online; accessed 1-Nov-2014]

wik14f. wiki: *Signal to noise ratio*. http://en.wikipedia.org/wiki/Signal-to-noise_ratio. Version: 2014. – [Online; accessed 1-Nov-2014]

wik14g. wiki: *Weingartenmap*. http://en.wikipedia.org/wiki/Differential_geometry_of_surfaces#Second_fundamental_form. Version: 2014. – [Online; accessed 1-Nov-2014]

YQ07. Yang, Pinghai ; Qian, Xiaoping: *Direct Computing of Surface Curvatures for Point-Set Surfaces*. 2007