TOWARDS A DATA-DRIVEN OBJECT RECOGNITION FRAMEWORK USING TEMPORAL DEPTH-DATA

David Birkas

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

December 2015

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By:	**David Birkas**
Entitled:	**Towards a Data-driven Object Recognition Framework using Temporal Depth-data**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________________ Chair
______________________________ Examiner
______________________________ Examiner
______________________________ Examiner
______________________________ Supervisor
______________________________ Co-supervisor

Approved ______________________________
Chair of Department or Graduate Program Director

__________ 20 _____ ______________________________
Rama Bhat, Ph.D.,ing., FEIC, FCSME, FASME, Interim Dean
Faculty of Engineering and Computer Science

Abstract

Towards a Data-driven Object Recognition Framework using Temporal Depth-data

David Birkas

Object recognition using depth-sensors such as the Kinect device has received a lot of attention in recent years. Yet the limitations of such devices such as large noise and missing data makes the problem very challenging. In this work I propose a framework for data-driven object recognition that uses a combination of local and global features as well as time varying depth information.

Acknowledgments

I would like to thank my mother and father for their unconditional support. My fiance for her patience and encouragement. My brother for his help through my entire work. Last but not least for my supervisor who always kept me on the right track.

Contents

List of Figures

List of Tables

Dedicated to my family.

Chapter 1

Introduction

Object detection and object retrieval is a huge research area in computer science. It includes many fields within computer science, like computer vision [8], [33] and computer graphics [3], [32]. The reason behind why this task does not belong clearly to one another lies in the diversity of the use cases of the final application. Moreover the pipeline of such application may need to solve several complex tasks, which one by one may belong to one of the above mentioned fields, but they solve a bigger complication altogether. Furthermore inspecting solutions from different perspectives is beneficial. The combined knowledge of the field of Computer Graphics and Vision can highlight and create a wide range of novel solutions and applications. The reason behind this is that while the filed of Computer Graphics is making efforts to answer the question: how to produce image data from models, the filed of Computer Vision is making efforts to answer the question: how models are produced from image data. All in all, this research area is clearly benefiting from the results of more than one field in computer science and hence it is located in the joint-section of these fields.

The main question in object detection is how can computers detect, recognize, distinguish or identify objects from an input data, like an image or video stream, as we humans do. For us it is an obvious thing and we are pretty good at it even under certain "complicating" circumstances, such as when an object is partially occluded. However for computers this is not so trivial. There are many factors that need to be taken into consideration. Just to name one, for example lighting conditions. These circumstances can be narrowed down by establishing a specification towards our goals. For instance if the colour information will not be used, then the problem of lighting condition might be omitted. Therefore it is important to define the answers to the questions in *table 1.1* before designing such system. See also the corresponding image in *figure 1.1*).

Subject	Question
Domain	What kind of object(s) do we want to detect?
Accuracy	Do we want to detect the exact same object or just the same type?
Motion	Are we going to detect objects in motion or in a static state?
Input type	What kind of input data is used for gathering information of the object?
Number of inputs	Are we using a single image or a video stream?
Feature set	What kind of information are we extracting from the input?
Approach	How we going to use up our feature set for detection purposes?

Table 1.1: Questions what need to be answered before developing an object detection system

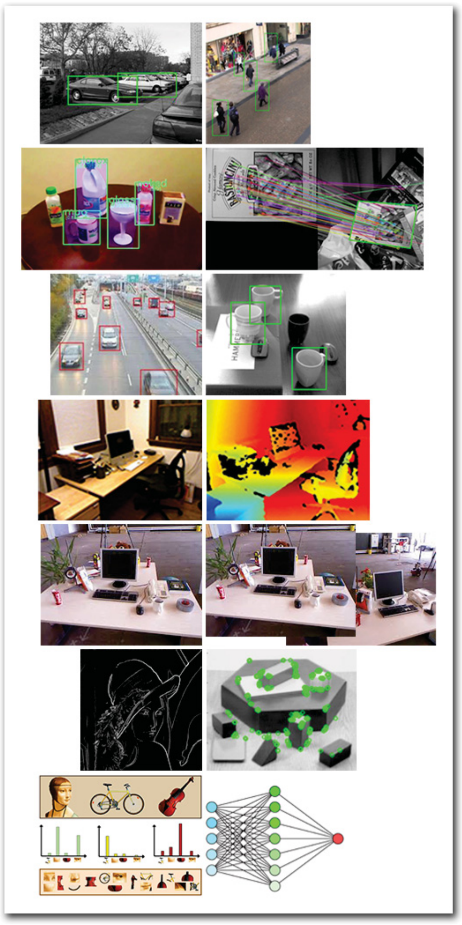


Figure 1.1: Visualization of *table 1.1*'s subjects. Each row corresponds to the same row in the table

Object detection systems can be used up in a wide range of applications to improve, automate, help, solve problems in everyday life of humanity. As it was stated above the research part of object detection is a joint-section of many computer science fields, but solutions to this problem cover an even broader area, where it can be exceedingly useful. Just to name a few:

- Security systems to detect certain unwanted objects.
- Medical diagnoses to help detect doctors some anomalies .
- Self driving cars to reduce accidents.
- Generating 3D scenes to accelerate certain phases of 3D modelling.
- etc.

The presented solution in this thesis is designed according to *table 1.1* as follows. For the domain I choose objects which are related to office assets and furniture, for example computer screens, keyboards, tables, chairs, etc. Yet leaving the opportunity to extend the object database according to the specific application. The reason why I favored these objects as my domain is because detecting everyday like objects reliably is a useful application. This statement was supported also by a company, who deals within the security sphere. They would be for example very interested in applications, which can detect and identify missing objects, which were previously detected. Continuing with the decisions according to *table 1.1*, the detection part is focusing on detecting the general type of the objects in static scenes. Detecting the exact same object requires a more complex feature set, which in parallel increases the "complicating" circumstances what needs to be considered. In my application's point of view the gain of detecting the exact same object does not scale compared to how more complex the task can become. As for input type I choose depth images without the color information captured by the Microsoft Kinect [11] device. The reason why I have neglected the color information is because it can make the system sensitive to lighting conditions and objects with the same shape but different textures may result in different features because edges in the color domain do not represent valley and ridges. Since I want the system to be as robust as it can, I have decided not to use it. Due to the above reasons extracting information from the pure depth image of an object may produce a more general feature set. This is because only the intensity values are taken into consideration, which reflect only the geometric information of the object in 3D. Furthermore some models in the supporting database may have very poor textures or even none, which makes the color information unstable. Additionally depth images can boost the segmentation and help in the camera localization because it contains 3D information. To improve the accuracy I am using multiple images and evaluating the final results based on the partial results of the single images. As for the feature set base, I choose edges, namely Canny lines [6] obtained from the object's depth map. These lines can capture the hidden information of the corresponding object and this information can be used to identify the type of the object for example. For the approach a feature based method, particularly the well known bag of feature words model was chosen. This technique is the state of the art approach in feature based methods and is robust against occlusions and noise [8], [9]. A summary can be seen in *table 1.2* below.

Subject	Choice
Domain	Office objects
Accuracy	General
Motion	Static
Input type	Depth image
Number of inputs	Multiple images
Feature set	Canny lines
Approach	Bag of features

Table 1.2: My answers to *table 1.1* questions

The main idea behind my solution was to generate a sketch like line drawing from the object's depth image and find the best corresponding match in the supporting database. Capturing the silhouettes and the reciprocal internal lines, which heavily depend on the depth map encapsulates $3D$ information in a $2D$ setup. This makes the system more simple and thus robust. Furthermore sketch based shape retrievals were widely studied in the past couple of years [2], [3], [4]. The solutions presented in these studies prove accurate and reliable retrieval results, which raises the question: "Why don't we use it for detection purposes also?". These methods all rely on a sketch based user interface, where the user can draw the desired model or scene. However instead of drawing the model or the scene as a sketch, it could be generated automatically with some input device like a camera. Hence the user has to only capture the desired model or scene. One of the biggest and hardest problem of sketch based shape retrieval is the vagueness of how people sketch [3]. This solution would not just adopt the advantages of such introduced systems, but also solve this ambiguity. Generating the sketches automatically will lead to a more consistent matching and due to this consistency the fine tuning of certain thresholds are more tangible.

1. In this thesis, I present a **pipeline** and a corresponding application for object detection.
2. In my knowledge this is the first approach to **detect objects from depth images** by tracing it back to a sketch based object retrieval dilemma.
3. Furthermore another innovation in this method is, that it uses **multiple depth images** to make the detected results more accurate.

The setup is a moving utility car with a computer running the application along with the camera on top observing the scene. The system is capable of localizing the camera in space while detecting objects in the input stream. It is also prepared for 3D scene reconstruction in future development, by showing corresponding 3D meshes of the detected objects. The presented system can be used up in a wide range of applications, like in security systems detecting missing objects, in game development for scene generation, or in robotics to identify objects and their position in space. A snap shot of the setup and the application can be seen in *figure 1.2.* Also a global pipeline of the system can be seen in *figure 1.3.*

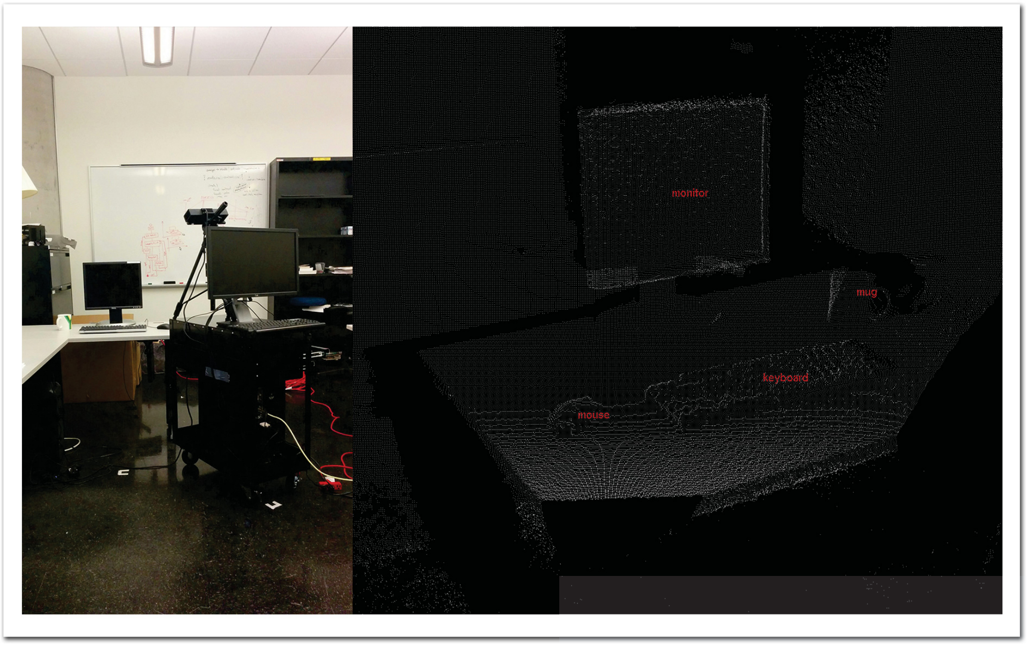


FIGURE 1.2: Left: the setup, utility car with the computer and the camera on top. Right: the system identifying the objects on the table correctly.

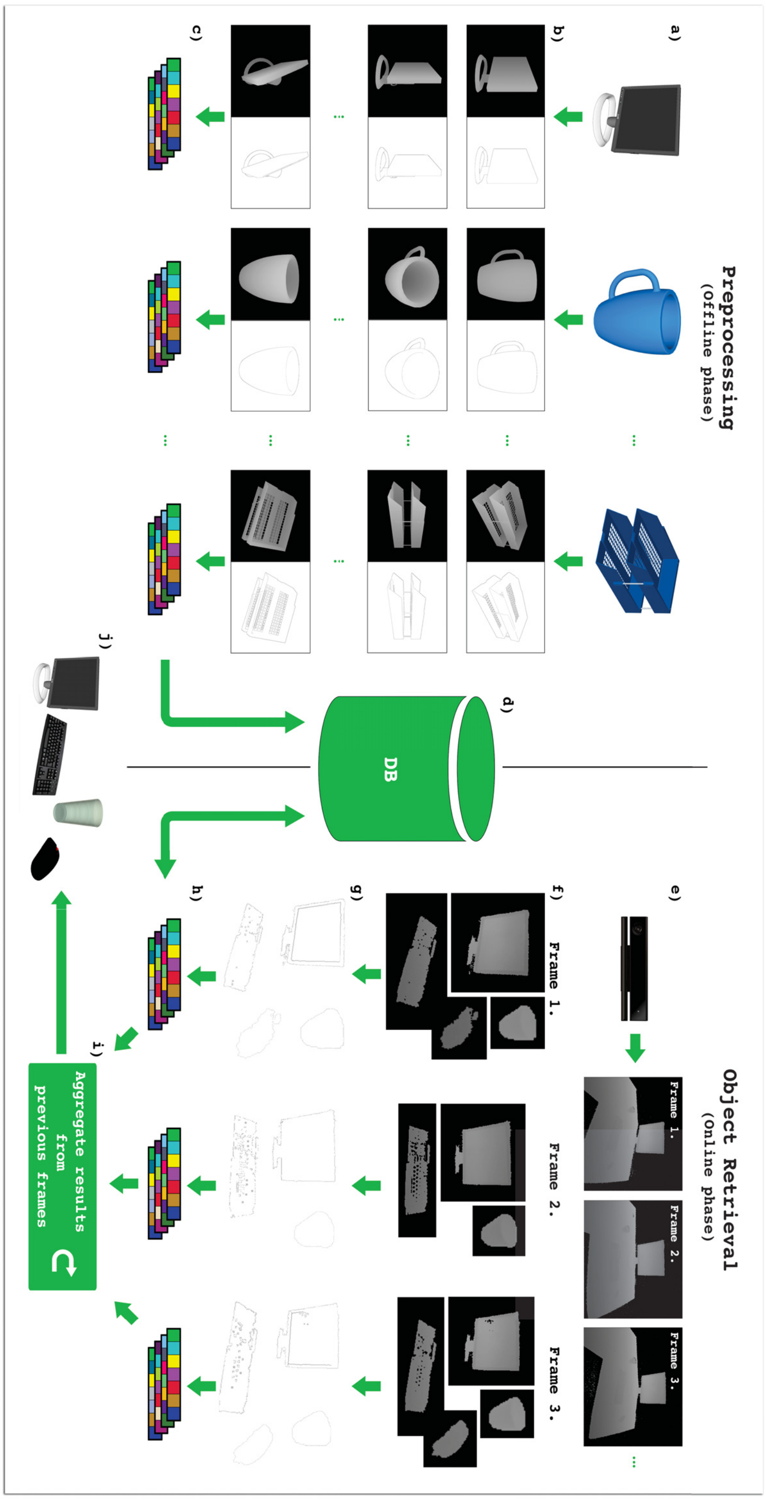


FIGURE 1.3: There are two main phases in the system. The offline, preprocessing phase and the online, detection phase. a) the input of the preprocessing phase, namely meshes. b) creating snapshots of each mesh and obtain the corresponding depth image, then generate the canny lines from it. c) gather the local descriptors from the edge images. d) create the supporting database from the local descriptors. e) input of the detection phase, namely depth maps of the viewed scene. f) segment the scene into objects. g) obtain the edge images of each segmented object's depth image. h) generate the local descriptors from the edge images and find the closest one in the database. i) aggregate results from previous frames. j) final result.

Chapter 2

Related Work

Object detection is a widely studied research area, but it still draws lot of attention as a research filed since most of the solutions concentrate only on a specific portion of the detection. There is no universal object detector, which would work under any circumstances yet. This thesis tries to focus on expanding an already functioning system to a broader use-case with some additional improvements. As I mentioned in *Chapter 1* this solution is tracing back the problem of object detection to a sketch based object retrieval proceeding. This chapter will break down the related previous work and fundamental algorithms used in this thesis in some way.

2.1 Previous Work

This chapter introduces some previous work, which provided guidance for designing the presented system in *Chapter 3*. I will also describe one related work from each important stage of the presented system separately. These important main stages are: segmentation, descriptor design, detection, registration.

2.1.1 Comparison

There are several work out there, which try to address and solve the problem of object detection. I am highlighting three present work here for comparison purposes. The first work [36] is using a single deep neural network to detect objects in regular images. When a system detects an object in an image it draws a bounding box around it, with the title of the object type. Their results are promising, but training the system is tricky compared to the presented system in this thesis. They have to provide images with ground truth bounding boxes around the objects, which they want to detect. In the presented system training the system or in other words generating the database is much more easier, discussed later in *Chapter 3.1.* On the other side using regular 2D images for object detection makes the system more feasible to integrate it into already existing systems. The second work [32], which I am highlighting here is related in a way, that they are using the depth image from the Kinect camera also to obtain some geometric features from the observed objects. Their system is robust and works well on big objects like couches and tables, but fails on smaller objects like cups or monitors. However the presented system in this thesis overcomes this issue and works well on both small and bigger objects too. One other advantage towards the presented system compared to theirs is that my system is robust against rotation, while their system would fail to detect a chair laying on its side for example. These features will be demonstrated in *Chapter 5.3.* The third work [4], which I am highlighting here is using the contour information of the objects encoded it with the Fourier descriptor. The presented system in their paper works reliable till there is no occlusion. Of course occlusion was not the case in their setup, but it is in my case. My system is robust against occlusion, which will be also shown in *Chapter 5.3.*

2.1.2 Segmentation

Segmentation is the base phase of the system presented in this thesis. The more accurate the results are of this stage, the more reliably will the detection part work. The reason for this is because the descriptors will become more robust and unique towards the object they represent if the output is more accurate of this stage. To make sure the results are satisfiable I used depth images as an input for the segmentation. Depth images represent a snap shot of the scene in the system as a normal image, however they also provide additional information by introducing the third dimension. Moreover I convert the depth images into point cloud representation, which is a more feasible representation of the depth image. More on depth images and point clouds in *Chapter 2.2.1.* The obtained point clouds need to be processed in a way, that the system can separate the objects into clusters. To segment the scene I am using a plane detection algorithm described below.

Since I am receiving the point clouds in an organized fashion it enables the use of graph-based [12] or connected-component [13] approaches. In this thesis I followed the method proposed in this paper [1], which uses the connected-component approach.

The main idea behind separating the objects into independent clusters is first finding planar regions in the viewed scene. For plane representation the following normal form is used:

$$ax + by + cz + d = 0$$

Hence such a planar equation is calculated for each point in Euclidean space. To achieve this the first step is to calculate each point's normal. Calculation of the normals can be done in real time exploiting the organized property with this proposed technique [14]. After calculating the normals, a point in the point cloud can be represented as follows:

$$p = \{x, y, z, n_x, n_y, n_z\}$$

where p is a point in the point cloud with x, y, z coordinates and n_x, n_y, n_z are the corresponding normals. The only remaining variable to represent a point with the above planner equation is the d component which can be calculated as the dot product of the coordinates and the normals.

$$n_d = (x, y, z) \cdot (n_x, n_y, n_z)$$

Therefore the final representation for a point in a point cloud is:

$$p = \{x, y, z, n_x, n_y, n_z, n_d\}$$

To see if two neighbouring points belong to the same plane some distance metric has to be proposed. For distance metric a range distance is used between the d components:

$$dist_{range}(p_1, p_2) = |p_{1n_d} - p_{2n_d}|$$

and the distance between the normal directions, which is the dot product of the two point's normal:

$$dist_{normal}(p_1, p_2) = p_{1n} \cdot p_{2n}$$

Now we can proceed with the connected-component algorithm. This algorithm works in a way, that it segments an organized point cloud into a set of partitions. This is done by labeling with an integer each point in the point cloud. Those points which belong to the same cluster will be labeled with the same integer. So if

$$P(x_1, y_1) \in S_i \quad \text{and} \quad P(x_2, y_2) \in S_i, \quad \text{then} \quad L(x_1, y_1) = L(x_2, y_2)$$

where $P(x, y)$ is a point in the point cloud, S_i is a cluster of points and $L(x, y)$ is the label of the given point. The points are compared using a comparison function, where the above mentioned distance metrics are used:

$$C(p_1, p_2) = \begin{cases} true, \quad if((dist_{normal} < thresh_{normal}) \quad \text{and} \\ \qquad (dist_{range} < thresh_{range})) \\ \qquad false, \quad otherwise \end{cases}$$

where $thresh_{normal}$ and $thresh_{range}$ are the set thresholds for the distance metrics respectively. The algorithm starts by assigning the first point with a label, in this case 0. Then the first row and column are compared with the above comparison function to have assigned the appropriate labels. The remaining points are then treated by checking their neighbouring points, $P(x-1, y)$ (left) and $P(x, y-1)$ (top). The following scenarios can happen:

$$C_1(P(x, y), P(x-1, y)) \quad \text{and} \quad C_2(P(x, y), P(x, y-1))$$

$$if(C_1 \quad \&\& \quad C_2) == true, \quad \text{the two segments has to be merged}$$

$$if(C_1 \quad \&\& \quad C_2) == false, \quad \text{a new label is assigned to the current point}$$

$$if(C_1 \quad || \quad C_2) == true, \quad \text{the point is assigned with matched label}$$

After the label image is produced some refinement has to be done in order to make sure we have real planner regions, not just locally planner ones. A least squares plane fit is done on each labeled segment, which has at least $T_{inliers}$ inliers. To make sure the results are really planer the curvature is also computed and a $T_{curvature}$ threshold is used to filter out which are smooth but not planner. The result of this algorithm can be seen in *figure 2.1.*

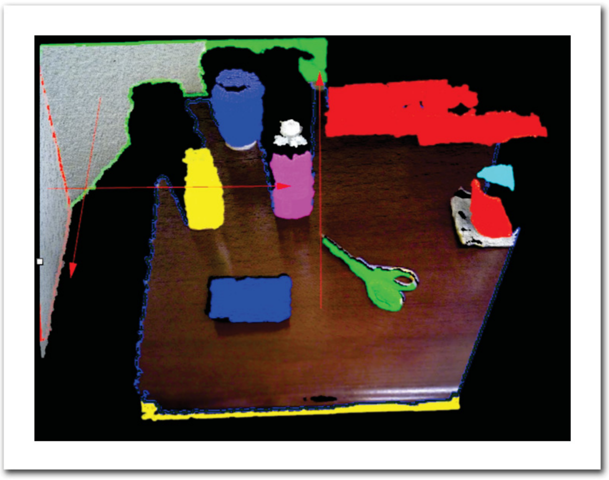


FIGURE 2.1: The result of the above described algorithm. Image from [1].

2.1.3 Descriptor Design

Designing or choosing the descriptor, which will encapsulate all the necessary information, to have the system operate as it is planned is an important step, when planning such system as it is represented in this thesis. In this thesis I choose the state of the art descriptor for sketch based object retrieval, which is presented in the next chapter, in *Chapter 2.1.3*, but there are several other descriptors, like [23] or [34]. In this chapter I will present Fourier descriptor [23], which gave me lot of guidance and understanding towards the final decision.

Fourier descriptors are obtained by applying the Fourier transform on the shape's signature. The resulting coefficients are the Fourier descriptors. The shape signature is derived from the boundary curve of the shape, in other words the silhouette. There are several methods how this signature is exploited from the shapes boundary. The most commonly used methods are complex coordinates, curvature function and centroid distance, but it was proven by [29], that centroid distance method outperforms the others in overall.

The first step to compute the centroid distance dependent Fourier descriptors is to obtain the boundary coordinates.

$$(x(t), y(t)), \quad t = 0, 1, ..., N-1$$

where N equals to the number of boundary points. The centroid distance function is expressed in the following formula:

$$r(t) = ([x(t) - x_c]^2 + [y(t) - y_c]^2)^{\frac{1}{2}}, \quad t = 1, 2, ..., N - 1$$

which represents the distance from the boundary point to the centroid. (x_c, y_c) represents the centroid of the shape. The calculation of x_c and y_c is done by these formulas:

$$x_c = \frac{1}{N} \sum_{N-1}^{t=0} x(t), \quad y_c = \frac{1}{N} \sum_{N-1}^{t=0} y(t)$$

The discrete Fourier transform of $r(t)$, which is called *dft* also in scientific literature is then declared with the following formula:

$$a_n = \frac{1}{N} \sum_{N-1}^{t=0} r(t) exp(\frac{-j2\pi nt}{N}), \quad n = 0, 1, ..., N - 1$$

where a_n represent the coefficients of the Fourier transform.

The acquired Fourier coefficients are translation invariant due to the translation invariance of the centroid distance. In other words, it does not matter where the shape is in space, since the distance between its centroid and its boundary points is independent to the shape's location. To achieve rotation invariance the phase information of the coefficients are ignored by using only magnitudes $|a_n|$. Furthermore scale invariance is also achieved, by dividing the coefficients with the DC component, namely a_0. Also since the centroid function is a real value function only half of the coefficients are needed to index the shapes. The final f feature vector is used as the Fourier descriptor.

$$f = [\frac{|a_1|}{|a_0|}, \frac{|a_2|}{|a_0|}, ..., \frac{|a_{\frac{N}{2}}|}{|a_0|}]$$

The similarity measure between two Fourier descriptor is a simple Euclidean distance between the two feature vector. Fourier descriptor captures coarse or global features in the lower order coefficients and finer shape features in the higher order coefficients. Therefore it is robust against noise and irregularities, since they only appear in very high frequencies, which are usually neglected. Since mostly lower order frequencies are used it is also a compact descriptor. Fourier descriptor is also capable of reconstructing the boundary form the given coefficients, which is an important compression feature, see *figure 2.2.* However, since Fourier descriptor only captures silhouette information it can fail when there are large boundary indentations or protrusions in the input shape, like what occlusion can cause for example.

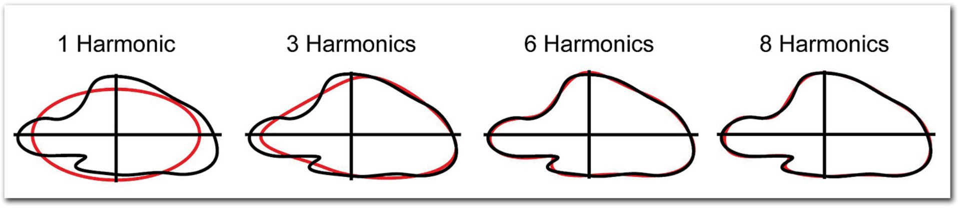


FIGURE 2.2: In this image the viewer can observe, that the more frequencies are used to represent a boundary curve, the more information it has to reproduce the curve.

2.1.4 Detection

There are lot of recent work on object detection, like [32] or [33]. There are also some recent work on sketch based object retrieval, like [2] or [3]. Since I am combining object detection problem with a sketch based object retrieval dilemma, in this section I am briefly describing how does the state of the art sketch based retrieval and the corresponding GALIF feature work [3]. I integrated this algorithm into my system.

The system takes an image with a sketch as an input. First this sketch image or image of edges, where the edges represent the sketch lines is divided into smaller local image patches. This is done by generating $N \times N$ key points evenly distributed over the image. A local image patch is centered around a given key point and is represented by an $n \times n$ cell. A pixel is inside this cell if its (x, y) coordinate is inside the cell's bounding box. Each patch size is determined by relatively to the image area.

For each local image patch there is a corresponding local feature F. These features are obtained by first applying k number of different orientation Gabor filters to the original image. The Gabor filter in the frequency is defined as:

$$g(u, v) = exp(-2\pi^2((u_\theta - w_0)^2\sigma_x^2 + v_\theta^2\sigma_y^2))$$

where $(u_\theta, v_\theta) = R_\theta(u, v)^T$ is the standard coordinate system rotated by θ, w_0 is the peak response frequency, θ is the filter orientation, σ_x frequency bandwidth, σ_y angular bandwidth.

Each orientation of the filter, masks all content that does not possess the right frequency and orientation. This means that the filter only responds to a subset of the lines in the image. All parameters are fixed of the Gabor filter g_i except the rotation. Then each image is convolved with a k set of Gabor filters as I mentioned above, which results in k set of filtered response images.

$$R_i = \|idft(g_i * dft(I))\|$$

where I is the input image, $*$ denotes point-wise multiplication and $idft$ and dft stand for inverse/forward discrete Fourier transform. Please see an illustration in *figure 2.3*.

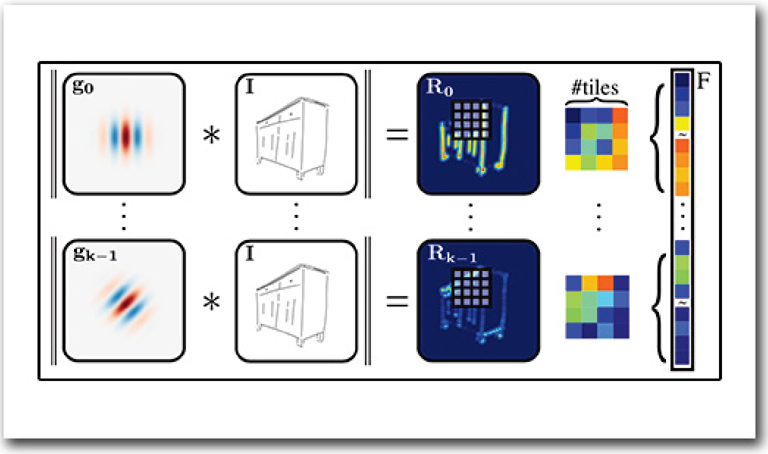


FIGURE 2.3: Result of the Gabor filter and the local feature pipeline. The image is from [3].

This means that the dimension of a local feature vector F for each image, is $k \times n \times n$. For each dimension of F there is an average Gabor filter response within the $n \times n$ cell for orientation i.

$$F(i) = \sum_{(x,y)\in n\times n} R_i(x, y)$$

All features that do not contain edges are discarded and then are normalized such that $\|F\|_2 = 1$.

After calculating all local feature vector F for an image they are represented as visual word frequencies. This means that all local feature vector x_i is quantized against the visual vocabulary, where they are represented as an index q_{ij} to their closest visual word.

$$q_{ij} = argmin_j \|x_i - c_j\|$$

where c_j is a cluster centroid of one of the visual words. Finally the final histogram of visual word h representation is defined by

$$h_j = |\{q_{ij}\}|$$

To do this we need to generate a visual vocabulary. The vocabulary is generated by randomly sampling generated local features and clustering them by using k-means clustering [10]. The number of clusters will determine the size of the vocabulary. This number is very important because it affects the retrieval performance.

2.1.5 Registration

Registration is an important part of the presented system in this thesis. It is important because this part solves the camera localization problem and the correspondence between the detected objects through different input frames. There are lot of work on this direction, like [30] and [31] or [35].

In this chapter, I will briefly describe the method called Kinect Fusion used in [30], because this is the method of my choice. The reason behind it in summary is, that it is fast due to the GPU implementation and tested on the Kinect camera, which showed reliable results.

Kinect Fusion uses the well known ICP algorithm to calculate the 6DOF transformation of the current frame, to the previous one (more on general ICP in *Chapter 2.2.5*). Each 6DOF transformation result is a local transformation between the current and the previous frame. Incrementally applying these local transformation to each other produces the global transformation of the camera. To find the correspondances between frame F_{i-1} and F_i, they use the projective data association technique. One of the innovation of their GPU implementation is, that is uses all points instead of down-sampling or searching for key-points in the cloud. The result of their algorithm can be seen in *figure 2.4* below.

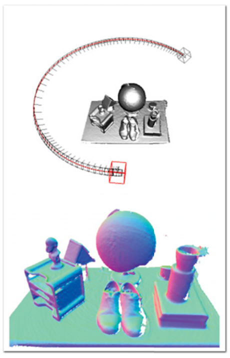


FIGURE 2.4: Result of the Kinect Fusion algorithm. The image shows the position and orientation of the camera around the scene. The image is from [31].

2.2 Fundamental Algorithms

This chapter will highlight the most important data structures, algorithms and methods used in the presented system. These methods are fundamental because they are well proven throughout the time or they present the state of the art solution in the given context.

2.2.1 Depth Images and Point Clouds

As an input I am using depth images. Depth images are images like regular images, but they also carry depth information for each pixel:

$$p = \{r, g, b, d\}$$

where p represents a pixel in the image and r, g, b is the colour, d is the depth information. To capture depth images we need a specialized camera, like the Kinect from Microsoft [11]. Usually these depth images are converted into point cloud data structures which allows easier management over the captured data and projects the depth information into 3D Euclidean coordinate space:

$$p = \{r, g, b, x, y, z\}$$

where p represents a point in the point cloud and r, g, b is the colour, x, y, z is the projected depth into Euclidean space information. Moreover devices like Kinect are capable of giving these structures in an organized fashion (i.e. matrix):

$$\begin{array}{cccc} p_{0,0} & p_{0,1} & p_{0,2} & \cdot\cdot \\ p_{1,0} & p_{1,1} & p_{1,2} & \cdot\cdot \\ p_{2,0} & p_{2,1} & p_{2,2} & \cdot\cdot \\ : & : & : & \end{array}$$

where $p_{x,y}$ represent a point in the point cloud which corresponds to the appropriate pixel in the depth image. From this we can quickly see that neighbouring pixels (or points) can be accessed in constant time, which can be advantageous when processing these point clouds. An example of a depth image and the corresponding point cloud can be seen in *figure 2.5*.

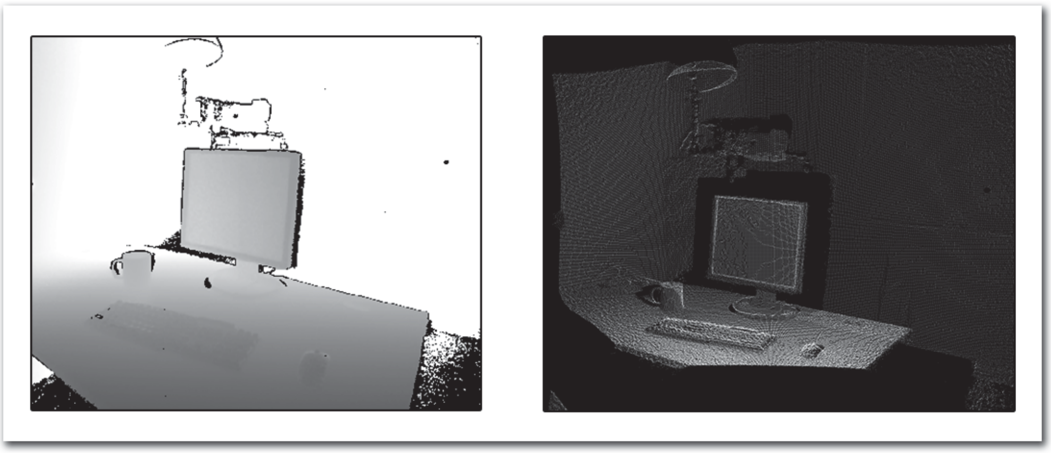


FIGURE 2.5: Image of a depth map and the corresponding point cloud

2.2.2 RANSAC

For refining the results of the plane detection algorithm I am using the RANSAC or random sample consensus method [25][26][27]. This method is an iterative algorithm, which calculates the parameters of certain mathematical models, in this case planes from a set of points, which might contain outliers. Outliers are points, which does not belong to the specified model and inliers are points, which are part of the given model. This method is non-deterministic, which means that it produces a result in a certain probability. This result can be improved, by increasing the number of iterations. One of the biggest advantages of this algorithm is, that it is highly robust against noise.

The algorithm takes a set of points and does the following:

1. Randomly selects n points, in this case $n = 3$, since with three points a plane can be defined.
2. Calculates the parameters of the plane from the 3 random points.
3. All other points are then tested against the previously calculated model and are marked as an outlier or inlier, within a certain threshold.
4. Step 1. to 4. is repeated until the number of iteration is set or enough inliers are found.
5. The plane's parameters are recalculated by considering all inlier points.

The final result can be seen in *figure 2.6*.

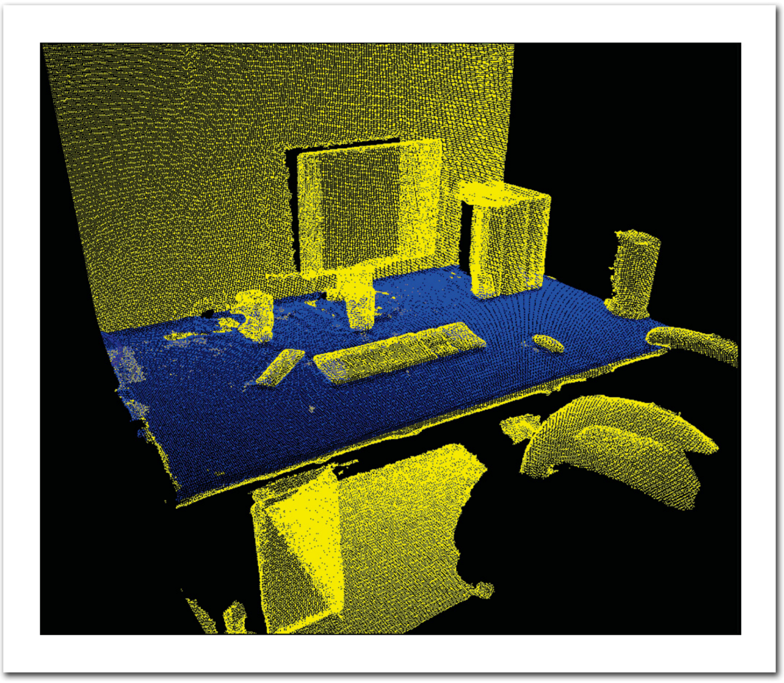


FIGURE 2.6: Result of the RANSAC algorithm described above.

2.2.3 Canny Lines

I am using edges as features, gathering them from the depth map of the object. These edges are the well known Canny lines [6]. In this section I am going to briefly discuss this technique.

The Canny algorithm first finds the intensity gradient of the image. It applies a pair of convolution masks (G_x, G_y) and finds the gradient strength and direction for each pixel with the following formula:

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

$$\theta = arctan(G_y/G_x)$$

The direction is rounded to four angle intervals, namely 0, 45, 90, 135. After this a non-maximum suppression is applied [7], where the direction information is used to remove pixels that are not considered to be part of the edge. Hence only a thin edge will remain. This is why it is also called an edge thinning technique. The last step is a threshold check. There are two thresholds an upper (t_{upper}) and a lower (t_{lower}) bound. If

$$G > t_{upper}$$

the pixel is accepted as an edge. If

$$G < t_{lower}$$

the pixel is rejected as an edge. If

$$t_{upper} \geq G \geq t_{lower}$$

the pixel is accepted only if it is connected to a pixel, which is above t_{upper}. A result of this algorithm can be seen in *figure 2.7*.

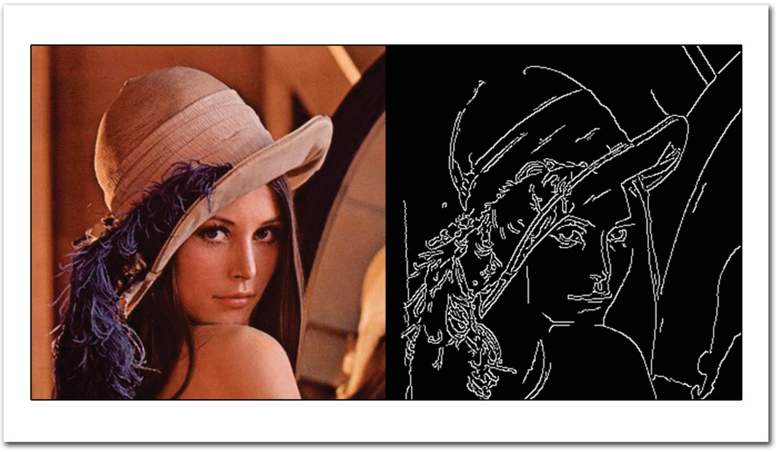


FIGURE 2.7: left: the input image, right: the out put of the Canny algorithm

2.2.4 Bag of Features

My selected approach for using my features to detect objects is the bag of feature model [8], [9]. This model has become the method of choice for affine invariant image retrieval. Since in this work I actually gathering my features from the grey scale image of the object's depth map this technique is feasible.

The bag of feature model compares images based on histogram features. The idea behind the histograms is basically to divide a range of values into series of intervals (e.g. bins). This reduces the size of the data and speeds up the comparison between two set of values. In the bag of feature model a test feature set is used to generate the vocabulary. The vocabulary is an n dimension vector, which determines the intervals of the histogram. The vocabulary is generated by clustering the input feature samples into bigger chunks. The end result of the clustering determines the value of n. After the vocabulary is generated each value of a feature set of an image is determined to its best fit in the vocabulary's interval field and only the number of occurrence of the given interval is stored in the descriptor. Therefore the end result is a sparse vector of the given image. It is sparse since an image usually will not hold feature values for each entry of the vocabulary. In this thesis I follow the rules of creating such vocabulary and descriptors by [2] and [3]. An illustration can be seen in *figure 2.8* of the bag of feature model.

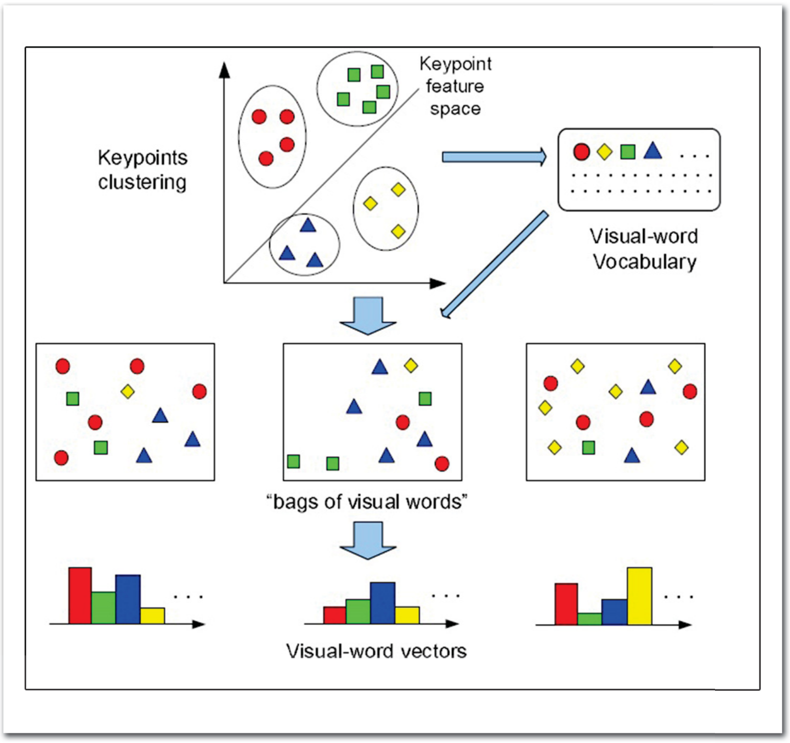


FIGURE 2.8: Illustration of the bag of feature model

2.2.5 ICP

For finding the correspondence between two affined point clouds I am using the well known ICP, namely iterative closest point algorithm [18][19][20][21][22]. This algorithm tends to minimize the difference between two set of points and find the corresponding transformation matrix. It is used in a wide variety of applications, like surface reconstruction or robot localization. I will be using it for localization and finding the correspondence between cloud clusters.

The algorithm takes two set of points, in this case two point clouds. One is the target or the reference cloud, which will be kept fixed and the other one is the source cloud, which will be transformed to match the target as best as it can. The transformation is done in an iterative fashion, hence the "iterative" in the name. The transformation contains both translation and rotation. The general main steps of the algorithm is the following:

1. For each point in the reference cloud find the closest points in the target cloud.
2. Using a mean squared error cost function, estimate the transformation, that will align each point found in step 1. in the best possible way.
3. Apply the transformation to the source cloud obtained in step 2.
4. Iterate through step 1. and 3., till the desired result is achieved.

An illustration of the method can be observed below in *figure 2.9*.

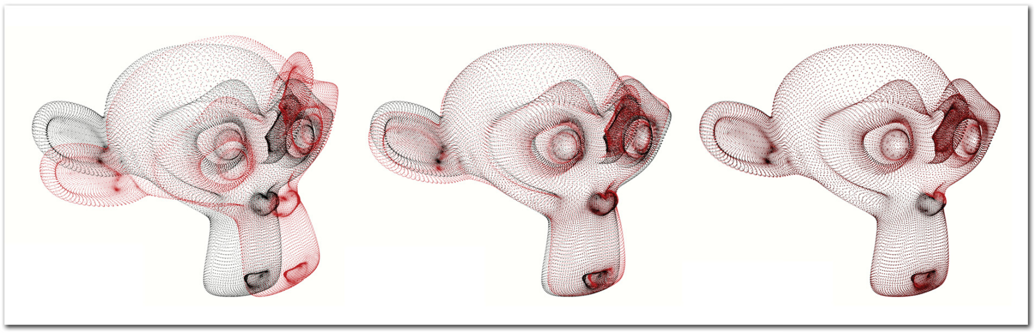


Figure 2.9: The result of the ICP method. From left to right the algorithm converges.

Chapter 3

Proposed Method

The proposed method consist of two main part, an offline (preprocessing) and an online part (detection). The offline section covers the pipeline of creating the supporting database and the corresponding vocabulary, while the online section covers the pipeline of the object detection system. Each pipeline consist of several phases. Each phase has a specific task, which converts the given input to an output for the next phase. All phases for both offline and online are explained below with their corresponding input and output in a separate subsection. There is also two independent subsections besides the above two, namely the correspondence between the inputs, which runs parallel with the online phase and the retrieval refinement.

3.1 Offline Section

The offline section's pipeline inhere from three main phases.

1. Depth View Generation
2. Edge View Generation
3. Vocabulary and Descriptor creation

You can see the pipeline in *figure 3.1.* This part is offline because it runs only once before the online part of the system is used. Therefore for example speed is not a critical issue in this stage of the system. However without this part the whole online pipeline would be useless. The input is a set of 3D meshes, from which the vocabulary and the corresponding supporting database is generated. The supporting database consists of several local descriptors for each object in the database. The three main phases are described below in the subsections.

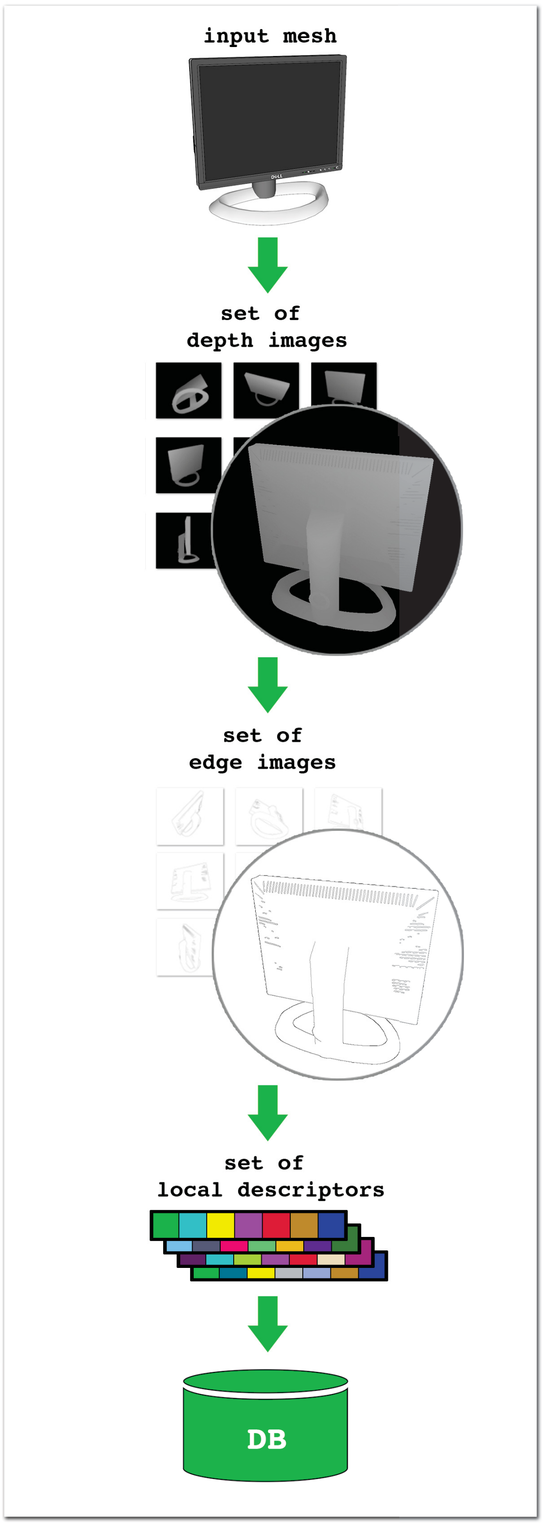


FIGURE 3.1: The offline pipeline

3.1.1 Depth View Generation

The input and output of this phase of the offline pipeline can be seen in *table 3.1* below.

Input	Output
x Set of 3D meshes	$102 \times x$ Set of depth images

Table 3.1: Input and output of phase 1 in the offline section of the system, where x is the number of input 3D meshes.

As *table 3.1* states, to generate the database a set of 3D meshes is required as an input. In this thesis this set contains meshes of office objects, like computer screens, keyboards, mugs, etc. A sample set can be seen in *figure 3.2* below. The size of the database should be at least 100 to 200 meshes to have enough samples for generating the vocabulary *(more on this later in subsection 3.1.3.)*

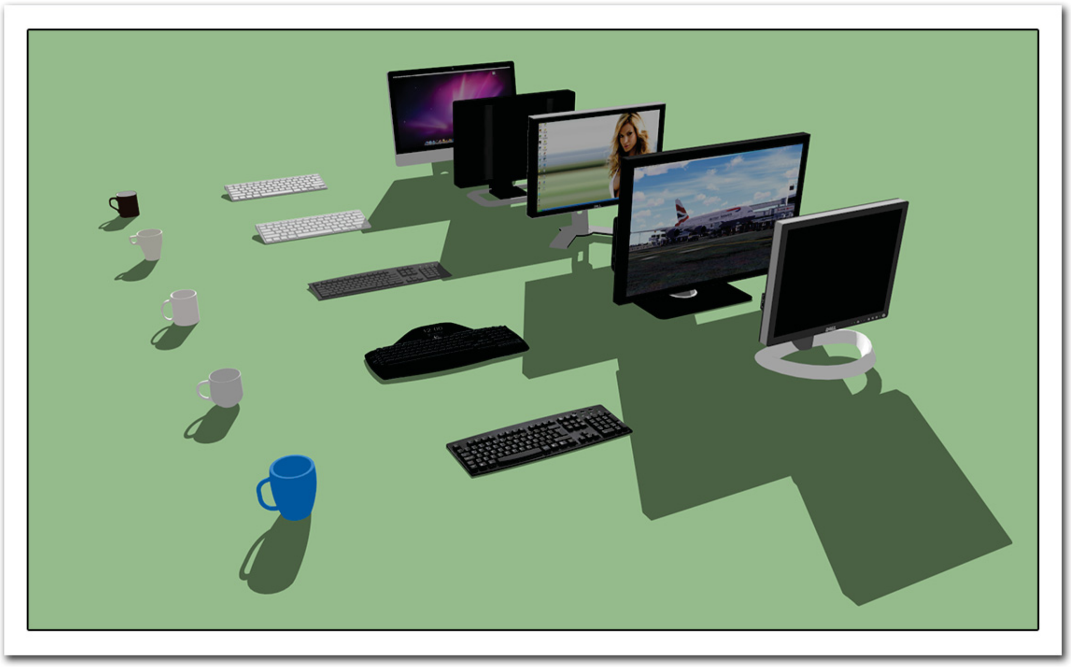


Figure 3.2: Sample of the office 3D mesh database set.

During the database generation each mesh is loaded and scaled to fit inside the center of a unit sphere. This sphere is tessellated in a way that it has 102 vertices. A visualization can be seen in *figure 3.3* off the tessellated sphere. Each frame the camera is placed to a different vertex of the sphere looking at the center with an up vector of $(0, 1, 0)$. As soon as the camera has been at each vertex a new model is loaded and the procedure repeats itself. From each vertex the camera creates a snapshot of the model's depth map.

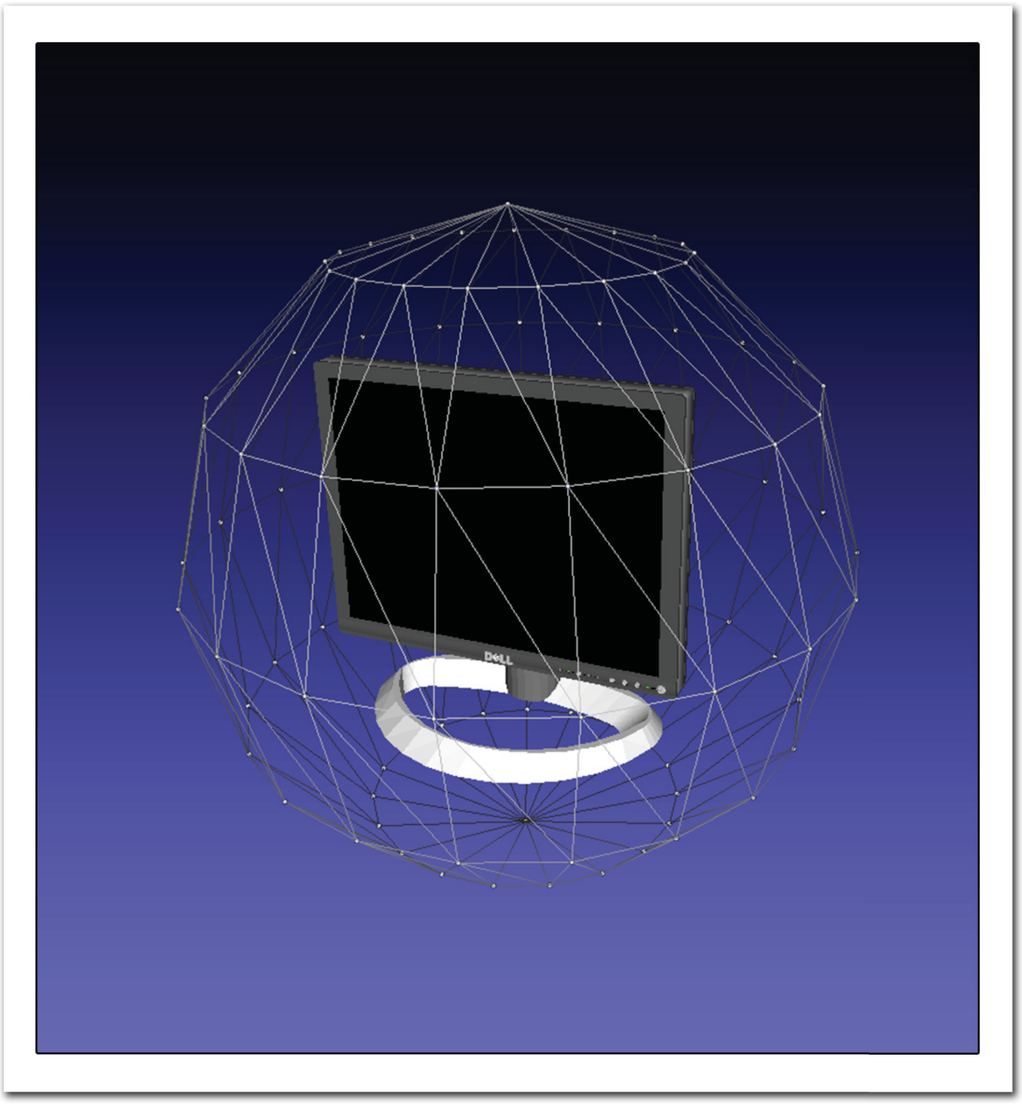


FIGURE 3.3: Tessellated unit sphere with an object scaled and centered inside it.

The final output is a set of depth images of each mesh from the input set. Precisely as *table 3.1* says, 102 depth image of each mesh. A snapshot of the output can be seen in *figure 3.4* below.

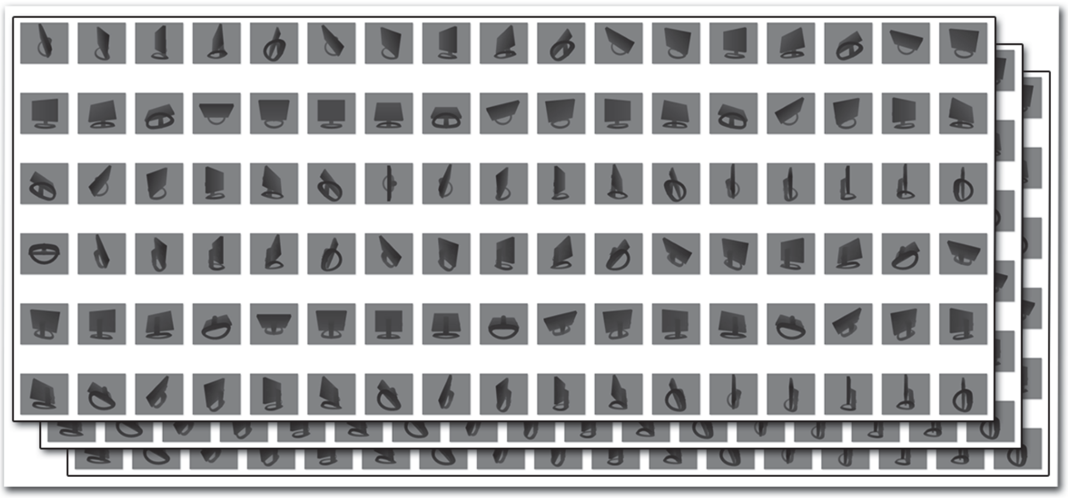


Figure 3.4: Snapshot of the database depth image series.

3.1.2 Edge View Generation

The input and output of this phase of the offline pipeline can be seen in *table 3.2* below.

Input	Output
$102 \times x$ Set of depth images	$102 \times x$ Set of edge images

Table 3.2: Input and output of phase 2 in the offline section of the system, where x is the number of input 3D meshes.

The next step is to detect the edges in the generated depth images in the previous phase and render them as lines in a separate image. To generate the edge images I tried the following edge detection approaches:

1. Silhouettes
   The problem with these type of curves, that they are not descriptive enough to detect an object with high success rate for the reasons discussed in *Chapter 2.1.2.* For example if we look at the contour of a monitor and a mug in *figure 3.5* even us humans can hardly guess which one is which.

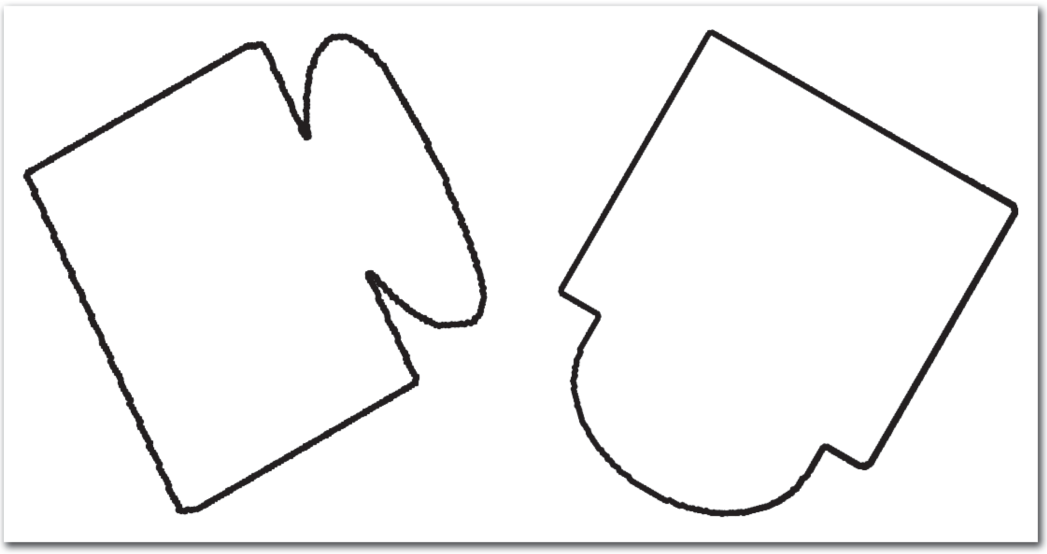


Figure 3.5: Left: monitor boundary curve. Right: mug boundary curve

2. Suggestive contours [5]

   This contour generating technique generates enough edges to be descriptive, but unfortunately the algorithm generates poor contours on models with poor mesh connectivity [2]. See the results of this technique on poor meshes in *figure 3.6.*

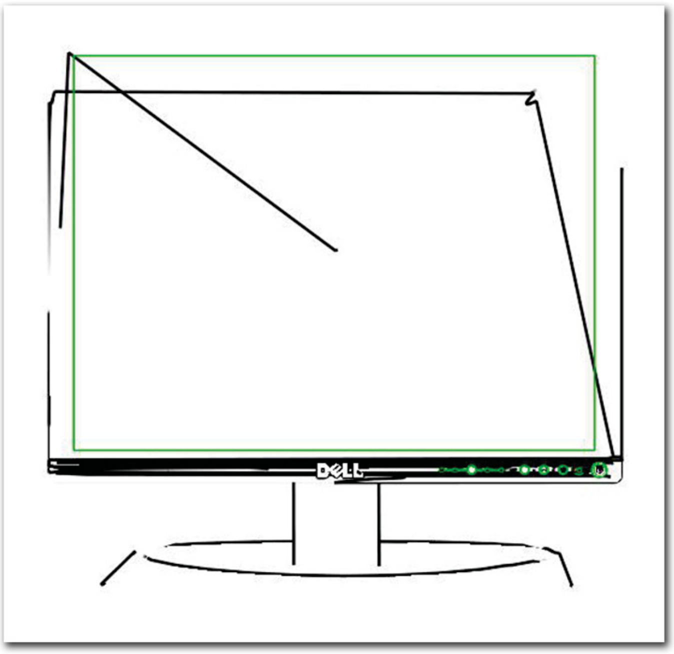


Figure 3.6: Suggestive contour result on a poor mesh.

3. Canny lines [6]
   As it was suggested in this [2] paper, if suggestive contours fail the Canny lines algorithm from depth images is a good alternative, since it also generates enough curves to become descriptive enough. Hence to generate the edge images I used the Canny lines technique. Results can be seen in *figure 3.7*.

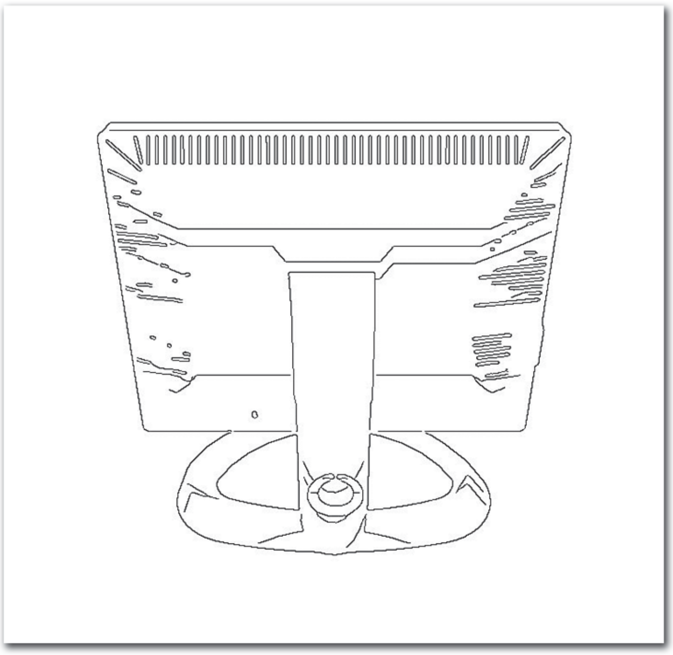


Figure 3.7: Results of the Canny line algorithm

The Canny operator have two thresholds as it was explained in *Chapter 2.2.3*, which need to be fine tuned in order to produce the desired results. If the thresholds are too sensitive unwanted lines can appear in the resulting edge images, due to the intensity jumps in the resolution of the depth image. This is true also the other way around, which means that if the thresholds are set too tough, edges which are holding useful information may not appear in the resulting edge image. My Canny thresholds are 10 for the lower bound and 30 for the upper bound. These threshold values are for the Canny operator used in the OpenCV library. These threshold values produce similar results for both the database generated depth images and for a real captured object's depth images, which will be explained in *Chapter 3.2.3*.

The final output is a set of edge images similarly to the previous phase. As *table 3.2* states this phase generates the same amount of edges images as the number of input depth images. A snapshot of the output can be viewed in *figure 3.8*.

3.1.3 Vocabulary and Descriptor creation

The input and output of this phase of the offline pipeline can be seen in *table 3.3* below.

Input	Output
$102 \times x$ Set of edge images	$102 \times x$ Set of local descriptors and the corresponding vocabulary

Table 3.3: Input and output of phase 3 in the offline section of the system, where x is the number of input 3D meshes.

To generate the local descriptors and the vocabulary I am using the technique described in *Chapter 2.1.3*. In this chapter I will quantify the general variables used in that chapter by the guide of [3].

So far each input model has 102 different images containing its corresponding Canny lines, these are called the edge images. Now for each image a descriptor will be generated, called as the local descriptor. These local descriptors are generated as follows.

Each image is divided into local image patches. $32 * 32 = 1024$ evenly distributed key points are generated on the given image and a patch is created by centering a 4×4 matrix on each key point. The size of the patch is relative to the image size. This value is 0.2, which means that 20% of the image is covered by an image patch. For each patch a corresponding local feature is generated, but first 4 different orientation Gabor filter is applied to the original line image (the parameters of the Gabor filters can be found in [3]). This means that the local feature's final dimension will be a $4 * 4 * 4 = 64$. Each local feature of one image is than quantized against the vocabulary to generate a sparse local descriptor. The resulting histograms are then stored in an inverted index data structure [16] in order to achieve faster query during the online stage. The dimension of this vector depends on the dimension of the vocabulary, which is 1000 in this system. The vocabulary is generated by randomly sampling 1 million local features. To achieve a dense pile of distinguished local features the number of input 3D meshes should not be less then 100. A summary of these values can be seen in *table 3.4* below.

Variable	Value
Number of key points	1024
Number of tiles in local image patch	16
Size of local image patch	0.2
Number of Gabor filters applied	4
Dimension of local feature	64
Dimension of local descriptor	1000
Size of vocabulary	1000

Table 3.4: The quantified values of *chapter 2.5*

3.2 Online Section

The online section's pipeline has four main phases.

1. Depthmap Processing
2. Segmentation
3. Local Descriptor Generation
4. Retrieval

You can see the pipeline in *figure 3.8*. The online part of the system is where the actual detection takes place. This is the part which is closer to the user, hence fast feedback is an important aspect for example. However the results of this pipeline is highly dependent on the supporting database generated in the offline phase. The input is a (set of) depth map of the currently viewed scene. It is then processed through several phases to convert them into separate objects with corresponding local descriptors. These descriptors are then compared against the database and the best match will be the current result. The system gathers more then one view of the viewed scene, therefore the actual final result is refined by multiple current results. The correspondence between the processed views is also calculated. The below subsections serve as a more spacious explanation of the various phases of the online pipeline.

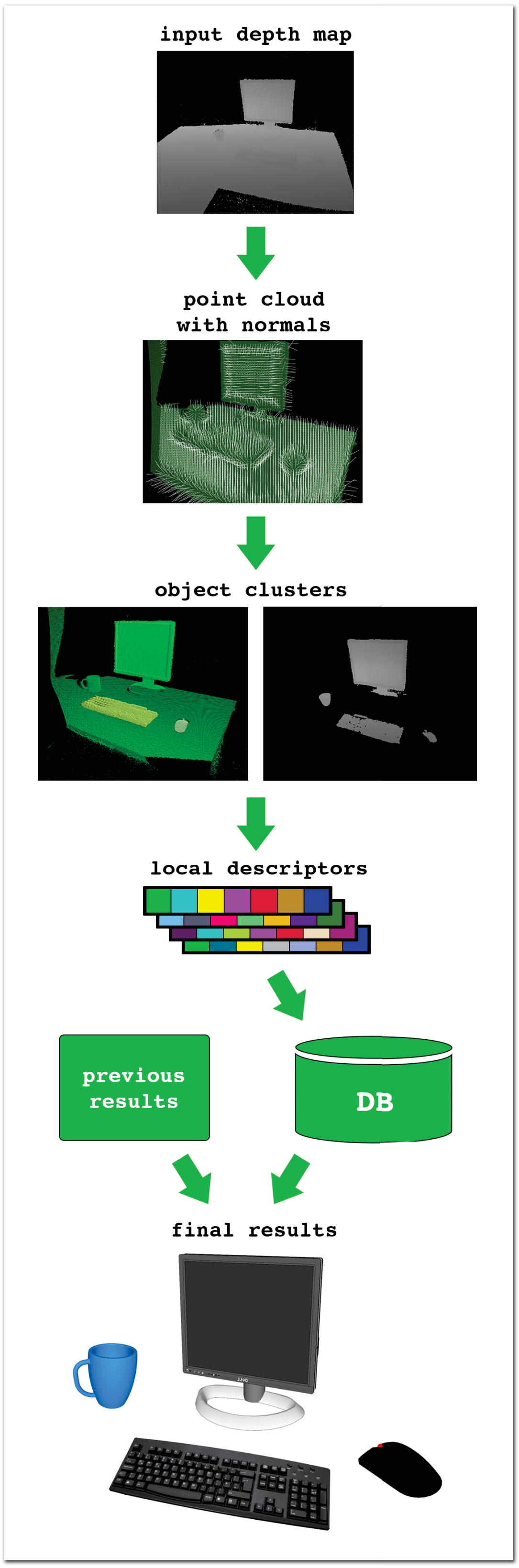


FIGURE 3.8: The online pipeline

3.2.1 Depthmap Processing

The input and output of this phase of the online pipeline can be seen in *table 3.5* below.

Input	Output
Depth image of a scene	Corresponding point cloud

TABLE 3.5: Input and output of phase 1 in the online section of the system.

This phase is a preparation for the segmentation. It starts by projecting the depth value's into 3D Euclidean space to generate a point cloud data structure, as it is explained in *Chapter 2.2.1*. After having the point cloud structure ready, normals are calculated [14] for each point. A result of the calculated normals can be seen in *figure 3.9*. By having the normals calculated the segmentation can take place.

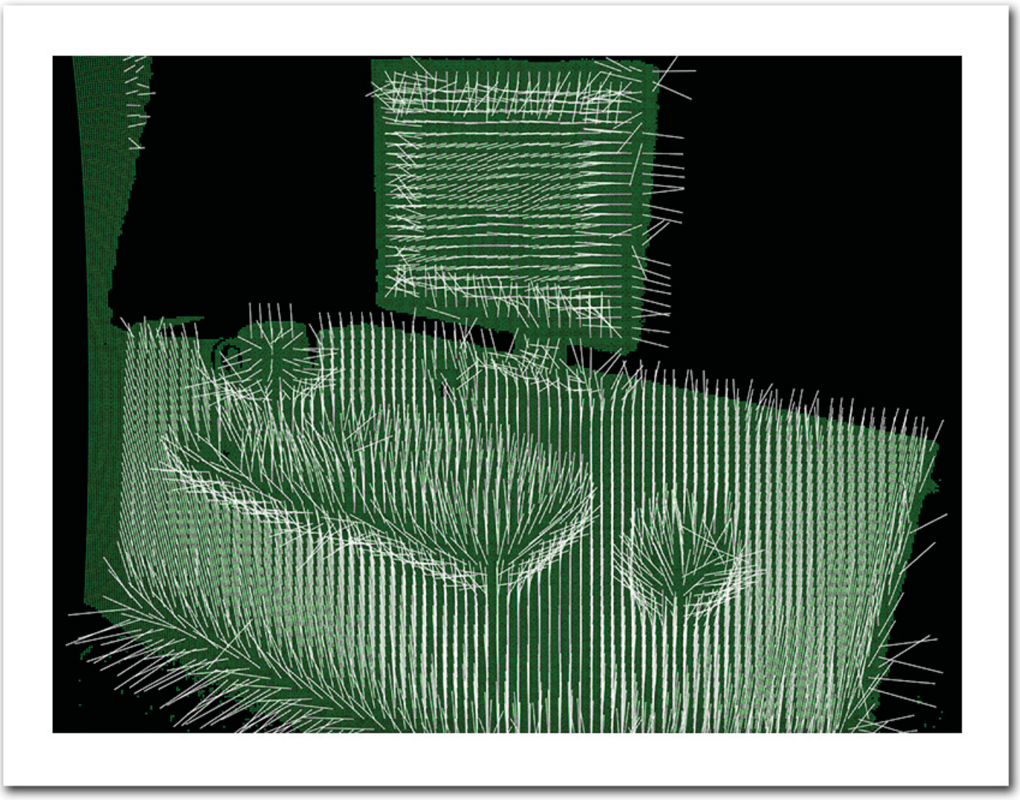


FIGURE 3.9: Result of the normal calculation. White lines represent the normals.

3.2.2 Segmentation

The input and output of this phase of the online pipeline can be seen in *table 3.6* below.

Input	Output
Point cloud of a scene	Object clusters

Table 3.6: Input and output of phase 2 in the online section of the system.

The segmentation part is a key phase of the system, since the detection will depend on its results. Therefore the quality and robustness of the segmentation has to be acceptable. The segmentation part relies on the normals calculated in the previous phase. By having the normals the plane detection can take place. The algorithm of the technique is described in *Chapter 2.1.1.* The result of the algorithm is refined with the technique called RANSAC, explained in *Chapter 2.2.2.* If more then one plane is detected a top to bottom approach is used. First the normals of the detected planes are compared and sorted into two separate list. One which represents the horizontal planes, like floor or table tops and one which represents the vertical planes, like walls for example. After clustering the planes a sorting procedure takes place on the horizontal list. The sorting is based on the altitude of the planes. If plane P_1 is higher, then $P_1 > P_2$, which means that P_1 will be ahead of P_2 in the list. After detecting and arranging the planes in the point cloud the object clustering part is next. The first input for the clustering is the list of the horizontal planes and starts with the first element of the list, e.g. the highest one.

For each plane, which was detected in the previous step a 2D concave hull is calculated, see *figure 3.10 below.* This concave hull is then projected in 3D within a given height as a 3D polygonal prism. All points which are inside this prism are segmented from the point cloud. The coincident is, that these points are actually the points of those objects which are sitting on this plane. Since we know the indices of these points in the original point cloud, a binary cloud can be generated as follows:

$$P(x,y) \in C = \begin{cases} if \quad true, & \text{intensity value is 1} \\ if \quad false, & \text{intensity value is 0} \end{cases}$$

where $P(x,y)$ is a point in the original point cloud and C is the cluster inside the 3D polygonal prism. An example of such binary cloud can be seen in *figure 3.11* below.

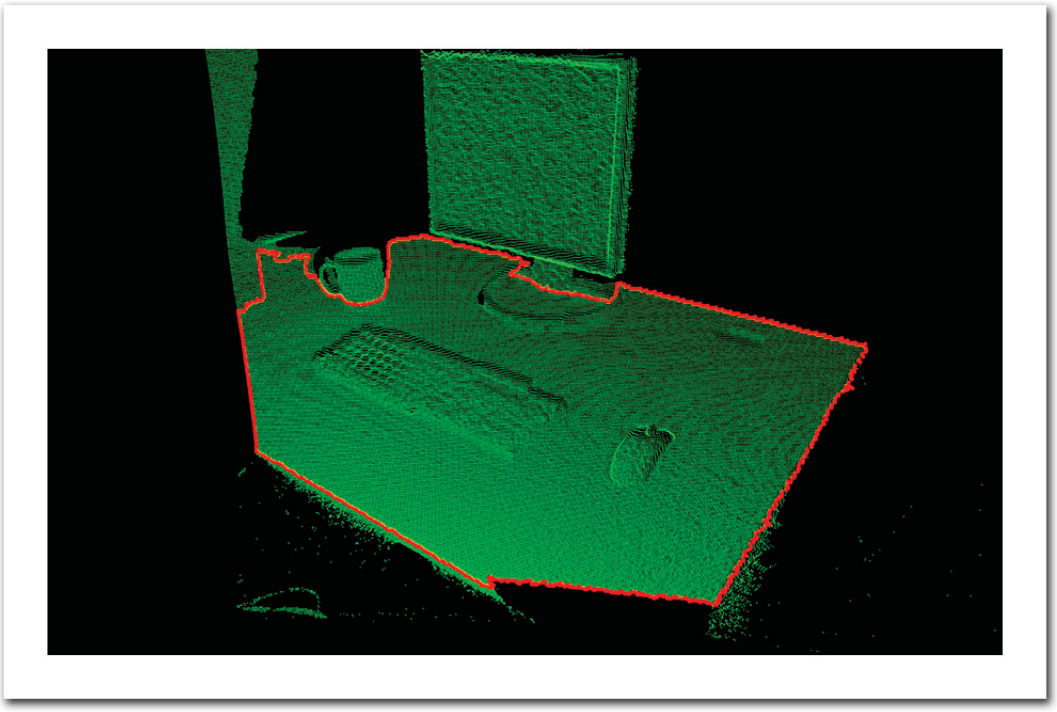


FIGURE 3.10: Result of the plane's concave hull calculation.

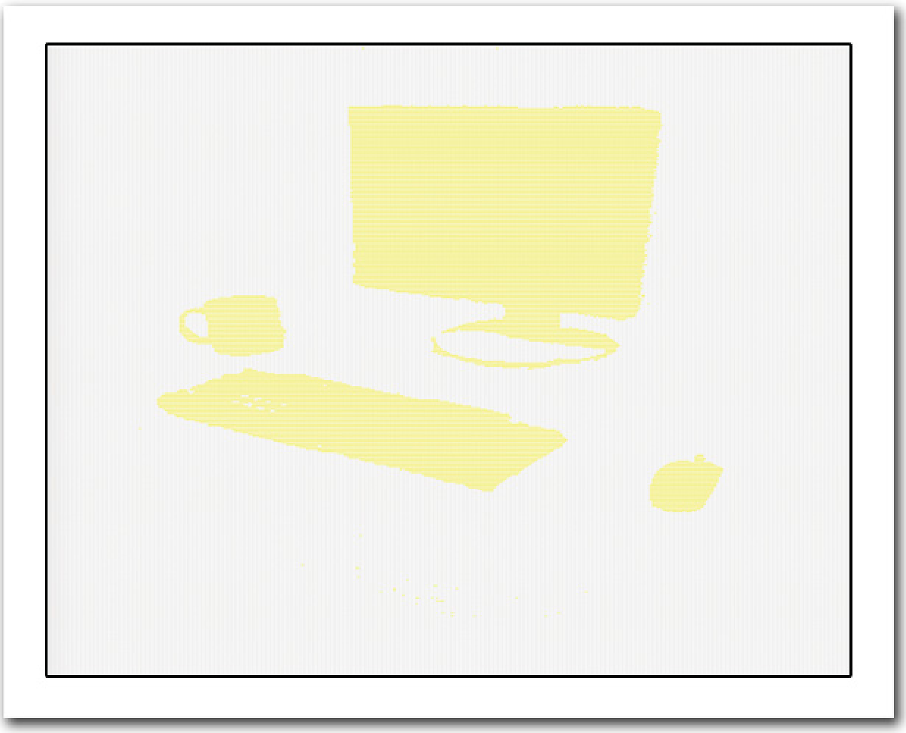


FIGURE 3.11: Visualization of *figure 3.11's* point cloud as a binary cloud in a text editor. Yellow color represents 1's and rest are 0's.

From the generated binary cloud the object clusters can be seen easily. The algorithm goes through the binary cloud and only points with intensity value of 1 are considered. For each of these point's neighbours are accessed in constant time due to the organized property of the point cloud. To generate the clusters the following formula is then applied:

$$C(p, p_n) = \begin{cases} true, & if(p_n \neq 0 \ \&\& \ dist(p, p_n) < t_{distance}) \\ \qquad false, & \text{otherwise} \end{cases}$$

where p represent a point from the binary cloud with an intensity value of 1, p_n is a neighbour of p and $dist(p, p_n)$ is the distance function between two given points, which is a simple Euclidean distance [17]. Based on the above formula a point's neighbour is in the same cluster if its value is 1 and is within a certain distance threshold compared to the given point. If its value is 1, but the distance part of the formula fails, then it means it is most likely two objects overlapping each other and belongs to a separate cluster. On the other hand it still might belong to the same object, but due to depth discontinuity the distance measuring fails. Therefore there are two problems yet what needs some attention.

1. Depth discontinuity
2. If two object overlap each other partially

The first problem can occur if there is an object, which occludes some of its own part, like a computer screen its holder or a table its leg. The second problem is strait forward. There are solutions for both cases, but unfortunately they cancel out each other in a way. The solution for the depth discontinuity is if the clustering of the objects is without any distance data between the neighbouring points in the binary cloud. This means, that the algorithm does not use any distance metric, but then overlapping objects would be considered as one. Again, introducing a distance metric would solve the overlapping problem, but as it was stated above it fails when depth discontinuity comes along. An observation was made about the point clouds generated by Kinect v2, the second generation depth camera from Microsoft. It tends to create some tail like noise in the point cloud, where discontinuity happens. As a result of this a well set distance threshold will still consider a monitor's holder as one object with the monitor and two objects which are overlapping each other. The result of the clustering can be seen in *figure 3.12*.

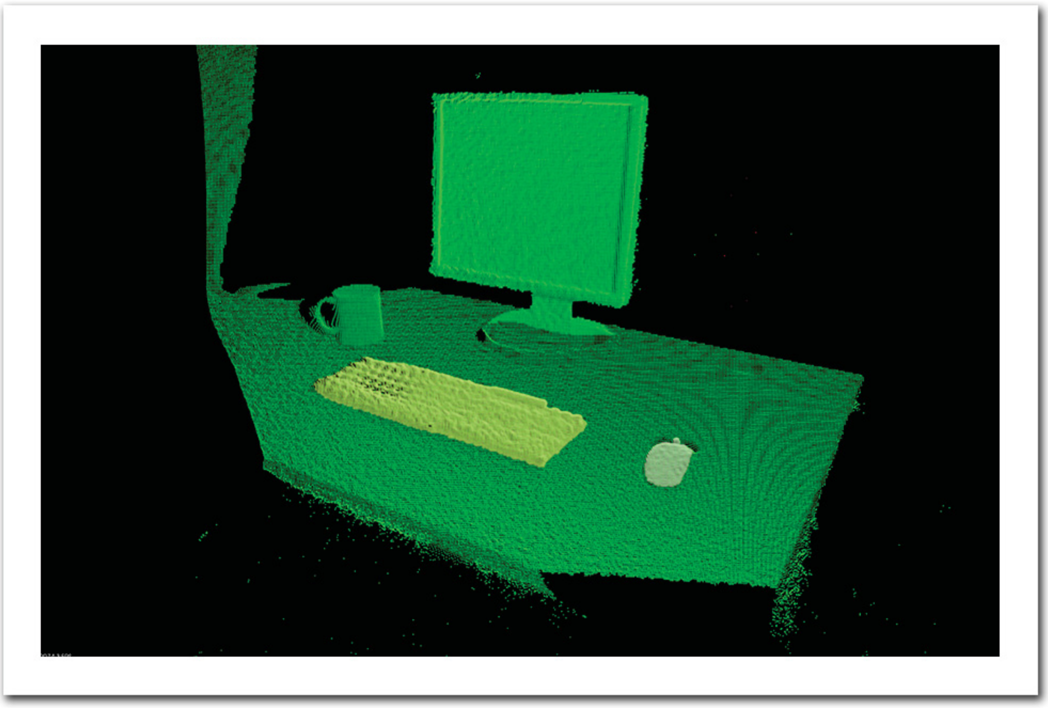


FIGURE 3.12: Object clustering results. Each cluster represents a different color, in total 4.

3.2.3 Local Descriptor Generation

The input and output of this phase of the online pipeline can be seen in *table 3.7* below.

Input	Output
Object clusters	Local descriptors

TABLE 3.7: Input and output of phase 3 in the online section of the system.

The input of this phase is a set of object clusters. These object clusters are then converted back to their depth image state. Since the Kinect V2's depth image resolution is only 512 $\times$ 424, an object cluster's corresponding depth image is relatively small. Therefore these depth image patches are scaled up uniformly to have their width 256 pixel wide. The ratio between the original depth image patches and the scaled version can bee seen in *figure 3.13* and *figure 3.14*. Note that there is also an empty border added to the scaled image.

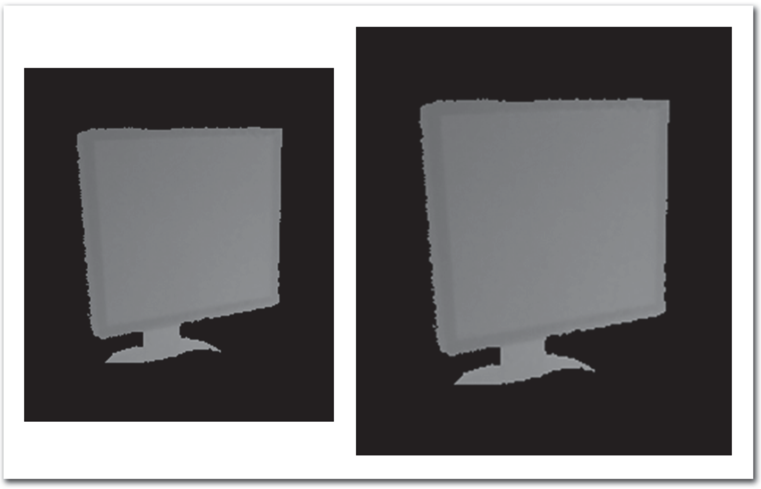


Figure 3.13: Left: original depth image patch. Right: scaled depth image patch.

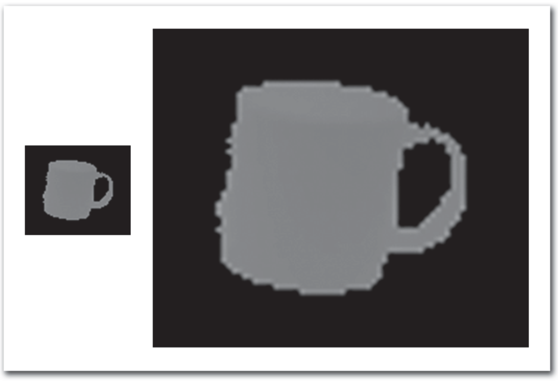


Figure 3.14: Left: original depth image patch. Right: scaled depth image patch.

After having the scaled depth image patches, the same edge detection algorithm is applied as in the offline section, namely the Canny line algorithm. A visualization of the final result can be seen in *figure 3.15* below. The local descriptors are calculated for each line image as it is defined in *Chapter 2.1.3* with the values of *table 3.1.3*.

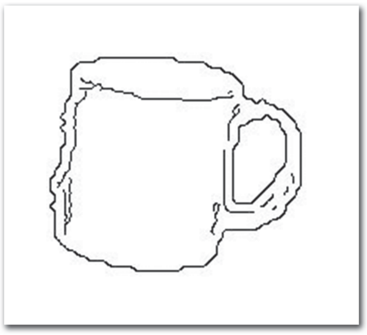


Figure 3.15: Edge image of *figure 3.13*.

The similarity between the database and the online phases's edge images can be seen in *3.16* below.

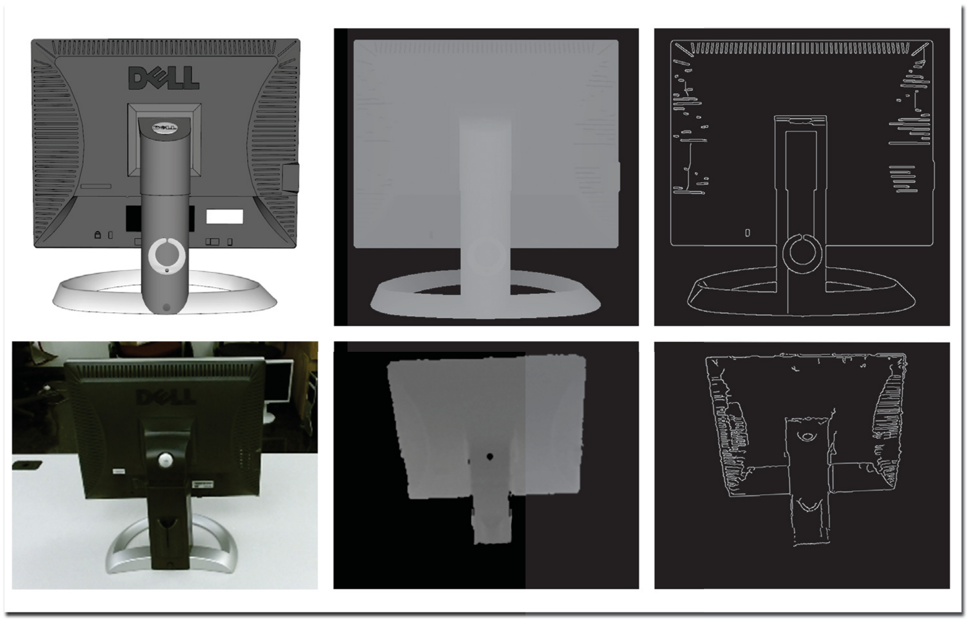


FIGURE 3.16: Top database generation, bottom online phase generation. From left to right: model, depth map of the model, corresponding edge image.

3.2.4 Retrieval

The input and output of this phase of the online pipeline can be seen in *table 3.8* below.

Input	Output
Local descriptors	Detection results

TABLE 3.8: Input and output of phase 4 in the online section of the system.

The values of the local descriptors represent raw word counts of the given visual word. This can be misleading since some word might need bigger weights as it can be more important in the given context. Therefore I am using the tf-idf model (term frequency-inverse document frequency) [16] to determine the weight of a given visual word. The idea behind using this technique is that a visual word is more important, which appears more often in a sketch, but also less distinctive if it occurs often in the collection [3]. For computing weights the following formula is used [9]:

$$h_j = \frac{h_j}{\sum_i h_i} log(\frac{N}{f_i})$$

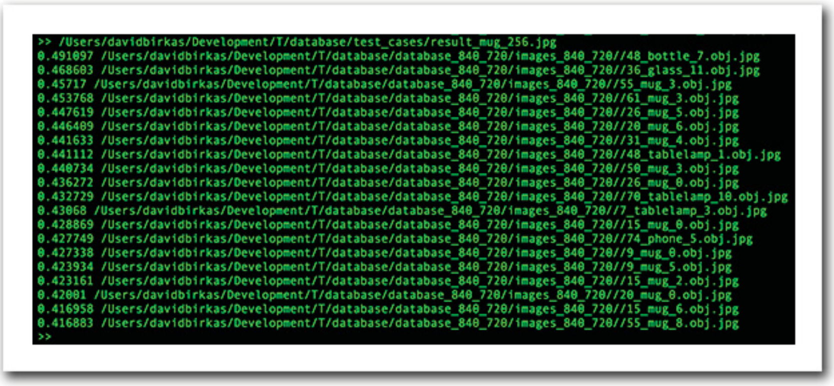
where h_j is an entry of a local descriptors, N is the total number of views in the collection and f_i is the frequency of visual word j in the whole collection.

After calculating the right weights a similarity metric is used to measure the similarity between two local descriptors. If D_1 and D_2 are two local descriptors representing an image, then the similarity between them is calculated with the following formula [16]:

$$S(D_1, D_2) = \frac{\langle D_1, D_2 \rangle}{\|D_1\| \, \|D_2\|}$$

This means, that two local descriptors are similar if they point into the same direction. All descriptors are normalized, so descriptors with higher word counts are not benefiting from it.

In the end the local descriptors are compared against the local descriptors in the supporting database with the above similarity metric. The comparison is fast due to inverted index structure created during the database generation. The final top 20 views, whose local descriptor was the most similar with the examined object's local descriptor is evaluated to have a final result. The decision is made by counting the occurrence of each object type. The one with the most occurrence is the final result of the object detector. This also means that it does not have to be the object which had the smallest difference. For example in *figure 3.17* the object which is getting detected by the system is a mug. The first two results are not mugs, but the majority in the top 20 result is actually a mug, hence the current result is a mug.



```
>> /Users/davidbirkas/Development/T/database/test_cases/result_mug_256.jpg
0.491097 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//48_bottle_7.obj.jpg
0.468603 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//36_glass_11.obj.jpg
0.45717 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//55_mug_3.obj.jpg
0.453768 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//61_mug_3.obj.jpg
0.447619 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//26_mug_5.obj.jpg
0.446409 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//20_mug_6.obj.jpg
0.441633 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//31_mug_4.obj.jpg
0.441112 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//48_tablelamp_1.obj.jpg
0.440734 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//50_mug_3.obj.jpg
0.436272 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//26_mug_0.obj.jpg
0.432729 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//70_tablelamp_10.obj.jpg
0.43068 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//7_tablelamp_3.obj.jpg
0.428869 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//15_mug_0.obj.jpg
0.427749 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//74_phone_5.obj.jpg
0.427338 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//9_mug_0.obj.jpg
0.423934 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//9_mug_5.obj.jpg
0.423161 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//15_mug_2.obj.jpg
0.42001 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//20_mug_0.obj.jpg
0.416958 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//15_mug_6.obj.jpg
0.416883 /Users/davidbirkas/Development/T/database/database_840_720/images_840_720//55_mug_8.obj.jpg
>>
```

Figure 3.17: Testing the retrieval system by inputting *figure 3.16* image patch of a mug

3.2.5 Correspondence

The input and output of this separate phase of the online pipeline can be seen in *table 3.7* below.

Input	Output
Point cloud	Transformation matrix

Table 3.9: Input and output of the correspondence in the online section of the system.

This phase runs parallel to the above explained online pipeline. Its responsibility is to be able to localize the camera in space and to be able to tell the correspondence between detected objects in different frames.

As *table 3.7* claims the input of this phase is a point cloud. This point cloud is registered to the previous frame's point cloud with the technique discussed in *Chapter 2.1.4* and *Chapter 2.2.5*. It is important to note, that the transformation matrix is calculated always to the previous frame and then it is multiplied with the previous frame's transformation matrix to obtain the global transformation.

$$T_1 * T_2 * ...T_n = T_{global}$$

T_1, T_2, ... T_n represents the local transformation matrix of the current frame and T_{global} represents the global transformation matrix. It is also important to note, that there is no T_0, since the first transformation is applied after the second frame.

After the registration is done the transformation matrix is obtained. The resulting transformation matrix is applied to each object detected in the current frame. After the transformation matrix is applied to the detected object clusters a simple distance calculation can prove which cluster corresponds to which previous detected object clusters, if any. This distance is a simple Euclidean distance, which was discussed in several chapters above already. The distance is measured between the centroid of the clusters. As always there is a certain threshold for the distance. If two centroids are within this certain threshold, then the two cluster represent the same object. In my application this threshold is 0.05, which represents 5 cm in the point cloud space. An illustration of this phase can be seen in *figure 3.18* below.

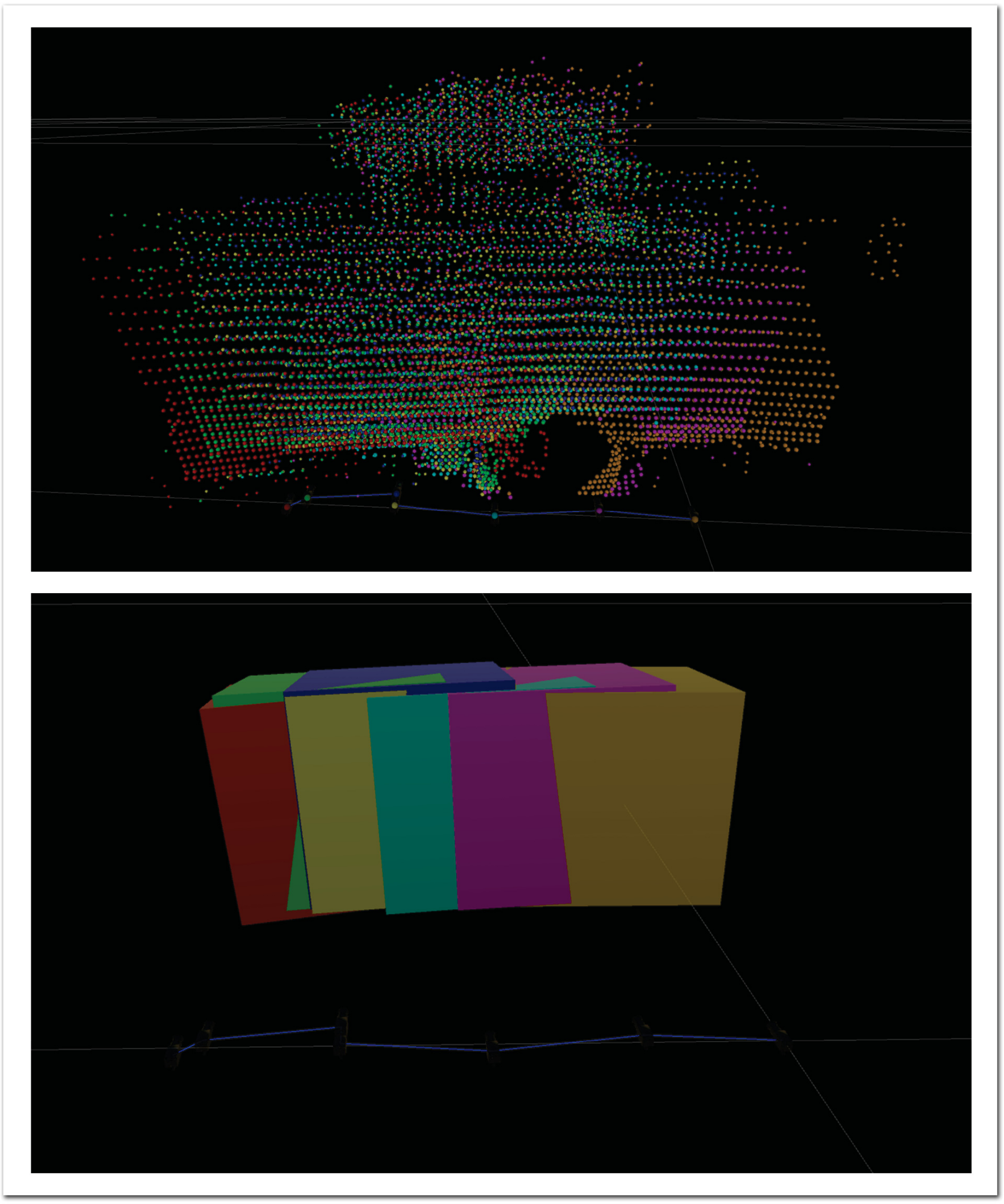


Figure 3.18: Top: Each color represents a separate frame and each camera is marked with the corresponding frame's color. Bottom: The same image, but instead of visualizing the point clouds cubes are used to be able to see the transformations more clearly.

3.2.6 Retrieval refinement

The input and output of this separate phase of the online pipeline can be seen in *table 3.10* below.

Input	Output
Results of the online phase	Refined results

TABLE 3.10: Input and output of the refinement phase in the online section of the system.

This is an independent phase of the above presented system. It tends to refine the output of the online part's current results. The refinement works in an iterative way on top of the online pipeline. The input as it is in *table 3.10* is a set of results from the online pipeline. The different input results are separated into vectors. Each vector holds results to the same corresponding object in the scene. The acquisition of the proper correspondence between these results are described in the previous chapter, namely *Chapter 3.2.5*. There has to be at least two result vectors to have the results refined. As a result of this there is no refinement phase after the results of the first processed input depth map frame of the online phase. Furthermore the refinement of a result list of vectors triggers only for the objects, which are the current results of the online pipeline.

To illustrate the above mechanism the following simple example will guide through the process. Suppose there were several depth map frames processed via the online phase already. Therefore there should be already several list of vectors with results of the corresponding object. A new input depth frame is then processed. There were two objects clusters at the end with a result set. Both set of results are then added to the right list of result vectors. Clearly if there was no corresponding list of result vectors yet to the given object, then a new one is created. As soon as the new results are added to the proper list the refinement mechanism triggers.

The refinement takes the result list vectors as an input and finds the result in the same way as in *Chapter 3.2.4*. The difference is that now there are $x * top20$ results, where x is the size of the list and $x > 1$. An example of this can be seen below.

$$\begin{bmatrix} 5 \\ 0 \\ 3 \\ 6 \\ 0 \\ 2 \end{bmatrix} + \begin{bmatrix} 8 \\ 1 \\ 0 \\ 3 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 13 \\ 1 \\ 3 \\ 9 \\ 0 \\ 3 \end{bmatrix}$$

Where the first two vector represents two result vector of the same object cluster from a different view point. Each dimension of the vector represents the occurrence of a given type of object. This means, that there are 6 type of objects in this example. The result of the two result vector is the refined result. As it can be seen the first two vector would have different results and after refinement the entry zero becomes the dominant.

My observation was, that the retrieval refinement improved the successful detection rate. Some input depth frames might be from a less advantageous view and hence it can influence the detection. Considering multiple views as the final result can overcome this issue. The following images in *figure 3.19* demonstrates the difference between one view and multiple view results.

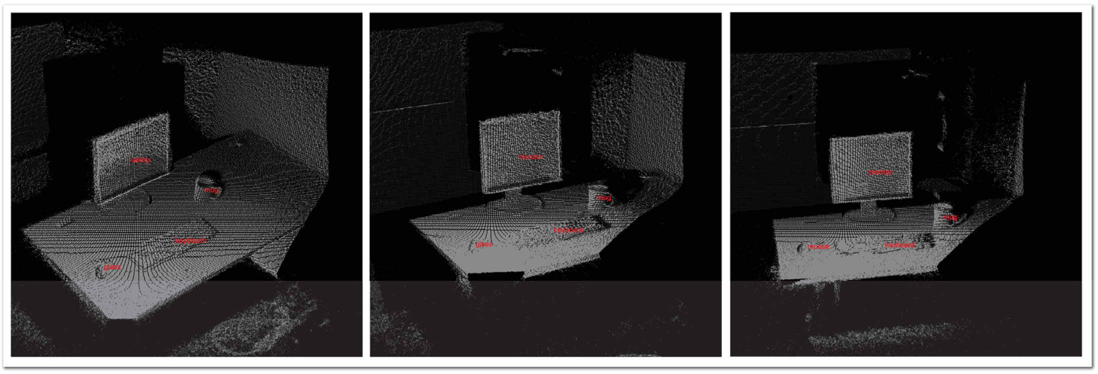


FIGURE 3.19: From left to right: 1., 2., 3., input frame. Frame 1. gets false positives for the monitor as laptop and for the mouse as glass. Frame 2. refinement phase fixes false positive for the monitor. Frame 3. refinement phase fixes the false positive for the mouse.

Chapter 4

Implementation Details

My chosen language for developing both parts of the application (online part - object detection, offline part - database generator) was C++. As for developing environment I used Microsoft Visual Studio 2013 under Windows 8.1 operating system. The reason for this is, that the new Kinect V2 camera's API requires at least Windows 8 and Microsoft Visual Studio 2013. I also tried some open source API's for the new Kinect, like libfreenect but I didn't find them reliable enough yet. Still I wanted to have the chance to make the application cross platform in the future, therefore I designed an interface between the Windows dependent Kinect code and the application's other parts. As for an application framework I used the well known library Qt. For visualization and artificial depth map generation I used OpenGL. All the point cloud processing tasks are done with PCL [28], the point cloud library. Since PCL library has no official release for Microsoft Visual Studio 2013 I had to create my own version. Edge detections and image manipulations are done with OpenCV. Additional library used in the project is Boost C++. A summery of the used libraries and APIs can be seen in *table 4.1* below.

Library/API	Version
Kinect	2.0
OpenGL	2.1
PCL	1.7.2
OpenCV	2.4.9
Boost C++	1.57.0

TABLE 4.1: Summery of the libraries and APIs used in the project.

It is also important to note, that the application, which realizes the online part of the system is a multithreading application. This is important in order to achieve real-time feedback to the user. There are three important threads in the application.

- The main thread, which is responsible for the user inputs and the rendering, plus converting each depth map frame into a point cloud structure and calculating the corresponding normals. This thread is running with 28-30 fps.
- The registration thread, which is responsible for finding the transformation between the input frames, so the camera can be localized and the correspondence can be found between the detected objects. This thread is running with 20-22 fps.
- The processing thread, which is responsible to process the input frame by segmenting it into separate object clusters and identify them by object type. This thread is running with 2-3 fps.

Input frames above consist of the point cloud and the corresponding normals. There is also communication between some of the threads. The processing thread for example queries the registration thread about the actual global transformation to be able to find correspondences between the identified and the already identified objects. This is important from the refinement phase of point of view. On the other hand the main thread for visualization purposes queries the processing thread for the identified object results to be able to show feedback to the user and to be able to indicate any changes happened during the refinement phase. An illustration of the system can be seen in *figure 4.1* below.

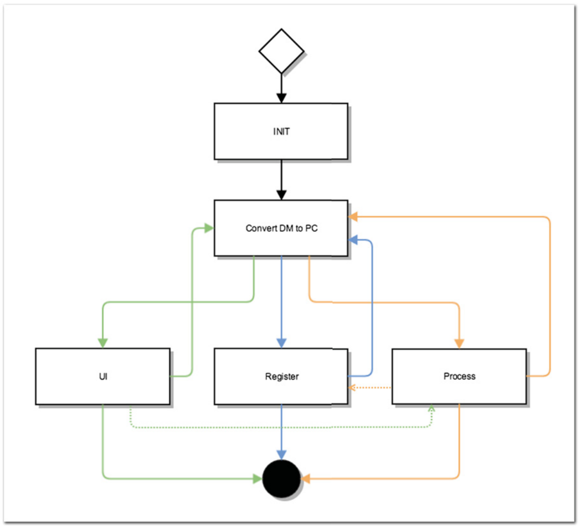


FIGURE 4.1: A flow diagram of the system. Each color represents a different thread. Dotted lines mean, that the thread is depending on some variables from the other thread.

Chapter 5

Discussion

In this chapter some analysis will be provided of the system, limitations will be discussed and a summary of the results with some evaluation will be provided. The chapter will finish with some thoughts on future work. Limitations will cover the edge cases of the applications and cases where failure can happen. The results section will cover an evaluation of the used techniques and how they improved the results. There will be words about the difference between the one view and multiple view results. Last but not least some future improvements will be discussed.

5.1 Analysis

As it was described in *Chapter 4* the system has three threads. Out of these three threads two is basically real time, which means 15+ fps. The third thread is running with 2 fps. This thread is responsible for the segmentation and detection part. While the corresponding application is designed in a way, that the user doesn't necessarily notices this slowness, it would be feasible to speed up this part of the system.

On the other hand, even with 2 fps the system is accurately detecting the objects in the scene. Some objects are detected correctly from the first frame and some are detected correctly after several frames. This is due to the fact that some views are more descriptive of a given object, then others. Therefore it is important to mention, that as more frames come into the system the detection's reliability grows.

The system is also robust against occlusion unlike the Fourier descriptor method [4] described in *Chapter 2.1.1.* The system is capable of detecting large (e.g. chairs) and small (e.g. cups) objects and is scale and rotation invariant unlike this [32] method also mentioned in *Chapter 2.1.1.*

Last but not least the presented system is an extension of a state of the art sketch based method [3], by automatizing the sketch generation and implanting it into a full object detection pipeline.

5.2 Limitations

The limitations of the system in the current stage are the following:

- Detecting certain very small objects is hard due to the noise of the input depth map and errors of the plane detection. Objects, which can belong into the "very small" object category do not extrude enough from the supporting plane, like cell phones or pens laying on a table for example. This means, that during the plane detection phase they can get neglected and counted as part of the plane.
- On the other side, the noise from the depth camera can influence the plane detection algorithm and corners of the table can be falsely detected as objects.
- Also very dense scenes where objects are touching and are close together are very hard to segment rightfully. Therefore on these kind of scenes the system can fail to segment and detect the right objects.

5.3 Evaluation and Results

This section describes the main tricks, which improved and made the system more robust. The methods will be discussed in a bottom to top fashion.

First of all, one of the key phases in the system is the segmentation of the objects. The object segmentation relies on detecting the supporting plane structure. Using only the plane detection method explained in *Chapter 2.1.1* was not enough, because the algorithm is not able to fully filter out points, which are not part of the plane, but it gives a reasonable good guess. Using the resulting points of this phase, I used an additional refinement phase with the help of the well known RANSAC algorithm, explained in *Chapter 2.2.2*. Using RANSAC alone on a big point cloud data as obtained from the Kinect is too slow, but using it on a smaller point set is satisfying and fast.

An other important aspect, which should be highlighted is the chosen descriptor. When I first started implementing the system I used the a descriptor called Fourier descriptor [23], described in *Chapter 2.1.2*. This descriptor works fine until the case of partial matching does not come into the picture. Since this descriptor relies on the silhouette or in other words the contour of the object, as soon as the input is just a part of the given object it fails to detect it and gives false positive results. On the other hand GALIF features and the bag of feature method does not depend on full object observation and partial matching is way more robust and reliable. Therefore I switched to this feature and method instead, which improved the results successfully.

The biggest improvement and contribution of this thesis is the using of multiple views. Each view is processed and evaluated separately, but after each processed phase a refinement is taking place. This refinement is considering all previous results and reevaluates the results if necessary. This contribution improves the accuracy of the detection and filters out false positive detections, which makes the system more reliable. The tests showed, that the accuracy of the presented object detection pipeline is 75%, but 50% of the failed cases are actually fall into a similar object type, like detecting a cup instead of a mug or detecting a laptop instead of a monitor.

As a conclusion, tracing back the object detection dilemma to a sketch based object retrieval problem, combining it with additional techniques from the filed of computer graphics and computer vision is a viable approach. Some results can be seen in *figure 5.1* below.

5.4 Future work

One of the most important things which would largely improve the robustness and reliability of the presented system is improving the segmentation part. There are several other techniques out there, which address a solution to this problem, like [24]. Hence trying out some other methods should be competent. To make the system more fast and more real-time, some parallelization could be added. There are several phases in the system, which could be implemented on GPU, like one of the point cloud processing steps. Since right now the whole process of detecting without the registration and refinement part is 2 fps. As from a development point of view an additional feature, like reconstruction would greatly boost the benefits of the application. The system is already prepared for such a feature, just some additional development is need to find the proper oriented bounding boxes for each object cluster and replace it with a polygonal model from the supporting database. Along with this, there are endless range of possibilities to shape, improve the system in a way so it can become beneficial to a certain user base.

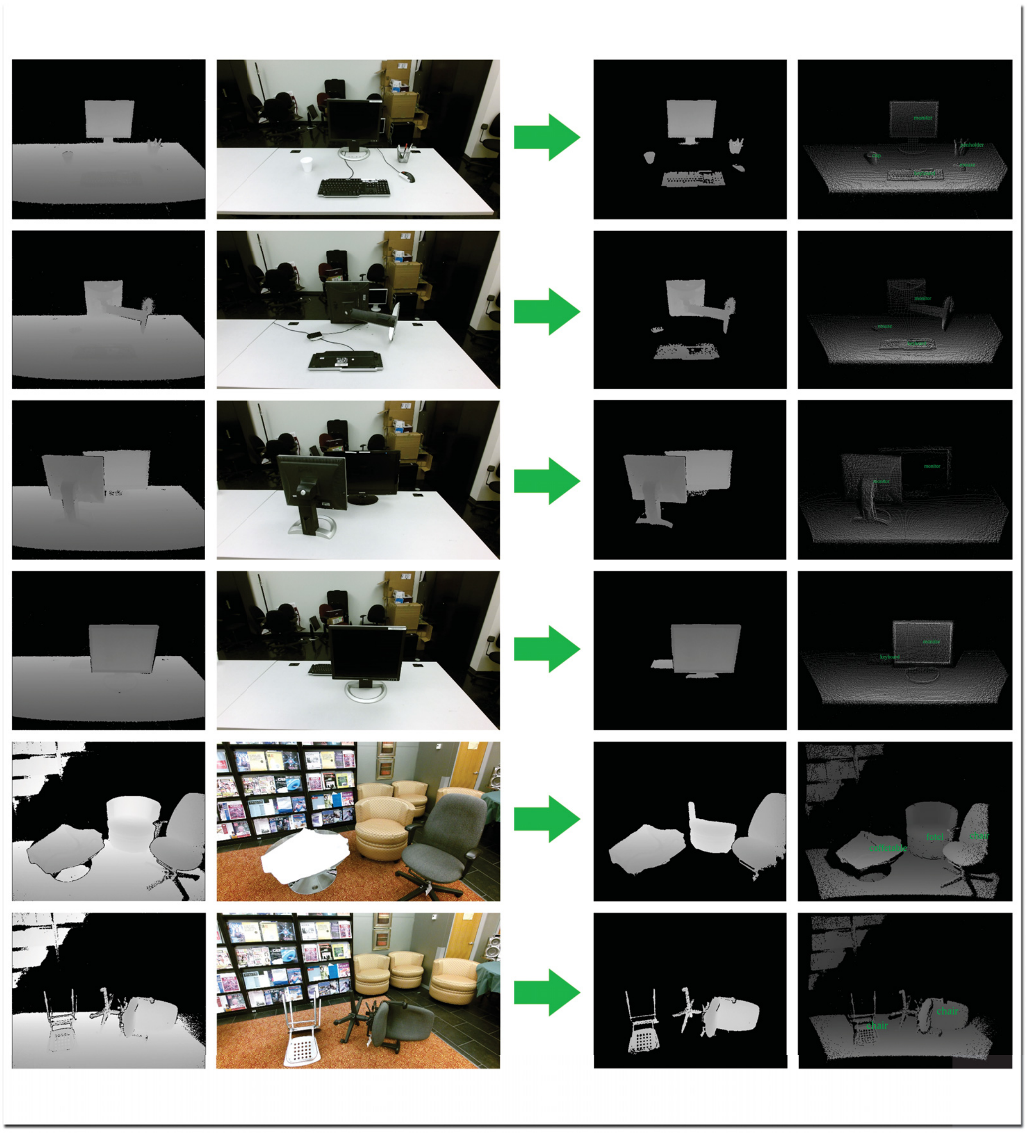


FIGURE 5.1: Some example results of the system. From top to bottom: general office scene, general office scene with objects not in their standard position, general office scene with an occlusion example (1), general office scene with an occlusion example (2), general furniture scene, general furniture scene with objects not in their standard position.

Bibliography

[1] Trevor, a, Gedikli, S., Rusu, R. B., Christensen, H. I. (2013). Efficient organized point cloud segmentation with connected components. Proceedings of Semantic Perception Mapping and Exploration, 16.

[2] Xu, K. (2012). Sketch2Scene: Sketch-based Co-retrieval and Co-placement of 3D Models.

[3] Eitz, M., Richter, R., Boubekeur, T., Hildebrand, K., Alexa, M. (2012). Sketch-based shape retrieval. ACM Transactions on Graphics, 31(4), 110.

[4] Chen, D.-Y., Tian, X.-P., Shen, Y.-T., Ouhyoung, M. (2003). On Visual Similarity Based 3D Model Retrieval. Computer Graphics Forum, 22(3), 223232.

[5] DeCarlo, D., Finkelstein, A., Rusinkiewicz, S., Santella, A. (2003). Suggestive contours for conveying shape. ACM Transactions on Graphics, 22(3), 848.

[6] Canny, J. (1986). A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(6), 679698.

[7] Lindeberg, T. (1996). Edge detection and ridge detection with automatic scale selection. Computer Vision and Pattern Recognition, 1996. Proceedings CVPR 96, 1996 IEEE Computer Society Conference on, 30(2), 465470.

[8] Squire, D. M., Mller, W., Mller, H., Pun, T. (2000). Content-based query of image databases: Inspirations from text retrieval. Pattern Recognition Letters, 21(13-14), 11931198.

[9] Sivic, J., Zisserman, A. (2003). Video Google: a text retrieval approach to object matching in videos. IEEE International Conference on Computer Vision, 14701477. doi:10.1109/ICCV.2003.1238663

[10] Lloyd, S. (1982). Least squares quantization in PCM. IEEE Transactions on Information Theory, 28(2), 129137.

[11] Zhang, Z. (2012). Microsoft kinect sensor and its effect. IEEE Multimedia, 19(2), 410.

[12] Felzenszwalb, P. F., Huttenlocher, D. P. (2004). Efficient graph-based image segmentation. International Journal of Computer Vision, 59(2), 167181.

[13] Hulik, R., Beran, V., Spanel, M., Krsek, P., Smrz, P. (2012). Fast and accurate plane segmentation in depth maps for indoor scenes. 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, 16651670.

[14] Holzer, S., Rusu, R. B., Dixon, M., Gedikli, S., Navab, N. (2012). Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images. IEEE International Conference on Intelligent Robots and Systems, 26842689.

[15] John A. Hartigan. 1975. Clustering Algorithms (99th ed.). John Wiley Sons, Inc., New York, NY, USA.

[16] Witten, I., Moffat, A., BELL, T. 1999. Managing giga- bytes: compressing and indexing documents and images. Mor- gan Kaufmann.

[17] Deza, M. M., Deza, E. (2009). Encyclopedia of Distances. Media (Vol. 2006).

[18] Besl, P., McKay, N. (1992). A Method for Registration of 3-D Shapes. IEEE Transactions on Pattern Analysis and Machine Intelligence.

[19] Systems, I. (1991). Object Modeling by Reg strat ion of Multiple Range Images*, (April), 27242729.

[20] Zhang, Z. (1994). Iterative point matching for registration of free-form curves and surfaces. International Journal of Computer Vision, 13(2), 119152.

[21] Pomerleau, F., Colas, F., Siegwart, R. (2015). A Review of Point Cloud Registration Algorithms for Mobile Robotics. Foundations and Trends in Robotics, 4(1-104).

[22] Rusinkiewicz, S. (2001). Efficient Variants of the ICP Algorithm a r c Levoy.

[23] Zhang, D., Lu, G. (2002). An Integrated Approach to Shape Based Image Retrieval. Computer, 23(January), 16.

[24] Silberman, Nathan; Hoiem, Derek; Fergus, Rob; Kohli, P. (2012). Indoor Segmentation and Support Inference from RGBD Images. Lecture Notes in Computer Science, 7576(Part 5), 746760.

[25] Schnabel, R., Wahl, R., Klein, R. (2007). Efficient RANSAC for point-cloud shape detection. Computer Graphics Forum, 26(2), 214226.

[26] Hast, A., Nysj, J. (2013). Optimal RANSAC - Towards a Repeatable Algorithm for Finding the Optimal Set. Journal of WSCG, 21(1), 2130.

[27] Oehler, B., Stueckler, J., Welle, J., Schulz, D., Behnke, S. (2011). Efficient multi-resolution plane segmentation of 3D point clouds. Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7102 LNAI, 145156.

[28] Rusu, R. B., Cousins, S. (2011). 3D is here: Point Cloud Library (PCL). Proceedings - IEEE International Conference on Robotics and Automation, 14.

[29] Zhang, D., Lu, G. (2001). A comparison of shape retrieval using Fourier descriptors and short-time Fourier descriptors. Advances in Multimedia Information Processing PCM 2001, 24, 855860.

[30] Izadi, S., Davison, A., Fitzgibbon, A., Kim, D., Hilliges, O., Molyneaux, D., Freeman, D. (2011). Kinect Fusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology - UIST 11, 559.

[31] Newcombe, R. a, Molyneaux, D., Kim, D., Davison, A. J., Shotton, J., Hodges, S., Fitzgibbon, A. (2011). KinectFusion: Real-Time Dense Surface Mapping and Tracking. IEEE International Symposium on Mixed and Augmented Reality, 127136.

[32] Li, Y., Dai, A., Guibas, L., Niener, M. (2015). Database-Assisted Object Retrieval for Real-Time 3D Reconstruction. Computer Graphics Forum, 34(2), 435446.

[33] Leng, B., Zeng, J., Yao, M., Xiong, Z. (2015). 3D Object Retrieval With Multitopic Model Combining Relevance Feedback and LDA Model. Image Processing, IEEE Transactions on, 24(1), 94105.

[34] Scovanner, P., Ali, S., Shah, M. (2007). A 3-dimensional sift descriptor and its application to action recognition. Proceedings of the 15th International Conference on Multimedia - MULTIMEDIA 07, (c), 357.

[35] Davison, A. J. (2003). Real-time simultaneous localisation and mapping with a single camera. Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on, 2, 14031410.

[36] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S. (2015). SSD: Single Shot MultiBox Detector. Arxiv.