Escuela
Politécnica
Superior

Towards a real-time 3D object recognition pipeline on mobile GPGPU computing platforms using low-cost RGB-D sensors

Bachelor's Thesis

Bachelor's Degree in Computer Engineering

Bachelor's Thesis

Author:
Alberto García García

Supervisors:
José García Rodríguez
Sergio Orts Escolano

June 2015

Universitat d'Alacant
Universidad de Alicante

UNIVERSITY OF ALICANTE

Bachelor's Degree in Computer Engineering

Bachelor's Thesis

Towards a real-time 3D object recognition pipeline on mobile GPGPU computing platforms using low-cost RGB-D sensors

Author

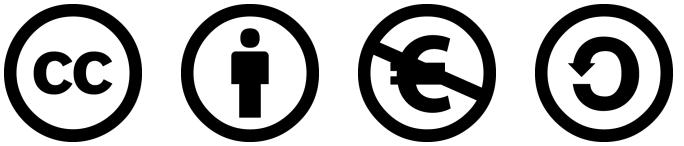
Alberto García García

Supervisors

José García Rodríguez, PhD.
Sergio Orts Escolano, PhD.

Department of Computer Technology (DTIC)

Alicante, June 9, 2015



Let the future tell the truth.
Let the future evaluate each one according to his work and accomplishments.
The present is theirs; the future, for which I have really worked, is mine.
Nikola Tesla

Acknowledgments

For the lazy reader who does not want to be bored with irrelevant details, here you have a generic acknowledgment so that you do not lose any second of your precious time: thanks to anyone who has contributed direct or indirectly to this project, it wouldn't have been possible without you.

For those of you who would like to stop for a while, and listen to what a random student has to say after finishing one of the most important stages of his life: sorry for the mess, and please play *Time* by *Pink Floyd* in the background (a live version if possible) while you read to make it at least worthwhile.

It is difficult to overstate my gratitude to my supervisors, Jose and Sergio. Your enthusiasm, knowledge and great efforts to help me with absolutely everything made this possible. Wherever I end up in a few years, I wouldn't be there if it wasn't for you. I could not wish for better or friendlier supervisors.

In this regard, I would also like to thank Higinio, who introduced me to the research community and offered his help when I was just a freshman, and Jerónimo for his support when we were working together.

I wish to thank all my colleagues for the experiences we have shared. We have been together four though years but we had a lot of fun in the process. Thanks Manu, Caye, Víctor, Alex, Ginés, Jorge, Brayan, Sergiu, Pablo and Javi. You made it ***worth*** every single day. Also thanks to all the good people that I have met either in the classroom or in the laboratory: Marcelo, Vicente, Óscar, Joan Carles, Pablo, Paco, Jose Manuel, Fran, Joselu and Jesús. Always ready for a cup of coffee or a beer.

In addition, would like to express my sincere gratitude to all those good teachers and professors who love their work and try to get the best out of us, either in the University or either in highschool. In particular, thanks to my former teachers and now friends Carlos and Jose Juan, you belong to the rare kind of teachers who are able to inspire students, so never stop lecturing.

Also thanks to NVIDIA for their generous hardware donation, which allowed us to carry out this project using their Jetson TK1 platform.

I would also like to thank the food franchise *Mucho+QuePizza* for providing $3x1$ offers in familiar size pizzas with five ingredients. Additionally, thanks to *Foster's Hollywood* for their *Bacon Cheese Fries*, the *Director's Choice* burger and the *Banana Split* dessert. I couldn't have completed this work without the necessary nutrients.

Last but not least, I wish to thank my parents and brother. They bore me, raised me, supported me, taught me, and loved me. To them I dedicate this thesis.

Abstract

In this project, we propose the implementation of a 3D object recognition system which will be optimized to operate under demanding time constraints. The system must be robust so that objects can be recognized properly in poor light conditions and cluttered scenes with significant levels of occlusion.

An important requirement must be met: the system must exhibit a reasonable performance running on a low power consumption mobile GPU computing platform (NVIDIA Jetson TK1) so that it can be integrated in mobile robotics systems, ambient intelligence or ambient assisted living applications.

The acquisition system is based on the use of color and depth (RGB-D) data streams provided by low-cost 3D sensors like *Microsoft Kinect* or *PrimeSense Carmine*.

The range of algorithms and applications to be implemented and integrated will be quite broad, ranging from the acquisition, outlier removal or filtering of the input data and the segmentation or characterization of regions of interest in the scene to the very object recognition and pose estimation.

Furthermore, in order to validate the proposed system, we will create a 3D object dataset. It will be composed by a set of 3D models, reconstructed from common household objects, as well as a handful of test scenes in which those objects appear. The scenes will be characterized by different levels of occlusion, diverse distances from the elements to the sensor and variations on the pose of the target objects. The creation of this dataset implies the additional development of 3D data acquisition and 3D object reconstruction applications.

The resulting system has many possible applications, ranging from mobile robot navigation and semantic scene labeling to human-computer interaction (HCI) systems based on visual information.

Resumen

En este proyecto proponemos el diseño y la implementación de un sistema de reconocimiento de objetos 3D optimizado para funcionar bajo restricciones temporales exigentes. El sistema debe permitir un reconocimiento robusto de objetos en escenas con iluminación deficiente y oclusión entre los elementos que las conforman.

Se plantea como requisito que el sistema funcione con un rendimiento adecuado sobre una plataforma de computación GPU móvil de bajo consumo (NVIDIA Jetson TK1) para permitir su utilización en aplicaciones de robótica móvil, de ambientes inteligentes o de vida asistida por el entorno.

El sistema de adquisición de datos se basa en el procesamiento de flujos de información de color y profundidad (RGB-D) proporcionados por sensores de bajo coste como *Microsoft Kinect* y *PrimeSense Carmine*.

El abanico de algoritmos y aplicaciones a implementar e integrar cubrirá varios niveles, desde la adquisición y mejora o filtrado de los datos de entrada, pasando por la segmentación y caracterización de zonas de interés en la escena y terminando por la detección y el reconocimiento de los objetos así como la estimación de su pose.

Adicionalmente, para poder validar el sistema, crearemos un dataset de objetos compuesto por un conjunto de modelos 3D reconstruidos a partir de objetos reales así como una serie de escenas en las que aparecen dichos objetos con diversos niveles de oclusión, diferentes distancias al sensor y poses distintas. La creación de este dataset implica el desarrollo de herramientas adicionales de captura de datos procedentes de los sensores previamente mencionados y de reconstrucción de objetos.

El sistema resultante puede ser de aplicación a múltiples sistemas, ya sea de navegación robótica, etiquetado semántico de escenas e incluso para sistemas de interacción hombre-máquina basados en información visual.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Chapter 1

Introduction

This first chapter introduces the main topic of this work. It is organized in six different sections: Section 1.1 sets up the framework for the activities performed during this project, Section 1.2 introduces the motivation of this work, Section 1.3 elaborates a state of the art of existing 3D object recognition systems, Section 1.4 lays down the proposal developed in this work, Section 1.5 presents the main and specific goals of this project. Finally, Section 1.6 details the structure of this document.

1.1 Outline

In this bachelor thesis we have performed a theoretical and practical research focused on the problem of 3D object recognition in cluttered and occluded environments under demanding time constraints. The project comprises the study of 3D object recognition systems and 3D feature descriptors; all of it combined with 3D data acquisition by means of low-cost sensors, like the *Microsoft Kinect 2* device, and the data processing capabilities of low power consumption parallel computing platforms such as the *NVIDIA Jetson TK1*.

The main goal is the proposal, design and development of an efficient and accurate solution for the 3D object recognition in cluttered scenes problem with the addition of time constraints to make the system able to run in real time so that it can be integrated on a low-cost mobile robotic platform.

This work has been performed under the frame of the following national project: *SIRMAVED: Development of a comprehensive robotic system for monitoring and interaction for people with acquired brain damage and dependent people*, ID

code (DPI2013-40534-R), funded by the *Ministerio de Economía y Competitividad (MEC)* of Spain with professors José García-Rodríguez and Miguel Ángel Cazorla-Quevedo from the University of Alicante as main researchers.

Moreover, part of this work was done during an eight month research collaboration programme in the *Department of Computer Technology (DTIC)* at the University of Alicante. This collaboration was funded by the *Ministerio de Educación, Cultura y Deporte (MECD)* of Spain by means of a research grant for the project: *Implementation of 3D vision algorithms under time constraints: human-computer interaction and object recognition applications*. The work carried out was led and supervised by professor José García-Rodríguez and postdoctoral researcher Sergio Orts-Escolano from the *Industrial Informatics and Computer Networks* research group.

1.2 Motivation

This document presents the results of the work carried out to prove the knowledge acquired during the *Bachelor's Degree in Computer Engineering*, taken between the years 2011-2015 at the *University of Alicante*. The motivation of this work arises from the collaboration with the *Department of Computer Technology* in research tasks related to computer vision, high performance computing and the *SIRMAVED* project.

The main goal of the previously mentioned *SIRMAVED* project is the development of a robotic system for monitoring dependent people or persons with acquired brain damage and interacting with them to perform common daily tasks like recognition and manipulation of small sized objects. In order to do that, the integration of multiple technologies is required; particularly, an object recognition system is needed to identify instances in the environment before handling or grasping them. In that sense, it is necessary to design and develop an onboard 3D object recognition system which must be able to operate on cluttered and occluded environments. The system must meet low power, small size and real-time performance requirements.

1.3 Related works

In the context of computer vision, object recognition is the process of detecting and identifying objects in images or video sequences as well as determining

their pose, that is, their positions and orientations. This task is still one of the hardest challenges of computer vision systems so that multiple approaches have been taken and implemented over many years of research in the field.

Traditionally, object recognition systems made use of bidimensional images with intensity information. Those systems apply machine learning and matching algorithms which are based either on significant features of the objects or their appearance. However, technological advances made during last years have caused a huge increase in the use of tridimensional information in the field of computer vision in general and, in particular, in object recognition systems; this is due to the capability of acquiring real-time 3D data and efficiently process all this information.

Nowadays, the use of 3D information for this task is in a state of continuous evolution, still far behind of the maturity achieved by the systems which make use of 2D intensity images. Nevertheless, despite being a relatively mature area thanks to the extensive research performed during the last decades [1], the use of 2D images exhibits a number of problems which hinder the progress in lines of research that follow this approach. Oppositely, the use of range images or point clouds, which offer 2.5D/3D information, presents many significant benefits over traditional 2D ones. Some of the main advantages are the following ones [2]: (1) they provide more geometrical information thus removing surface ambiguities, (2) many of the features that can be extracted are not affected by illumination or even scale changes, (3) pose estimation is more accurate due to the increased amount of geometrical information. Therefore, the use of 3D images has become a solid choice to overcome the inherent hurdles of the traditional 2D methods.

In addition, recent advances in low-cost range sensors such as the *Microsoft Kinect* device [3] have enabled a widespread adoption of this technology and have increased the usage of 3D information. Accessibility and affordability are two of the key factors which have contributed to the development of this research area.

Together with the technological development of tridimensional information acquisition systems, another key factor in the evolution of the 3D object recognition field is the creation and development of computing devices which are able to process, in an efficient manner, the huge amount of data representing the tridimensional information provided by range sensors. In this sense, the continuous improvements introduced in the fields of *General Purpose computation on GPUs (GPGPU)* and low power consumption parallel computing

devices like the *NVIDIA Jetson TK1* platform [4] have made available to the researchers the necessary computational resources for the execution of 3D object recognition algorithms under time constraints.

The combination of these three factors (the advantages of 3D images, low-cost sensors and parallel computing devices) has transformed this area into one of the most active at the moment; therefore, creating a robust 3D object recognition system, which is also able to work in real time, has become one of the main goals towards which many efforts of computer vision researchers are directed [5].

Regardless of the kind of input data, object recognition methods are divided into four broad categories depending on the approach taken to describe the object: (1) model-based, (2) shape-based, (3) appearance-based and (4) feature-based methods. Model-based methods try to approximately represent the object as a collection of geometric primitives. Shape-based methods represent the object by using its contour or other shape cues. The main idea behind appearance-based methods is the representation of individual object views as points in one or more multidimensional spaces whose basis are obtained through the statistical analysis of the image training set; unknown views are then projected to those spaces and then they are classified according to the nearest view among the ones projected by the training set. Appearance-based methods are attractive due to their robustness against shape or pose changes; in addition, they do not need to compute image features nor geometric primitives in order to perform the matching. However, their main drawbacks include the necessity of dense sampling of training views and the lack of robustness against occlusion [6]. On the other hand, feature-based methods are frequently used due to their robustness when dealing with cluttered scenes and partial occlusions. Those methods focus on extracting features, i.e., pieces of information which describe simple but significant properties of the object; after the feature extraction phase, machine learning techniques are applied to train a system with those known features so that it becomes able to classify features extracted from unknown objects [7]. Nowadays, appearance-based and feature-based methods are the most popular choices. Since our proposal is mainly focused on object recognition in cluttered environments with occlusions, we will proceed with an in-depth review of feature-based approaches.

Feature-based 3D object recognition methods can be classified into two broad categories according to the way they process the object itself: global feature based methods and local feature based ones. On the one hand, global

methods are characterized by dealing with the object as a whole. They define a set of global features which effectively describe the whole object. Those methods are widely used in the context of tridimensional shape classification [8], but their global approach renders them unsuitable for recognizing partially visible objects in cluttered and occluded scenes. On the other hand, local feature based methods describe local surface patches of the object which are located around highly distinctive points named *keypoints*. In this sense, local feature based methods are able to cope with cluttered environments and occlusions [9] since they do not need the whole object to describe and use its information. Since this work is framed under the area of 3D object recognition in cluttered environments with temporal constraints, we will focus this review on local feature based 3D object recognition methods.

At the present time, local feature based 3D object recognition is one of the main research areas in the computer vision field. In addition, the inclusion of temporal constraints to this recognition process has many practical applications in multiple areas like robotics, biometric analysis, surveillance, auxiliary medical systems, mobile systems manipulation, automatic assembly and many others [10, 11, 12, 13, 14, 15, 16, 17]. In the latter years, computer vision in general and object recognition in particular are increasingly becoming more mature. As the techniques have become more powerful, the industry has been able to create a wide variety of computer vision products and services [18].

There exist many reviews about 3D object recognition in the literature, including the seminal works of Besl and Rain [19], Brady *et al.* [20], Arman *et al.* [21], Campbell and Flynn [6], and Mamic and Bennamoun [22]. All of them perform a general review of the 3D object recognition problem with varying levels of detail. However, none of them addresses recent progresses on the problem nor are significantly focused on the problem that concerns us: local surface feature based 3D object recognition methods.

The work of Guo *et al.* [2] is characterized by its comprehensive analysis of different local surface feature based 3D object recognition methods which were published between the years 1992 and 2013. In that review, they explain the main advantages and drawbacks of each one of them. They also provide an in-depth survey of the techniques used in each phase of a 3D object recognition pipeline, from the keypoint extraction stage to the surface matching phase, including the extraction of local surface descriptors. The freshness and the level of detail of the work make it the most remarkable review (the only one, according to the authors) focused on local surface feature based methods.

It is important to remark the part of the review devoted by Guo *et al.* in their work to enumerate, taking into account the most recent works, a set of possible future research directions about local surface feature based 3D object recognition, including the following ones: (1) the creation of a benchmarking method for existing methods, (2) non-rigid object recognition, (3) fusion of photometric and geometric information (RGB-D), (4) object recognition from 3D video sequences and (5) the creation or adaptation of current methods to make them able to work with low resolution images provided by low-cost sensors like *Microsoft Kinect*.

Our work follows some of the research directions suggested by Guo *et al.* [2]. On the one hand, we will try to fuse photometric and geometric information into a 3D descriptor to obtain a robust object recognition system on multiple scenarios in which the use of exclusively one kind of information would fail. On the other hand, we will use depth and color data streams provided by low-cost sensors. Our main goal is to achieve a system's implementation that deals with demanding time constraints, i.e., real-time recognition tasks.

Given the posed problems, we will perform a brief review of the state of the art of each of them, in order to properly contextualize our proposal. In this sense, this general review about the state of the art of object recognition will be further divided into the following sections: Section 1.3.1 describes the local surface feature based methods, including their main stages and the core techniques used in them. Section 1.3.2 reviews a set of 3D descriptors which are currently used for object recognition applications. Section 1.3.3 explores the possibilities of using low-cost 3D sensors in object recognition systems to achieve real-time implementations. Section 1.3.4 reviews the current state of the systems which make use of feature descriptors that append photometric information to the traditional geometric one in order to improve the recognition rate.

1.3.1 3D object recognition pipeline

As we previously stated, local surface feature based object recognition methods codify, by means of a descriptor, the information provided by high descriptive regions of the object. These descriptors are later compared to elaborate hypotheses which are assessed to extract conclusions about unknown objects. The main advantage of these kind of methods is their ability to recognize objects in cluttered scenes with a certain level of occlusion.

Traditionally, local surface feature based methods are structured in three main stages [2]: *keypoint detection*, *local surface feature description* and *surface matching*. During the *keypoint detection* phase, a highly descriptive 3D point set is selected so that those points which do not provide enough information to discriminate the surface are discarded. The *local surface feature description* stage consists of the creation of descriptors, i.e., codifications of the characteristic information of the surface, around the selected keypoints. At last, the *surface matching* process takes place. The descriptors extracted from a particular scene are compared with those obtained from isolated objects to determine the possible matchings. Later, those matchings are used to generate a set of hypotheses which are grouped, refined and verified to classify the descriptors thus recognizing the objects represented by scene surfaces.

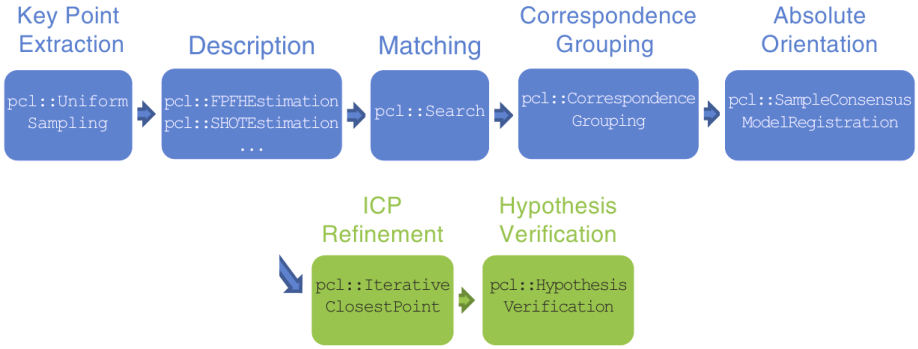


Figure 1.1: Local 3D recognition pipeline (Figure reproduced from [23]).

After this brief introduction to local surface feature based methods, we will provide a detailed description of each of the main three phases.

1.3.1.1 Keypoint detection

The first stage of a local feature based recognition system is typically devoted to the selection of a set of characteristic points of the surface around which the surface information will be encoded into descriptors. This selection guarantees the extraction of local and significant object surface information. In addition, the keypoint detection has two main goals: on the one hand, reducing the computational complexity of encoding the information of the whole surface instead of local patches; on the other hand, discard ambiguous regions which do not provide enough information.

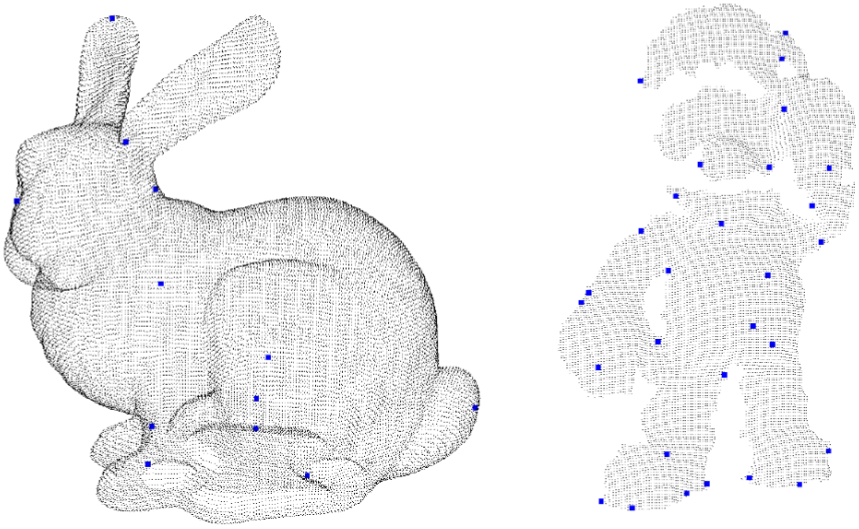


Figure 1.2: Keypoint detection using curvature SIFT. Detected keypoints are marked using blue dots. (Left) Complete Stanford bunny model. (Right) 3D partial view of a Mario Bros model. (Figure reproduced from *A three-dimensional representation method for noisy point clouds based on growing self-organizing maps accelerated on GPUs* [24]).

The simplest way to perform that selection is applying some kind of mesh decimation or surface sparse sampling techniques. However, those methods do not take into account the information richness of those keypoints so the selected points may end up having a low distinctiveness and repeatability. In this sense, keypoint detection methods must enforce three constraints on the detected keypoints in order to guarantee the uniqueness of the local descriptors extracted from the keypoints's surrounding surfaces [25]: (1) repeatability between different views of the same object, (2) a unique 3D coordinate basis defined from the neighborhood surface, and (3) enough distinctive information in that neighborhood surface.

Depending on how keypoint detection methods deal with the scale, we can classify them into two different categories [26]: fixed-scale and adaptive-scale. Fixed-scale methods choose points which are highly distinctive within a neighborhood whose size is fixed as an input parameter to the method. Those methods are divided into two categories depending on which measures are used to determine the distinctiviness of a keypoint [2]: curvature based methods

(CBM) and other surface variation measure methods (OSV). Adaptive-scale methods create a scale space for each 3D image and then select those points which are distinctive within a neighborhood, whose size is adaptively determined, in both spatial and scale spaces. Those methods are classified into four different categories depending on the way they build the scale space [2]: transform based methods (TB), coordinate smoothing based (CS), surface variation based (SV), and geometric attribute smoothing based ones (GAS). For an in-depth description of the techniques, the reader is encouraged to consult the review by Guo *et al.* [2].

Adaptive-scale methods are the most popular ones due to their ability to detect the scale of the keypoint. This information improves the precision of the whole system when dealing with scale changes in the scenes. However, they tend to be temporally and spatially complex. Fixed-scale methods are simpler and less complex from a computational point of view but they do not take scale into account and the best neighborhood size has to be empirically determined.

In conclusion, this stage increments the efficiency and sometimes the precision of the following phases of the object recognition system.

1.3.1.2 Local surface feature description

The second phase of a local feature based recognition system consists of describing the features of the local surface around the detected keypoints. As we said before, keypoints are stable locations of the shape whose neighborhood surface features are highly distinctive. A feature is a persistent element of a surface which captures its relevant information [27]. In this stage, the features of each local surface patch are encoded into a descriptor. These shape representations are used to train the object recognition during the learning phase or to recognize during the surface matching stage, i.e, generate a set of possible hypotheses about unknown objects.

Feature descriptors must be distinctive and robust so pose invariance is required. In order to make that happen, the most common solution consists of defining the descriptor with respect to a local frame instead of a global one. The vast majority of descriptors define first a local reference frame at a keypoint and then express the local neighborhood surface features with respect to it. A local reference frame *(LRF)* is a basis of three orthogonal unit vectors defined upon a local support [28]. They provide a way to achieve invariance to rigid transformations (rotations and translations) and increase the robustness

against noise and clutter.

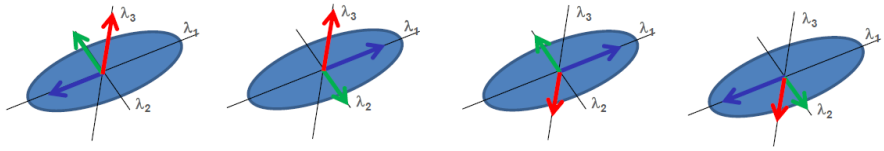


Figure 1.3: Different Local Reference Frames obtained by enforcing the right hand rule. The orthogonal unit vectors are shown in blue, green and red (denoted by λ_1, λ_2 and λ_3 respectively). The keypoint is located at the origin, the blue area is the local surface neighborhood. (Figure reproduced from [28]).

Sometimes, a single LRF might provide an ambiguous definition; in those cases, feature descriptors resort to defining multiple LRFs at each keypoint to provide various descriptions at the same keypoint thus removing the possible ambiguities. Figure 1.3 shows an example of multiple LRFs at the same keypoint. However, defining more LRFs increases the complexity of the system since it has to compute more descriptors and the matching phase has to solve the ambiguity problem [28].

In the same way as the keypoints, the quality of a local reference frame depends on its repeatability and robustness to noise. Using a repeatable and robust LRF raises the distinctiveness of the descriptor dramatically increasing the performance of the surface matching phase [29]. In that sense, the quality of the LRF has a significant impact on the effectiveness of the whole object recognition system [30].

Once the LRF is computed, another key parameter of a surface descriptor is its support size, i.e., the size of the region around the keypoint whose features will be encoded. Examples of different supports are shown on Figure 1.4. Depending on the support size, descriptors can be classified into three categories: pointwise, local/regional and global descriptors. The first ones are simple and straightforward but they are not able to deal with noise or occlusions since they do not provide enough information. Global descriptors are highly descriptive and they are also extremely invariant and robust to noise due to the amount of considered information, but they need the complete surface to be computed so they are not suitable for recognition with clutter or occlusions. Local descriptors do not need the whole surface to be computed so they are able to handle occlusions and clutter. Their locality depends on the size of the support so they can be adapted to increase their distinctiveness and robustness to noise while

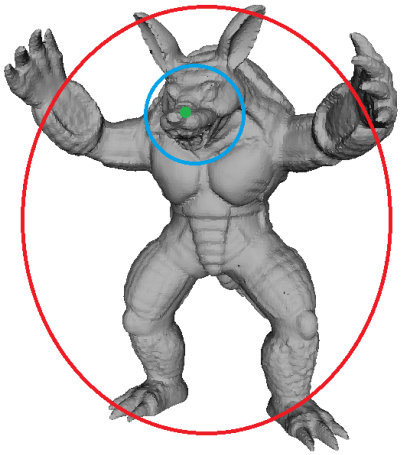


Figure 1.4: Different support sizes for a descriptor over the popular Armadillo model from the *Stanford 3D scanning repository* [31] : global descriptor (red), local descriptor (blue) and pointwise descriptor (green).

keeping their clutter and occlusion handling capability. Local surface feature based methods use local descriptors.

Choosing the right size for the support is critical for local feature descriptors. As we said before, their strength lies on their adaptability. Increasing the neighborhood size increments the amount of information that can be encoded making it more distinctive. However, it also increases the sensitivity to noise and occlusions.

Once the LRFs and the support size are defined, the last step for creating the descriptor consists of enconding the information, with respect to the LRF, of the local surface delimited by the support around the keypoint. According to the information encoding method, descriptors can be classified into three categories [2]: signature based, histogram based, and transform based descriptors. Signature based methods encode geometric measures for each point of a subset of the neighborhood. Histogram based methods accumulate geometric measures into histograms according to a specific domain. Transform based methods transform the 3D image to another domain and encode the local surface information in it.

1.3.1.3 Surface matching

The last stage of a typical local feature based object recognition system is named *surface matching*. In this stage, the feature descriptors extracted from the scene

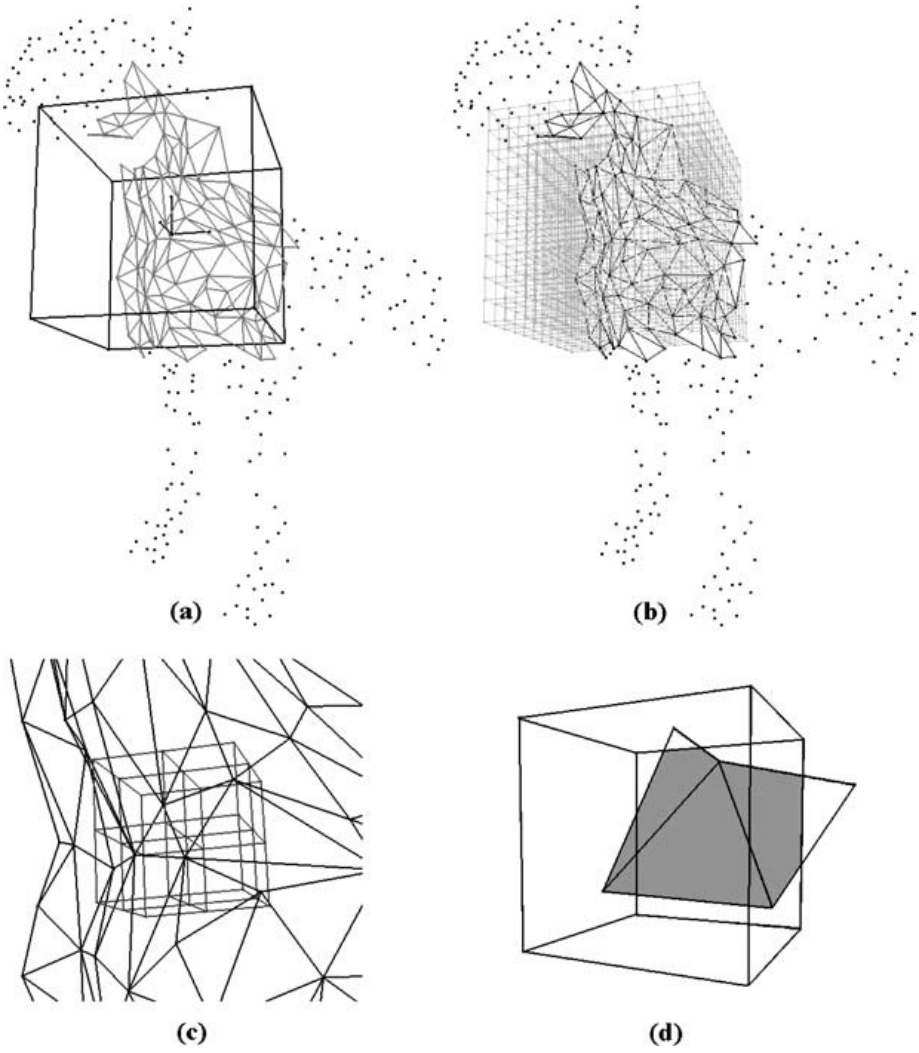


Figure 1.5: Example of an histogram based 3D descriptor: the Mian tensor which divides the support into bins and accumulates the area of surface which intersects each one of them. (a) 3D basis (LRF) and support defined over the surface of the dog. (b) A 10x10x10 support grid defined over the surface of the dog centered at the LRF origin. (c) A zoomed in view of eight bins around the LRF origin. (d) A single bin intersecting three triangular faces. The shaded area is the intersection. (Figure reproduced from [16]).

are matched against the descriptors of the trained system to generate a set of hypotheses by voting for the possible candidates. At last, those hypotheses are verified to identify the different objects in the scene. In this sense, this stage can

be subdivided into three phases [2]: feature matching, hypothesis generation and verification.

The goal of the feature matching phase is to obtain a set of correspondences between the extracted scene descriptors and the ones from the models used to train the system. In order to do that, a matching strategy and a search method must be defined. The most used strategy is the nearest neighbor (NN) search. The simplest search method is the brute-force one; however, comparing each scene descriptor with all the stored ones is extremely inefficient. Efficient data structures or index based methods are commonly used to alleviate the complexity of the search. Hash tables and k-d trees are the most popular alternatives.

Once a set of correspondences between the scene descriptors and the stored ones has been established, the system must determine the possible candidate object models that might be present in the scene due to those correspondences and then generate transformation hypotheses for each one of them. However, those transformations tend to be inaccurate because of false matches. The most common techniques that are used to generate transformation hypotheses and improve their accuracy are *geometric consistency*, *random sample consensus* and *pose clustering*.

In the end, the generated hypotheses are accepted or rejected during the verification phase. The goal is to discard false hypotheses, i.e., transformations whose accuracy fall below a predefined threshold. The most commonly used methods are the so called individual verification ones. They align the candidate model with the scene using the transformation hypothesis, then the alignment is refined with registration methods like the Iterative Closest Point algorithm. The hypothesis is accepted if its error is below certain threshold. Then the scene points which correspond to the aligned model are segmented thus recognizing the object.

1.3.2 3D local feature descriptors

Local surface feature descriptors are a key part of an accurate and robust object recognition system. As we previously stated, using local surface features is the current trend due to their robustness to clutter and occlusions. These descriptors encode the characteristics of the local neighborhood of feature points. Since they do not need the whole object for describing it, they are therefore suitable for partially visible object recognition under clutter conditions.

Arguably, the most important characteristic of a local feature descriptor is pose invariance. This is due to the fact that recognition systems must be able to detect objects which can be scanned from different viewpoints or appear in a scene with any arbitrary pose. A common way to achieve pose invariance is to resort to a reference axis or define a local reference frame. Both solutions are object-centered coordinate systems. In contrast with viewer-centered coordinate systems, object-centered ones are view-independent thus making the surface descriptions invariant to pose. Besides that, each descriptor delimits and describes the local surface using various techniques so that the performance of the system may vary.

Throughout this section we will describe the following 3D local feature descriptors: SI, SHOT, FPFH, 3DSC, USC, Tensor, RoPS and TriSI. These descriptors were chosen following popularity, novelty and performance criteria.

1.3.2.1 Spin Images (SI)

Spin Images (SIs) is a descriptor initially introduced by Johnson [32] for surface matching and object recognition systems. In order to achieve pose invariance, Spin Images resort to defining local coordinate systems at oriented points (3D surface points with an associated direction). Oriented points are defined by a surface vertex p and its surface normal n thus formulating the local coordinate system (p, n). Then the plane P through p is defined as perpendicular to n and the line L through p as parallel to n. This creates a cylindrical coordinate system (α, β) in which α is the perpendicular distance of a point x to L while β is the signed perpendicular distance of x to P. Figure 1.6 shows the creation of the coordinate system.

The next step of the descriptor consists of creating a spin map S_0 by projecting the 3D points of the mesh to the oriented point basis. Given a point x the projection function $S_0(x)$ gives us the coordinates (α, β) in the basis (p, n). Once the points have been projected, a 2D array representing the spin image is generated using the oriented point basis. This array is divided into bins so that a histogram-like representation is generated. Then, the corresponding bin for each projected point $S_O(x)$ is calculated taking into account its (α, β) coordinates. If the point x meets some established constraints, i.e., distance of x to O or angle between O and the surface normal of x (namely, the support angle), the array is updated by incrementing the four surrounding bins of the bin which corresponds to x. The contribution of x to the four surrounding bins is

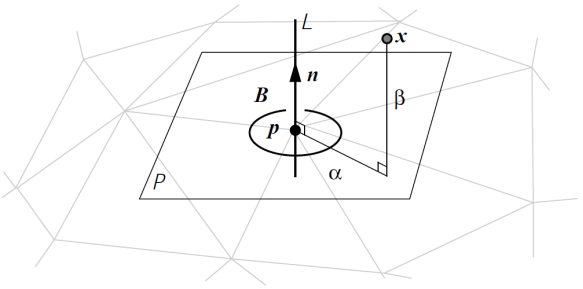


Figure 1.6: An oriented point basis created at a vertex in a surface mesh. The position of the oriented point is the 3-D position of the vertex, and the direction of the oriented point is the surface normal at the vertex. (Figure reproduced from [32]).

determined by using bilinear interpolation so that the array is less sensitive to the position of the point thus dealing with noisy data. Figure 1.7 illustrates the 2D array generation process and 1.8 shows three 2D arrays and spin images generated using three points on the surface of a duckling model.

Regarding to the spin image generation, there are three parameters which control the process: support angle, image width and bin size. The bin size controls the descriptiveness of the spin image while the width determines the amount of information taken into account. The support angle controls the effect of self-occlusion and clutter. It is important to remark that spin images are not scale invariant [32].

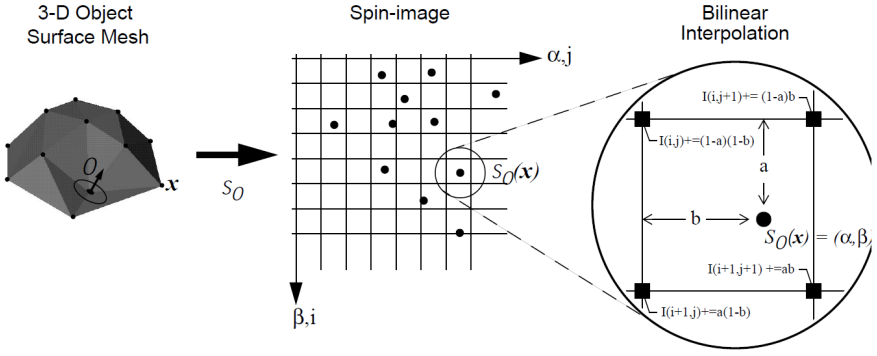


Figure 1.7: The addition of a point to the 2-D array representation of a spin-image. From left to right: projection, 2D array generation and contribution calculation with bilinear interpolation. (Figure reproduced from [32]).

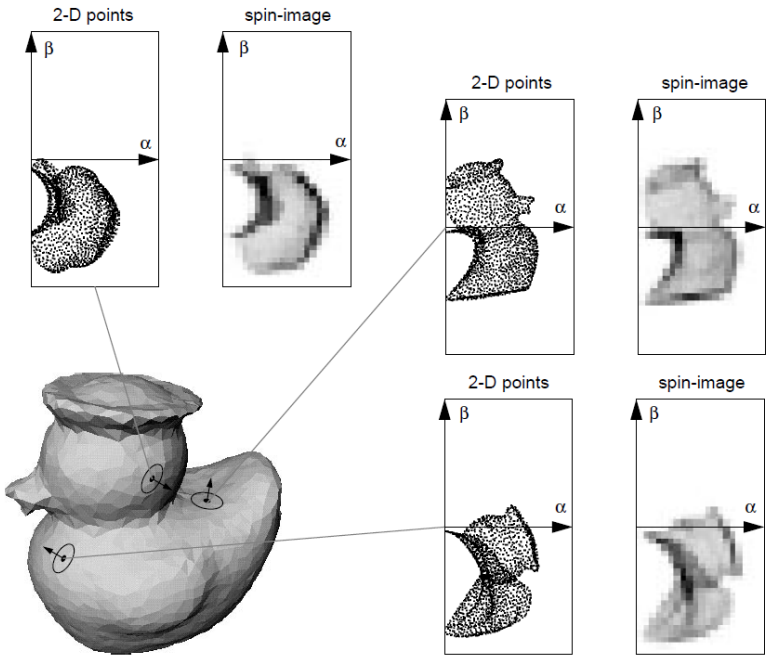


Figure 1.8: Spin images and 2D arrays for three oriented points on the surface of a duckling model. Darker areas correspond to bins with higher number of points projected into them. (Figure reproduced from [32]).

1.3.2.2 Fast Point Feature Histogram (FPFH)

The Fast Point Feature Histograms (FPFH) [33] descriptor is an efficient variant of the original Point Feature Histograms (PFH) [34] both introduced by Rusu *et al.*. The PFH descriptor tries to encode the geometrical properties of the neighborhood of a keypoint generalizing the mean curvature around that point and storing the information on a multi-dimensional histogram. This generates a pose invariant descriptor which is also able to deal with different resolutions and levels of noise.

The first step for computing the descriptor of a determined point consists of delimiting its neighborhood. In order to do that, a 3D sphere with radius r is placed with its center at the keypoint. All its neighbors whose distance is smaller than the radius r are taken into account for the next step. Figure 1.9 shows this process.

Then, for each pair of points p_s and p_t ($s \neq t$ and p_s being the point with a smaller angle between its normal and the line connecting both points) in the

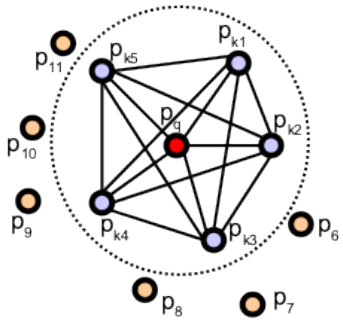


Figure 1.9: Neighborhood for keypoint p_q in red, delimited by a 3D sphere represented by a dotted circle. (Figure reproduced from *Pointclouds.org*).

neighborhood, a triplet (α, ϕ, θ) is computed. In order to do that, a fixed coordinate frame (u, v, w) at one of the points is defined to achieve pose invariance. Being $u = n_{p_s}, \quad v = u \times (p_t - p_s)$ and $w = u \times v$ where n_{p_s} is the normal associated to p_s.

Once the coordinate frame has been established, the difference between the normals of both points are expressed as a set of angular features, i.e., the triplet (α, ϕ, θ) that we mentioned earlier: $\alpha = v \cdot n_{p_t}, \quad \phi = u \cdot \frac{p_t - p_s}{d}$ and $\theta = \arctan(w \cdot n_{p_t}, u \cdot n_{p_t})$. where d is the distance from p_s to p_t and n_{p_t} the normal associated to p_t.

Having all the triplets computed for each pair of points in the neighborhood, the actual descriptor is generated by binning that set of triplets into the point feature histogram. In order to do that, the value range of each feature (α, ϕ and θ) is divided into b subintervals. This way, a histogram with b^3 bins is created. The histogram is generated by counting the number of occurrences in each subinterval. The computational complexity of the PFH descriptor for a cloud with n points is $O(nk^2)$ where k is the number of neighbors for each point.

The Fast Point Feature Histograms (FPFH) is a simplified version of this process which reduces the computational complexity to $O(nk)$ and keeps most of the representational power of the PFH descriptor. The FPFH first computes a set of triplets (α, ϕ, θ) between the keypoint and its neighbors and then generates a histogram called Simplified Point Feature Histogram (SPFH). Then that simplified histogram and the weighted ones of the keypoint's neighbors are combined to form the final histogram: $FPFH(p) = SPFH(p) + \frac{1}{k}\sum_{i=1}^{k}\frac{1}{w_k}SPFH(p_k)$

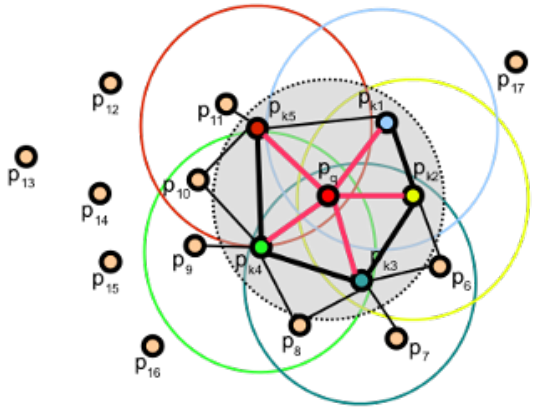


Figure 1.10: FPFH Neighbors of the current keypoint p_q in red linked by pink lines. For those points, each neighborhood is delimited by a colored circle. Bold lines represented relationships which are computed twice. (Figure reproduced from *Pointclouds.org*).

where p is the current point, p_k are the neighbors of p and w_k is a score given to the (p, p_k) pair which is usually a distance in some given metric space.

1.3.2.3 3D Shape Context (3DSC)

The 3D Shape Context descriptor [35] was originally introduced by Frome *et al.* as a three dimensional extension of 2D shape contexts introduced by Belongie *et al.* [36]. 3DSC builds a 3D sphere as the support for the descriptor and then divides it into bins to generate a histogram-like representation. That sphere is centered on the keypoint p and its north pole is oriented with the surface normal of that point. The sphere is then divided into bins by logarithmically spaced boundaries along the radial dimension and equally spaced ones in the azimuth and elevation dimensions. Figure 1.11 shows the support of the 3DSC descriptor.

Frome *et al.* perform $J+1$ radial divisions $R = \{R_0, ..., R_J\}$, $K+1$ elevation divisions $\theta = \{\theta_0, ..., \theta_K\}$ and $L+1$ azimuthal ones $\phi = \{\phi_0, ..., \phi_L\}$. In this way, each bin corresponds to one element in the $J \times K \times L$ feature vector (R, θ, ϕ). Each bin (j, k, l) accumulates the weighted count $w(p_i)$ for each point p_i whose spherical coordinates relative to the keypoint or sphere center p fall within the corresponding ranges $[R_j, R_{j+1})$, $[\theta_k, \theta_{k+1})$ and $[\phi_l, \phi_{l+1})$. Each point p_i contributes with a certain amount to the bin count following Equation 1.1. In this sense, the contribution of a certain point depends on the volume of the

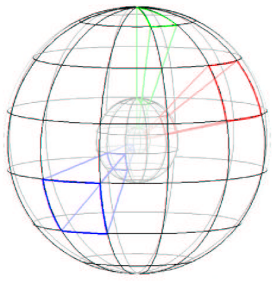


Figure 1.11: 3D Shape Context support sphere. (Figure reproduced from [35]).

bin $V(j,k,l)$ and the local point density around that bin p_i, which is estimated as the count of points in a sphere of radius δ around p_i.

$$w(p_i) = \frac{1}{p_i \sqrt[3]{V(j,k,l)}} \tag{1.1}$$

1.3.2.4 Unique Shape Context (USC)

The Unique Shape Context (USC) is a modified version of the 3DSC descriptor which was introduced by Tombari *et al.* [37] in order to improve its accuracy and reduce its memory footprint. Tombari *et al.* identified that the lack of uniqueness of the local reference frame reduces the accuracy of the descriptor and increases its memory footprint since the computation of multiple descriptors at each feature point is required in order to achieve a unique description at each point.

The USC is an improved 3DSC which does not need to compute the descriptor over multiple rotations on azimuth directions. This improvement consists of employing an unique and unambiguous local reference frame which yields a repeatable normal axis and two unique directions lying on the tangent plane. In this sense, only a single descriptor has to be computed over the rotation indicated by the vectors over the tangent plane thus reducing the memory footprint of the descriptor and increasing the overall accuracy of the recognition process.

The unambiguous local reference frame is computed as follows. Given a keypoint p and a spherical support centered on it with a determined radius R (note that this support is the same that is later employed to compute the descriptor), the covariance matrix M of the points of the neighborhood is com-

puted as shown on Equation 1.2 where $d_i = ||p_i - p||_2$ and $Z = \sum_{i:d_i \leq R} (R - d_i)$. Note that smaller weights are assigned to distant points in the covariance matrix since those are likely lying out of the region of interest.

$$M = 1/Z \sum_{i:d_i \leq R} (R - d_i)(p_i - p)(p_i - p)^T \tag{1.2}$$

Then, the Total Least Squares estimation of the three unit vectors of the local reference frame is obtained by computing the eigenvector decomposition of the covariance matrix. Also, a sign disambiguation step is performed to overcome the eigenvector decomposition ambiguity so that a unique and repeatable local reference frame is generated. Once the LRF is obtained, the descriptor is computed using that reference frame in the same way as 3DSC does.

1.3.2.5 3D Tensor (Tensor)

The Tensor is a 3D local feature descriptor introduced by Mian *et al.* [16] which relies on building a three dimensional histogram that accumulates the amount of surface intersecting each bin instead of just holding a point count. This means that the descriptor computation requires a triangular mesh which is usually decimated for performance reasons. First of all, the vertices of the mesh are paired such that they satisfy certain distance and angle constraints: two vertices are paired only if their distance falls in a predefined interval $[d_{min}, d_{max}]$ and if the angle between their normals θ is greater than 5 degrees. In order to avoid the combinatorial explosion of pairs, each vertex is only paired with the nearest three ones which satisfy the previous constraints.

Then, for each pair of vertices a coordinate basis is defined. The origin is established at the center of the line which joins both vertices. The average of the normals of the vertices makes the z-axis, the cross product of them makes the x-axis and again the cross product of the z-axis and the x-axis makes the y-axis. Once the coordinate based is established, a thee-dimensional grid is defined at its origin with a certain number of bins (usually 10x10x10) and a predefined bin size (normally derived as $d_{min}/5$). Once the grid has been defined, the Hodgman's polygon clipping algorithm is used to find the surface area of intersection of each bin and the mesh. Each bin holds the amount of surface area of the mesh which intersects that bin. Since most bins tend to be empty, the tensor is compressed to a sparse form.

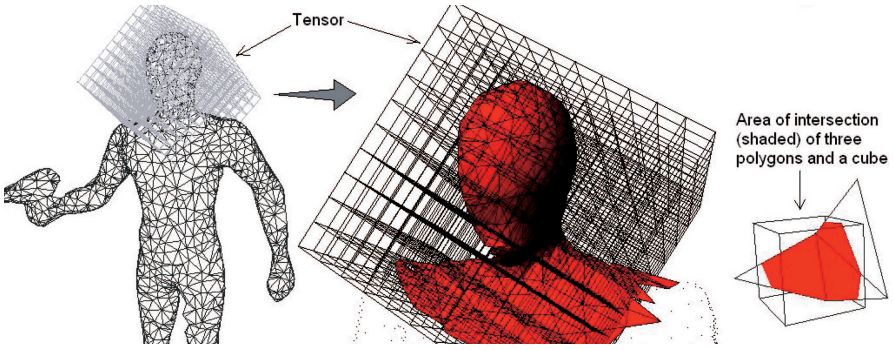


Figure 1.12: Tensor computation. (Figure reproduced from [15]).

1.3.2.6 Signature of Histograms of Orientations (SHOT)

SHOT is a descriptor originally introduced by Tombari *et al.* [30] which encodes a signature of histograms representing topological traits of points within a spherical support. The signature structure of the SHOT descriptor is an isotropic spherical grid which encompasses partitions along the radial, azimuth and elevation axes as shown on Figure 1.13. The usual number of spatial bins is 32 (8 azimuth divisions, 2 elevation and radial divisions) generating a small cardinality descriptor. For each volume or spatial bins, a local histogram is computed. Then all local histograms are grouped together to form the actual descriptor which lays at the intersection between a histogram-based method and a signature-based one.

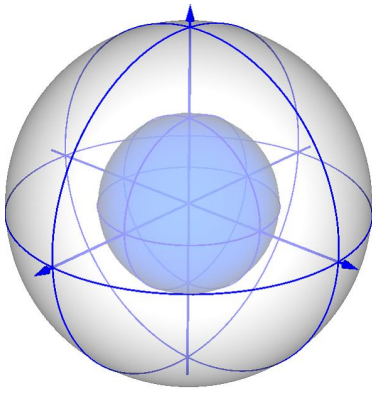


Figure 1.13: SHOT signature structure. (Figure reproduced from [30]).

For each one-dimensional local histogram, point counts are accumulated into bins according to a function of the angle θ_i between the normal at each

point within the volume n_i and the normal at the keypoint n or center of the support. The function used to accumulate point counts is $\cos\theta_i$ which can be computed fast as $n_i \cdot n$.

It is important to remark that before creating the volumes and computing the descriptor, a mechanism like the one used for the USC descriptor is employed to generate a repeatable and unambiguous local reference frame: eigenvector decomposition of the covariance matrix of the point coordinates within the support.

1.3.2.7 Rotational Projection Statistics (RoPS)

Rotational Projection Statistics is a local feature descriptor proposed by Guo *et al.* in their paper [38] with the main goal of developing a method robust to noise and variations in mesh resolutions. RoPS first crops a local surface of the mesh using a sphere of radius r centered at a keypoint p which can be detected with any available method. Next, a LRF is generated by calculating the eigenvectors of the covariance matrix of the local surface. In order to build the covariance matrix, RoPS needs a surface mesh S which contains N triangles and M vertices.

Once the LRF has been computed, the neighboring points within the support radius are relocated in the LRF to achieve pose invariance. This results in a transformed point cloud Q'. Then, a feature descriptor is generated by encoding the information of that local surface from different viewpoints.

In order to do that, the neighboring points are rotated around the LRF x-axis by an angle θ_k and then they are projected onto the local xy, yz and xz planes using the LRF axes. Each plane is partitioned into $L \times L$ bins which count up the number of points which fall in them. This results in a distribution matrix D which partially records the local surface information from a particular viewpoint. To make the feature descriptor compact, four central moments and Shannon entropy are calculated for each D. These values are concatenated to form a sub-feature $f_x(\theta_k)$.

However, performing this step only on one axis and one rotation yields a low descriptive descriptor. In order to overcome this problem, the information of the local surface is recorded from various viewpoints. The neighboring points are rotated around the local x-axis by a series of angles $\theta_k, k = 1, 2, ..., T$ to generate a set of sub-features for the x-axis $\{f_x(\theta_k)\}, k = 1, 2, ..., T$. The process is repeated rotating Q' around the y and z axes to generate the set of

sub-features $\{f_y(\theta_k)\}$ and $\{f_z(\theta_k)\}, k = 1, 2, ..., T$.

In the end, the overall feature descriptor is generated by concatenating all these sub-features into a vector f.

$$f = \{f_x(\theta_k), f_y(\theta_k), f_z(\theta_k)\}, \quad k = 1, 2, ..., T \tag{1.3}$$

1.3.2.8 Tri-Spin-Image (TriSI)

The TriSI feature descriptor is an improvement of the RoPS descriptor [38] also introduced by Guo *et al.*[39]. It consists of an improved LRF and a new 3D local surface feature which is very robust to noise, varying mesh resolutions, occlusions and clutter. This is achieved by encoding the information of a local surface around three orthogonal axes using one spin image for each one of them.

Guo *et al.* [39] first build a LRF which is an enhanced version of the one presented in [38] but considering a different weighting strategy to increase the robustness against noise and outliers. In the end, the LRF is constituted by the keypoint O_k and three unambiguous vectors $\{\vec{v}_1, \vec{v}_2, \vec{v}_3\}$.

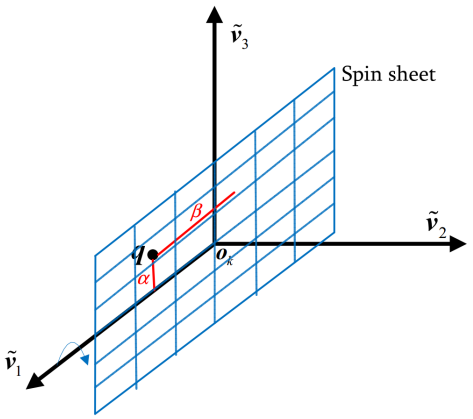


Figure 1.14: TriSI spin sheet. (Figure reproduced from [39]).

Once the LRF has been defined, the information of the local surface is encoded by three signatures $\{SI_1, SI_2, SI_3\}$ around the three axes $\{\vec{v}_1, \vec{v}_2, \vec{v}_3\}$. A signature SI_1 is generated by encoding the point distribution around $\vec{v}_1$ using a spin image approach. A point q of the local surface is mapped to a 2D space represented by two parameters (α, β). α represents the perpendicular distance of q from the line that passes through O_k and is parallel to $\vec{v}_1$, while β is the

signed perpendicular distance to the plane that goes through O_k and is perpendicular to $\vec{v_1}$. Figure 1.14 shows a graphical representation of α and β for a point q whose mapping is being calculated for SI_1 using the coordinate system defined by O_k and $\vec{v_1}$.

Equations 1.4 and 1.5 show the operations needed to compute both α and β to obtain the mapping for a point q when generating the signature SI_1 using $\vec{v_1}$.

$$\alpha = \sqrt{||q - O_k||^2 - (\vec{v_1} \cdot (q - O_k))^2} \tag{1.4}$$

$$\beta = \vec{v_1} \cdot (q - p) \tag{1.5}$$

Once all the points are mapped to their (α, β) coordinates, the 2D space is discretized into bins which count the number of points falling in each one of them, this process generates a 2D histogram. The contribution of each point to the bin is bi-linearly interpolated to its neighboring bins so that the final feature is less sensitive to positional noise that might affect the points of the local surface.

The other two signatures SI_2 and SI_3 are generated following the same process adopted to create SI_1. In this case, SI_2 and SI_3 might be generated by substituting the reference axis $\vec{v_1}$ in Equations 1.4 and 1.5 with $\vec{v_2}$ and $\vec{v_3}$ respectively. $\vec{v_1}$ is used in Equations 1.4 and 1.5 for mapping a point q to the 2D coordinate system defined by O_k and $\vec{v_1}$, by substituting $\vec{v_1}$ with $\vec{v_2}$ and $\vec{v_3}$ the coordinate system is changed to be defined by $(O_k, \vec{v_2})$ and $(O_k, \vec{v_3})$ respectively.

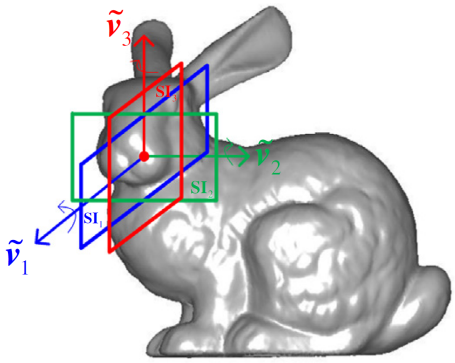


Figure 1.15: An illustration of a TriSI feature. (Figure reproduced from [39]).

In the end, the three signatures are concatenated to obtain a high discriminative feature $f = SI_1, SI_2, SI_3$. Because the three axes are orthogonal to each other, the information from the three signatures is complementary and relatively non-redundant. As a last step, to make the feature vector compact, f is compressed by projecting it onto a PCA subspace.

1.3.3 Real-time object recognition with low-cost sensors

In general, object recognition methods exhibit a considerable execution time, mainly due to two factors: (1) most methods make use of high precision tridimensional information which implies a high computational cost because of the complexity of the data that must be processed and (2) this high precision information is often provided by sweeping range laser scanners which are extremely slow and require a lot of time to perform a complete sweep and provide a full capture [40].

Figure 1.16: SICK S3000 laser scanner. (Image reproduced from *Sick.com*).

Those factors render impossible the use of traditional object recognition systems when demanding time constraints must be met. In our context of mobile robotics and continuous interaction with the user, traditional systems are not able to satisfy the requirements regarding to execution time, low power consumption and cost.

Besides the laser scanning, time-of-flight cameras (*ToF*), such as the *SR4000* camera from *Mesa Imaging*, provide depth maps by using a modulated light source. Devices which make use of this technology are able to achieve considerable higher capture rates per second than sweeping laser scanners (see Figure 1.17). Due to this fact, *ToF* devices are suitable for real-time robotic ap-

plications [41]. However, their high economic cost significantly reduces their usage.

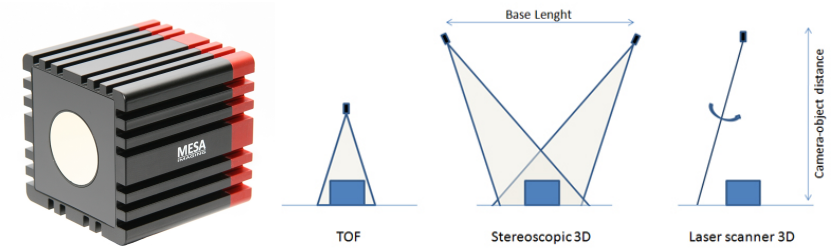


Figure 1.17: Left: SR4000 camera (Figure reproduced from *SR4000 dataset* [42]), Right: basic functioning schema of *ToF*, stereoscopic 3D and laser scanner systems (Figure reproduced from *Robotshop.com*).

As an alternative to the already shown sensor devices, RGB-D cameras emerged. These devices are characterized by their sensing technology which is able to provide high quality and synchronized color and depth information.

Some successful examples of this kind of devices are the *PrimeSense* Carmine, shown in Figure 1.19, and the *Microsoft* Kinect [1], shown in Figure 1.18. These devices make use of active sensing techniques to provide robust depth estimates in real-time. The basic functioning consists of projecting a known pattern by using an infrared emitter, then the pattern reflected by the surfaces is captured by an infrared sensor. Changes in the depth of the surfaces of the scene deform the original pattern. Those deformed patterns are correlated with the original ones to create an approximated depth map [43][44][45].

Figure 1.18: Left: first revision of the Kinect camera, Right: Kinect version 2. (Figures reproduced from *Wikipedia.org* under *Wikimedia Commons* license).

This kind of sensors exhibit a set of advantages which makes them suitable for real-time computer vision applications, including object recognition. Their

[1] `https://www.microsoft.com/en-us/kinectforwindows/`

most remarkable advantages are a reduced size, low power consumption and low economic cost so that they can be massively distributed. However, in order to keep those advantages, many sacrifices are required: they provide high quality depth maps but their range is quite limited in comparison with a high precision laser scanner; in addition, depth maps are noisy due to limitations of the technology. As we can observe, the advantages are significant but their associated drawbacks render the traditional approaches, which make use of high precision data, unsuitable for working with low cost sensors. In that sense, they have to be adapted to cope with those downsides.

Figure 1.19: Left: Asus Xtion Pro (Figure reproduced from *Asus.com*), Right: PrimeSense Carmine 1.08 (Figure reproduced from *Primesense.com*).

1.3.4 Fusion of photometric and geometric information

The vast majority of current 3D object recognition methods exclusively work with geometrical information to describe surfaces and detect objects in scenes [2]. Photometric information (colors and textures) is usually ignored. However, there are multiple psychological studies which confirm that colors have a significant influence on human vision when recognizing objects. In that sense, color should be considered in order to improve the efficacy of current models [46].

With the advent of low-cost 3D sensors which are capable of providing both geometrical and photometric information, researchers have started to develop tridimensional surface feature descriptors which include color and texure information. It is the case of the *CSHOT* descriptor [47] created by Tombari *et al.* as a generalization of the *SHOT* descriptor [30]. Its goal is to take into account mutiple kinds of surface information besides its geometry. Experiments show that the *CSHOT* descriptor notably improves the precision of the recognition system in scenes characterized by the presence of occlusions [47].

1.4 Proposal

After describing the motivation of this work and analyzing the state of the art of 3D object recognition systems that exist nowadays or have been proposed, we propose the implementation of a local surface feature based object recognition system which will operate over data streams provided by low-cost 3D sensors. This system will resort to the power of a mobile GPU parallel computing platform (*NVIDIA* Jetson TK1 [2]) to be able to operate in real-time. In addition, the system will be highly optimized for multi-threaded CPU execution on modern multi-processor architectures so that we can exploit all the capabilities of heterogeneous CPU/GPU platforms.

1.5 Goals

The main goal of this project is the development of an interactive 3D object recognition system. First, a sequential CPU pipeline will be implemented. That system will be optimized for multi-threaded CPU execution. In addition, it will be reimplemented to fit into a GPU computing platform to accelerate certain phases.

An extensive state of the art of 3D object recognition systems will be carried out. It will be mainly focused on 3D feature descriptors and real-time object recognition.

Furthermore, we will analyze the precision and efficiency of all the implemented surface descriptors. An extensive experimentation will be carried out to determine which descriptor offers a good balance between precision and runtime. That descriptor will be selected for the final implementation of the system.

As a secondary goal, but directly related to the previous one, we will create a 3D object dataset comprised of a set of reconstructed 3D object models and a handful of validation scenes. For this purpose, we will develop a 3D object reconstruction pipeline to generate models that will be used to train our system. The test scenes will be utilized to assess the precision and runtime of our proposed pipeline and its optimizations.

To sum up, the main goals of this project are the following ones:

- State of the art of existing 3D object recognition systems.

[2] `https://developer.nvidia.com/meet-jetson-embedded-platform`

- 3D object dataset and evaluation scenes.
- CPU implementation of a 3D object recognition pipeline.
- Descriptor precision and performance analysis.
- Multi-threaded and optimized CPU implementation.
- GPU acceleration of certain stages of the pipeline.

Figure 1.20 shows the proposed timeline for accomplishing the aforementioned general goals. Note that in order to achieve the objectives, they will be dealt with in a sequential manner. The project will start on Thursday 29th January, 2015.

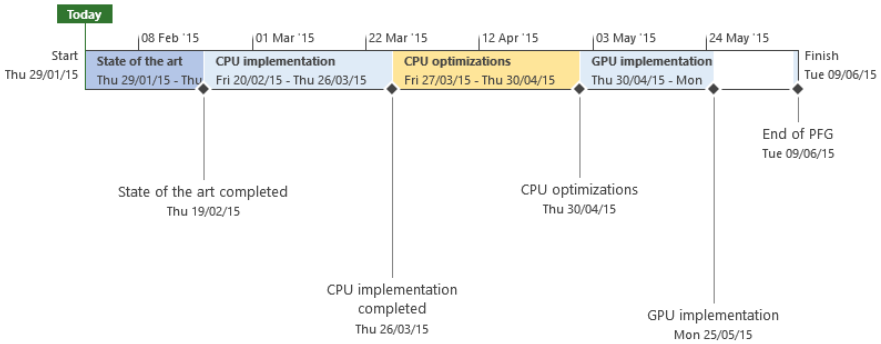


Figure 1.20: Project timeline.

Next, we will decompose the general goals into specific ones and distribute them over the available time in form of tasks that will compose our project. In addition, we will show detailed Gantt diagrams for each task, providing their duration, dependencies and start/finish dates.

The first task will be **1. State of the art**. This task spans 19 days of work from January 29th to February 19th and consists of the following subtasks:

- **1.1. Selection of relevant descriptors**
- **1.2. Description of selected descriptors**
- **1.3. Document: Introduction**

Figure 1.21 shows the Gantt diagram for the task *1. State of the art*.

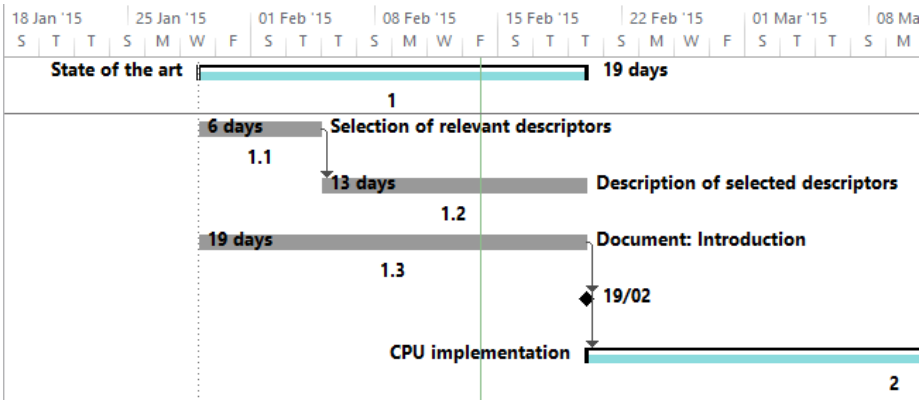


Figure 1.21: Gantt diagram for Task 1: State of the art.

The second task will be **2. CPU implementation**. This task spans 30 days of work from February 20th to March 26th and consists of the following subtasks:

- **2.1. Pipeline implementation with Point Cloud Library.**
- **2.2. Inclusion of Point Cloud Library descriptors.**
- **2.3. Refactoring and multi-object pipeline.**
- **2.4. Comparison of descriptors.**
- **2.5. Document: Proposal - CPU implementation.**

Figure 1.22 shows the Gantt diagram for the task *2. CPU implementation*.

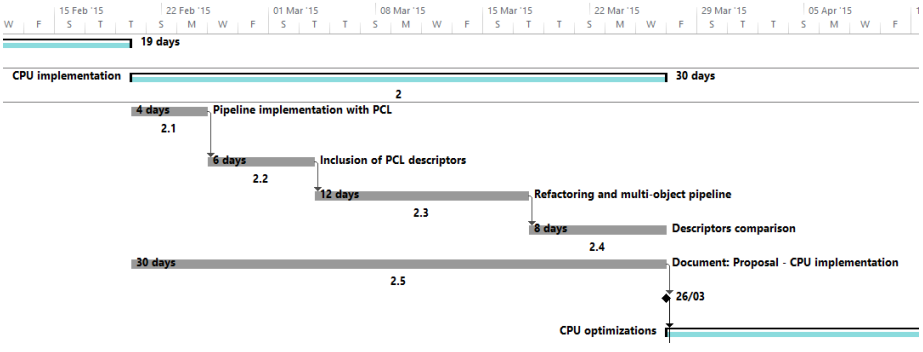


Figure 1.22: Gantt diagram for Task 2: CPU Implementation.

The third task will be **3. CPU optimizations**. This task spans 30 days of work from March 27th to April 30th and consists of the following subtasks:

- **3.1. Optimization study.**
- **3.2. Optimization tools configuration.**
- **3.3. Optimized CPU implementations.**
- **3.4. CPU optimizations experimentation.**
- **3.5. Document: Proposal - CPU optimizations.**

Figure 1.23 shows the Gantt diagram for the task *3. CPU optimizations*.

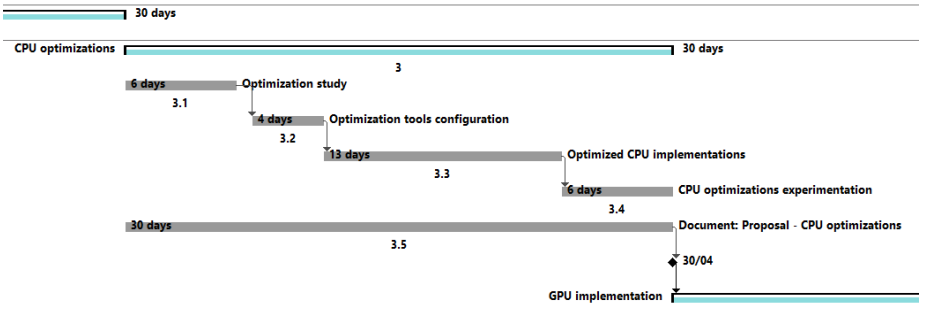


Figure 1.23: Gantt diagram for Task 3: CPU optimizations.

The fourth task will be **4. GPU implementation**. This task spans 21 days of work from April 30th to May 25th and consists of the following subtasks:

- **4.1. Descriptor improvement study.**
- **4.2. Jetson TK1 configuration.**
- **4.3. Port object recognition pipeline to Jetson TK1.**
- **4.4. CUDA implementations.**
- **4.5. Document: Proposal - GPU implementation.**
- **4.6. Document: Methodology.**
- **4.7. Document: Conclusions.**

Figure 1.24 shows the Gantt diagram for *4. GPU implementation*.

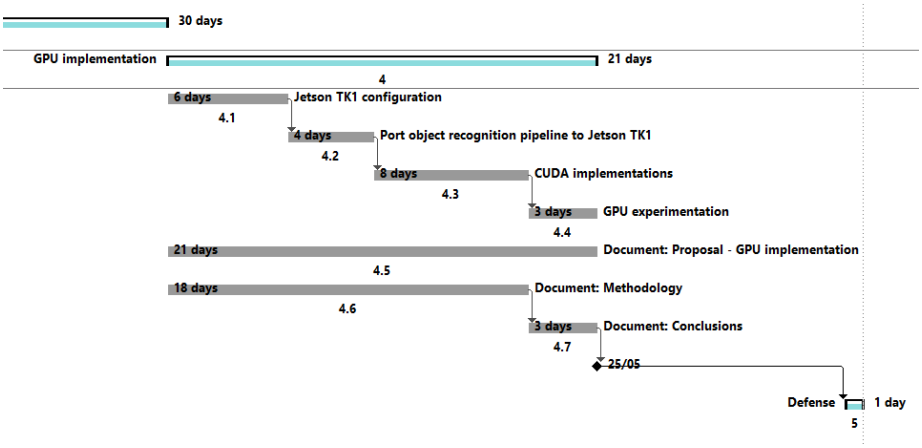


Figure 1.24: Gantt diagram for Task 4: GPU implementation.

1.6 Structure

This document is structured as follows: Chapter 1 outlined the project and presented a detailed state of the art of each issue covered in this work. It exposed the motivation of the work and its general and specific goals. Chapter 2 presents the methodology followed during the development of this project. It exposes the different techniques, technologies and tools used to carry out this work. Chapter 3 presents our proposed solution. It first describes the proposal in depth. Then it presents the CPU implementation, the CPU optimizations and the GPU implementation. The experiments and discussion for each part of the implementation are detailed throughout the chapter. Finally, Chapter 4 details the conclusions extracted from this project and draws future work and research directions.

Chapter 2

Methodology

In this chapter we will describe the methodology employed in this work. It is organized into three different sections. Section 2.1 introduces the purpose of this chapter. Section 2.2 explains the multiple technologies that were used to obtain data, develop our applications or even improve them. At last, Section 2.3 describes the experimentation methodology that was followed to assess the different object recognition systems or phases.

2.1 Introduction

In order to carry out this project, we need the support of many different technologies. First, we need a set of data acquisition systems to feed our applications with 3D information. Subsequently, we need tools for developing those applications and, as far as possible, ease and streamline the development process. Section 2.2 describes both kinds of technologies that were used during the development of the project.

In addition, we will eventually need to compare our systems with others or even with itself in order to verify the improvements introduced by changes in some of its phases. In this sense, objectively quantifying the performance of the system turns out to be critical if we want to improve the system and work on the right track. Section 2.3 deals with this problem and establishes the experimentation methodology that will be followed during our tests.

2.2 Technologies

In this section we will describe the different technologies which were directly or indirectly used during the development of this project. Subsection 2.2.1 describes the software tools and Subsection 2.2.2 focuses on hardware platforms and devices.

2.2.1 Software

The object recognition system will be implemented using C/C++ as the programming language of choice. We are going to use Visual Studio 2013 as the IDE and Git (with a private repository hosted on *BitBucket.org*) for version control. CMake will be used for managing the build process of the project. We will also make use of *Boost* libraries for some particular tasks and the *STL* containers, iterators and algorithms.

In order to manage the acquisition devices, we will resort to the *Open Natural Interaction* (OpenNI) project for handling the data streams from compatible cameras like *Microsoft Kinect* and *PrimeSense Carmine*. The new *Kinect 2.0* is not compatible out of the box with it so we will make use of the *Kinect for Windows SDK 2.0* to obtain data from it. In addition, we will explore the use of *libfreenect2* and its corresponding driver since the aforementioned SDK is only compatible with *Microsoft Windows* systems.

The *Point Cloud Library* (PCL) will be used to process the information provided by the sensors. The main reason for using this library is streamlining the development process. We will make use of many algorithms which are inherently complex. The purpose of this work is not implementing those algorithms one by one but knowing and applying them with a higher purpose and even improve them if possible. Since we are not going to reinvent the wheel, the *Point Cloud Library* provides us with all the functionality and necessary tools to implement our object recognition system at a higher level without losing time with unnecessary details.

The CPU implementation will be optimized using *Open Multi-Processing* (OpenMP) to get all the possible performance on multi-core processors (parallel programming).

The GPU accelerated version will be implemented into a *Jetson TK1* parallel platform using *Compute Unified Device Architecture* (CUDA) for parallelizing parts of the CPU implementation, and its environment for profiling and improving the system.

2.2.1.1 OpenNI

Open Natural Interaction (OpenNI) is an open source software project which offers a set of APIs for accessing natural interaction devices. Its open source framework is also referred as the OpenNI SDK and is has become the recognized standard for developing computer vision middleware and 3D solutions. The original OpenNI project, founded by PrimeSense, was shutdown when Apple acquired PrimeSense in 2013. Since then, Occipital and Structure are keeping a forked version of OpenNI named OpenNI 2.0 with their own SDK.

The OpenNI 2.0 API provides a single and unified interface for accessing depth, RGB and IR streams from PrimeSense compatible depth sensors. It also provides a simple interface which can be used by middleware libreries to interact with depth sensors. In this sense, applications can make use of the basic data provided by supported sensors and third party middleware such as skeletal tracking.

The architecture or stack of the OpenNI 2.0 SDK is shown in Figure 2.1. As we can observe, the hardware abstraction layer provides a unified interface so that the device driver layer is completely abstracted from the rest of the OpenNI stack so OpenNI might be installed with different drivers for using various devices. The *OpenNI Programmer's Guide* provides further useful information about the API [48].

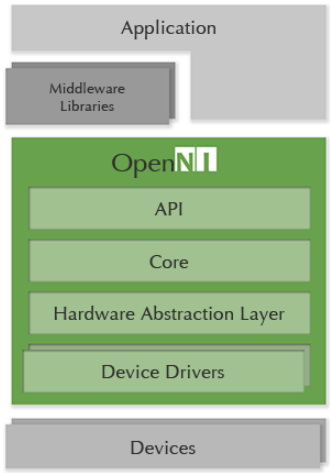


Figure 2.1: OpenNI 2.0 SDK Architecture.

2.2.1.2 Point Cloud Library

The Point Cloud Library (PCL) [49] is an open project which is meant to provide support for the common 3D building blocks that computer vision and robotics applications might need. The library contains state-of-the-art algorithms for filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation. In addition, the library is supported by an international community of robotics and perception experts and more recently by the *OpenPerception* organization. The PCL is released under the terms of the *3-clause BSD* license and it is open source software and more importantly, it is cross-platform.

Figure 2.2: Point Cloud Library (PCL) logo.

From an architectural and implementation point of view, the PCL is a fully templated, modern C++ library for 3D point cloud processing which makes use of modern CPUs optimizations like SSE to achieve outstanding levels of efficiency and performance. In addition, the library is based on *Eigen* and provides support for *OpenMP* and *Intel Threading Building Blocks* libraries for multi-core parallelization. Furthermore, all the modules and algorithms pass data around using Boost shared pointers, thus avoiding the need to re-copy data. For more information, we suggest reviewing the introductory paper *3D is here: Point Cloud Library* [49].

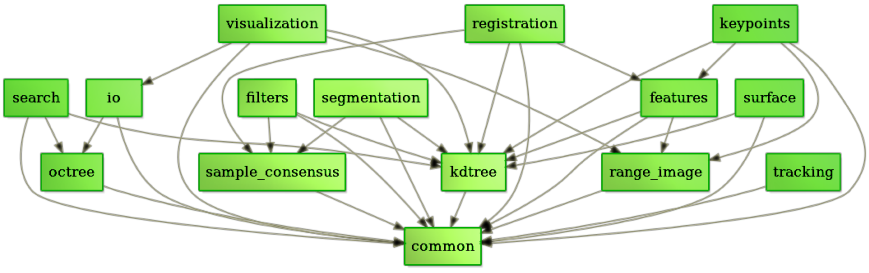


Figure 2.3: Point Cloud Library (PCL) dependency graph.

2.2.1.3 OpenMP

Open Multi-Processing (OpenMP) is a set of compiler directives and callable runtime library routines that extend Fortran and C/C++ to express shared-memory parallelism [50]. It was designed to be a flexible standard and easily implemented across different platforms. Nowadays, it is the industry-standard API for high-level shared-memory parallel programming due to its simplicity and effectiveness.

From the inside, its model uses multithreading to parallelize tasks using threads, forked by a master thread, which run concurrently with a custom runtime environment mapping those threads to cores. In this way, both task and data parallelism can be achieved thanks to work sharing constructs provided by the API.

The OpenMP standard comprises four distinct parts: control structures, work sharing constructs, the data environment, synchronization and the runtime library.

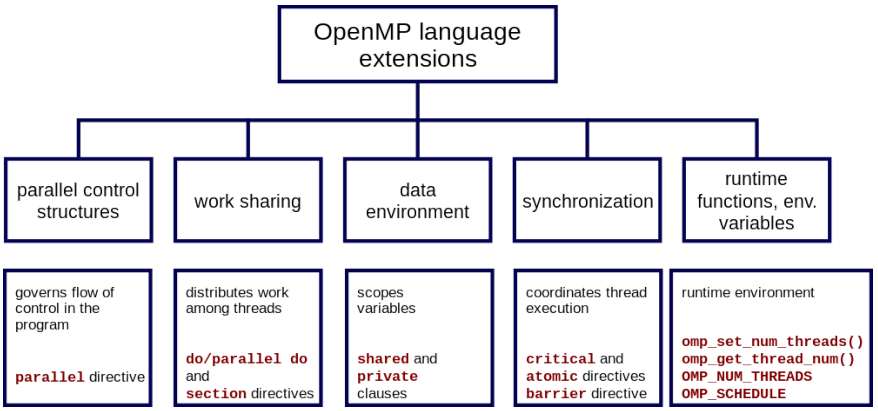


Figure 2.4: OpenMP language extensions (Figure reproduced from *Wikimedia.org*).

Listing 2.1 shows a brief example of OpenMP usage to parallelize a simple vector addition by using the preprocessor directive *omp parallel for*.

Listing 2.1: Simple OpenMP loop parallelization example.

```
1 void sum_vectors(int *a, int *b, int *c) {
2   #pragma omp parallel for
3   for (int i = 0; i < N; i++)
4     c[i] = a[i] + b[i];
5 }
```

2.2.1.4 NVIDIA CUDA

The Compute Unified Device Architecture (CUDA) [51] is a parallel computing platform and programming model which allows using a GPU for general purpose computing. From a software point of view, CUDA is a heterogeneous programming model in which both the CPU (host) and GPU (device) are used. The distinctive features of this model are simplicity and elegance. Programs are written in the familiar C/C++, Fortran, Python and many more languages whilst a set of extensions in the form of basic keywords is added to those languages in order to let the developer express the parallelism of the application thus directing the compiler for mapping that portion of the application for GPU execution on CUDA compatible devices.

A typical CUDA program is comprised of kernels, i.e., functions which are executed in parallel across a set of GPU threads. Those threads are organized in blocks of threads and grids of blocks. Figure 2.5 shows the aforementioned hierarchy.

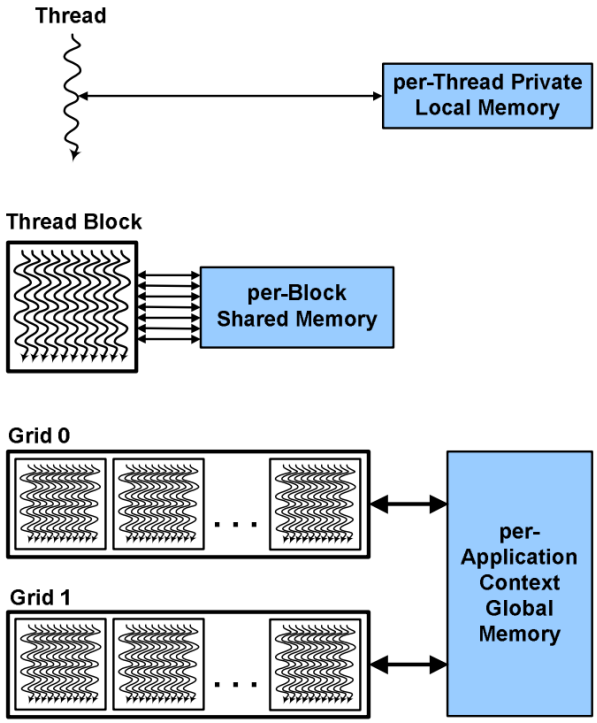


Figure 2.5: CUDA hierarchy of threads, blocks and grids with corresponding per-thread, per-block and per-application memory spaces.

Each thread within a block executes a copy of the kernel function. Threads of the same block are concurrently executed and cooperate through memory sharing and synchronization barriers. Blocks are further organized into grids which are arrays of blocks which execute the same kernel.

As an illustration, the code sample shown on Listing 2.2 adds two vectors A and B of N elements and stores the result into C by using the kernel *VecAdd* which is executed on a grid of one block with N threads in that block. Each thread computes the index of its corresponding element and adds it.

Listing 2.2: Simplified CUDA kernel for vector addition.

```
__global__ void VecAdd (float* A, float* B, float* C) {
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
int main() {
  // Memory allocation and copy...
  // Kernel call with N threads
  VecAdd<<<1, N>>>(A, B, C);
  // Memory copy and free...
}
```

In addition to that programming model, CUDA also consists of a custom driver for the GPU as well as a compiler (NVCC for C) which takes an integrated CPU and GPU source code to produce GPU assembly code (PTX) and CPU code for the host which is later compiled by a typical C compiler. What's more, the CUDA toolkit provides a profiler and a debugger for GPU programs. The full CUDA SDK stack is shown in Figure 2.6.

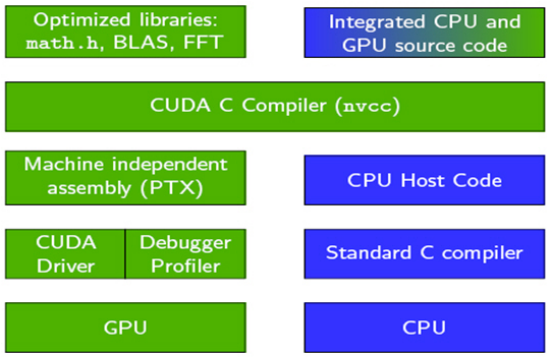


Figure 2.6: CUDA SDK stack (GPU parts in green, blue for CPU ones).

2.2.2 Hardware

Apart from all the software tools used to develop the system, this one has to be deployed into an embedded platform where many different devices will have to cooperate to provide the necessary data streams.

We have chosen the NVIDIA Jetson TK1 parallel computing platform to deploy our final and accelerated object recognition system due to its unprecedented performance for embedded computing systems. Also, many devices will be used to acquire depth and RGB frames: Kinect and PrimeSense Carmine (both PrimeSense compatible devices), and the recently introduced Kinect 2.0. We will review the aforementioned hardware devices one by one during the following sections.

2.2.2.1 Jetson TK1

During last years, general purpose computing on GPUs (GPGPU) has enabled crucial acceleration of various applications in many fields. This fact has increased the pace of the research progress and multiple lines of work which were previously discarded due to their computational demands are now taken into account.

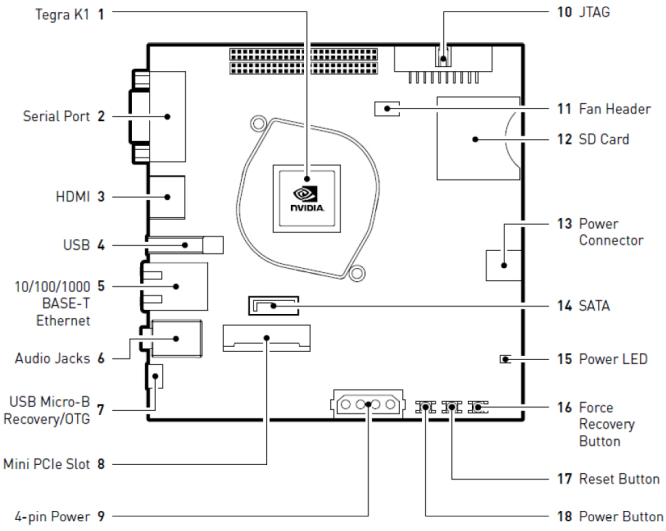


Figure 2.7: Jetson TK1 layout (Figure reproduced from [52]).

Nowadays, the need for computing power is complemented by the need for energetic efficiency (green computing). This characteristic is mainly rele-

vant in those scenarios in which energy consumption is a critical factor but a superior computing power still makes the difference. In this sense, embedded computing has become a great target for bringing up the computational horsepower provided by GPUs due to their increasing energetic efficiency. This is where the *Jetson TK1* platform [4] comes into play.

The following list shows the set of parts which compose the development board, its layout is shown in Figure 2.7. A detailed list of features can be consulted in [52], this summarized version was extracted from the quick start guide [53].

- **Processor:** NVIDIA Tegra K1 Mobile Processor
  - ARM Cortex A-15 CPU 2.32 GHz ARM quad-core
  - NVIDIA Kepler GK20a GPU 192 CUDA cores
- **Memory**
  - 2 GB DDR3L 933MHz EMC x16 64-bit data width system RAM
  - 16 GB fast eMMC 4.51 storage
- **Network**: Realtek RTL8111GS (10/100/1000 BASE-T Ethernet)
- **Audio**: Realtek ALC5639 HD Audio (Microphone and headphone jacks)
- **Input/Output:**
  - USB Type-A Host 3.0 / Micro USB connector
  - Display HDMI connector
  - Half mini-PCIe expansion slot
  - SATA connector / 4-pin power connector
  - RS232 serial port (DB9)
  - SD card connector / JTAG connector / 2x I/O expansion headers
- **Power:** External 12V AC Adapter

The most important component of this development board is its *NVIDIA Tegra K1* mobile processor [4]. It was designed to revolutionize the possibilities of embedded computing by integrating the power of GPU processing in a consumption-aware platform. The technical details of the processor can be consulted on its corresponding *Technical Reference Manual* [54].

Mainly, two components of this processor are responsible for its high computational power together with its reduced energetic consumption: the ARM **Cortex A15** CPU and the NVIDIA **Kepler GK20a** GPU, both with a set of additional features.

Figure 2.8: Block diagram of GK110 SMX, composed by 192 CUDA cores (single precision), 64 double precision units, 32 special function units and 32 load-/store units. (Figure reproduced from the document *Next Generation CUDA Compute Architecture: Kepler GK110* [57]).

On the one hand, the energetic efficiency is due to the fact that the CPU is not a typical ARM Cortex A15, it includes an additional core named *Battery Saver Core*. The original four cores of the ARM Cortex A15 and this additional one compose the **4-PLUS-1** architecture designed by NVIDIA. The technical specifications of this architecture are fully described on its corresponding *whitepaper* [55]. The **4-PLUS-1** architecture makes use of the *vSMP* (Variable Symmetric Multiprocessing) technology. In this way, the *Battery Saver Core* is responsible for executing normal tasks at low frequency while the other four

main cores, which work at a higher clock speed, are activated when a computational demanding task is executed. This technology allows a selective activation of each one of the five cores to optimize the resources for the workload, this has become a crucial optimization for mobile systems together with the proven efficiency of four core architectures [56].

On the other hand, the main source of computational power is the NVIDIA Kepler GK20a [57]. The architecture of this GPU is essentially the same as the one of high-end graphics cards but it also includes a set of optimizations to reduce power consumption while preserving performance. The GK20a features 192 CUDA cores organized into a *GPC* (Graphics Processing Cluster) with a single *SMX* (Advanced Symmetric Multiprocessor) and a memory controller. Figure 2.8 shows a block diagram of an GK110 SMX. Beyond these hardware improvements, this architecture supports a set of APIs specifications like OpenGL 4.4, DirectX 11.2 and CUDA 6.

To sum up, thanks to the 4-PLUS-1 and GK20a architectures, the Tegra K1 processor provides high performance for both single and multithreaded applications on mobile systems with an extraordinary energetic efficiency and supporting many graphics and computation APIs specifications. In this sense, the Jetson TK1 board becomes an ideal platform for developing applications for high performance embedded systems.

2.2.2.2 Depth sensors

Nowadays, many depth measurement techniques are available including ultrasonic waves, microwaves, and even light waves. The most popular devices resort to light waves by making use of different strategies: triangulation, time-of-flight, structured light or laser scan. In Section 1.3.3 we reviewed those different systems and we concluded that structured light sensors as the PrimeSense Carmine or the original Kinect were perfect for our application since they offer a considerable quality with a reduced economic cost. In this sense, we decided to use both sensors for this project.

The Kinect and Carmine sensors consist of an infrared laser emitter, an infrared camera and an RGB one. The laser emits a single beam which is split into multiple ones by a diffraction grating in order to create a constant speckle pattern. This pattern is projected onto the scene and its reflection is captured by the infrared camera. The captured pattern is then correlated against a reference one to obtain a disparity image. For each pixel, the distance to the sensor can

be computed from the corresponding disparity. Figure 2.9 shows the speckles pattern projected by a Kinect device and the corresponding depth map for that image.

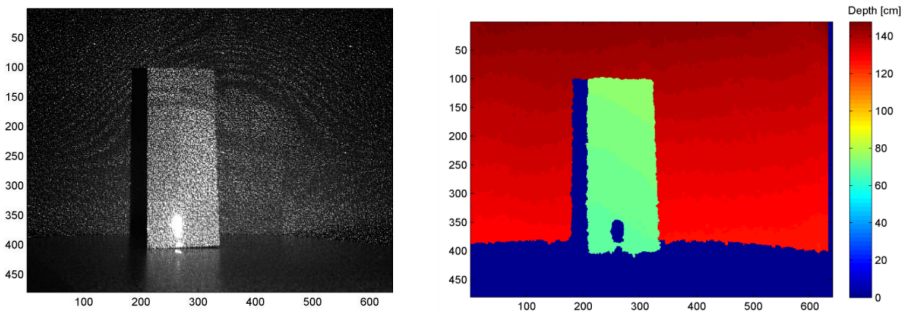


Figure 2.9: Speckle pattern and depth map (Figure reproduced from [58]).

The depth measurement is performed by a process named depth triangulation [59]. First, a set of reference patterns are obtained by projecting those patterns on a plane at a know distance. Kinect and PrimeSense sensors use three patterns for three different regions of distance: 0.8 - 1.2 meters, 1.2 - 2.0 meters and 2.0 - 3.5 meters. The nearest region has a higher accuracy than the farthest one.

Later, the IR emitter projects those patterns onto unknown scenes. The speckles are then projected onto surfaces whose distance to the sensor is different than the one of the reference plane for that pattern. Because of that, the speckles on the captured infrared image will be shifted in the direction of the baseline between the IR emitter and the perspective center of the IR camera [58]. The shifts can be measured for all speckles using an image correlation procedure to generate a disparity image which can be used to compute the distance of each pixel to the sensor.

Figure 2.10 shows the mathematical model explained by Khoshelham *et al.* [58] which illustrates the relationship that exists between the distance of an object point k to the sensor and the measured disparity d, assuming a depth coordinate system in which the origin is the perspective center of the IR camera, the Z axis is orthogonal to the image plane, the X axis is perpendicular to the Z one in the direction of the baseline b and the Y axis is orthogonal to both of them creating a right handed coordinate system. As we have previously noted, if an object is displaced closer or further away from the sensor, the speckle is displaced in the X axis direction thus allowing the measurement

of the disparity.

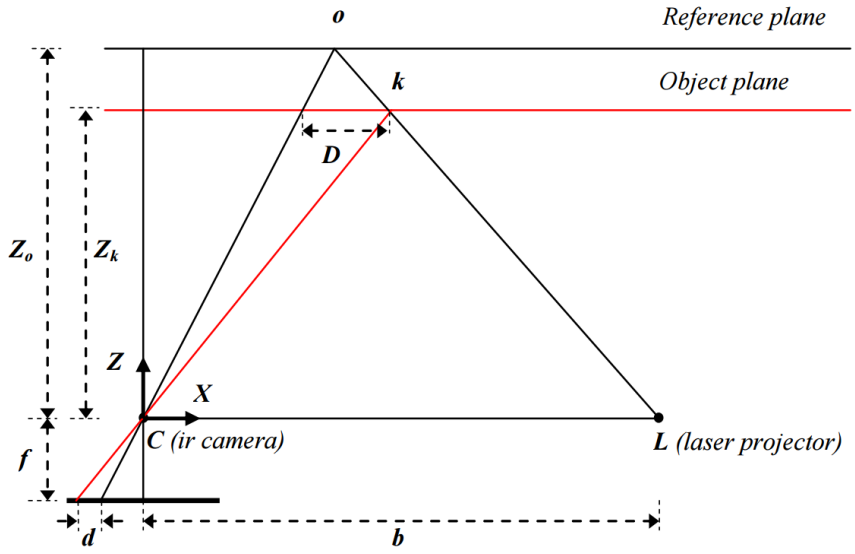


Figure 2.10: Kinect mathematical model (Figure reproduced from [58]).

Applying triangle similarity rules we can extract the equivalences shown on Equation 2.1 and Equation 2.2. Substituting D from Equation 2.2 into Equation2.1 we can express Z_k (the depth of the point k in the scene) as shown on Equation 2.3.

$$\frac{D}{b} = \frac{Z_o - Z_k}{Z_o} \tag{2.1}$$

$$\frac{d}{f} = \frac{D}{Z_k} \tag{2.2}$$

$$Z_k = \frac{Z_o}{1 + \frac{Z_o}{fb}d} \tag{2.3}$$

This mathematical model based on triangulation allows us to derive the depth of a pixel from the measured disparity. The planimetric object coordinates can be also computed as shown on Equations 2.4 and 2.5.

$$X_k = -Z_k f(x_k - x_o + \delta x) \tag{2.4}$$

$$Y_k = -Z_k f(y_k - y_o + \delta y) \tag{2.5}$$

As we can see, many calibration parameters are involved in this mathematical model: the focal length f, the principal point offsets (x_o, y_o), the lens distortion coefficients $(\delta x, \delta y)$, the base length b and the distance to the reference pattern Z_o. These parameters are determined by a calibration process whose explanation is beyond the scope of this work, readers may consult [58] for a detailed analysis.

We will also make use of the recently introduced Kinect 2.0 device. This scanner is a revised version of the original Kinect with a higher resolution, wider field of view and other improvements. A comparison between the presented depth sensors is shown in Table 2.1.

Sensor	Color	Depth	Min./Max. Dist.	FOV (H/V)	Power Consumption
Kinect	640x480 @30fps	320x240	40cm-4.5m	57/43 °	12 W
Kinect 2.0	1920x1080 @30fps	512x424	50cm-4.5m	70/60 °	12 W
Carmine 1.09	640x480 @30fps	640x480	0.35cm-1.4m	57.5/45 °	2.25 W

Table 2.1: Comparison of depth sensors.

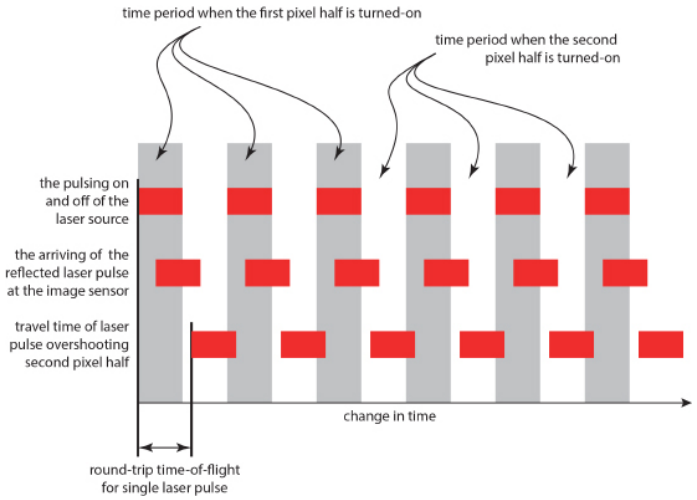


Figure 2.11: The indirect Time-of-Flight method (Figure reproduced from [61]).

This new camera makes use of the time of flight method. In this case, a clock signal strobes an array of laser diodes whose beams go through diffusers projecting short pulses of infrared light onto the scene. These pulses reflect on the different surfaces and are captured by the IR camera of the Kinect 2.0.

This IR camera is special because each pixel is divided into two parts which act as accumulators of photons of laser light. The aforementioned clock signal controls which half of each pixel is turned on and registering light pulses. Each half is 180 degrees out of phase from the other one, so that when the first is on the other is off and vice versa. At the same time, the laser light source is pulsed in phase with the first pixel half. In this sense, the clock signal determines which half is actively registering reflected light as shown in Figure 2.11. This method is named *indirect Time-of-Flight* and the distance can be inferred by comparing the ratio of light received in each pixel half, i.e., the more light that arrives in the second accumulator against the first, the farther that part of the scene is from the sensor [60].

2.3 Experimentation

In this section, we will briefly introduce the experimentation methodology which will be followed during the rest of this document.

2.3.1 Measuring performance

We will measure the performance of our implementations from two points of view: accuracy and efficiency. The first one will be analyzed using typical classifier metrics: true positives, false positives, true negatives, false negatives, precision and recall. The latter one will be measured by resorting to wall clock execution time which will be obtained using C++ high-resolution timers from the *Boost* library. A more detailed explanation of this methodology is deferred to Section 3.3.3.

It is worth noting that all timing results are the average of 100 executions which were performed to avoid possible performance issues that might arise due to processes executing in background or system operations. A single execution could be significantly affected by these circumstances. Assuming that these effects do not always happen, we can obtain more reliable results by averaging a set of executions.

2.3.2 Test systems

The test machine used for running the CPU related experiments is an Intel(R) Core i5-3570 CPU @ 3.40 GHz (4 CPUs), 3.4GHz with 8 GiB RAM DDR3 1866

MHz and an ASUS P8H77-M Pro motherboard with a chipset Intel H77. The operating system is Windows 8.1 Pro 64-bit (6.3, Build 9600). The programs were compiled using the Visual Studio 2010 Professional integrated compiler on Visual Studio 2013 Professional using Release settings for maximum speed optimization.

The test machine for other CPU and GPU experiments was the Jetson TK1 board, previously described in Section 2.2.2.1. The operating system is a custom Ubuntu (Linux For Tegra r21.3) included in the NVIDIA Jetson Jetpack. The programs were compiled using *g++ 4.8.2* for ARM and *nvcc* from CUDA Toolkit 6.5 with Release settings for speed optimization.

Chapter 3

A 3D object recognition pipeline

In this chapter we will describe the implementation of our system proposal introduced in Chapter 1. This chapter is organized in six sections. Section 3.1 introduces our proposed solution. Section 3.2 describes the CPU implementation of the recognition system. Section 3.3 shows the experimentation which was carried out to validate the CPU implementation and assess its performance. Section 3.4 repeats this experimentation on the Jetson TK1 to determine the performance of our system in that platform. Section 3.5 describes the CPU optimizations and their results and achieved speedup. Section 3.6 is devoted to the GPU implementation of the aforementioned pipeline, including the experimentation and discussion.

3.1 Introduction

After reviewing the state of the art of 3D object recognition systems and defining the typical stages of a recognition pipeline we will develop a CPU implementation. This sequential implementation will be programmed in C++, making use of the *Point Cloud Library* to perform 2D/3D image and point cloud processing operations.

In addition, experiments will be carried out to determine the computational load of each part of the system. This information will be used to accelerate the whole pipeline by implementing the critical phases, which are suitable to be

parallelized in a GPU-based parallel computing platform like the *Jetson TK1* using *CUDA*.

3.2 3D object recognition pipeline on CPU

The first step for implementing the object recognition system is the sequential CPU implementation of a full pipeline. Figure 3.1 shows a block diagram of our proposed pipeline which is based on the one proposed by Aldoma *et al.* [23]. This pipeline is divided into six main stages: keypoint extraction, descriptor extraction, feature matching, correspondence grouping, instance alignment and hypothesis verification. The system takes a scene point cloud as an input and outputs the recognized models and their estimated poses. This whole pipeline is described throughout Subsections 3.2.2, 3.2.3, 3.2.4, 3.2.5, 3.2.6 and 3.2.7. In addition, a preprocessing step is performed. It is described on Subsection 3.2.1.

Some parts of this pipeline are performed offline for training the system with 3D reconstructed models. This step is described in Subsection 3.2.8.

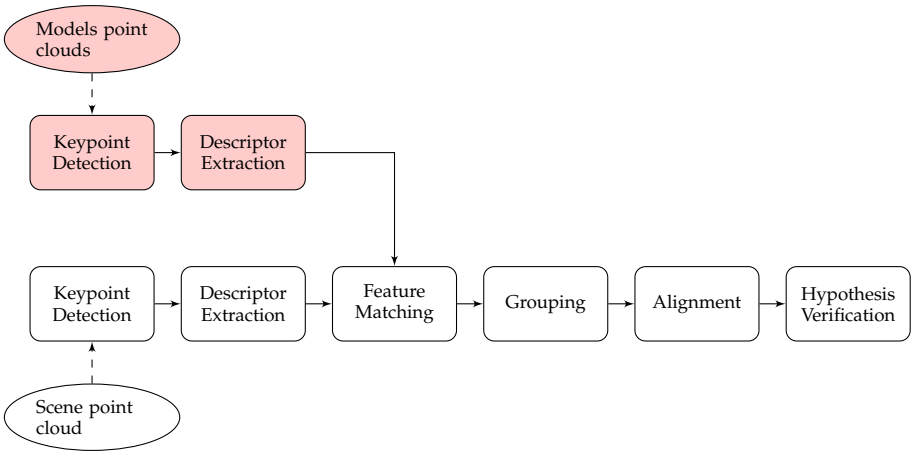


Figure 3.1: Object recognition pipeline implemented on the CPU. Red phases are performed offline while blank ones are live as the sensor provides information.

3.2.1 Preprocessing

Before entering the object recognition pipeline, point clouds go through a preprocessing step whose goals are removing non-relevant information, reducing the levels of noise, or computing additional information for later use. The main preprocessing steps performed in our implementation are bilateral filtering, normal estimation, plane segmentation, resolution computation and k-d tree generation.

3.2.1.1 Bilateral filtering

Due to the technologies used by the different 3D acquisition devices, observed surfaces are inherently noisy and some preprocessing steps are required to extract coherent information by minimizing this observation error. One of the most popular noise reduction techniques is the bilateral filter [62] which is able to remove the noise of the input data whilst preserving edge and curvature information. This filter was originally used in color and grayscale images but has been recently adapted to work with depth maps and 3D point clouds.

The original bilateral filter is a non-linear and edge-preserving filter in which the intensity value of each pixel in an image is replaced by a weighted average of the intensity values of the neighborhood pixels. The bilateral filter is a combination of two Gaussian kernels, one which gives priority to pixels that are close to the target pixel in the image plane, and a range kernel, which gives priority to the pixels which have similar intensity values as the target pixel. Thus, the new intensity value of a filtered pixel is given by Equation 3.1:

$$P_f = \frac{1}{K_p} \sum_{q \in \omega} V_q f(||p - q||) g(||V_p - V_q||) \tag{3.1}$$

$$K_p = \sum_{p \in \omega} f(||p - q||) g(||V_p - V_q||) \tag{3.2}$$

where K_p is a normalization factor (see Equation 3.2), ω is the neighborhood of the pixel, V_p and V_q are the intensity values of the target pixel and the current neighbor respectively, q and p are the positions on the image plane of the target pixel and the current neighbor respectively, and P_f is the new intensity value of the target pixel. The function contains the domain and range kernels $f(||p - q||)$ and $g(||V_p - V_q||)$ respectively. Those kernels, f and g are often Gaussian functions with standard deviations σ_s and σ_r.

This filter turns out to be quite important in our system because reducing

the noise of the point cloud improves the normal estimation process producing more stable normals. Since most 3D feature descriptors are based on the curvature of the geometry, obtaining stable normals indirectly increases the precision of the system.

We integrated this filter in our pipeline by using the implementation provided by the *Point Cloud Library*. The library provides a special implementation as a fast approximation of the bilateral filter, we will use this optimized method due to its efficiency. Listing 3.1 shows the developed implementation using the *Point Cloud Library*. Figure 3.2 shows the result of applying a bilateral filter to a point cloud.

Listing 3.1: Bilateral filtering with PCL.

```
1  void filter_bilateral (
2    const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
3    const float & pSigmaR,
4    const float & pSigmaS,
5    pcl::PointCloud<TPoint>::Ptr & pDstCloud)
6  {
7    pcl::FastBilateralFilterOMP<TPoint> bf;
8    bf.setInputCloud(pSrcCloud);
9    bf.setSigmaR(pSigmaR);
10   bf.setSigmaS(pSigmaS);
11   bf.applyFilter(*pDstCloud);
12 }
```

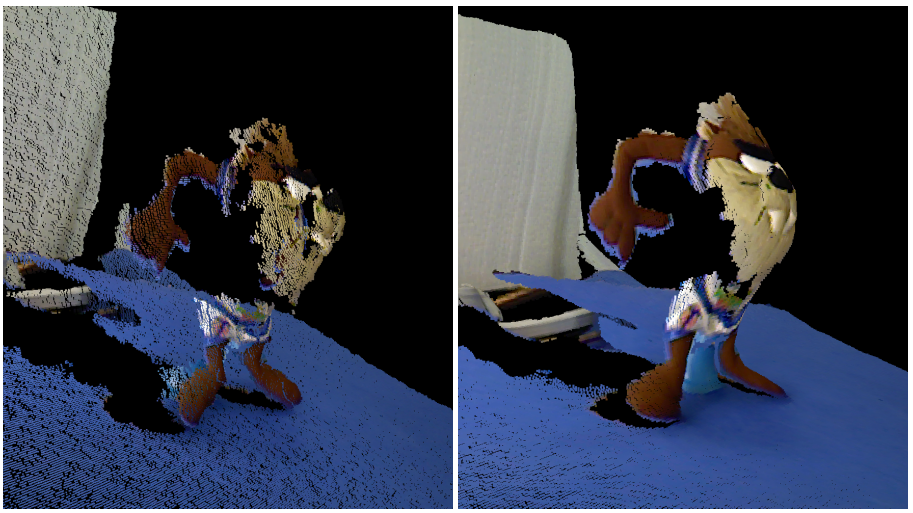


Figure 3.2: Raw point cloud of a scene with the *Tasmanian* model on it (left) and the same point cloud applying a bilateral filter with $\sigma_s = 15.0f$ and $\sigma_r = 0.05f$ (right). In this case, the filtering is excessive and it leads to information loss.

3.2.1.2 Normal estimation

Estimating surface normal information is a key step for the object recognition pipeline since they are often used to detect and describe point cloud features. Computing the normals of a surface is trivial but doing so with a point cloud is not so easy. Two options arise: transform the point cloud into a surface mesh and then compute the normals or use approximations to infer the surface normals from the point cloud directly. Since we don't want to introduce any overhead, we will approach this problem efficiently by using the point cloud approximation.

The most popular estimation method approximates the normal to a point on the surface by estimating the normal of the tangent plane to the surface which is also estimated by fitting a plane to the neighborhood of the point whose normal is being estimated using least squares fitting. The estimation of this normal is reduced to a *Principal Component Analysis* (PCA) of the covariance matrix C created using the neighbors p_i of the point p whose normal is being estimated:

$$C = \frac{1}{k}\sum_{i=1}^{k}(p_i - \vec{p}) \cdot (p_i - \vec{p})^T, C \cdot \vec{v}_j = \lambda_j \cdot \vec{v}_j, j \in \{0,1,2\} \tag{3.3}$$

where k is the number of neighbors considered, p_i are neighbors, $\vec{p}$ is the centroid of the neighborhood, λ_j is the j-th eigenvalue of the covariance matrix, and $\vec{v}_j$ the j-th eigenvector. The three eigenvalues are computed analytically and then if we have two of them which are close together and one significantly smaller, the eigenvectors for the first two eigenvalues determine the plane, while the eigenvector for the smallest eigenvalue determines the normal to this plane. This normal is then oriented towards the viewpoint by flipping it if needed to satisfy Equation 3.4.

$$\vec{n_i} \cdot (\vec{v_p} - p_i) > 0 \tag{3.4}$$

where $\vec{n_i}$ is the estimated normal, $\vec{v_p}$ is the viewpoint and p_i is the point whose normal is $\vec{n_i}$. It is important to remark that this approach works properly when the point cloud was acquired from a single point of view, but the re-orientation method does not hold if it has multiple acquisition viewpoints.

We integrated the normal estimation process in our pipeline by using the implementation provided by the *Point Cloud Library*. The search complexity is reduced by using a k-d tree. Figure 3.2 shows the developed implementation

using the *Point Cloud Library*. Figure 3.3 shows normals computed using this normal estimation method and how the noise reduction affects their stability.

Listing 3.2: Normal estimation implementation with PCL.

```
void compute_normals (
  const pcl::PointCloud<TPoint>::ConstPtr & pCloud,
  const pcl::search::KdTree<TPoint>::Ptr & pKdTree,
  pcl::PointCloud<TNormal>::Ptr & pNormals,
  const float & pRadius)
{
  pcl::NormalEstimation<TPoint, TNormal> ne;
  ne.setSearchMethod(pKdTree);
  ne.setRadiusSearch(pRadius);
  ne.setInputCloud(pCloud);
  ne.compute(*pNormals);
}
```

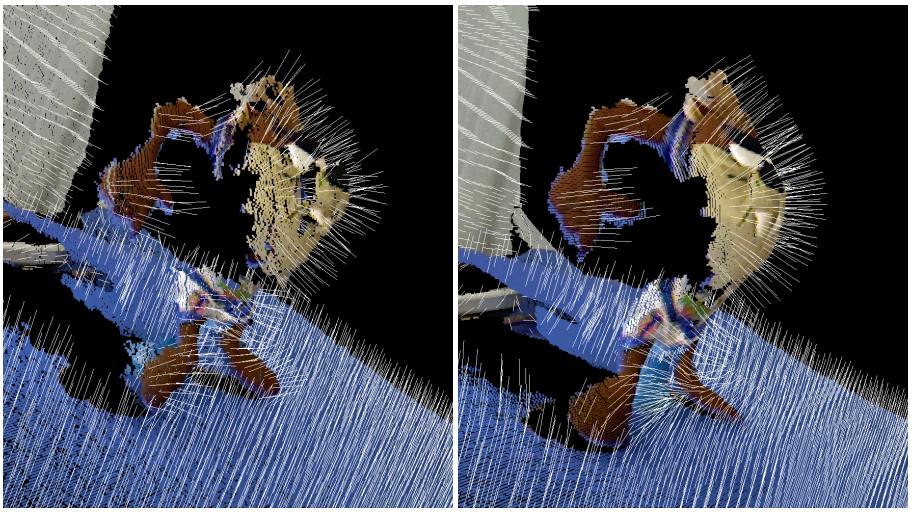


Figure 3.3: Normals of a raw point cloud of a scene with the *Tasmanian* model on it (left) and normals of the same point cloud applying a bilateral filter with $\sigma_s = 15.0f$ and $\sigma_r = 0.05f$ (right). The normals are much more stable thanks to noise reduction. It is necessary to find a balance between noise removal and geometric surface smoothing.

3.2.1.3 Plane segmentation

The vast majority of the information contained in the scene point cloud is irrelevant for recognizing objects since it does not represent objects at all and does not provide any cue about them. In fact, this information is a burden in terms of efficiency and precision for the following phases. For example, most surfaces are useless but their points are still taken into account to detect keypoints and compute descriptors. In addition, this extra information usually leads to bad matches and false hypotheses which have a negative impact on performance and precision. In this sense, segmentation is a key step for a successful detection and recognition so fast and efficient segmentation methods are required if we want our system to run in real time.

When dealing with objects, removing irrelevant information and segmenting them out is a relatively easy problem if we assume that objects tend to lie on flat surfaces. Most approaches for object segmentation detect planes in the scene and remove them, then a refinement process is applied to cluster objects so that they are easily segmented. The vast majority of the approaches rely on RANSAC or convex hulls. However, RANSAC takes a lot of time and convex hulls poorly represent the shape of many surfaces. Another approach is the one presented by Trevor *et al.* [63] which is a combination of an efficient *Organized Multi Plane Segmentation* (OMPS) and *Organized Connected Components Segmentation* (OCCS). This method achieves exceptional efficiency by taking advantage of organized point clouds thus enabling real-time plane segmentation for typical RGB-D data.

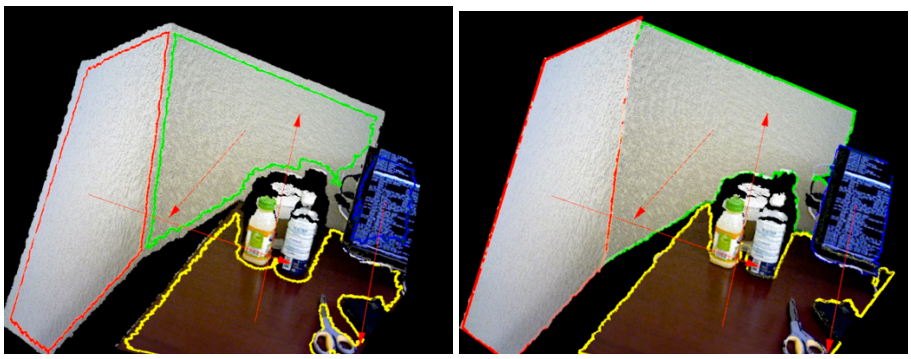


Figure 3.4: Segmented planes with OMPS outlined in red, green and yellow (left), and the result after refinement process (right). (Figure reproduced from [63]).

The *OMPS* approach segments the scene to detect large connected components corresponding to planar surfaces. The algorithm for computing the connected components explores the neighborhood of each point exploiting the organization of the point cloud to avoid searching in the whole point cloud for neighbors. The algorithm works by partitioning a organized point cloud P into a set of segments. A label L is assigned to each point of the organized cloud $P(x, y)$ denoted by $L(x, y)$. Points which belong to the same segment will be assigned the same label. In order to determine if two points, namely $P(x_1, y_1)$ and $P(x_2, y_2)$, belong to the same segment they are compared using a predicate $C(P(x_1, y_2), P(x_2, y_2))$. If the predicate is *true* then $L(x_2, y_2) = L(x_1, y_1)$. Otherwise, a new label is assigned to the point. The algorithm is similar to the two-pass binary image labeling one [64] modified to label point cloud data with continuous values based in some predicate [65].

In order to compute the predicate, the algorithm computes the *Hessian-normal* planar equation ($ax + by + cz + d = 0$) for each point in the Euclidean space. In order to do that, surface normals are required. After computing them (see 3.2.1.2), a point p can be represented as $p = \{x, y, z, n_x, n_y, n_z\}$. This representation is extended by computing the perpendicular distance to this point with normal (d in the planar equation) using the dot product $n_d = (x, y, z) \cdot (n_x, n_y, n_z)$. In this sense, the point representation is augmented as $p = \{x, y, z, n_x, n_y, n_z, n_d\}$. Using this point representation, the aforementioned predicate yields *true* if the angular difference between the normals and the distance in the range or n_d component fall below some predefined thresholds (see Equations 3.5 and 3.6).

$$n_{P(x_1,y_1)} \cdot n_{P(x_2,y_2)} < Threshold_{normal} \tag{3.5}$$

$$|d_{P(x_1,y_1)} - d_{P(x_2,y_2)}| < Threshold_{range} \tag{3.6}$$

After each point has been labeled, planes are fitted for each segment using least square fitting with a minimum number of inliers to accept those planes. The curvature of the segments is also computed to ensure that they are actually planar by filtering out segments whose curvature exceeds a determined threshold.

Once planes are fitted, they are refined using the algorithm described in [63] to be segmented out later as the last step of the OMPS method. In the end,

the OCCS method takes a greedy approach to euclidean clustering [33] using a comparison function based on a distance threshold to extract euclidean clusters which are supposed to be the final segmented objects.

The OMPS and OCCS segmentation methods were integrated in our pipeline by using their respective PCL implementations: *OrganizedMultiPlaneSegmentation* and *OrganizedConnectedComponentSegmentation*. Listings 3.3 and 3.4 show the implementations of the OMPS and the OCCS algorithms respectively.

Listing 3.3: Implementation of OMPS with PCL.

```
void segment_objects (
  const pcl::PointCloud<TPoint>::ConstPtr & pCloud,
  const pcl::PointCloud<TNormal>::ConstPtr & pNormals,
  const int & pMinInliers,
  const double & pAngularThreshold,
  const double & pDistanceThreshold,
  pcl::PointCloud<TPoint>::Ptr & pDstCloud)
{
  pcl::OrganizedMultiPlaneSegmentation<TPoint, TNormal, pcl::Label> mps;
  mps.setMinInliers(pMinInliers);
  mps.setAngularThreshold(pAngularThreshold*0.017453);
  mps.setDistanceThreshold(pDistanceThreshold);
  mps.setInputNormals(normalCloud);
  mps.setInputCloud(pCloud);

  std::vector<pcl::PlanarRegion<TPoint>,
    Eigen::aligned_allocator<pcl::PlanarRegion<TPoint>>> regions;
  std::vector<pcl::ModelCoefficients> modelCoefficients;
  std::vector<pcl::PointIndices> inlierIndices;
  pcl::PointCloud<pcl::Label>::Ptr labels(new pcl::PointCloud<pcl::Label>);
  std::vector<pcl::PointIndices> labelIndices;
  std::vector<pcl::PointIndices> boundaryIndices;
  mps.segmentAndRefine(
    regions,
    modelCoefficients,
    inlierIndices,
    labels,
    labelIndices,
    boundaryIndices);
[...]
}
```

The *OrganizedMultiPlaneSegmentation* outputs a set of labeled regions and the indices of the points in the original point cloud which belong to each particular segment or region (*labelIndices*). This information is used to discard the segmentation of those regions which contain less points than the specified inlier threshold.

Those planes which were filtered out are passed to the OCSS as a boolean mask (*planeLabels*) for efficiency reasons. After the segmentation, the OCCS provides a set of labels for the regions (*euclideanLabels*) and the indices of the points in the original point cloud which belong to each segment (*euclideanLabelIndices*).

Listing 3.4: Implementation of OCCS with PCL.

```
[...]
pcl::PointCloud<pcl::Label> euclideanLabels;
std::vector<pcl::PointIndices> euclideanLabelIndices;
pcl::EuclideanClusterComparator<TPoint, TNormal, pcl::Label>::Ptr ecc(
new pcl::EuclideanClusterComparator<TPoint, TNormal, pcl::Label>());
ecc->setInputCloud(pCloud);
ecc->setLabels(labels);
ecc->setExcludeLabels(planeLabels);
ecc->setDistanceThreshold(pDistanceThreshold, false);
pcl::OrganizedConnectedComponentSegmentation<TPoint, pcl::Label> occs(
ecc);
occs.setInputCloud(pCloud);
occs.segment(euclideanLabels, euclideanLabelIndices);
[...]
```

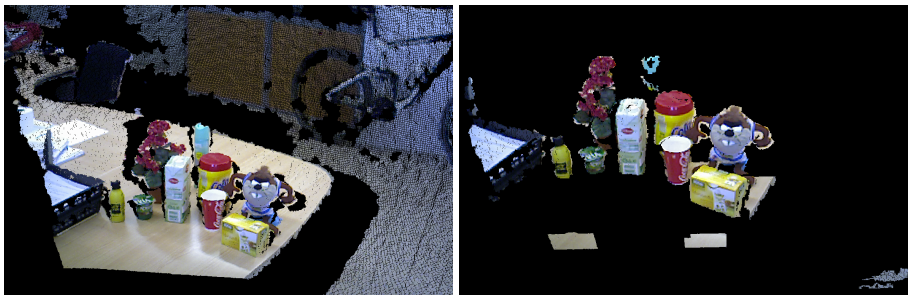


Figure 3.5: Sample scene captured with PrimeSense Carmine (left), same scene segmented with OMPS and OCCS (right) approximately 14% of the original points.

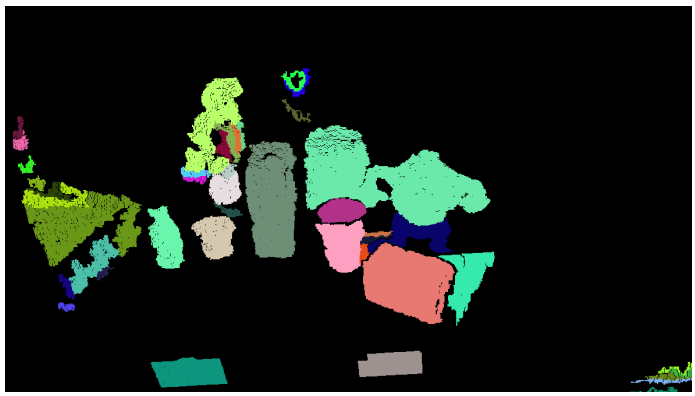


Figure 3.6: Clusters extracted by OMPS/OCCS.

3.2.1.4 Resolution computation

The resolution of a point cloud is defined as the average distance between each cloud point and its nearest neighbor. Estimating the spatial resolution of the model cloud might be useful to achieve some kind of resolution invariance by considering the different spatial parameters of the system as if they were given in units of cloud resolution. Listing 3.5 shows how the resolution of a point cloud is computed using PCL functions and a k-d tree to optimize the nearest neighbors search.

Listing 3.5: Cloud resolution computation with PCL.

```
double compute_resolution (
  const pcl::PointCloud<TPoint>::ConstPtr & pCloud,
  const pcl::search::KdTree<TPoint>::ConstPtr & pKdTree)
{
  double resolution = 0.0;
  int points = 0;
  int nres;

  std::vector<int> indices(2);

  std::vector<float> sqrDistances(2);

  for (size_t i = 0; i < pCloud->size(); ++i)
  {
    // Skip NaNs
    if (!pcl_isfinite((*pCloud)[i].x))
      continue;

    nres = pKdTree.nearestKSearch(i, 2, indices, sqrDistances);

    if (nres == 2)
    {
      resolution += sqrt(sqrDistances[1]);
      ++points;
    }
  }

  if (points != 0)
  {
    resolution /= points;
  }

  return resolution;
}
```

Once the resolution has been computed, we can set all the parameters of the system to express units in terms of cloud resolution. For example, instead of defining a fixed 15mm support size for a descriptor we can define it as 15 times the resolution, thus achieving some kind of resolution invariance.

3.2.1.5 k-d tree generation

A k-d tree [66] is a data structure used for organizing a set of points in a k-dimensional Euclidean space. They are a particular case of the *Binary Space Partitioning* with additional constraints: only using perpendicular planes to the axes of the coordinate system and storing a point in each node of the tree. The main advantage of a k-d tree is the efficiency for performing range search operations with $O(\log N)$ complexity.

The process for building a k-d tree is simple. A k-d tree is a binary tree, i.e., every non-leaf node has two children nodes, namely left and right. Each level of the tree splits the space on a specific dimension. In our case, a 3D space, all children of the root node on the first level will be split based on the first dimension X so that points with greater X values will be placed on the right subtree and those with lesser values will go to the left one. At the next level, the recently created nodes will be split on the Y axis. Then on the Z axis and then the splitting process gets back to the X axis. The median point is chosen as the root at each level. A k-d tree for a point cloud can be created using the PCL as shown in Listing 3.6. This tree will be used later by many stages of the pipeline to perform efficient search operations.

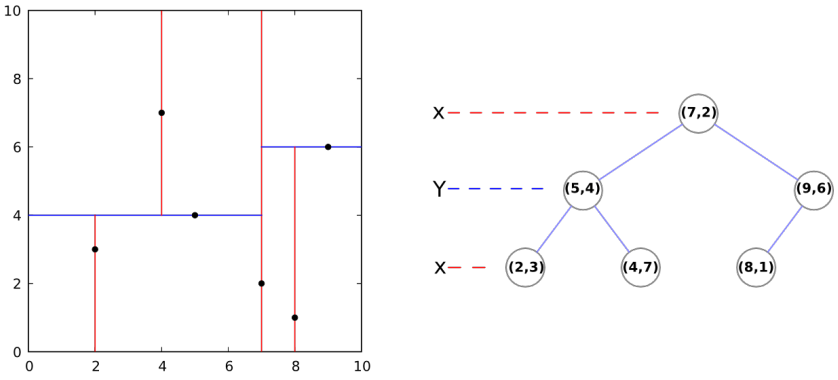


Figure 3.7: Example of a 2D k-d tree. (Figure reproduced from *Wikipedia*).

Listing 3.6: Building k-d trees for point clouds with PCL.

```
1 pcl::search::KdTree<TPoint>::Ptr modelKdTree(
2   new pcl::search::KdTree<TPoint>);
3 modelKdTree->setInputCloud(model);
```

3.2.2 Keypoint Detection

The first step of the actual recognition pipeline consists of detecting a set of feature points as a subset of the input point clouds. The contribution of the keypoint extraction phase is twofold. On the one hand, if the keypoints are properly detected, the information of their neighborhoods is representative enough to effectively describe the underlying surface. On the other hand, it reduces the number of descriptors needed by only computing them on the specific keypoints thus increasing the efficiency of the system. In this sense, the detection of interest points is an important step to reduce the search space for feature extraction and focus attention on significant structures. We have integrated three different and well known keypoint extractors: *Uniform Sampling*, *Intrinsic Shape Signatures* and *Scale Invariant Feature Transform*.

3.2.2.1 Uniform Sampling

Extracting keypoints with Uniform Sampling is simple and straightforward. Basically, the algorithm creates a 3D voxel grid over the input cloud data and approximates all the points which are present in the same voxel with their centroid. Figure 3.8 shows a set of keypoints extracted with Uniform Sampling.

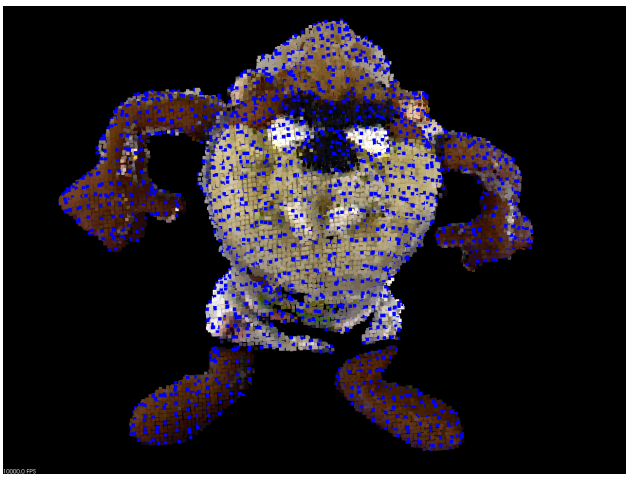


Figure 3.8: Keypoints extracted with uniform sampling in blue.

The *Point Cloud library* offers a simple implementation of this sampling. Listing 3.7 shows the integration of uniform sampling using PCL.

Listing 3.7: Implementation of Uniform Sampling with PCL.

```
void detect_keypoints_uniform_sampling (
  const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
  pcl::PointCloud<TPoint>::ConstPtr &pDstCloud,
  const float & pRadiusSearch)
{
  pcl::PointCloud<int> sampledIndices;

  pcl::UniformSampling<TPoint> us;
  us.setInputCloud(pSrcCloud);
  us.setRadiusSearch(pRadiusSearch);
  us.compute(sampledIndices);

  pcl::copyPointCloud(*pSrcCloud, sampledIndices.points, *pDstCloud);
}
```

Uniform Sampling is a quick and simple keypoint extractor but the keypoints provided as a result lack distinctiveness and repeatability. As shown in Figure 3.9 the keypoint set describes poorly the underlying surface since they are not detected using any intelligent strategy or underlying saliency.

Figure 3.9: Keypoints extracted with uniform sampling in blue. Notice the low distinctiveness of the keypoints as they are detected everyhwere.

3.2.2.2 Intrinsic Shape Signatures

A more complex keypoint detector is available in PCL, the *Intrinsic Shape Signatures* (ISS) detector. Originally, ISS is a descriptor proposed by Zhong [67] but it is also considered a detector since the author included an algorithm for

choosing keypoints which fit the descriptor. The algorithm performs a search through the surface and selects points with large variations in the principal direction. Listing 3.8 shows the ISS implementation with PCL. A detection example is shown in Figure 3.10.

Listing 3.8: Implementation of ISS keypoint detector with PCL.

```
void detect_keypoints_iss (
  const pcl::PointCloud<TPoint>::ConstPtr & pCloud,
  const float & pSalientRadius,
  const float & pNonMaxRadius,
  const int & pMinNeighbors,
  const float & pThreshold21,
  const float & pThreshold32,
  const double & pResolution,
  const pcl::search::KdTree<TPoint>::ConstPtr & pKdtree,
  pcl::PointCloud<TPoint>::Ptr & pDstCloud)
{
  pcl::ISSKeypoint3D<TPoint, TPoint> iss;
  iss.setInputCloud(pCloud);
  iss.setSearchMethod(pKdtree);
  iss.setSalientRadius(pSalientRadius * pResolution);
  iss.setNonMaxRadius(pNonMaxRadius * pResolution);
  iss.setMinNeighbors(pMinNeighbors);
  iss.setThreshold21(pThreshold21);
  iss.setThreshold32(pThreshold32);
  iss.compute(*pDstCloud);
}
```

Figure 3.10: Keypoints extracted with ISS in blue (salientRadius = 6, nonMaxRadius = 4, minNeighbors = 5, threshold21 = 0.975, threshold32 = 0.975).

3.2.2.3 Scale Invariant Feature Transform

The Scale Invariant Feature Transform (SIFT) is one of the most popular keypoint detectors since its introduction by Lowe [68]. The method was originally proposed for 2D images but Flint *et al.* [69] presented a 3D version. It works by building a scale space over a density function. The cloud is convolved with a set of Gaussian filters with varying standard deviations, the adjacent convoluted images are subtracted to generate Difference of Gaussian (DoG) clouds. At last, it looks for local maxima of a 3D version of the Hessian determinant over that density function. Listing 3.9 shows its implementation with PCL and Figure 3.11 shows a detection example.

Listing 3.9: SIFT detector implementation with PCL.

```
void detect_keypoints_sift (
  const pcl::PointCloud<TPoint>::ConstPtr & pCloud,
  const float & pMinScale,
  const float & pNrOctaves,
  const float & pNrScalesPerOctave,
  const float & pMinConstrast,
  const pcl::search::KdTree<TPoint>::ConstPtr & pKdtree,
  pcl::PointCloud<TPoint>::Ptr & pDstCloud)
{
  pcl::PointCloud<pcl::PointWithScale>::Ptr keypointsWithScale(
    new pcl::PointCloud<pcl::PointWithScale>);
  pcl::SIFTKeypoint<TPoint, pcl::PointWithScale> sift;
  sift.setSearchMethod(pKdtree);
  sift.setScales(pMinScale, pNrOctaves, pNrScalesPerOctave);
  sift.setMinimumContrast(pMinConstrast);
  sift.setInputCloud(pCloud);
  sift.compute(*keypointsWithScale);
  pcl::copyPointCloud(*keypointsWithScale, *pDstCloud);
}
```

Figure 3.11: Keypoints extracted with SIFT in blue (minScale = 0.005, nrOctaves = 8, scalesPerOctave = 8, minimumContrast = 0.005).

3.2.3 Descriptor Extraction

Once relevant keypoints have been detected, we have to describe their neighborhoods by extracting a descriptor for each keypoint. This step of the pipeline was described in depth during Section 1.3.1. In addition, the theoretical basis of each descriptor was also described in Section 1.3.2. We have integrated a subset of those descriptors, which are implemented in the *Point Cloud Library*, into our pipeline: *Fast Point Feature Histograms* (FPFH), *Spin Images* (SI), *Unique Shape Context* (USC), *3D Shape Context* (3DSC), *Signatures of Oriented Histograms* (SHOT), *Color-SHOT* (CSHOT) and *Rotational Projection Statistics* (ROPS).

3.2.3.1 FPFH

Details regarding the FPFH descriptor are described in Section 1.3.2.2. Listing 3.10 shows the integration of the FPFH descriptor using the PCL implementation.

Listing 3.10: FPFH descriptor implementation with PCL.

```
void compute_fpfh_descriptors (
  const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
  const pcl::PointCloud<TNormal>::ConstPtr & pNormals,
  const pcl::PointCloud<TPoint>::ConstPtr & pKeypoints,
  const pcl::search::KdTree<TPoint>::ConstPtr & pKdTree,
  const float & pSearchRadius,
  pcl::PointCloud<pcl::FPFHSignature33>::Ptr & pDstCloud)
{
  pcl::FPFHEstimation<TPoint, TNormal, pcl::FPFHSignature33> fpfh;
  fpfh.setSearchMethod(pKdTree);
  fpfh.setInputCloud(pKeypoints);
  fpfh.setInputNormals(pNormals);
  fpfh.setRadiusSearch(pSearchRadius);
  fpfh.setSearchSurface(pSrcCloud);
  fpfh.compute(*pDstCloud);
}
```

3.2.3.2 SI

The *Spin Images* descriptor was previously defined in Section 1.3.2.1. Listing 3.11 shows how to extract SI descriptors using the implementation provided by the *Point Cloud Library*. It is important to remark that our input clouds have to be converted to *PointXYZ* clouds to be accepted by the *SpinImageEstimation* class.

Listing 3.11: SI descriptor implementation with PCL.

```
void compute_spinimage_descriptors (
  const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
  const pcl::PointCloud<TNormal>::ConstPtr & pNormals,
  const pcl::PointCloud<TPoint>::ConstPtr & pKeypoints,
  const float & pSearchRadius,
  pcl::PointCloud<SpinImage>::Ptr & pDstCloud)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(
    new pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr keypoints(
    new pcl::PointCloud<pcl::PointXYZ>);
  pcl::copyPointCloud(*pSrcCloud, *cloud);
  pcl::copyPointCloud(*pKeypoints, *keypoints);

  pcl::SpinImageEstimation<pcl::PointXYZ, TNormal, SpinImage> si;
  si.setInputCloud(keypoints);
  si.setInputNormals(pNormals);
  si.setRadiusSearch(pSearchRadius);
  si.setImageWidth(8);
  si.setSearchSurface(cloud);
  si.compute(*pDstCloud);
}
```

3.2.3.3 3DSC

The fundamentals of the *3D Shape Context* descriptor were previously explained in Section 1.3.2.3. Listing 3.12 shows its integration using PCL. Input clouds are expected to use the *PointXYZ* point type.

Listing 3.12: 3DSC descriptor implementation with PCL.

```
void compute_3dsc_descriptors(
  const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
  const pcl::PointCloud<TNormal>::ConstPtr & pNormals,
  const pcl::PointCloud<TPoint>::ConstPtr & pKeypoints,
  const float & pSearchRadius,
  pcl::PointCloud<pcl::ShapeContext1980>::Ptr & pDstCloud)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(
    new pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr keypoints(
    new pcl::PointCloud<pcl::PointXYZ>);
  pcl::copyPointCloud(*pSrcCloud, *cloud);
  pcl::copyPointCloud(*pKeypoints, *keypoints);

  pcl::ShapeContext3DEstimation<pcl::PointXYZ, TNormal,
    pcl::ShapeContext1980> sc3d;
  sc3d.setInputCloud(keypoints);
  sc3d.setInputNormals(pNormals);
  sc3d.setRadiusSearch(pSearchRadius);
  sc3d.setMinimalRadius(pSearchRadius / 10.0);
  sc3d.setPointDensityRadius(pSearchRadius / 5.0);
  sc3d.setSearchSurface(cloud);
  sc3d.compute(*pDstCloud);
}
```

3.2.3.4 USC

The functioning of *Unique Shape Context* descriptor was depicted in Section 1.3.2.4. Listing 3.13 shows the code needed to integrate that descriptor into our pipeline using PCL. Also notice that point clouds are expected to use the *PointXYZ* type.

Listing 3.13: USC descriptor implementation with PCL.

```
void PointCloudOperations::compute_usc_descriptors(
  const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
  const pcl::PointCloud<TNormal>::ConstPtr & pNormals,
  const pcl::PointCloud<TPoint>::ConstPtr & pKeypoints,
  const float & pSearchRadius,
  pcl::PointCloud<pcl::UniqueShapeContext1960>::Ptr & pDstCloud)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(
    new pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr keypoints(
    new pcl::PointCloud<pcl::PointXYZ>);
  pcl::copyPointCloud(*pSrcCloud, *cloud);
  pcl::copyPointCloud(*pKeypoints, *keypoints);
  pcl::UniqueShapeContext<pcl::PointXYZ, pcl::UniqueShapeContext1960,
  pcl::ReferenceFrame> usc;
  usc.setInputCloud(keypoints);
  usc.setRadiusSearch(pSearchRadius);
  usc.setMinimalRadius(pSearchRadius / 10.0);
  usc.setPointDensityRadius(pSearchRadius / 5.0);
  usc.setLocalRadius(pSearchRadius);
  usc.setSearchSurface(cloud);
  usc.compute(*pDstCloud);
}
```

It is important to remark that, in both 3DSC and USC implementations, we have chosen a set of default values for the minimal radius and point density radius parameters. In this case, the minimal and point density radiuses have been set to a tenth and a fifth part of the search radius respectively. These values were selected based on experiments presented in the original paper or in the PCL implementation.

3.2.3.5 SHOT

The *Signatures of Histograms* descriptor was previously described during Section 1.3.2.6. Listing 3.14 shows the integration of the SHOT descriptor using the implementation provided by the PCL. In this case, no special parameters have to be set. We can use a *k-d tree* as a search method. In addition, PCL provides a multi-core implementation named *SHOTEstimationOMP*. We show the basic one that will be used during the experimentation for a fair comparison in terms of efficiency.

Listing 3.14: SHOT descriptor implementation with PCL.

```
void compute_shot_descriptors (
  const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
  const pcl::PointCloud<TNormal>::ConstPtr & pNormals,
  const pcl::PointCloud<TPoint>::ConstPtr & pKeypoints,
  const pcl::search::KdTree<TPoint>::ConstPtr & pKdTree,
  const float & pSearchRadius,
  pcl::PointCloud<pcl::SHOT352>::Ptr & pDstCloud)
{
  pcl::SHOTEstimation<TPoint, TNormal, pcl::SHOT352> shot;

  shot.setSearchMethod(pKdTree);
  shot.setInputCloud(pKeypoints);
  shot.setInputNormals(pNormals);
  shot.setRadiusSearch(pSearchRadius);
  shot.setSearchSurface(pSrcCloud);
  shot.compute(*pDstCloud);
}
```

3.2.3.6 CSHOT

As we have previously mentioned in Section 1.3.4, *Color-SHOT* is an extension of SHOT which adds color information to the description. Listing 3.15 shows the usage of the PCL interface *SHOTColorEstimation* for integrating the CSHOT descriptor.

Listing 3.15: CSHOT descriptor implementation with PCL.

```
void compute_cshot_descriptors (
  const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
  const pcl::PointCloud<TNormal>::ConstPtr & pNormals,
  const pcl::PointCloud<TPoint>::ConstPtr & pKeypoints,
  const float & pSearchRadius,
  pcl::PointCloud<pcl::SHOT1344>::Ptr & pDstCloud)
{
  pcl::SHOTColorEstimation<TPoint, TNormal, pcl::SHOT1344> shot;

  shot.setSearchMethod(pcl::search::KdTree<TPoint>::Ptr(
    new pcl::search::KdTree<TPoint>));
  shot.setInputCloud(pKeypoints);
  shot.setInputNormals(pNormals);
  shot.setRadiusSearch(pSearchRadius);
  shot.setSearchSurface(pSrcCloud);
  shot.compute(*pDstCloud);
}
```

3.2.3.7 RoPS

The *Rotational Projection Statistics* theoretical primer was exposed in Section 1.3.2.7. Listing 3.16 shows the integration of the ROPS descriptor using the recently added implementation to the *Point Cloud Library*.

As we can observe, *ROPSEstimation* works with *PointXYZ* point clouds and requires surface information in form of polygons so we used an efficient triangulation algorithm named *Greedy Projection Triangulation* [70].

Listing 3.16: ROPS descriptor implementation with PCL.

```
void compute_rops_descriptors (
  const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
  const pcl::PointCloud<TNormal>::ConstPtr & pNormals,
  const pcl::PointCloud<TPoint>::ConstPtr & pKeypoints,
  const float & pSearchRadius,
  pcl::PointCloud<ROPS135>::Ptr & pDstCloud)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(
    new pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr keypoints(
    new pcl::PointCloud<pcl::PointXYZ>);

  pcl::copyPointCloud(*pSrcCloud, *cloud);
  pcl::copyPointCloud(*pKeypoints, *keypoints);

  pcl::PointCloud<pcl::PointNormal>::Ptr cloudWithNormals(
    new pcl::PointCloud<pcl::PointNormal>);
  pcl::concatenateFields(*cloud, *pNormals, *cloudWithNormals);
  pcl::search::KdTree<pcl::PointNormal>::Ptr kdtreeNormals(
    new pcl::search::KdTree<pcl::PointNormal>);
  kdtreeNormals->setInputCloud(cloudWithNormals);

  pcl::GreedyProjectionTriangulation<pcl::PointNormal> gp3;
  pcl::PolygonMesh triangles;

  gp3.setSearchRadius(0.025f);
  gp3.setMu(2.5);
  gp3.setMaximumNearestNeighbors(100);
  gp3.setMaximumSurfaceAngle(M_PI / 4);
  gp3.setMinimumAngle(M_PI / 18);
  gp3.setMaximumAngle(2 * M_PI / 3);
  gp3.setNormalConsistency(false);

  gp3.setInputCloud(cloudWithNormals);
  gp3.setSearchMethod(kdtreeNormals);
  gp3.reconstruct(triangles);

  pcl::ROPSEstimation<pcl::PointXYZ, ROPS135> rops;
  rops.setSearchMethod(pcl::search::KdTree<pcl::PointXYZ>::Ptr(
    new pcl::search::KdTree<pcl::PointXYZ>));
  rops.setInputCloud(keypoints);
  rops.setSearchSurface(cloud);
  rops.setRadiusSearch(pSearchRadius);
  rops.setNumberOfPartitionBins(5);
  rops.setNumberOfRotations(3);
  rops.setSupportRadius(pSearchRadius);
  rops.setTriangles(triangles.polygons);
  rops.compute(*pDstCloud);
}
```

We have used the default set of parameters recommended by the authors in the PCL documentation for the number of rotations (3) and partition bins (5).

3.2.4 Feature Matching

Once the local descriptors have been computed for each keypoint in the cloud, we have to find correspondences between the descriptors extracted from the stored models and the ones computed over the scene. The correspondences are determined in a process named *feature matching*. Every descriptor in the scene will be matched against the descriptors of each model in the database to account for the presence of multiple instances of the same model.

The matching process consists of finding the nearest neighbor to a certain descriptor in terms of Euclidean distance between them. Usually, descriptors are encoded so that distance operations can be easily computed. In addition, a maximum distance value can be enforced so that matchings are rejected if the distance between descriptors exceeds a predefined threshold.

For this purpose, efficient search structures like k-d trees are often used to perform nearest neighbor search operations. The *Point Cloud Library* implements a generic 3D spatial locator using k-d trees and the FLANN (*Fast Library for Approximate Nearest Neighbor*) library [71] for extreme efficiency. Listing 3.17 shows the implementation of the matching process using PCL and the *KdTreeFLANN* locator.

Listing 3.17: Finding correspondences with PCL.

```
1  void find_correspondences (
2    const pcl::PointCloud<TDescriptor>::ConstPtr & pModelDescriptors,
3    const pcl::PointCloud<TDescriptor>::ConstPtr & pSceneDescriptors,
4    const float & pThreshold,
5    pcl::CorrespondencesPtr & pCorrespondences)
6  {
7    pcl::KdTreeFLANN<TDescriptor> kdtree;
8    kdtree.setInputCloud(pModelDescriptors);
9
10   for (size_t i = 0; i < pSceneDescriptors->size(); ++i) {
11     std::vector<int> neighborsInd(1);
12     std::vector<float> neighborsSqrDists(1);
13
14     int neighborsFound = kdtree.nearestKSearch(
15       pSceneDescriptors->at(i), 1, neighborsInd, neighborsSqrDists);
16
17     if (neighborsFound == 1 && neighborsSqrDists[0] < pThreshold)
18     {
19       pcl::Correspondence correspondence(neighborsInd[0],
20         static_cast<int>(i), neighborsSqrDists[0]);
21       pCorrespondences->push_back(correspondence);
22     }
23   }
24 }
```

3.2.5 Correspondence Grouping

Once the matching phase has finished, we have a list of correspondences between keypoints in the current scene and the ones detected in the models of the database. Only with those plain correspondences we are not yet able to determine if an object is present in the scene or not. In order to determine that a certain object is present in the scene we need at least three correspondences between a model's keypoints and the scene ones in order to retrieve the hypothesis, i.e., 6 Degree-of-Freedom (DoF) pose of the model in the scene. However, finding a correspondence or even three does not necessarily mean that the object is present in the scene. For example, we can have two keypoints $k1_m$ and $k2_m$ which are quite close in a certain model, then we find two descriptors $d1_s$ and $d2_s$ which are extremely far away in the scene but are similar to the ones extracted from the model $d1_m$ and $d2_m$ so that two correspondences are established $d1_s - d1_m$ and $d2_s - d2_m$. Since we are only taking rigid transformations into account, it is impossible for both correspondences to belong to the same instance.

In order to deal with this situation, the correspondence grouping step is introduced in the pipeline. On a high level, this process groups correspondences that are geometrically consistent, for a given model, into clusters and discards the ones which are not consistent. The criteria for determining that consistency varies depending on the method but they are usually based on allowing rigid transformations (rotations and translations) to a certain extent.

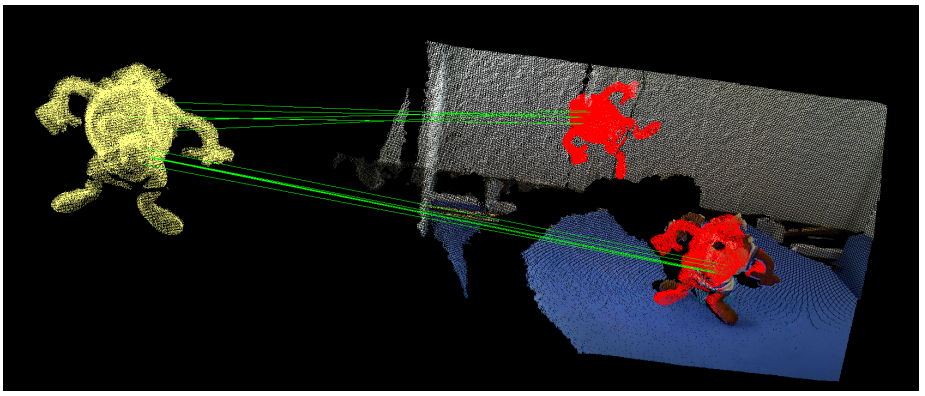


Figure 3.12: Example of correspondence grouping. The yellow object is the model and the red ones are the transformed hypotheses. As we can see, two clusters of correspondences are grouped based on their geometric consistency.

The most popular and simple method for performing correspondence grouping is the so called *Geometric Consistency Grouping* (GCG) implemented in PCL and based on the proposal presented by Chen and Bhanu [72]. It is a geometric consistency clustering algorithm which enforces simple geometric constraints between pairs of correspondences. The algorithm just iterates over all correspondences not yet grouped, adding them to the current cluster if they are consistent or creating a new one for them if not. Listing 3.18 shows the use of the GCG algorithm with PCL.

Listing 3.18: Correspondence grouping using GCG with PCL.

```
void cluster_correspondences_gcg (
  const pcl::PointCloud<TPoint>::ConstPtr & pModel,
  const pcl::PointCloud<TPoint>::ConstPtr & pModelKeypoints,
  const pcl::PointCloud<TPoint>::ConstPtr & pScene,
  const pcl::PointCloud<TPoint>::ConstPtr & pSceneKeypoints,
  const pcl::CorrespondencesPtr & pModelSceneCorrespondences,
  const float & pClusterSize,
  const float & pClusterThreshold,
  std::vector<Eigen::Matrix4f, Eigen::aligned_allocator
    <Eigen::Matrix4f>> & pRotoTranslations,
  std::vector<pcl::Correspondences> & pClusteredCorrespondences)
{
  pcl::GeometricConsistencyGrouping<pcl::PointXYZ, pcl::PointXYZ> gcg;

  pcl::PointCloud<pcl::PointXYZ>::Ptr modelKeypoints(
    new pcl::PointCloud<pcl::PointXYZ>);

  pcl::copyPointCloud(*pModelKeypoints, *modelKeypoints);

  pcl::PointCloud<pcl::PointXYZ>::Ptr sceneKeypoints(
    new pcl::PointCloud<pcl::PointXYZ>);

  pcl::copyPointCloud(*pSceneKeypoints, *sceneKeypoints);

  gcg.setGCSize(pClusterSize);
  gcg.setGCThreshold(pClusterThreshold);
  gcg.setInputCloud(modelKeypoints);
  gcg.setSceneCloud(sceneKeypoints);
  gcg.setModelSceneCorrespondences(pModelSceneCorrespondences);
  gcg.recognize(pRotoTranslations, pClusteredCorrespondences);
}
```

A set of parameters can be customized to alter the end result of the algorithm. For example, a minimum number of correspondences per cluster might be enforced using a cluster threshold. In addition, the resolution of the consensus set used to cluster correspondences can be modified to make the algorithm more or less sensitive to geometric consistency constraints. Figure 3.12 shows an example of correspondence grouping using GCG with a minimum of three correspondences per cluster and a cluster set resolution of 8 times the model cloud resolution.

3.2.6 Pose Estimation and Alignment

After clustering the correspondences, we need to estimate the pose of the model in the scene for each detected instance. Some correspondence grouping methods as the aforementioned GCG provide the transformation for each correspondence cluster (rotation and translation) so that this step is unnecessary. In our case, we use the rototranslations generated by the correspondence grouping stage to transform the corresponding model to an estimated pose in the scene which will hopefully perform a coarse alignment with model instance in the scene. This 6DoF estimated pose is usually refined using the *Iterative Closest Point* (ICP) algorithm [73] [74].

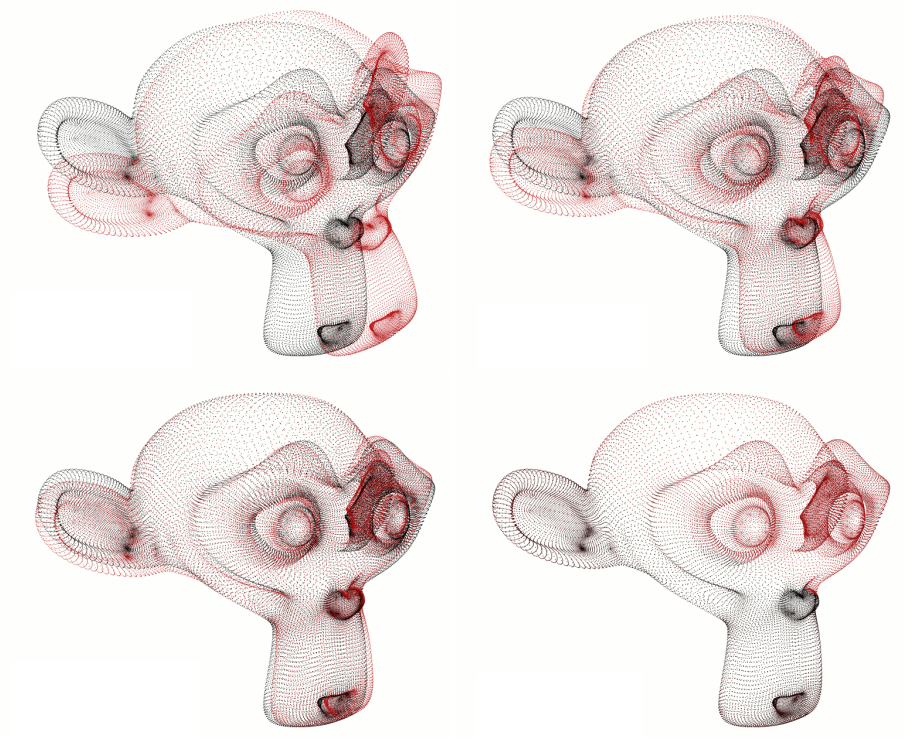


Figure 3.13: Cloud alignment refinement with ICP algorithm. The original point cloud in gray and the ICP aligned cloud in red. Initial positions (top left), four ICP iterations after (top right), eight ICP iterations after (bottom left) and twelve iterations after (bottom right). (Figure reproduced from *Pointclouds.org*).

The ICP is a widely used algorithm for fine geometric alignment of 3D point clouds when a coarse initial estimation of the pose is known. The algorithm

starts with two clouds and an initial guess of their relative rigid transformation which is iteratively refined by repeatedly generating pairs of corresponding points using a closeness criteria and minimizing a distance error metric. Many variants of the algorithm have been proposed whose goal is to increase the efficiency or precision of the method, the seminal work of Rusinkiewicz and Levoy [75] surveys and carries out a detailed comparison of the most relevant ones. The *Point Cloud Library* provides various implementations, we will use the basic one as shown in Listing 3.19.

Listing 3.19: Cloud alignment using ICP with PCL.

```
1  void PointCloudOperations::register_cloud (
2    const pcl::PointCloud<TPoint>::ConstPtr & pModel,
3    const pcl::PointCloud<TPoint>::ConstPtr & pScene,
4    const int & pMaxIterations,
5    const float & pMaxCorrespondenceDistance,
6    pcl::PointCloud<TPoint>::Ptr & pDstCloud)
7  {
8    pcl::IterativeClosestPoint<TPoint, TPoint> icp;
9    icp.setMaximumIterations(pMaxIterations);
10   icp.setMaxCorrespondenceDistance(pMaxCorrespondenceDistance);
11   icp.setInputCloud(pModel);
12   icp.setInputTarget(pScene);
13   icp.align(*pDstCloud);
14 }
```

The ICP takes as input two point clouds and the initial estimation of the transformation with a certain stop criteria (error threshold or maximum number of iterations) and it outputs the refined transformation and even the transformed cloud. The main steps of the algorithm are:

1. Establish correspondences between points using the nearest neighbor criteria.
2. Estimate the transformation by minimizing a mean square error cost function.
3. Apply and accumulate the transformation to align the second cloud.
4. Iterate again to refine if stop criteria is not met.

If the correspondence grouping algorithm does not provide the full pose, it is possible to compute it manually using methods like *RANdom Sample Consensus* to obtain the rigid transformation which best aligns the model to the scene instance using the previously clustered correspondences.

3.2.7 Hypothesis Verification

In the end, the local recognition pipeline has generated a set of object or model hypotheses $H = \{h_1, h_2, ..., h_n\}$ where a certain hypothesis h is composed by a model M and the rigid transformation (R, t) applied to align that instance in the scene $h_i = \{M_i, R_i, t_i\}$. However, a difficult problem arises: several hypotheses are false positives and they have to be discarded without compromising the recall of the system. This is the goal of the hypothesis verification stage.

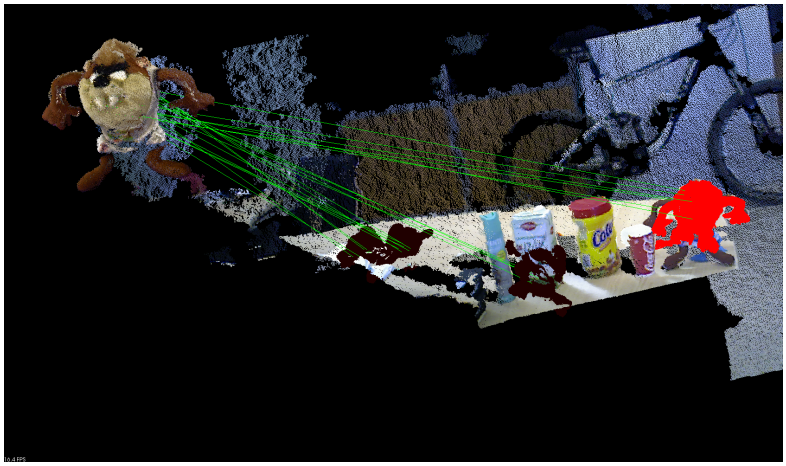


Figure 3.14: Recognition hypotheses verified with Global Hypothesis Verification; rejected false positive hypotheses in dark red, accepted true positive hypothesis in bright red. Green lines represent model-scene correspondences.

Typical hypothesis verification methods enforce certain geometric cues in a sequential manner, i.e., one hypothesis at a time. Aldoma *et al.* [23] proposed a method for simultaneous geometric verification of all object hypotheses. The *Global Hypothesis Verification* (GHV) algorithm considers two possible states $x_i = \{0, 1\}$ of a single hypothesis h_i since it can be inactive or active. Given a set of hypotheses $H = \{h_1, h_2, ..., h_n\}$ a possible solution vector is defined as $X = \{x_1, x_2, ..., x_n\}$. A global cost function ζ is defined as $\zeta(X) : B^n \rightarrow R, X = \{x_1, ..., x_n\}, x_i \in B = \{0, 1\}$. This function estimates how good the current solution X is by considering the whole set of hypotheses as a global scene model instead of considering each model hypothesis separately.

The cost function integrates four geometrical cues as shown on Equation 3.7: Ω_x is the number of scene inliers, Λ_x represents the number of points explained by multiple hypotheses, γ_x is the clutter for h_i and $|\phi_{h_i}|$ is the number

of outliers for h_i.

$$\zeta(X) = \sum_{p \in S} (\Lambda_x(p) + \gamma_x(p) - \Omega_x(p)) + \lambda \sum_{i=1}^{n} |\phi_{h_i}| x_i \tag{3.7}$$

This optimization problem is typically solved using *Simulated Annealing* or other metaheuristics. The target is to maximize the number of explained scene points and minimize the number of model outliers, multiple explained scene points and unexplained scene points close to active hypotheses.

The Point Cloud Library contains an implementation of the Global Hypotheses Verification method. Listing 3.20 shows the integration of the GHV method in our pipeline using PCL. This method takes a scene cloud and a set of model hypotheses as inputs, as well as some extra parameters for tuning the algorithm. The output consists of a boolean vector which represents the optimal solution X which minimizes the aforementioned cost function. This vector can be seen as a mask which activates (true) or deactivates (false) each model hypothesis.

Listing 3.20: Implementation of GHV with PCL.

```
void verify_hypotheses (
  const pcl::PointCloud<TPoint>::Ptr & pScene,
  std::vector<pcl::PointCloud<TPoint>::ConstPtr> & pModels,
  const float & pInlierThreshold,
  const float & pOcclusionThreshold,
  const float & pRadiusClutter,
  const float & pRegularizer,
  const float & pClutterRegularizer,
  const bool & pDetectClutter,
  const float & pRadiusNormals,
  std::vector<bool> & pHypothesesMask)
{
  pcl::GlobalHypothesesVerification<TPoint, TPoint> ghv;
  ghv.setSceneCloud(pScene);
  ghv.addModels(pModels, true);
  ghv.setInlierThreshold(pInlierThreshold);
  ghv.setOcclusionThreshold(pOcclusionThreshold);
  ghv.setRegularizer(pRegularizer);
  ghv.setRadiusClutter(pRadiusClutter);
  ghv.setClutterRegularizer(pClutterRegularizer);
  ghv.setDetectClutter(pDetectClutter);
  ghv.setRadiusNormals(pRadiusNormals);

  ghv.verify();
  ghv.getMask(pHypothesesMask);
}
```

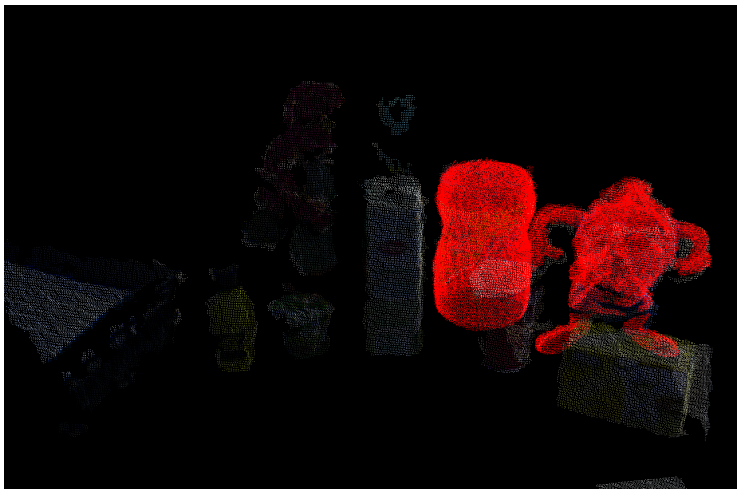


Figure 3.15: Global Hypotheses Verification example with multiple accepted hypotheses of different object models present in the scene.

3.2.8 Offline training

When a scene point cloud is provided to the system, it has to go through the whole pipeline since it is new information which is not previously available. However, the 3D models which will be recognized are available prior to the recognition process. In this regard, it is not needed to recompute model information each time a new scene has to be processed.

Instead of computing model information for each scene, we will generate all of it offline and then load it once when the system is started. This stage is commonly referred as *offline training*.

For each model we want to recognize, we computed resolution, estimated normals, detected keypoints, and extracted descriptors. The normal estimation method, keypoint detector and descriptor extractor are the same ones that will be used later to process scene point clouds. The reason for using the same methods is obvious: ensure representation consistency.

The precomputed information is stored in a set of configuration files with proper file formats to optimize its loading process when the system starts (XML files for parameter configuration and PCD ones for point cloud information like keypoints, normals and descriptors).

3.3 Descriptors and pipeline performance study

In order to validate the CPU implementation and assess the performance and precision of the available feature descriptors, we carried out a set of experiments which consisted of recognizing a particular object in a series of heterogeneous test scenes. The main goal of the experimentation was to find out the best descriptor for our purpose in terms of efficiency and precision.

In addition, the main goal of this project is to implement and deploy a real time object recognition system. In this sense, the experimentation allowed us to determine the computational cost of the different phases of the system. Taking it into account, we will focus on parallelizing and accelerating those stages with higher cost by developing an optimized CPU implementation and a GPU one.

In this section we will describe the experimentation carried out, from the model reconstruction and scene capturing to the methodology and results.

3.3.1 Model reconstruction

For this first round of experiments, we tested our system with two different objects which appeared in all the test scenes. The first step of this experimentation consisted of reconstructing full 3D models from the real-life objects (see Appendix A). For this purpose, we used a rotating platform to capture 64 partial views of the objects from a determined angle (approximately 45° respect to the platform plane) and position (roughly a distance of 90 cm from the camera sensor and the center of the platform). Figure 3.16 shows some of the RGB images captured from those points of view. For these initial tests we used the Primesense Carmine sensor.

Given the 64 partial views, we developed a reconstruction application to merge and align all generated point clouds with color information. This application used the known angular step information (64 views imply a step of 5.625 ° between each view) to roughly align each view to the previous one. The ICP algorithm is also used to refine that approximate transformation. Figure 3.17 shows the fully reconstructed 3D model. In addition, statistical outlier removal and euclidean cluster extraction operations were performed to segment the object and reduce noise levels.

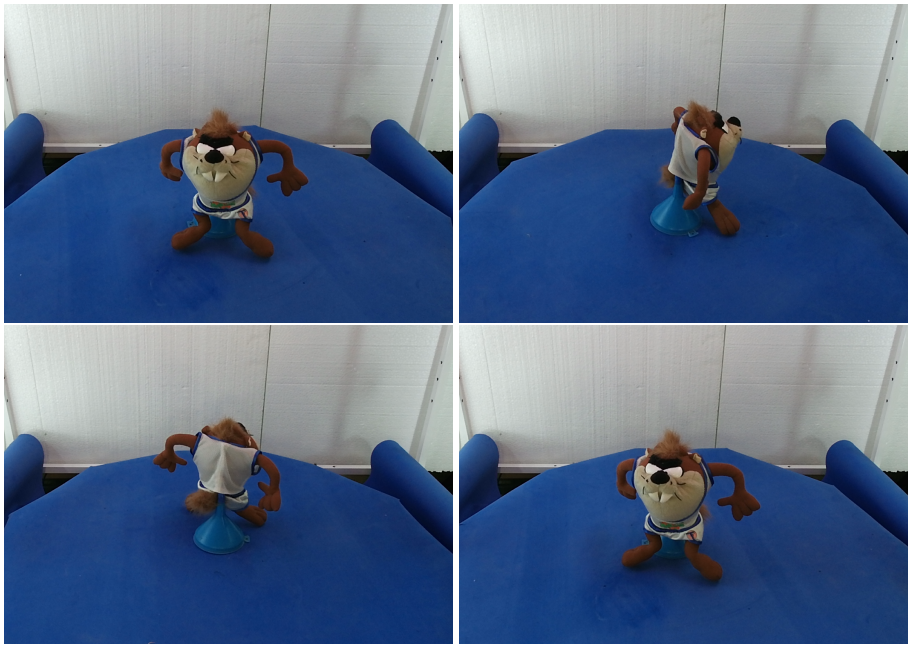


Figure 3.16: Tasmanian object captured from different points of view using a rotating platform and the Primesense Carmine sensor. 64 partial views were captured in total, with an angular step of 5.625 degrees. View 0 (top left), view 43 (top right), view 23 (bottom left) and view 63 (bottom right).

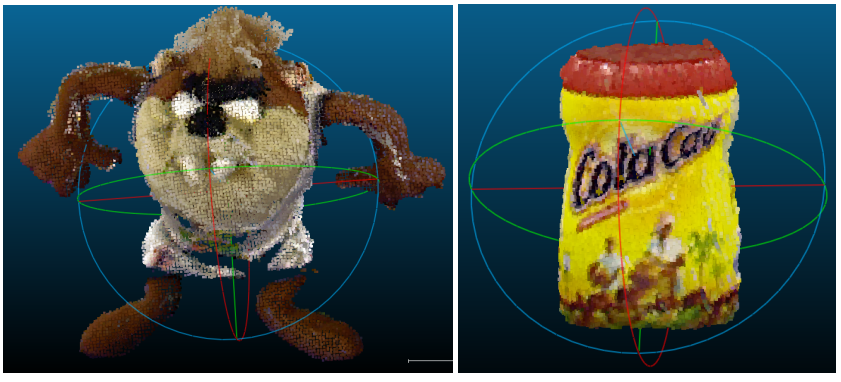


Figure 3.17: Reconstructed *tasmanian* model point cloud (left) and *Colacao* model point cloud (right). 64 partial views of the objects were used to fully reconstruct them. Normals estimated using Meshlab normal estimation feature.

3.3.2 Evaluation scenes

A total of nine test scenes were captured using the same Primesense Carmine which was used to capture the partial views of the object. The purpose of these scenes is to generate a heterogeneous set of possible scenarios with varying number of objects, poses, clutter and occlusions. Figure 3.18 shows all the scenes. First, three scenes with three objects, including the reconstructed ones, were captured at different distances. Then, two captures were taken by grouping those objects to produce clutter and occlusions and even turning the model backwards. Three more captures were taken replicating this procedure, this time with five different objects. A last capture was taken with nine objects at intermediate distance with occlusions.

Figure 3.18: Heterogeneous test scenes captured with Primesense Carmine acquistion device. First row: Scene01, Scene 02 and Scene03. Second row: Scene04, Scene05 and Scene06. Third row: Scene07, Scene08 and Scene09.

3.3.3 Performance study methodology

We have tested seven feature descriptors which are implemented by the *Point Cloud Library*: Rotational Projection Statistics (ROPS), Unique Shape Context (USC), Fast Point Feature Histograms (FPFH), Unique Signatures of Histograms (SHOT), 3D Shape Context (3DSC), and Color-SHOT (CSHOT).

Each descriptor has been used for recognizing the reconstructed models in the test scenes. For that, they have been integrated in our pipeline while the rest of it remained the same for all the descriptors. We used *Uniform Sampling* to extract keypoints, *Geometric Consistency Grouping* for correspondence grouping and *Global Hypothesis Verification* for rejecting false matches. Pose refinement with *Iterative Closest Point* was disabled in order to assess the quality of the pose estimation using only the established correspondences which depend on the quality of the descriptor.

A confusion matrix or contingency table was created for each descriptor, representing the four possible outcomes of the predictions or hypotheses produced by the system: if the hypothesis properly identifies the object in the scene, then it is called a *true positive*(TP); however, if the hypothesis does not correspond to the proper object in the scene then it is said to be a *false positive* (FP). A *true negative* (TN) occurs when no hypothesis is produced for objects in the scene which do not correspond to the model and a *false negative* (FN) is when the model is present in the scene but no hypothesis recognizes it. The contingency table helps us to derive several evaluation metrics to place each descriptor in a *Receiver Operating Characteristic* (ROC) space.

Figure 3.19: Validation scene (left) and recognition results with SHOT (right) one true positive (the recognized tasmanian model) and eight true negatives.

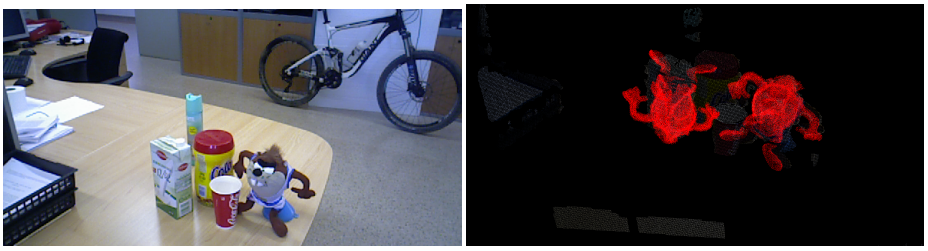


Figure 3.20: Validation scene (left) and recognition results with SHOT (right) one true positive (the recognized tasmanian model), one false positive (the wrongly recognized tasmanian in the milk) and seven true negatives.

Figure 3.21: Validation scene (left) and recognition results with CSHOT (right) one false negative (the unrecognized tasmanian model) and four true negatives.

A ROC space is defined by two metrics: sensitivity (y-axis) and 1-specificity (x-axis). The sensitivity is represented by the *true positive rate* (TPR) and the *false positive rate* (FPR) is equal to (1-specificity). Each instance of a confusion matrix represents a point in the ROC space which depicts relative trade-offs between false positives (costs) and true positives (benefits) of a prediction system. We also computed the F1 score for each descriptor. The accuracy of a system (ACC) can be biased by a lack of equilibrium between positive and negative instances. The F1 scores applies a harmonic mean of precision and recall for a better measurement of a test's accuracy.

A default set of parameters was fixed throughout the entire experimentation in order not to bias the results. However, variations were allowed in the matching threshold since distances between descriptors are not consistent across different types, i.e., SHOT distances are normalized while FPFH ones are not. Some specific tuning was performed as well in the hypothesis verification threshold.

3.3.4 Results

The results of the experimentation will be presented from two points of view: precision or accuracy, and efficiency. Section 3.3.4.1 is devoted to the former and Section 3.3.4.2 to the latter one.

3.3.4.1 Precision

Table 3.1 shows the results obtained by the different descriptors during the experimentation in terms of true/false positives/negatives rates and the subsequent ROC metrics. Figure 3.22 shows those results placed in a ROC space.

Table 3.1: Accumulated experimentation rates (TP, FP, TN and FN) throughout all scenes for the selected descriptors including ROC space metrics and F1 score.

	Rates				ROC Metrics			
Descriptor	**TP**	**FP**	**TN**	**FN**	**TPR**	**FPR**	**ACC**	**F1 SCORE**
CSHOT	14	0	60	4	0.780	0.000	0.950	0.875
SHOT	14	2	58	4	0.780	0.030	0.920	0.824
ROPS	14	2	58	4	0.780	0.030	0.920	0.824
3DSC	10	3	57	8	0.560	0.050	0.860	0.645
USC	10	3	57	8	0.560	0.050	0.860	0.645
FPFH	7	7	53	11	0.390	0.120	0.770	0.438

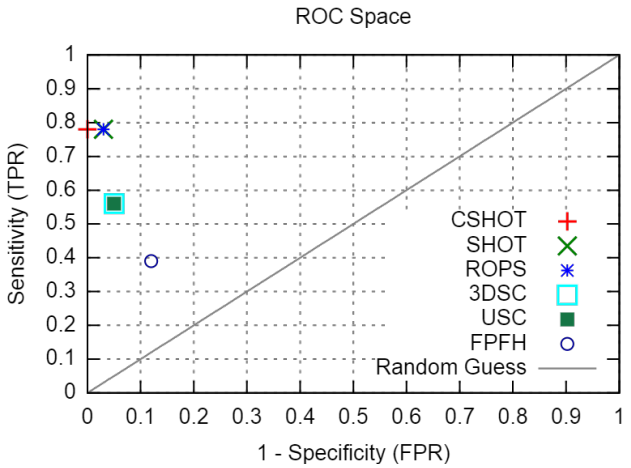


Figure 3.22: Descriptor metrics in the ROC space. Top left represents the perfect classifier with full sensitivity (all true positives) and specificity (no false positives).

3.3.4.2 Efficiency

Another important aspect of this experimentation is the execution time needed by each descriptor to extract descriptors in the whole scene. Table 3.2 shows the execution times for each scene and the mean for all of them. Figure 3.23 represents this data for a clearer visualization.

Table 3.2: Experimentation results (descriptor extraction times for a whole scene cloud) for the selected descriptors and each validation scene.

	Descriptor extraction times (s) for the scenes									
Desc.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Mean (s)**
SHOT	0.612	1.093	1.340	1.141	1.174	1.161	1.663	1.377	1.883	1.271
CSHOT	0.715	1.310	1.579	1.325	1.318	1.262	1.945	1.649	2.227	1.481
FPFH	2.424	4.929	8.196	6.879	7.009	4.365	8.736	6.574	8.398	6.390
3DSC	3.075	5.823	8.318	7.033	7.035	5.481	9.624	7.456	9.941	7.087
USC	3.392	6.275	8.865	7.505	7.585	5.945	10.406	8.117	10.776	7.652
ROPS	4.230	7.369	8.565	7.387	7.408	7.140	10.622	9.259	12.198	8.242

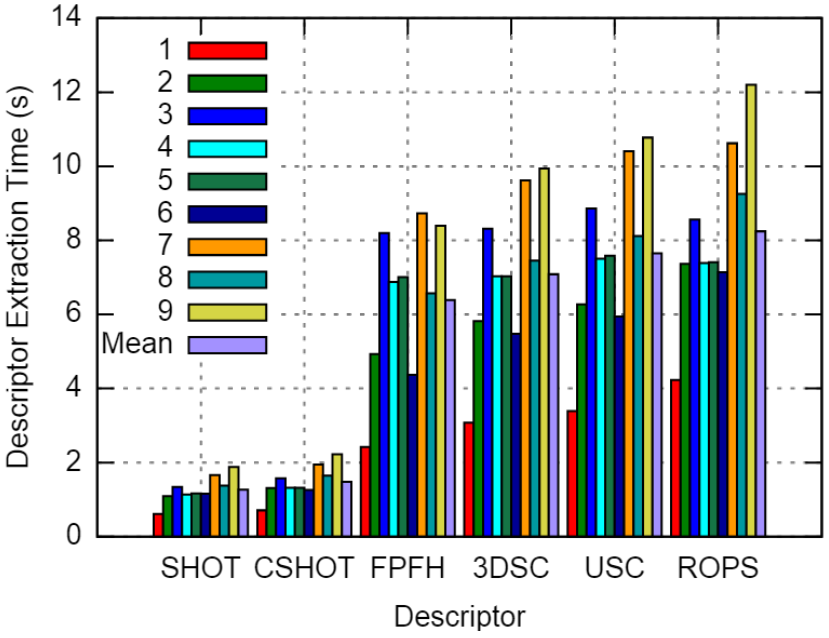


Figure 3.23: Extraction times for each descriptor and test scene.

We also computed the execution times for each phase of the pipeline using the SHOT descriptor for reference. Table 3.3 shows those execution times taking into account the normal estimation, OMPS, OCCS, DS (downsampling with uniform sampling), SHOT, matching (CORR), GCG and GHV. Figure 3.24

shows the mean runtime distribution of the pipeline executed for all scenes using SHOT with a single model.

Table 3.3: Experimentation results (execution times for the different pipeline's phases) for the SHOT descriptor and each test scene recognizing a single model.

	Execution times (s) for the different scenes									
Phase	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Mean (s)**
NE (I)	0.968	0.928	0.940	0.932	0.929	0.932	0.939	0.987	0.976	0.948
OMPS	0.162	0.158	0.163	0.172	0.165	0.159	0.167	0.168	0.165	0.164
OCCS	0.084	0.075	0.069	0.075	0.067	0.066	0.071	0.068	0.072	0.072
NE (II)	0.059	0.099	0.119	0.102	0.099	0.092	0.139	0.125	0.165	0.111
DS	0.005	0.009	0.011	0.009	0.010	0.009	0.013	0.012	0.016	0.010
SHOT	0.615	1.092	1.336	1.141	1.160	1.020	1.638	1.379	1.847	1.248
CORR	10.584	15.184	13.166	11.654	12.092	14.859	17.303	17.443	24.593	15.209
GCG	2.088	2.279	2.408	2.316	2.342	2.287	2.578	2.453	2.642	2.377
GHV	1.119	1.311	1.663	1.251	1.242	1.057	2.060	2.732	2.163	1.622
All	15.684	21.135	19.875	17.652	18.106	20.481	24.908	25.367	32.639	21.761

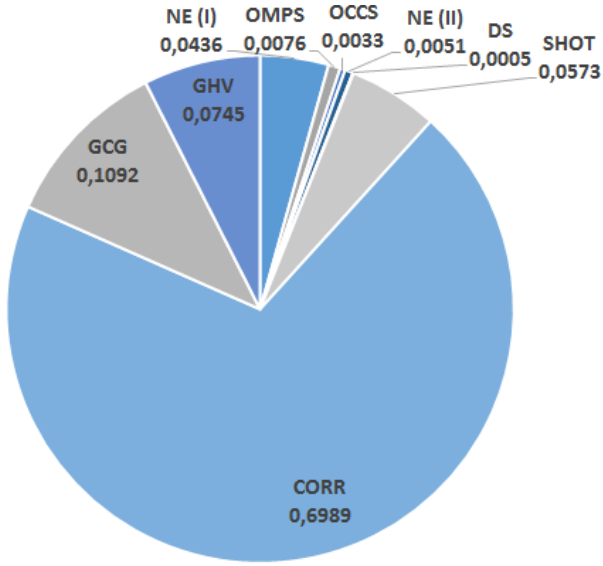


Figure 3.24: Mean execution times percentages for each phase using SHOT.

3.3.5 Discussion

Once the results have been shown off, we will analyze them from two points of view: accuracy (see Table 3.1) and efficiency (see Table 3.2).

On the one hand, CSHOT is the most accurate descriptor producing no

false positives at all. SHOT and ROPS offer a similar TPR but their specificity falls below CSHOT's one. 3DSC and USC are quite similar but, together with FPFH, they are far behind CSHOT and even SHOT/ROPS in terms of accuracy.

On the other hand, SHOT is the fastest descriptor. It achieved the best descriptor extraction execution times consistently throughout all the scenes. CSHOT shows slightly worse execution times, this is normal considering the additional amount of information that it encodes. ROPS is the slowest descriptor on average although 3DSC/USC are sometimes slower. FPFH is slightly faster than 3DSC/USC/ROPS. This difference is significant in scenes with high point count, i.e., scenes with a large number of objects. However, it is approximately six times slower than CSHOT on average.

Taking all of this into account, it is obvious that SHOT/CSHOT are the best descriptors for our purposes. Further experimentation will be required to choose one of them. CSHOT is more accurate but SHOT is faster so we will have to choose one according to the limits of the real-time implementation.

We will also analyze the efficiency of the pipeline (see Table 3.3) in order to determine the phases with higher computational cost so that they become possible targets for parallelization or rethinking to decrease their execution time.

The phase with the highest execution time is the matching one. This stage takes approximately a 70% of the total execution time. The main reason for this inefficiency is the use of non-optimized data structures for high dimensional data such as certain feature descriptors. In this sense, increasing the efficiency of this phase will dramatically reduce the pipeline's latency. Other phases which might as well be parallelized or redesigned due to their high execution times are Geometric Consistency Grouping (10% of the pipeline's execution time), Global Hypothesis Verification (7%) and the descriptor extraction (6%) stages.

3.4 Jetson TK1 sequential experimentation

In the previous section we have shown the execution times of our object recognition system running on a desktop *personal computer* with a high-end processor. In this section we are going to run the same experiments on the Jetson TK1 to determine the performance of our system in that platform and also compare it with the PC. We expect a performance loss due to the worse specifications of the Jetson TK1.

3.4.1 Results

Table 3.4 shows the experimentation results (execution times for the different pipeline's phases) using the SHOT descriptor for each test scene recognizing a single model.

Table 3.4: SHOT-based pipeline execution times on Jetson TK1.

	Execution times (s) for the different scenes									
Phase	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Mean (s)**
NE (I)	2.030	2.090	2.070	2.200	2.170	2.200	2.020	2.200	2.180	2.129
OMPS	0.220	0.230	0.220	0.259	0.241	0.220	0.230	0.230	0.230	0.231
OCCS	0.190	0.190	0.180	0.210	0.160	0.190	0.170	0.180	0.200	0.186
NE (II)	0.130	0.240	0.300	0.240	0.260	0.230	0.320	0.260	0.430	0.268
DS	0.013	0.021	0.025	0.020	0.020	0.020	0.040	0.020	0.040	0.024
SHOT	0.930	1.730	2.110	1.800	1.860	1.660	2.600	2.030	3.090	1.979
CORR	10.280	14.480	12.760	11.250	11.620	14.300	16.350	15.730	23.390	14.462
GCG	4.960	5.440	5.830	5.400	5.440	5.360	5.890	5.570	6.150	5.560
GHV	2.070	2.860	2.800	2.370	1.880	2.180	4.580	2.840	3.760	2.816
All	20.823	27.281	26.295	23.749	23.651	26.360	32.200	29.060	39.470	27.654

3.4.2 Discussion

As we can observe, the execution times are increased in comparison with the ones shown in Table 3.3 as expected. Comparing the mean execution times for all phases on all scenes, 27.654 seconds on Jetson TK1 and 21.761 seconds on the desktop CPU, an approximate speedup of $1.27x$ is achieved using the latter one.

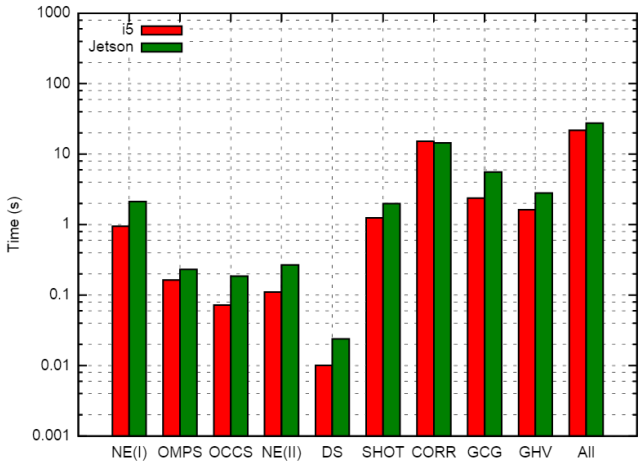


Figure 3.25: Mean execution times for all phases, Intel i5 vs Jetson TK1. Notice that the y-axis scale is logarithmic.

Figure 3.25 shows a graphical representation of the execution times of the desktop CPU versus the Jetson TK1 for each phase, taking into account the data shown in Tables 3.3 and 3.4. It is important to notice that the matching phase is faster on the Jetson TK1. This is due to floating point rounding issues which cause more aggressive segmentation and downsampling phases which in turn suppose less points to extract descriptors so less matchings have to be performed. However, this fact does not have a significant impact on the time distribution of the different phases as shown in Figure 3.26 in comparison with the Figure 3.24.

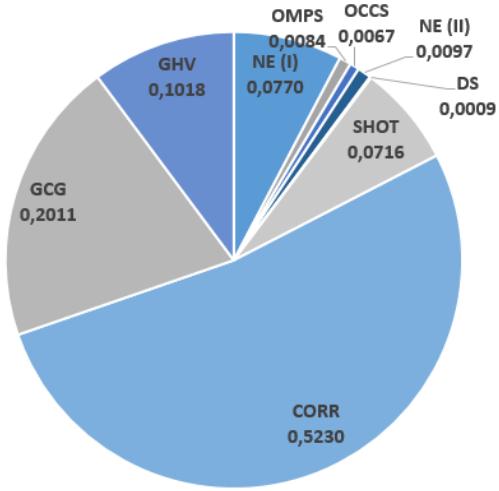


Figure 3.26: Phase time distribution using SHOT on Jetson TK1.

3.5 CPU optimizations

Before getting right into the GPU implementation, we decided to optimize the CPU one to get all the possible performance from the sequential implementation. These enhancements range from algorithmic improvements and architectural optimizations to efficient data structures and ad-hoc tricks for our application. In this section we will explain each optimization one by one and we will also carry out a brief set of experiments to determine the performance gain.

3.5.1 High-dimensionality optimized k-d tree

The current CPU implementation makes use of a typical k-d tree data structure to perform nearest neighbor searches during the matching phase. As we stated in Section 3.3, most of the time of the pipeline is spent finding correspondences. However, we expected k-d trees to be extremely efficient. The problem lies on the dimensionality of the descriptors. Common k-d tree implementations are optimized for low-dimensional queries like 3D points, but we are matching high-dimensional descriptors; for instance, SHOT has 352 dimensions. In this sense, simple k-d trees perform poorly when dealing with high-dimensional data [76, 77, 78].

In order to increase the efficiency of the matching process, we will make use of a special implementation of a k-d tree which is optimized for high-dimensional data: the *KDTreeIndexParams* from *FLANN* (Fast Approximate Nearest Neighbor Search) library [79]. This implementation was proposed by Silpa-Anan *et al.* in [80]. It is based on generating a set of multiple randomized trees which are later used to search in parallel. This approach has become one of the most effective methods for matching high-dimensional data.

Listing 3.21 shows the optimized version of the original matching phase using the aforementioned k-d tree. As we can observe, additional FLANN types are needed to convert data from PCL descriptors to FLANN matrices which can be used by FLANN indexes. A brief comparison between the non-optimized version and the optimized one was performed using the Jetson TK1. The results of that experimentation are shown in Table 3.5.

Listing 3.21: High-dimensionality optimized correspondence search.

```
1  int numInput = pModelDescriptors->points.size();
2  int inputsize = TDescriptor::descriptorSize();
3
4  flann::Matrix<float> data(
5    new float[numInput * inputsize], numInput, inputsize);
6
7  for (size_t i = 0; i < data.rows; ++i)
8    for (size_t j = 0; j < data.cols; ++j)
9      data[i][j] = pModelDescriptors->at(i).descriptor[j];
10
11 flann::Index<flann::L2<float>> index(data,
12   flann::KDTreeIndexParams(4));
13 index.buildIndex();
14
15 double distance = 0.0; int neighbors = 0; int kValue = 1;
16
17 for (size_t i = 0; i < pSceneDescriptors->size(); ++i)
18 {
19
20   flann::Matrix<float> p = flann::Matrix<float>(
```

```
21        new float[inputsize], 1, inputsize);
22
23      memcpy(
24        &p.ptr()[0],
25        &pSceneDescriptors->at(i).descriptor[0],
26        p.cols * p.rows * sizeof(float));
27
28      flann::Matrix<int> indices;
29      flann::Matrix<float> distances;
30
31      indices = flann::Matrix<int>(new int[kValue], 1, kValue);
32      distances = flann::Matrix<float>(new float[kValue], 1, kValue);
33
34      int neighborsFound = index.knnSearch(p, indices, distances,
35        kValue, flann::SearchParams(512));
36
37      if (neighborsFound == 1 && distances[0][0] < pThreshold)
38      {
39        pcl::Correspondence correspondence(indices[0][0],
40          static_cast<int>(i), distances[0][0]);
41        pCorrespondences->push_back(correspondence);
42      }
43    }
```

As we can observe in Table 3.5, a mean speedup of 3.982x is achieved by this optimized k-d tree implementation when using the SHOT descriptor.

Table 3.5: Execution times for the matching phase using the non-optimized and the high-dimension optimized k-d tree on Jetson TK1. Speedup is also shown.

	Execution times (s) for the different scenes									
k-d tree	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Mean (s)**
Original	10.280	14.480	12.760	11.250	11.620	14.300	16.350	15.730	23.390	14.462
Optimized	2.580	3.620	3.190	2.860	2.960	3.590	4.140	3.950	5.730	3.624
Speedup	3.984	4.000	4.000	3.934	3.926	3.983	3.949	3.982	4.082	3.982

3.5.2 Multi-core acceleration

Various phases of the pipeline can benefit from a multi-core implementation since most of their operations are suitable for concurrent execution on multi-threaded processors. In order to make our implementation executable on multi-core processors we will resort to *OpenMP*. Firstly, we optimized the previously modified matching phase by parallelizing the neighbor search loop as shown in Listing 3.22. It is important to remark that a critical section is needed to update the correspondence's vector and avoid race conditions due to reallocation operations like *push_back*.

Listing 3.22: Multi-core optimized correspondence search with four threads.

```
int numInput = pModelDescriptors->points.size();
int inputsize = TDescriptor::descriptorSize();

double distance = 0.0; int neighbors = 0; int kValue = 1;
flann::Matrix<float> data(
  new float[numInput * inputsize], numInput, inputsize);

for (size_t i = 0; i < data.rows; ++i)
  for (size_t j = 0; j < data.cols; ++j)
    data[i][j] = pModelDescriptors->at(i).descriptor[j];

flann::Index<flann::L2<float>> index(data, flann::KDTreeIndexParams(4));
index.buildIndex();

#pragma omp parallel for firstprivate(index) num_threads(4)
for (size_t i = 0; i < pSceneDescriptors->size(); ++i)
{
  flann::Matrix<float> p = flann::Matrix<float>(
    new float[inputsize], 1, inputsize);
  memcpy(
    &p.ptr()[0],
    &pSceneDescriptors->at(i).descriptor[0],
    p.cols * p.rows * sizeof(float));

  flann::Matrix<int> indices;
  flann::Matrix<float> distances;

  indices = flann::Matrix<int>(new int[kValue], 1, kValue);
  distances = flann::Matrix<float>(new float[kValue], 1, kValue);

  int neighborsFound = index.knnSearch(p, indices, distances,
    kValue, flann::SearchParams(512));

  #pragma omp critical
  {
    if (neighborsFound == 1 && distances[0][0] < pThreshold)
    {
      pcl::Correspondence correspondence(indices[0][0],
        static_cast<int>(i), distances[0][0]);
      pCorrespondences->push_back(correspondence);
    }
  }
}
```

We have carried out another brief experimentation to confirm the performance gain by using this optimized implementation. Table 3.6 shows the results of that experimentation. As we can observe, a mean speedup of 2.577 is achieved.

Table 3.6: Execution times and speedup for the matching phase using the high-dimension optimized k-d tree and its OpenMP optimized version on Jetson TK1.

	Execution times (s) for the different scenes									
Matching	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Mean (s)**
Original	2.580	3.620	3.190	2.860	2.960	3.590	4.140	3.950	5.730	3.624
Optimized	1.040	1.400	1.256	1.138	1.167	1.389	1.569	1.507	2.128	1.399
Speedup	2.481	2.586	2.540	2.513	2.536	2.585	2.639	2.621	2.693	2.577

In addition to the multi-core optimization of the matching phase, we have also modified the descriptor extraction stage to make use of a PCL multi-core optimized SHOT implementation named *SHOTEstimationOMP* as shown in Listing 3.23.

Listing 3.23: Multi-core optimized SHOT with OpenMP.

```
void compute_shot_descriptors (
  const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
  const pcl::PointCloud<TNormal>::ConstPtr & pNormals,
  const pcl::PointCloud<TPoint>::ConstPtr & pKeypoints,
  const pcl::search::KdTree<TPoint>::ConstPtr & pKdTree,
  const float & pSearchRadius,
  pcl::PointCloud<pcl::SHOT352>::Ptr & pDstCloud) {
  pcl::SHOTEstimationOMP<TPoint, TNormal, pcl::SHOT352> shot;
  shot.setSearchMethod(pKdTree);
  shot.setInputCloud(pKeypoints);
  shot.setInputNormals(pNormals);
  shot.setRadiusSearch(pSearchRadius);
  shot.setSearchSurface(pSrcCloud);
  shot.compute(*pDstCloud);
}
```

The results of the experimentation carried out to verify the performance improvement of this optimization are shown in Table 3.7. By using this multi-core implementation, a mean speedup of 2.724x is obtained.

Table 3.7: Execution times and speedup for the descriptor extraction phase using the original SHOT and its OpenMP optimized version on Jetson TK1.

	Execution times (s) for the different scenes									
SHOT	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Mean (s)**
Original	0.930	1.730	2.110	1.800	1.860	1.660	2.600	2.030	3.090	1.979
Optimized	0.393	0.602	0.800	0.660	0.679	0.568	0.983	0.747	1.069	0.722
Speedup	2.366	2.874	2.638	2.727	2.739	2.923	2.645	2.718	2.891	2.724

3.5.3 Organized normal estimation

The currently used normal estimation method has to perform a lot of neighbor search operations in order to build the covariance matrix of a certain neighborhood as we previously mentioned in Section 3.2.1. Although this process is highly optimized by using a k-d tree, those search operations still have a considerable associated cost. An alternative is the *Integral Image Normal Estimation* method. This approach needs a structured point cloud to exploit its matrix configuration so that no costly neighbor search operations are required. In our case, normals are estimated using raw point clouds which are organized so we

can apply this method. Listing 3.24 shows the implementation of this stage using PCL. Parameters must be tuned for each particular application.

Listing 3.24: Integral Image Normal Estimation with PCL.

```
void compute_normals_structured (
  const pcl::PointCloud<TPoint>::ConstPtr & pCloud,
  pcl::PointCloud<TNormal>::Ptr & pNormals)
{
  pcl::IntegralImageNormalEstimation<TPoint, TNormal> ne;
  ne.setNormalEstimationMethod (ne.COVARIANCE_MATRIX);
  ne.setMaxDepthChangeFactor(0.02f);
  ne.setNormalSmoothingSize(10.0f);
  ne.setInputCloud(pCloud);
  ne.compute(*pNormals);
}
```

In order to validate our improvement expectations, a similar experimentation to the previous ones was carried out by measuring the execution times of the normal estimation phase throughout the different test scenes using the non-optimized and the optimized implementations. Table 3.8 shows the results of this comparison. A mean speedup of 11.416x is proved.

Table 3.8: Execution times and speedup for the normal estimation (I) phase using the original method and its organized optimized version on Jetson TK1.

	Execution times (s) for the different scenes									
SHOT	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Mean (s)**
Original	0.930	1.730	2.110	1.800	1.860	1.660	2.600	2.030	3.090	1.979
Optimized	0.172	0.179	0.171	0.171	0.171	0.170	0.171	0.178	0.176	0.173
Speedup	5.407	9.665	12.339	10.526	10.877	9.765	15.205	11.404	17.557	11.416

3.5.4 Bounding box clipping

In the end, our application will be integrated on a robotic platform which is also able to recognize gestures and other human-computer interaction methods. One of the possible approaches consists of allowing the user to point the index finger at some part of the scene. By recognizing the direction of the pointing finger and the arm and projecting that vector onto the point cloud captured by the robot we will be able to segment a part of the scene using a bounding box. Listing 3.25 shows the implementation of that bounding box filter.

Listing 3.25: Bounding box filter implemented with PCL.

```
void bounding_box ( const pcl::PointCloud<TPoint>::ConstPtr & pSrcCloud,
  const float & pLimitMinX, const float & pLimitMaxX,
  const float & pLimitMinY, const float & pLimitMaxY,
  const float & pLimitMinZ, const float & pLimitMaxZ,
  pcl::PointCloud<TPoint>::Ptr & pDstCloud) {
  pcl::PointCloud<TPoint>::Ptr filtered_cloud_z(
    new pcl::PointCloud<TPoint>());
  pcl::PointCloud<TPoint>::Ptr filtered_cloud_x(
    new pcl::PointCloud<TPoint>());
  pcl::PassThrough<TPoint> pass;
  pass.setInputCloud(pSrcCloud);
  pass.setKeepOrganized(true);
  pass.setFilterFieldName("z");
  pass.setFilterLimits(pLimitMinZ, pLimitMaxZ);
  pass.filter(*filtered_cloud_z);
  pass.setInputCloud(filtered_cloud_z);
  pass.setFilterFieldName("x");
  pass.setFilterLimits(pLimitMinX, pLimitMaxX);
  pass.filter(*filtered_cloud_x);
  pass.setInputCloud(filtered_cloud_x);
  pass.setFilterFieldName("y");
  pass.setFilterLimits(pLimitMinY, pLimitMaxY);
  pass.filter(*pDstCloud);
}
```

We have repeated the Jetson TK1 experimentation (see Section 3.4) by running the whole pipeline with all the aforementioned optimizations and using a bounding box of $40x40x40cm$. The results are shown in Table 3.9. As we can observe, a mean speedup of $3.574x$ is achieved.

Table 3.9: Execution times and speedup for the whole pipeline using the original one and the fully optimized version on Jetson TK1.

	Execution times (s) for the different scenes									
Pipeline	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Mean (s)**
Original	20.823	27.281	26.295	23.749	23.651	26.360	32.200	29.060	39.470	27.654
Optimized	6.448	8.508	6.721	7.113	7.451	8.509	7.067	8.041	9.784	7.738
Speedup	3.229	3.207	3.912	3.339	3.174	3.098	4.556	3.614	4.034	3.574

3.6 GPU optimizations

By using certain CPU execution optimization techniques and redesigning some of the phases of the pipeline to use more efficient data structures or algorithms we achieved an approximate speedup of $3.5x$ cutting down the runtime of the whole pipeline from ~ 28 seconds to ~ 7.

However, we can still reduce the overall runtime of the pipeline by exploiting the implicit parallelism of some of its stages. The obvious targets are the bilateral filter, normal estimation and the cloud resolution computation. All

of these phases perform per-pixel or per-point operations in a completely independent manner. This means that those stages are perfectly suited to be deployed into a GPU to take advantage of its SIMT model (Single Instruction Multiple Threads). By doing this, we can process multiple pixels or points in parallel thus reducing the runtime.

The main goal of this implementation is achieving an efficient system which offers enough interactivity with the user. In this regard, we need to introduce some way to capture scenes in real-time to feed the system with them to achieve the aforementioned interactivity.

Up until now, we have been using scene clouds which were captured and generated offline. Now, we need a way of getting sensor's depth and RGB streams online to create a point cloud that will be eventually processed by our system. In order to do that, we have used the OpenNI 2 grabber interface provided by the *Point Cloud Library* to obtain depth and RGB maps in real-time as shown in Listing 3.26. Needless to say, the point cloud creation process is also a good candidate for parallelization.

Listing 3.26: OpenNI 2 grabber for getting depth and RGB images.

```
1  void Grabber::run ()
2  {
3    pcl::Grabber* interface = new pcl::io::OpenNI2Grabber();
4    boost::function<void (const boost::shared_ptr<pcl::io::Image> &,
5    const boost::shared_ptr<pcl::io::DepthImage> &, float constant)> f =
6    boost::bind(&Grabber::images_cb_, this, _1, _2, _3);
7    interface->registerCallback (f);
8    interface->start ();
9    while(1){}
10   interface->stop ();
11 }
```

3.6.1 Cloud projection

The grabber interface provides a RGB map with color information M_c and a disparity map M_d. In order to generate a point cloud, the depth and color information has to be projected in a three-dimensional space in which both maps are aligned thus producing the point cloud which represents the scene as shown in Figure 3.27.

By using the mathematical model discussed in Section 2.2, we can use the depth z of each point of the map to project it in a 3D space:

$$\begin{aligned} p_x &= z \cdot (x - x_c) \cdot 1/f_x \\ p_y &= z \cdot (y - y_c) \cdot 1/f_y \\ p_z &= z \end{aligned} \tag{3.8}$$

where $p \in R^3$, x and y are the row and the column of the projected pixel, x_c and y_c are the distances (in pixels) to the map center and f_x and f_y are the focal distances of the sensor.

As we can see, this projection is performed independently for each pixel of the map so it is perfectly suited for a parallel implementation. The pseudo-code of the cloud projection kernel is shown in Algorithm 1.

input : A depth map M_d of size 640×480
output: Projected point cloud P_{xyz} into 3D space

```
1  __global__ void gpuPointCloudProjectionKernel( Md, Pxyz );
2  {
3      //This kernel is executed creating one thread for each pixel in parallel;
4      int u = threadIdx.x + blockIdx.x * blockDim.x;
5      int v = threadIdx.y + blockIdx.y * blockDim.y;
6      float z = Md [v][u] / 1000.f; //Depth is stored in millimetres
7      float px = z * (u - cx) * fx_inv;
8      float py = z * (v - cy) * fy_inv;
9      float pz = z;
10     Pxyz[v][u].x = px;
11     Pxyz[v][u].y = py;
12     Pxyz[v][u].z = pz;
13 }
```

Algorithm 1: Pseudo-code of the GPU-based point cloud projection algorithm.

By executing this kernel with as many threads as pixels in the depth map with a proper block/grid distribution, each thread will process one pixel. It will retrieve its corresponding disparity value from the disparity map, perform the projection and store that 3D point in its corresponding position in the organized point cloud.

Figure 3.27: Depth, RGB map and point cloud (Figure reproduced from [17]).

3.6.2 Normal estimation

Once the organized point cloud has been projected, we can perform the normal estimation process previously described in Section 3.2.1.2. By taking a closer look to that formulation, it is obvious that the normal at each point can be computed independently thus making the normal estimation process ideal for a parallel implementation. Algorithm 2 shows the pseudo-code of a CUDA kernel for organized normal estimation using PCA.

input : A projected point cloud d_P_{xyz}
output: Point cloud of normals d_N_{xyz}

```
1 __global__ void gpuNormalEstimationKernel( P_xyz, k );
2 {
3     //This kernel is executed creating one thread for each point in parallel;
4     int u = threadIdx.x + blockIdx.x * blockDim.x;
5     int v = threadIdx.y + blockIdx.y * blockDim.y;
6     //Compute Covariance matrix centered at point p using k neighbours;
7     d_N_xyz[u][v] = compCovarianceMat(u,v,k,N);
8     d_N_xyz[u][v] = checkOrientation();
9 }
```

Algorithm 2: Pseudo-code of the GPU-based normal estimation algorithm

3.6.3 Bilateral filter

Another example of computation that is fitted for being offloaded to the GPU is the bilateral filter. Its principles were described in Section 3.2.1.1. In this filter, the intensity value at each pixel in an image is replaced by a weighted average of intensity values from nearby pixels. Since each pixel is computed independently, the bilateral filter is suitable for a GPU implementation.

The kernel shown in Listing 3.27 takes the depth map as input and each thread computes the new intensity for a single pixel whose value is later stored in an output depth map at its corresponding position.

Listing 3.27: Bilateral filter CUDA kernel.

```
__global__ void
bilateralKernel (const PtrStepSz<ushort> src, PtrStep<ushort> dst,
  float sigma_space2_inv_half, float sigma_color2_inv_half)
{
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  if (x >= src.cols || y >= src.rows)
    return;

  const int R = 6;
  const int D = R * 2 + 1;
  int value = src.ptr (y)[x];

  int tx = min (x - D / 2 + D, src.cols - 1);
  int ty = min (y - D / 2 + D, src.rows - 1);

  float sum1 = 0;
  float sum2 = 0;

  for (int cy = max (y - D / 2, 0); cy < ty; ++cy)
  {
    for (int cx = max (x - D / 2, 0); cx < tx; ++cx)
    {
      int tmp = src.ptr (cy)[cx];

      float space2 = (x - cx) * (x - cx) + (y - cy) * (y - cy);
      float color2 = (value - tmp) * (value - tmp);

      float weight = __expf (-(space2 * sigma_space2_inv_half + color2 *
        sigma_color2_inv_half));

      sum1 += tmp * weight;
      sum2 += weight;
    }
  }

  int res = __float2int_rn (sum1 / sum2);
  dst.ptr (y)[x] = max (0, min (res, numeric_limits<short>::max ()));
}
```

3.6.4 Cloud resolution

The last stage that will be optimized by resorting to a parallel implementation on the GPU is the cloud resolution computation. As we previously stated in Section 3.2.1.4, the cloud resolution is defined as the average distance between each cloud point and its nearest neighbor. This is again perfectly suited for a parallel implementation since we can process each point independently to compute its distance to the nearest neighbor and then calculate the average in a sequential manner.

The CUDA kernel shown in Listing 3.28 takes as input an organized cloud and its size and each thread processes a single point by computing the distance to the nearest neighbor and storing it in a linear array at its corresponding position. After the kernel has finished its execution, the CPU takes that linear array of distances and computes the average which is the cloud resolution.

Listing 3.28: Cloud resolution CUDA kernel.

```
__global__ void
computeCloudResolutionKernel (int rows, int cols,
  const PtrStep<float> vmap, float *nearest_neighbor)
{
  int u = threadIdx.x + blockIdx.x * blockDim.x;
  int v = threadIdx.y + blockIdx.y * blockDim.y;

  if (u >= cols || v >= rows) return;

  int global_index = (v*cols+u);
  float v_i_x = vmap.ptr (v)[u];
  float v_i_y = vmap.ptr (v +  rows)[u];
  float v_i_z = vmap.ptr (v +  2*rows)[u];
  float3 centroid = make_float3( v_i_x, v_i_y, v_i_z);
  int ty = min (v - ky / 2 + ky, rows - 1);
  int tx = min (u - kx / 2 + kx, cols - 1);

  float min_distance=numeric_limits<float>::max();
  for (int cy = max (v - ky / 2, 0); cy < ty; cy += STEP) {
    for (int cx = max (u - kx / 2, 0); cx < tx; cx += STEP) {
      float v_x = vmap.ptr (cy)[cx];
      if (!isnan (v_x) && cy != v && cx != u)
      {
        float v_y = vmap.ptr (cy + rows)[cx];
        float v_z = vmap.ptr (cy + 2 * rows)[cx];
        float3 v = make_float3(v_x,v_y,v_z) - centroid;
        float aux = norm(v);
        if( aux < min_distance )
          min_distance = aux;
      }
    }
  }
  nearest_neighbor[global_index]=min_distance;
}
```

3.6.5 Results

Once all the optimizations have been implemented, we tested the interactive system to measure the performance gain of the GPU accelerated stages. For this experimentation, we ran the online system and captured a scene with the OpenNI2 grabber. That scene was later processed by the GPU parallelized pipeline. We carried out a brief study by running the pipeline with different block configurations for the kernels in order to empirically determine the one which offers the best results. The runtimes are shown in Table 3.10.

Some configurations could not be tested due to the hardware limitations of the Jetson TK1 GPU. The specifications establish that a block can be composed by 1024 threads at most. In addition, a multiprocessor can execute a maximum of 2048 threads. Another important limitation is the amount of blocks that can be executed by a multiprocessor at a time which, 8 for this generation of CUDA devices [81].

We have tested configurations ranging from $(8, 8)$ (64 threads/block) to $(64, 64)$ blocks (4096 threads/block). Taking into account the aforementioned limitations, the following configurations couldn't be executed: $(32, 64)$, $(64, 32)$ and $(64, 64)$.

Table 3.10: Execution times for the GPU accelerated stages with different block configurations varying x and y dimensions.

	Runtime (s)				
Conf.	**Cloud proj.**	**Normal est.**	**Bilateral filt.**	**Cloud res.**	**Total**
8x8	0.0013	0.0118	0.0226	0.0164	0.0521
8x16	0.0009	0.0112	0.0217	0.0152	0.0491
8x32	0.0010	0.0115	0.0217	0.0153	0.0495
8x64	0.0009	0.0121	0.0218	0.0147	0.0494
16x8	0.0009	0.0078	0.0175	0.0115	0.0378
16x16	0.0009	0.0080	0.0183	0.0113	0.0385
16x32	0.0009	0.0080	0.0183	0.0100	0.0372
16x64	0.0010	0.0103	0.0179	0.0140	0.0432
32x8	**0.0009**	**0.0066**	**0.0176**	**0.0102**	**0.0353**
32x16	0.0009	0.0076	0.0175	0.0112	0.0373
32x32	0.0010	0.0084	0.0168	0.0124	0.0387
32x64	N/A	N/A	N/A	N/A	N/A
64x8	0.0009	0.0078	0.0173	0.0119	0.0379
64x16	0.0009	0.0088	0.0169	0.0116	0.0383
64x32	N/A	N/A	N/A	N/A	N/A
64x64	N/A	N/A	N/A	N/A	N/A

As we can observe, the $32x8$ configuration is the one which offers the best performance. The explanation is simple: each block consists of 256 threads and the Jetson TK1 has a maximum of 8 blocks and 2048 threads executing at the same time on a multiprocessor; with 256 threads per block, 8 blocks fit perfectly to both limits. In addition, $32x8$ is better than other similar sizes ($8x32$, $16x16$) due to memory access coalescence [82].

In Table 3.11 we can see the different steps that have been accelerated using the GPU and their different runtimes and the speedups. The acceleration is relative to the optimized CPU implementations of the different phases. The cloud

projection implementation on the CPU was extracted from the *PCL OpenNI wrapper* [83]. The rest of the phases are the organized normal estimation (see Section 3.5.3), bilateral filter (see Section 3.2.1.1) and the resolution computation (see Section 3.2.1.4).

Table 3.11: Runtime comparison between GPU accelerated stages and CPU ones.

	Cloud proj.	**Normal est.**	**Bilateral filt.**	**Cloud res.**	**Total**
CPU	0.0345	0.3496	0.1357	0.5830	1.1027
GPU	0.0009	0.0066	0.0176	0.0102	0.0353
Speedup	37.86	53.00	7.71	57.32	31.27

The previous runtimes do not include the time needed for data transfer operations. The reason for that is the CUDA zero-copy feature which can be used by the Tegra K1 device [84]. The Jetson TK1 has 2 GiB of RAM memory which is shared by the CPU and the GPU. This is why it can save memory transfers, that will be mandatory on discrete GPUs, using zero-copy access. Since we were using the *PCL DeviceArray2D* wrapper, we could not take advantage from the zero-copy access but a future work for this thesis could consist of including the feature in that wrapper.

It is proven that all stages are significantly accelerated, achieving a mean $x30$ speedup with peaks of $x50$ in some phases. The results confirm that GPUs are ideally suited for our application. One critical aspect of designing parallel algorithms is identifying the units of work and determining how they will interact via communication and synchronization. A second critical aspect is analyzing the data access patterns and ensuring data locality to the processing units. It is also necessary to consider the program execution pipeline in order to avoid unnecessary data transfers. These critical aspects have been satisfied by our GPU implementations.

Chapter 4

Conclusions

This chapter discusses the main conclusions extracted from the work presented in this document. The chapter is divided into three different sections: Section 4.1 presents and discusses the final conclusions of the work presented in this Bachelor's Thesis. Section 4.2 enumerates the most relevant highlights of this work. Finally, Section 4.3 presents future works: open problems and research topics that remain for future research.

4.1 Conclusions

In this work, we have implemented a pipeline for interactive 3D object recognition using the *Point Cloud Library*. The developed system is able to recognize multiple models in cluttered and occluded scenes by leveraging to local feature descriptors. The experiments that were carried out proved the accuracy of the proposal.

Once the basic implementation was finished, we deployed the system in a mobile GPU computing platform, namely *NVIDIA's Jetson TK1*, and tested its performance against the same pipeline running on a significantly better desktop computer. Through a extensive experimentation we quantified the performance loss and proved that the desktop computer presented an approximate speedup of 1.3 with respect to the Jetson TK1. At this point, the whole pipeline took an average runtime of 30 seconds to recognize a single object so the pipeline had to be optimized in order to achieve an interactive system for a mobile robotic platform.

In this regard, the initial CPU implementation was optimized at different levels either by reducing the computational complexity of certain phases or by applying common acceleration techniques and even some *ad hoc* simplifications which are acceptable inside the scope of our proposal. For instance, we used a special k-d tree optimized for high dimensional data to reduce the neighbor search complexity during the feature matching phase. We also changed the typical normal estimation approach to take advantage of the organization of the point cloud by using integral images; this optimization had a significant impact on the computational cost of the normal estimation phase. What's more, we resorted to *OpenMP* to accelerate various stages of the pipeline on multi-core processors. Given the context of this project, we also reduced the execution time of the whole pipeline by clipping the input clouds and keeping a reduced region of interest which can be determined by taking advantage of the gesture recognition and human-computer interaction capabilities of the robotic platform in which this system is intended to be integrated.

In addition, we took advantage of the GPU computing power of the Jetson TK1 and offloaded some of the massively parallel phases on the GPU. We developed CUDA implementations for creating the scene point cloud, computing the cloud resolution, estimating the normals and applying a bilateral filter. Those stages showed exceptional performance on the GPU, bringing us closer to an interactive system.

We also created a 3D object dataset for testing the system. It is composed of 30 fully reconstructed object models and a set of validation scenes in which the real-life objects appear with different poses, occlusion and lighting conditions. A simple 3D object reconstruction tool was developed to generate fully registered point clouds and meshes of those objects using a certain number of partial views.

From a personal point of view, I feel this project has allowed me to show off the knowledge acquired during the last four years and I am proud of the end result. The project was ambitious and the topic was nearly a complete unknown for me so it has been a real challenge to learn the rudiments and start moving on the right track. This implied a lot of trial and error with a lot of negative results mainly due to the novelty of the proposal. However, far from discouraging, this project motivated me to keep pushing forward, mostly thanks to my colleagues and advisors.

4.2 Highlights

The highlights of this work are the following:

- A 3D object recognition pipeline implemented with the *Point Cloud Library*. It is based on local feature descriptors and provides several capabilities.
  - Extensive experimentation with multiple descriptors.
  - Multiple models can be recognized.
  - Recognition can be performed under significant levels of occlusion.
  - Hypotheses are verified to reject false positives.
  - Full pose estimation is provided and optionally refined with ICP.
- A set of optimizations were included to make the system able to perform online object recognition.
  - High-dimensionality optimized k-d tree for descriptor matching.
  - Integral images based normal estimation for organized clouds.
  - Multi-threaded matching and descriptor extraction with OpenMP.
  - Bounding-box clipping based on HCI information.
- The system was deployed on the new Jetson TK1 GPU computing platform and some phases were accelerated using parallel CUDA implementations.
  - Parallel cloud creation through depth projection.
  - Parallel normal estimation for organized clouds.
  - Parallel bilateral filtering.
  - Parallel cloud resolution computation.
- A 3D object reconstruction tool implemented with the *Point Cloud Library* for creating models from a set of partial views of objects placed on a turntable.
- A multi-sensor 3D object dataset consisting of a set of reconstructed 3D models and a handful of validation scenes for the recognition system.

In addition, all the tools and applications which were developed during this project will be available as open-source software under the MIT license on *GitHub* [1]. Furthermore, this document and its LaTeXsource will be also released under the *Creative Commons Atribution-NonCommercial-ShareAlike 4.0 International License*.

4.3 Future work

Due to the time constraints of this project, a lot of possible improvements and ideas were left out for future work. Here we summarize them to conclude this thesis.

This work was mainly focused on local feature descriptors and so was the experimentation to determine the one which best suited our application. However, keypoint detectors are as important as feature descriptors to obtain good results in terms of performance and efficiency. In this sense, this work can be extended by implementing more keypoint detectors together with an extensive experimentation to determine the best combination of detector and descriptor.

Furthermore, some already existing descriptors were left out due to the absence of a PCL implementation. For instance, Mian's Tensor and TriSI can be implemented to extend the experimentation and even send a pull request to the *Point Cloud Library* public code repository as a contribution.

In addition, only a few phases were accelerated using CUDA and the power of GPU computing to take advantage of redesigned parallel implementations. Some of the slower phases were not optimized. As a future work, a parallelization study can be conducted over the correspondence grouping and hypotheses verification stages to rethink them and create a GPU accelerated implementation. The descriptor extraction and matching phases are also good candidates for parallelization.

Also related with the GPU implementation, it was programmed using a generic CUDA approach with no special optimizations for the target platform other than setting an appropriate size for the grid and the blocks and compiling for the specific architecture. In this regard, the system's performance can be enhanced by using specific CUDA features for the Jetson TK1 architecture, i.e., zero-copy.

[1] `https://github.com/Blitzman`

Appendix A

3D Object Reconstruction

In this appendix we present a pipeline for 3D object reconstruction using the Point Cloud Library. This pipeline may be used for reconstructing objects from partial views taken using a turntable so that we know the approximate transformations between views. Taking that set of partial views with their corresponding coarse transformations, a reconstructed mesh is generated together with a fully registered point cloud.

A.1 Introduction

In 3D computer vision, the problem of object reconstruction consists of generating three-dimensional models from different views of the same object. The goal is to generate a digital 3D model which accurately represents the real object.

This problem was usually tackled with 2D computer vision techniques using intensity images. However, state-of-the-art methods based on this approach present many problems which limit the quality of the reconstructed models. With the advent of low-cost 3D sensors like Microsoft Kinect, a whole new range of possibilities were unlocked so that a significant progress has been made in the field of object reconstruction. The use of 3D data with depth information for this purpose presents a lot of advantages with respect to only using 2D color images: additional geometry data, no surface ambiguities and invariance to illumination since most methods only make use of geometric information.

The reconstruction problem can be divided into two parts: data acquisition and surface registration and reconstruction. On the one hand, sensors must be handled to obtain the partial views of the objects. On the other hand, the acquired clouds must be registered to compute a mesh reconstruction. This process implies a lot of secondary operations that have to deal with point clouds.

In this work we will present a typical 3D object reconstruction pipeline for generating 3D meshes from a set of partial views. These views have been taken using a turntable to change the point of view with respect to the object. We will use *OpenNI* [85] together with the *Point Cloud Library* [49] to generate a set of point clouds for each partial view. The *Point Cloud Library* provides efficient ways for dealing with point clouds and implements a lot of useful operations for 3D processing.

The appendix is structured as follows. Section A.2 explains the acquisition problem and the approach taken to obtain point clouds using *OpenNI/PCL*. Section A.3 describes the reconstruction process using PCL. In the end, we will draw some conclusions about this problem and future improvements.

A.2 Acquisition

In order to capture the partial views of the object that we will be reconstructing, we have used the *Point Cloud Library* [23] together with *OpenNI* [48] and its *grabber* interface. Using this approach, a callback is registered for that interface so each time it captures a frame, a PCL point cloud with color information (XYZRGB) is generated and passed to the callback. This process is shown on Figure A.1.

Using this grabber, we will capture a total of 64 views for each object. The process or algorithm for capturing those views is the following one: 1) grab a point cloud using the aforementioned interface, 2) make a step with the turntable (a rotation of 5.265 $^\circ$ is performed to capture the 64 different and equally spaced points of view) and 3) wait for the turntable to stabilize. The interface for communicating with the turntable and operating it is beyond the scope of this work, in that sense, we omitted the code in Listing A.1.

Listing A.1: Grabbing point clouds with PCL and OpenNI.

```
class OpenNIGrabber
{
public:

  void run ()
  {
    pcl::Grabber* interface = new pcl::OpenNIGrabber();

    boost::function<void
      (const pcl::PointCloud<pcl::PointXYZRGBA>::ConstPtr&)>
      f = boost::bind (&SimpleOpenNIProcessor::cloud_cb_, this, _1);

    boost::signals2::connection c = interface->registerCallback (f);

    interface->start ();

    //Turntable code and periodic captures...

    interface->stop ();
  }

private:

  // [...]

  void cloud_cb_ (const pcl::PointCloud<pcl::PointXYZRGBA>::ConstPtr &c)
  {
    // Save point cloud in PCD format
    pcl::io::savePCDFile("scan_" + cloudNumber + ".pcd", c, false);
  }

};
```

Figure A.1 shows an example of point cloud of a partial view captured using the OpenNI grabber and later exported to PCD format for easier visualization with CloudCompare or Meshlab. Figures A.2 and A.3 show some of the RGB images and depth maps captured for different partial views.

A.3 Reconstruction

Once the full set of partial views has been acquired we end up with 64 point clouds (XYZRGB) stored in PCD format. The next stage is the actual reconstruction pipeline in which all those clouds are read and merged to generate the final reconstruction.

The outline of the reconstruction process is shown in Algorithm 3 where r is the registered point cloud, C is the set of partial views, c_0 is the first partial view, T is the approximate transformation between c_0 and the current view c_i, *origin* is the turntable center (constant across all views) and 5.625 is the turntable step rotation.

Figure A.1: Point cloud (XYZRGB) of a partial view of an object on the turntable captured using the aforementioned grabber with a PrimeSense Carmine device..

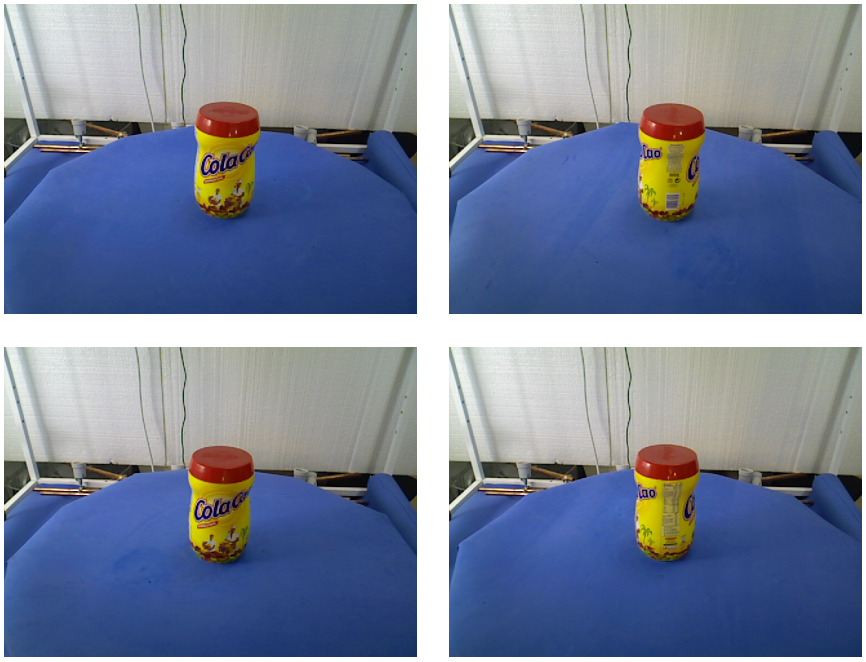


Figure A.2: RGB images of partial views (0, 16, 32 and 48).

As we can observe, the main process consists of the following steps: 1) pre-process the partial view to segment the object, simplify the cloud, remove out-

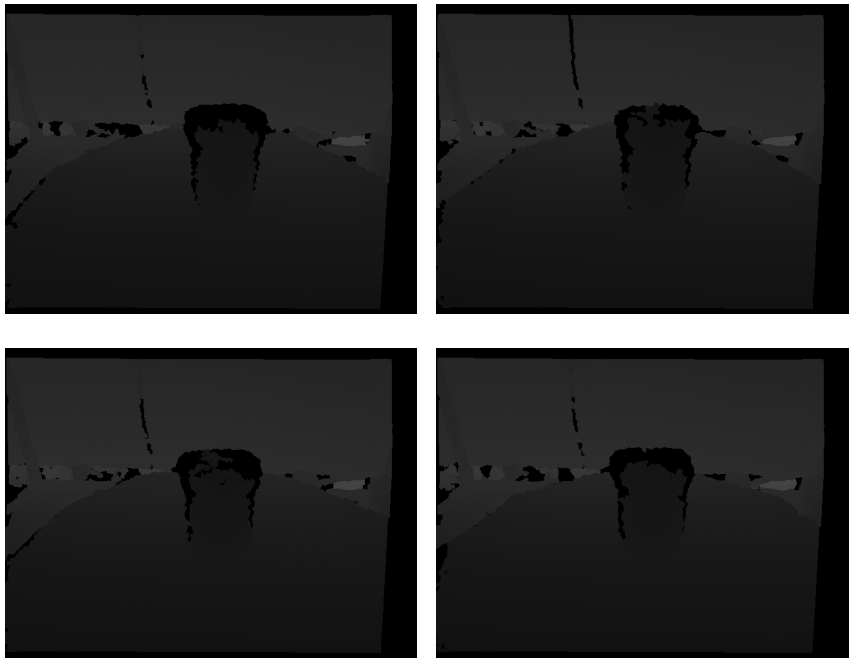


Figure A.3: Depth maps of partial views (0, 16, 32 and 48).

input : A set of 64 partial views $C = \{c_0, c_1, ..., c_n\}$
output: A reconstructed mesh m

```
1  reconstructObject( C, m )
2      r = c_0;
3      forall c_i ∈ C do;
4          c_i = preprocess(c_i);
5          T.rotation = i · 5.625;
6          T.translation = (origin.x, origin.y, origin.z);
7          c_i = transform(c_i, T);
8          r = register(c_i, r);
9      end
10     r = removeNoise(r);
11     r = downsample(r);
12     m = reconstructMesh(r);
```

Algorithm 3: General reconstruction process.

liers and noise and even smooth it, 2) apply the approximate transformation to the cloud using the known turntable displacement, 3) after that coarse alignment, apply a fine registration algorithm to refine the transformation. Once all clouds have been aligned, optionally downsample the registered cloud to simplify it and reduce the computational cost of the following phases, also optionally remove outliers and noise. Finally, apply the mesh reconstruction method to generate the model.

In this Section we will describe the aforementioned phases of the object reconstruction process. Subsection A.3.1 describes the preprocessing steps, Subsection A.3.2 explains the transformation and registration operations and finally A.3.3 covers the mesh reconstruction method.

A.3.1 Preprocessing

The main goal of the preprocessing phase is segmenting out the object and reduce the noise levels of the cloud to be registered.

In order to segment the object, a combination of filters is applied: bounding box filtering [86], plane segmentation filtering [87] and chroma key filtering.

To remove or reduce the noise levels of the cloud, another combination of filters is applied: statistical outlier removal (SOR) [88], radial outlier removal (ROR) [89], and euclidean cluster extraction (ECE) [90]. If the clouds are too dense, they are usually downsampled applying a Voxel Grid (VG) [91] filter.

First, we perform a basic segmentation using a bounding box filter by applying three different *PassThrough* filters for each dimension. Since we know that the objects will be placed in the center of the turntable we can approximately segment it out by defining a cube as a bounding box for that object. Listing A.2 shows the implementation of this bounding box filtering using the available Point Cloud Library *PassThrough* filters.

Listing A.2: Bounding box filtering using PassThrough.

```
1  pcl::PassThrough<pcl::PointXYZRGB> pass;
2
3  // passthrough depth (Z)
4  pass.setInputCloud(cloud);
5  pass.setKeepOrganized(true);
6  pass.setFilterFieldName("z");
7  pass.setFilterLimits(z_limit_min, z_limit_max);
8  pass.filter(*filtered_cloud_z);
9
10 // passthrough X
11 pass.setInputCloud(filtered_cloud_z);
12 pass.setFilterFieldName("x");
```

```
13    pass.setFilterLimits(x_limit_min, x_limit_max);
14    pass.filter(*filtered_cloud_x);
15
16    // passthrough Y
17    pass.setInputCloud(filtered_cloud_x);
18    pass.setFilterFieldName("y");
19    pass.setFilterLimits(y_limit_min, y_limit_max);
20    pass.filter(*filtered_cloud_y);
```

We also perform a plane segmentation operation to filter out all those points which can be fitted to a plane within a predefined distance. This step is normally optional because its aim is to remove the plane of the turntable which remains on the cloud after the bounding box filtering. Usually, it is easier and more accurate to perform a background subtraction operation or even a chroma key one, providing that the color of the object is not similar to the turntable's one. In this sense, we only apply plane segmentation for objects whose color could be filtered by chrome keying. Listing A.3 shows the implementation of a plane segmentation filter.

Listing A.3: Plane model segmentation

```
1    // Object for storing the plane model coefficients
2    pcl::ModelCoefficients::Ptr coefficients(new pcl::ModelCoefficients);
3    pcl::SACSegmentation<pcl::PointXYZRGB> segmentation;
4    segmentation.setInputCloud(filtered_cloud_y);
5    segmentation.setModelType(pcl::SACMODEL_PLANE);
6    segmentation.setMethodType(pcl::SAC_RANSAC);
7    segmentation.setDistanceThreshold(0.009);
8    segmentation.setMaxIterations(100);
9    pcl::PointIndices::Ptr inliers(new pcl::PointIndices);
10   segmentation.segment(*inliers, *coefficients);
11
12   // Extract the planar inliers from the input cloud
13   pcl::ExtractIndices<pcl::PointXYZRGB> extract;
14   extract.setInputCloud(filtered_cloud_y);
15   extract.setIndices(inliers);
16   extract.setNegative(true);
17   extract.setKeepOrganized(true);
18   extract.filter(*aux);
19   filtered_cloud_y = aux;
```

For those cases in which we can apply a simple chroma key filtering we use the code shown in Listing A.4.

Listing A.4: Chroma key filtering

```
1  const float bad_point = std::numeric_limits<float>::quiet_NaN();
2  pcl::PointXYZRGB pbad_point;
3  pbad_point.x = pbad_point.y = pbad_point.z = pbad_point.r =
4  pbad_point.g = pbad_point.b = pbad_point.a = bad_point;
5
6  for (size_t i = 0;i != filtered_cloud_y->size();i++)
7  {
8    // Do not consider NaN points
```

```
9     if (pcl::isFinite(filtered_cloud_y->points[i]))
10    {
11      // Do not consider bad points
12      if (filtered_cloud_y->points[i].y = 0.0f &&
13        filtered_cloud_y->points[i].z == 0.0f &&
14        filtered_cloud_y->points[i].x == 0) {
15        filtered_cloud_y->points[i] = pbad_point;
16      }
17      else {
18        uint8_t r = filtered_cloud_y->points[i].r;
19        uint8_t g = filtered_cloud_y->points[i].g;
20        uint8_t b = filtered_cloud_y->points[i].b;
21        // Determine the least intense channel
22        uint8_t min = (r < g) ? r : g;
23        min = (min < b) ? min : b;
24        // Determine the most intense channel
25        uint8_t max = (r > g) ? r : g;
26        max = (max > b) ? max : b;
27        // Check chroma key conditions
28        bool key =  b != min &&
29        (b == max || max - b < maximumCutoff) &&
30        (max - min) > differenceThreshold;
31        // Segment out the point
32        if (key) {
33          filtered_cloud_y->points[i] = pbad_point;
34        }
35      }
36    }
37  }
```

This method removes the points in which blue is not the less important channel and one of the following conditions is satisfied: either blue turns out to be the most intense channel, or either the difference between the blue channel intensity and the most intense channel falls below a predefined threshold (maximum cutoff). For removing that point, it is also necessary that the intensity difference between the most and the least intense channels exceeds a predefined threshold (difference threshold). Using both parameters (maximum cutoff and difference threshold) we can control the segmentation process to make it more or less sensitive to certain shades of blue thus achieving a segmentation as perfect as possible.

Figure A.4 shows a point cloud of a partial view before and after the previously defined preprocessing steps.

Once the object has been segmented out of the partial view we applied some noise removal filters. It is important to remark that depending on the camera, the point clouds will have higher or lower noise levels. The clouds shown in Figure A.4 were captured using a PrimeSense Carmine device and noise is barely present. However, Figure A.8 shows a segmented point cloud captured using a Kinect 2.0 device with high levels of noise (notice the trails in the borders of the object).

For those cases in which noise is a problem, we first applied a *Radius Outlier*

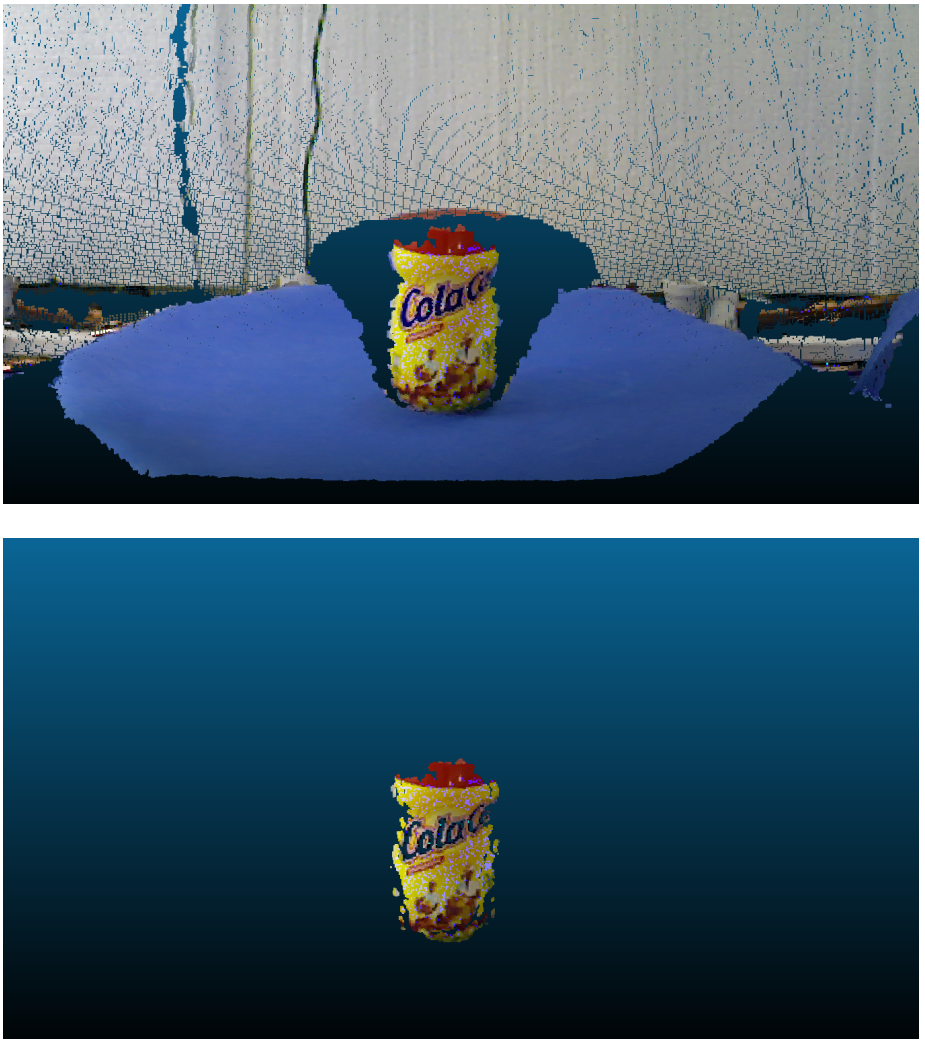


Figure A.4: Original point cloud of a partial view (top), segmented point cloud after preprocessing (bottom).

Removal filter as shown in Listing A.5.

Listing A.5: Radius Outlier Removal filtering.

```
1  pcl::RadiusOutlierRemoval<pcl::PointXYZRGB> ror;
2  ror.setInputCloud(pSrcCloud);
3  ror.setRadiusSearch(pRadius);
4  ror.setMinNeighborsInRadius(pNeighbors);
5  ror.filter(*pDstCloud);
```

This filter removes all points in the input cloud which do not have at least

a predefined number of neighbors within a certain radius. Figure A.5 helps to visualize what this filter does. The parameter configuration depends on the situation and nature of the noise so it is important to tune it properly in order to remove the noise while preserving valuable information.

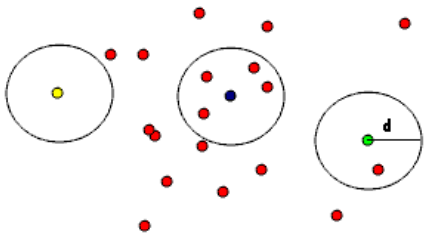


Figure A.5: RadiusOutlierRemoval filter. Using a radius *d* and only one neighbor the yellow point will be filtered, with two neighbors required per neighborhood both yellow and green points will be removed. (Figure reproduced from [89]).

Sometimes, ROR filtering is not enough or it is not able to properly remove the noise. In those cases, it is better to follow a statistical approach, namely *Statistical Outlier Removal* (SOR) filter.

Listing A.6 shows the integration of the SOR filter into our pipeline using the methods provided by the PCL.

Listing A.6: Statistical Outlier Removal filtering.

```
1  pcl::StatisticalOutlierRemoval<pcl::PointXYZRGB>sor;
2
3  sor.setInputCloud(pSrcCloud);
4  sor.setMeanK(pMeanK);
5  sor.setStddevMulThresh(pStdDev);
6
7  sor.filter(*pDstCloud);
```

This filter assumes that outliers can be removed by performing a statistical analysis on each point's neighborhood to discard those which do not meet a certain criteria. The SOR filter is based on the computation of the point to neighbors distance distribution in the input cloud. In this sense, the mean distance from each point to all its neighbors is computed. The filter assumes that the distance distribution is Gaussian with a predefined mean and standard deviation. Following this principle, those points whose mean distances are outside the interval defined by that deviation and mean are removed from the cloud. Figure A.6 helps to visualize this process.

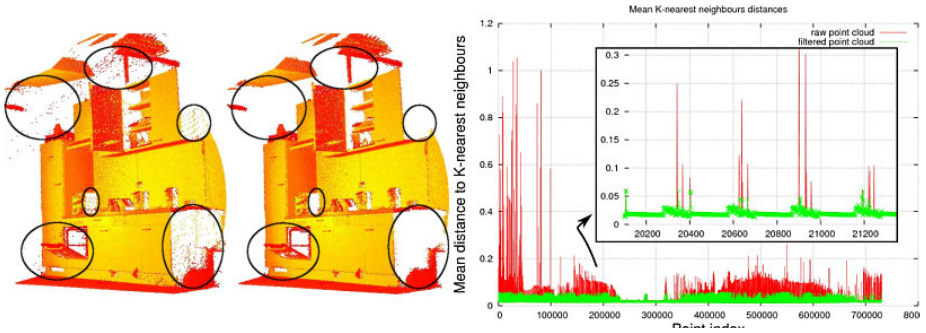


Figure A.6: Effects of the statistical outlier removal filter, original cloud (left) and filtered one (right). The plot shows the mean k-nearest neighbor distances in a point neighborhood before and after applying SOR filter (Figure reproduced from [88]).

Once these filters are applied, it is possible that some point clusters are generated if the cloud has dense outlier regions separated by sparse zones and we apply an aggressive filtering (see Figure A.8).

For those cases, it is important to remove those outlier clusters. In order to do that, we can assume that those cluster will be small and the big ones will contain the object itself. In that sense, we can apply a *Euclidean Cluster Extraction* (ECE) operation and keep the k biggest clusters. It is important to remark that there is a tradeoff between removing small clusters that may belong to the object and removing those that are noise. Listing A.7 shows the integration of that filter into our pipeline using PCL.

Listing A.7: Euclidean Cluster Extraction filter

```
pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree(
  new pcl::search::KdTree<pcl::PointXYZRGB>);
tree->setInputCloud(pSrcCloud);

std::vector<pcl::PointIndices> clusterIndices;
pcl::EuclideanClusterExtraction<pcl::PointXYZRGB> ec;
ec.setClusterTolerance(pClusterTolerance);
ec.setMinClusterSize(pMinClusterSize);
ec.setMaxClusterSize(pMaxClusterSize);
ec.setSearchMethod(tree);
ec.setInputCloud(pSrcCloud);
ec.extract(clusterIndices);

pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloudCluster(
  new pcl::PointCloud<pcl::PointXYZRGB>);

for (int i = 0; i < pClusters && !clusterIndices.empty(); ++i)
{
  int largestCluster = std::numeric_limits<int>::min();
  std::vector<pcl::PointIndices>::const_iterator largestClusterIt =
```

```
21      clusterIndices.begin();
22    int counter = 0;
23    for (auto it = clusterIndices.begin(); it != clusterIndices.end(); ++it)
24    {
25      if (it->indices.size() > largestCluster)
26      {
27        largestCluster = counter;
28        largestClusterIt = it;
29      }
30      counter++;
31    }
32
33    for (auto pit = clusterIndices[largestCluster].indices.begin();
34      pit != clusterIndices[largestCluster].indices.end(); pit++)
35      cloudCluster->points.push_back(pSrcCloud->points[*pit]);
36
37    cloudCluster->width = cloudCluster->points.size();
38    cloudCluster->height = 1;
39    cloudCluster->is_dense = true;
40
41    clusterIndices.erase(largestClusterIt);
42  }
43
44  pDstCloud->clear();
45  for (auto cit = cloudCluster->begin(); cit != cloudCluster->end(); ++cit)
46    pDstCloud->push_back(*cit);
47
48  pDstCloud->width = pDstCloud->points.size();
49  pDstCloud->height = 1;
50  pDstCloud->is_dense = true;
```

This method works like a flood fill algorithm. It chooses a point and initializes a new cluster. Then it checks the distance from that point to its neighbors and adds them to the cluster if the distance falls below a predefined threshold. Next, the added neighbors are checked against their corresponding ones until no more points can be added to that cluster. When this happens, a point which is not in a cluster is selected and the process starts again until all points are part of a cluster. Once the clusters are established, the k biggest ones are preserved. Figure A.7 shows a set of clusters extracted from a point cloud using ECE.

After removing the noise, we might consider simplifying the point cloud to reduce the computational cost of the following phases. This point cloud decimation is often performed by using the *Voxel Grid* filter. This algorithm creates a 3D voxel grid over the input cloud data and approximates all the points which are present in the same voxel with their centroid. Listing A.8 shows the integration of the voxel grid filter, the size of the voxels is determined by the *leaf size*.

Listing A.8: Voxel Grid filtering.

```
1   pcl::VoxelGrid<pcl::PointXYZRGB> vg;
2   vg.setInputCloud(pSrcCloud);
3   vg.setLeafSize(pLeafSize, pLeafSize, pLeafSize);
4   vg.filter(*pDstCloud);
```

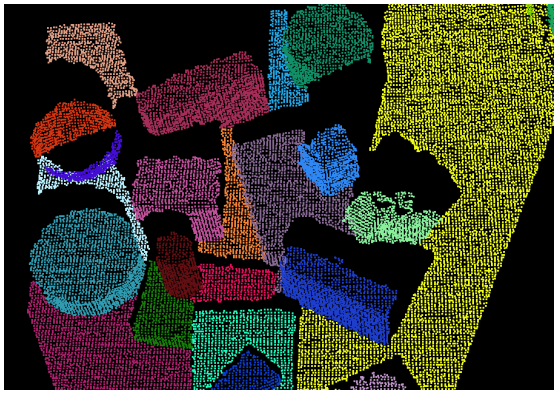


Figure A.7: Euclidean clusters extracted from a point cloud (Figure reproduced from *Pointclouds.org*).

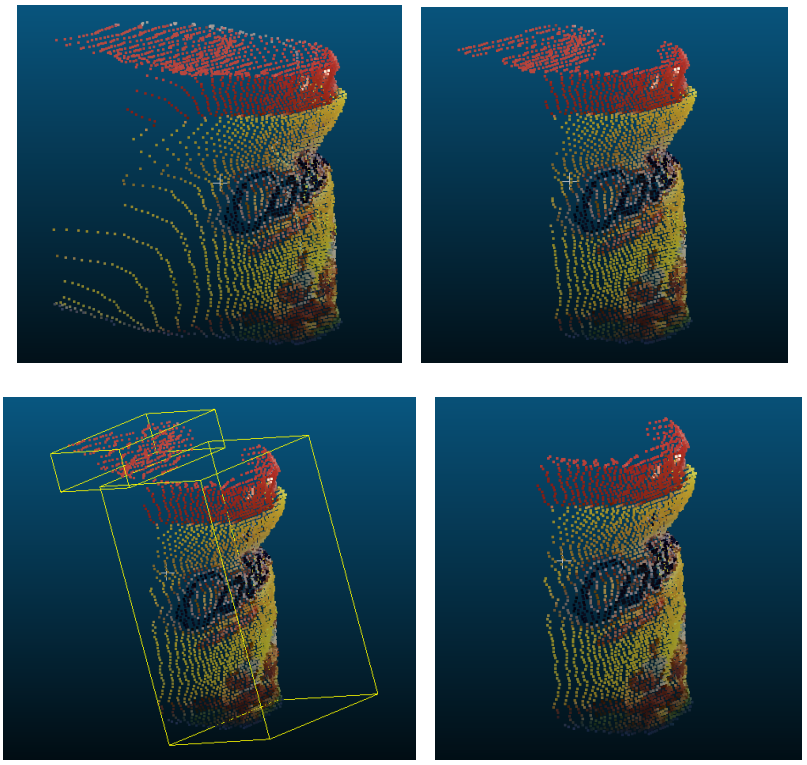


Figure A.8: Partial view with noise (top left), SOR/ROR filtered (top right), euclidean clusters detected (bottom left), keeping the biggest cluster (bottom right).

A.3.2 Transformation and registration

After all the preprocessing steps, the partial view is ready to be aligned with the model being built. In order to do that, first we will have to provide a coarse alignment which consists of rotating the partial view the same degrees the turntable has rotated from the starting position to capture that view.

To properly perform this rotation we need to translate all views to place the segmented object centered at a common origin. This is achieved by moving the origin of each view (the sensor) to the center of the turntable. For this purpose, we placed a marker of the actual center on the turntable and captured an empty view. Then we manually picked the marker point in the cloud and extracted its coordinates, as shown in Figure A.9.

That point is used to translate all partial views and modify their origin using the transformation shown in Listing A.9.

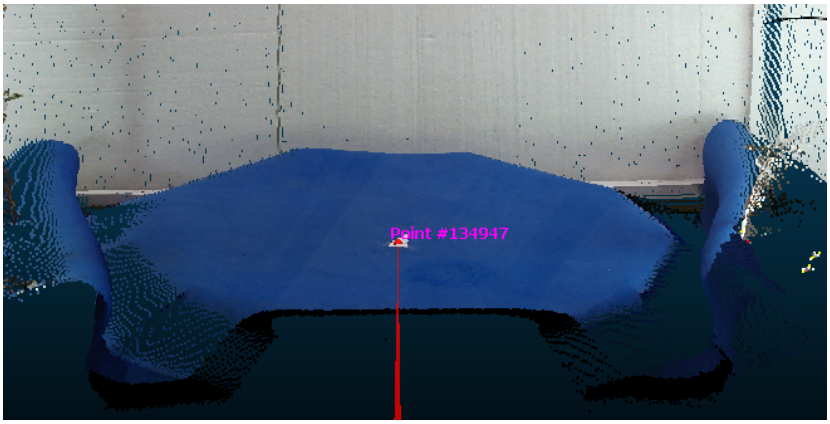


Figure A.9: Turntable center picked from an empty view.

Listing A.9: Transformation to translate a point cloud.

```
1  Eigen::Affine3f transform = Eigen::Affine3f::Identity();
2  transform.translation() << pTranslation[0], pTranslation[1], pTranslation[2];
3  pcl::transformPointCloud(*pSrcCloud, *pDstCloud, transform);
```

After that, we need an axis to rotate the view. If we use no custom axis, the object will not rotate around itself since the sensor's Z-axis is not parallel to the turntable plane normal. In this sense, we will obtain the normal of the plane of the turntable to rotate the view around it. In order to do that, we fitted a plane model to a section of the turntable and extracted its normal vector (see Figure A.10).

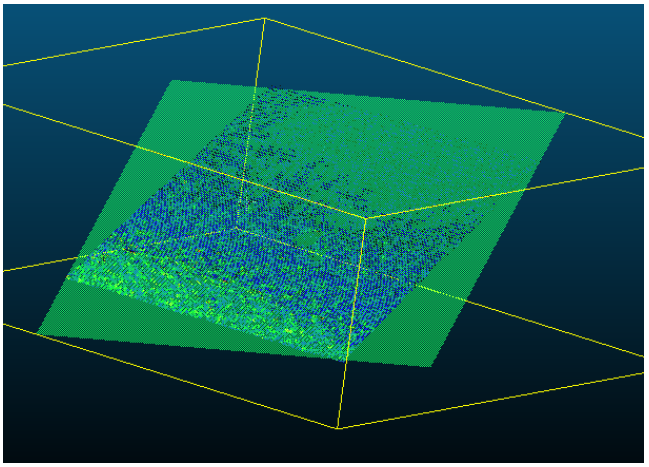


Figure A.10: Turntable section plane fitting.

That axis is used to rotate the current view using the transformation shown in Listing A.10. This works as some sort of odometry (taking into account the movement of the turntable, i.e., the first view will not be rotated, the second view will be rotated 5.625 °, the third one 11.25 ° and so on) to provide a coarse alignment.

Listing A.10: Transformation to rotate a point cloud.

```
1  float theta = pcl::deg2rad(pAngle);
2  Eigen::Affine3f transform = Eigen::Affine3f::Identity();
3  transform.translation() << 0.0, 0.0, 0.0;
4  transform.rotate(Eigen::AngleAxisf(theta, pAxis));
5  pcl::transformPointCloud(*pSrcCloud, *pDstCloud, transform);
```

Once the coarse transformation is applied, we refine it using the popular *Iterative Closest Point* algorithm [73] [74]. The ICP is a widely used algorithm for fine geometric alignment of 3D point clouds when a coarse initial estimation of the pose is known. The algorithm starts with two point clouds and an initial guess of their relative rigid transformation which is iteratively refined by repeatedly generating pairs of corresponding points using a closeness criteria and minimizing a distance error metric. Many variants of the algorithm have been proposed whose goal is to increase the efficiency or precision of the method, the seminal work of Rusinkiewicz and Levoy [75] surveys and carries out a detailed comparison of the most relevant ones. We used a basic brute-force implementation [92] shown in Listing A.11. Its results can be seen in Figure A.11.

Listing A.11: ICP alignment with PCL.

```
pcl::IterativeClosestPoint<pcl::PointXYZRGB, pcl::PointXYZRGB> icp;
icp.setInputSource(pSrcCloud);
icp.setInputTarget(pTgtCloud);
icp.setMaximumIterations(pMaxIterations);
icp.setTransformationEpsilon(pEpsilon);
icp.align(*pDstCloud);
```

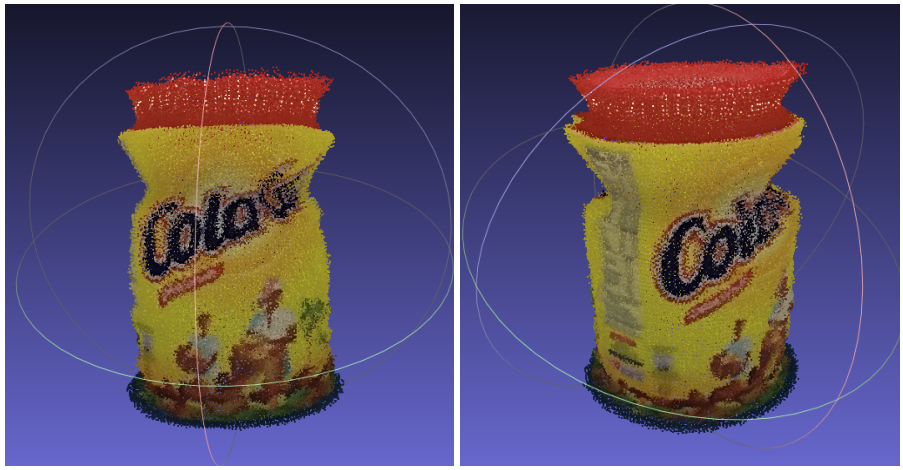


Figure A.11: Fully registered point clouds of an object.

A.3.3 Mesh reconstruction

The last phase of the reconstruction process is the actual mesh generation using the previously registered point cloud. There are many approaches to generate a triangular mesh from a dense point cloud. Two of the most popular ones are the *Greedy Projection Triangulation* (GPT)[93] [94] and the *Poisson Mesh Reconstruction* (PMR) [95] [96]. In our case, we have chosen the latter one because of its built-in smoothing component.

Listing A.12 shows the code needed to perform a *Poisson Mesh Reconstruction*, including extra required steps, given the previously registered point cloud.

Listing A.12: Poisson Mesh Reconstruction with PCL.

```
pcl::MovingLeastSquares<pcl::PointXYZRGB, pcl::PointXYZRGB> mls;
mls.setInputCloud(pSrcCloud);
mls.setSearchRadius(pMlsSearchRadius);
mls.setPolynomialFit(pMlsPolynomialFit);
mls.setPolynomialOrder(pMlsPolynomialOrder);
mls.setUpsamplingMethod(
```

```
7     pcl::MovingLeastSquares<pcl::PointXYZRGB,
8        pcl::PointXYZRGB>::SAMPLE_LOCAL_PLANE);
9   mls.setUpsamplingRadius(pMlsUpsamplingRadius);
10  mls.setUpsamplingStepSize(pMlsUpsamplingStepSize);
11
12  pcl::PointCloud<pcl::PointXYZRGB>::Ptr smoothedCloud(
13    new pcl::PointCloud<pcl::PointXYZRGB>());
14
15  mls.process(*smoothedCloud);
16
17  pcl::NormalEstimation<pcl::PointXYZRGB, pcl::Normal> ne;
18
19  pcl::search::KdTree<pcl::PointXYZRGB>::Ptr tree(
20    new pcl::search::KdTree<pcl::PointXYZRGB>());
21
22  ne.setSearchMethod(tree);
23  ne.setInputCloud(smoothedCloud);
24  ne.setRadiusSearch(pNeRadiusSearch);
25
26  Eigen::Vector4f centroid;
27
28  pcl::compute3DCentroid(*smoothedCloud, centroid);
29  ne.setViewPoint(centroid[0], centroid[1], centroid[2]);
30
31  pcl::PointCloud<pcl::Normal>::Ptr cloudNormals(
32    new pcl::PointCloud<pcl::Normal>());
33  ne.compute(*cloudNormals);
34
35  for (auto cit = cloudNormals->begin(); cit != cloudNormals->end(); ++cit)
36  {
37    (*cit).normal_x *= -1;
38
39    (*cit).normal_y *= -1;
40
41    (*cit).normal_z *= -1;
42  }
43
44  pcl::PointCloud<pcl::PointXYZRGBNormal>::Ptr cloudSmoothedNormals(
45    new pcl::PointCloud<pcl::PointXYZRGBNormal>());
46
47  pcl::concatenateFields<pcl::PointXYZRGB, pcl::Normal,
48    pcl::PointXYZRGBNormal>(
49    *smoothedCloud, *cloudNormals, *cloudSmoothedNormals);
50
51  pcl::Poisson<pcl::PointXYZRGBNormal> poisson;
52
53  poisson.setDepth(pPoissonDepth);
54
55  poisson.setInputCloud(cloudSmoothedNormals);
56
57  poisson.reconstruct(pDstMesh);
```

As we can observe, first we apply a *Moving Least Squares* [97] surface reconstruction method to smooth and upsample the point cloud. This resampling serves a double purpose: it removes noisy points and it also fills sparse regions of the point cloud thus smoothing it. This algorithm attempts to recreate the missing parts of the surface using analytical functions and polynomial fitting. It also helps correcting the known double-wall (multiple surfaces) artifacts which are usually a product from registering multiple scans.

Before starting the PMR process, we need to compute the normals for each point of the cloud. This step is not trivial since we are dealing with a com-

plete point cloud with no point of view in particular so we do not know where to orient the normals. A common approach is to compute the centroid of the cloud, estimate the normals for each point using the centroid as the viewpoint and reorient all of them to make sure they point outwards the object by negating their components. The most popular estimation method approximates the normal to a point on the surface by estimating the normal of the tangent plane to the surface which is also estimated by fitting a plane to the neighborhood of the point whose normal is being estimated using least squares fitting. The estimation of this normal is reduced to a Principal Component Analysis (PCA) of the covariance matrix created using the neighbors of the point whose normal is being estimated.

In the end, we provide the PMR method with the reconstructed point cloud together with the normal information by concatenating both point clouds into a single one (XYZRGBNormal). The *Poisson Mesh Reconstruction* algorithm works by expressing the surface reconstruction problem as the solution to a Poisson equation [98]. It uses an implicit solution to approximate the surface and then extracts the isosurface using a variation of the Marching Cubes [99] algorithm. An octree structure is applied to approximate the surface so various levels of detail can be obtained by changing the depth of the octree.

Figure A.12 shows two meshes reconstructed using the previously defined *Poisson Mesh Reconstruction* method with different depths for the octree. Both meshes were reconstructed using the point clouds shown in Figure A.11. As we can observe, this method is quite sensitive to noise so finding the right LOD is critical to avoid bad surface reconstructions due to excessive detail. It is important to remark that the clouds shown in Figure A.11 were not simplified using a voxel grid filter.

A.4 Conclusions

In this appendix, we have presented a full pipeline for reconstructing object meshes from a set of partial views taken using a turntable. An acquisition system has been implemented using PCL/OpenNI and a typical reconstruction process has also been developed with the *Point Cloud Library*. In the end, a 3D mesh in PLY format is generated as a model representing the real-life object.

As we have seen, the system works remarkably well thanks to knowing the turntable displacement and the effectiveness of noise reduction techniques. However, it still has a few weaknesses or possible improvements.

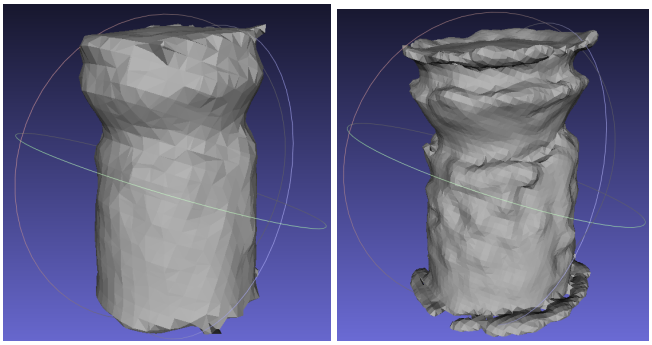


Figure A.12: Reconstructed meshes with PMR and depth values for the Poisson algorithm of 5 (left) and 7 (right).

First, the system relies on the turntable as a controlled environment with a fixed camera; in this sense, it is possible that some parts of an object can't be captured. One possible improvement consists of allowing the camera to freely move around the object and estimate the transformation between two poses using a typical descriptor matching pipeline. The estimated camera transformation is provided for a coarse alignment instead of the turntable rotation. This would allow the system to capture more details of the objects and increase the quality of the reconstruction.

We have seen that noise has a critical impact on the reconstruction so reducing it is a fundamental step of the pipeline. We have applied numerous techniques but there are many left to test. For instance, we could integrate a bilateral filter.

In order to estimate the normals of a fully registered point cloud we used a simple trick which consisted of flipping the normals outwards the centroid of the object. This trick is not assured to work on all objects and geometries. One possible upgrade is including some consistency propagation technique for normal orientations [100].

It is important to remark that the final generated mesh lacks color information. In this sense, a great upgrade could be including some kind of mapping and interpolation techniques to color the mesh using the point cloud data.

At last, if the main concern for the application is the efficiency of the system, most of the aforementioned steps are inherently parallel and perfectly suited for multi-core or SIMT (CUDA/OpenCL/C++AMP) implementations which can greatly reduce the execution time of the system and even achieve a real-time pipeline.

Bibliography

[1] K. Mikolajczyk and C. Schmid, "A performance evaluation of local descriptors," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 10, pp. 1615–1630, Oct 2005.

[2] Y. Guo, M. Bennamoun, F. Sohel, M. Lu, and J. Wan, "3d object recognition in cluttered scenes with local surface features: A survey," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 36, no. 11, pp. 2270–2287, Nov 2014.

[3] Z. Zhang, "Microsoft kinect sensor and its effect," *MultiMedia, IEEE*, vol. 19, no. 2, pp. 4–10, Feb 2012.

[4] NVIDIA. (2014) Technical Brief NVIDIA Jetson TK1 Development Kit Bringing GPU-accelerated computing to Embedded Systems. http://developer.download.nvidia.com/embedded/jetson/TK1/docs/Jetson_platform_brief_May2014.pdf.

[5] J. Ponce, S. Lazebnik, F. Rothganger, and C. Schmid, "Toward true 3d object recognition," in *Reconnaissance de Formes et Intelligence Artificielle*, 2004.

[6] R. J. Campbell and P. J. Flynn, "A survey of free-form object representation and recognition techniques," *Computer Vision and Image Understanding*, vol. 81, no. 2, pp. 166–210, 2001.

[7] A. Andreopoulos and J. K. Tsotsos, "50 years of object recognition: Directions forward," *Computer Vision and Image Understanding*, vol. 117, no. 8, pp. 827–891, 2013.

[8] U. Castellani, M. Cristani, S. Fantoni, and V. Murino, "Sparse points matching by combining 3d mesh saliency with statistical descriptors," in *Computer Graphics Forum*, vol. 27, no. 2. Wiley Online Library, 2008, pp. 643–652.

[9] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, vol. 2. Ieee, 1999, pp. 1150–1157.

[10] G. Foresti, "Object recognition and tracking for remote video surveillance," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 9, no. 7, pp. 1045–1062, Oct 1999.

[11] J. Wu and Z. Xiao, "Video surveillance object recognition based on shape and color features," in *Image and Signal Processing (CISP), 2010 3rd International Congress on*, vol. 1, Oct 2010, pp. 451–454.

[12] J. Stuckler and S. Behnke, "Integrating indoor mobility, object manipulation, and intuitive interaction for domestic service tasks," in *Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on*, Dec 2009, pp. 506–513.

[13] Y. Lei, M. Bennamoun, M. Hayat, and Y. Guo, "An efficient 3d face recognition approach using local geometrical signatures," *Pattern Recognition*, vol. 47, no. 2, pp. 509 – 524, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320313003166

[14] F. Sukno, J. Waddington, and P. Whelan, "Comparing 3d descriptors for local search of craniofacial landmarks," in *Advances in Visual Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7432, pp. 92–103. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33191-6_10

[15] A. Mian, M. Bennamoun, and R. Owens, "Three-dimensional model-based object recognition and segmentation in cluttered scenes," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 28, no. 10, pp. 1584–1601, Oct 2006.

[16] A. S. Mian, M. Bennamoun, and R. A. Owens, "A novel representation and feature matching algorithm for automatic pairwise registration of range images," *International Journal of Computer Vision*, vol. 66, no. 1, pp. 19–40, 2006.

[17] S. Orts-Escolano, V. Morell, J. Garcia-Rodriguez, M. Cazorla, and R. Fisher, "Real-time 3d semi-local surface patch extraction using gpgpu," *Journal of Real-Time Image Processing*, pp. 1–20, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11554-013-0385-7

[18] Y. Hirano, C. Garcia, R. Sukthankar, and A. Hoogs, "Industry and object recognition: Applications, applied research and challenges," in *Toward Category-Level Object Recognition*, ser. Lecture Notes in Computer Science, J. Ponce, M. Hebert, C. Schmid, and A. Zisserman, Eds. Springer Berlin Heidelberg, 2006, vol. 4170, pp. 49–64. [Online]. Available: http://dx.doi.org/10.1007/11957959_3

[19] P. J. Besl and R. C. Jain, "Three-dimensional object recognition," *ACM Computing Surveys (CSUR)*, vol. 17, no. 1, pp. 75–145, 1985.

[20] J. Brady, N. Nandhakumar, and J. Aggarwal, "Recent progress in the recognition of objects from range data," in *Pattern Recognition, 1988., 9th International Conference on*, Nov 1988, pp. 85–92.

[21] F. Arman and J. Aggarwal, "Model-based object recognition in dense-range images—a review," *ACM Computing Surveys (CSUR)*, vol. 25, no. 1, pp. 5–43, 1993.

[22] G. Mamic and M. Bennamoun, "Representation and recognition of 3d free-form objects," *Digital Signal Processing*, vol. 12, no. 1, pp. 47–76, 2002.

[23] A. Aldoma, Z.-C. Marton, F. Tombari, W. Wohlkinger, C. Potthast, B. Zeisl, R. Rusu, S. Gedikli, and M. Vincze, "Tutorial: Point cloud library: Three-dimensional object recognition and 6 dof pose estimation," *Robotics Automation Magazine, IEEE*, vol. 19, no. 3, pp. 80–91, Sept 2012.

[24] S. Orts Escolano, "A three-dimensional representation method for noisy point clouds based on growing self-organizing maps accelerated on gpus," 2014.

[25] A. Mian, M. Bennamoun, and R. Owens, "On the repeatability and quality of keypoints for local feature-based 3d object retrieval from cluttered scenes," *International Journal of Computer Vision*, vol. 89, no. 2-3, pp. 348–361, 2010. [Online]. Available: http://dx.doi.org/10.1007/s11263-009-0296-z

[26] F. Tombari, S. Salti, and L. Di Stefano, "Performance evaluation of 3d keypoint detectors," *International Journal of Computer Vision*, vol. 102, no. 1-3, pp. 198–220, 2013.

[27] A. Bronstein, M. Bronstein, and M. Ovsjanikov, "3d features, surface descriptors, and object descriptors," 2010.

[28] F. Tombari. (2014) Keypoints and features. http://www.pointclouds.org/assets/uploads/cglibs13_features.pdf.

[29] A. Petrelli and L. Di Stefano, "On the repeatability of the local reference frame for partial shape matching," in *Computer Vision (ICCV), 2011 IEEE International Conference on*, Nov 2011, pp. 2244–2251.

[30] F. Tombari, S. Salti, and L. Di Stefano, "Unique signatures of histograms for local surface description," in *Proceedings of the 11th European Conference on Computer Vision Conference on Computer Vision: Part III*, ser. ECCV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 356–369. [Online]. Available: http://dl.acm.org/citation.cfm?id=1927006.1927035

[31] M. Levoy, J. Gerth, B. Curless, and K. Pull, "The stanford 3d scanning repository," *URL http://www-graphics. stanford. edu/data/3dscanrep*, 2005.

[32] A. Johnson, "Spin-images: A representation for 3-d surface matching," Ph.D. dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, August 1997.

[33] R. B. Rusu, N. Blodow, Z. C. Marton, and M. Beetz, "Close-range scene segmentation and reconstruction of 3d point cloud maps for mobile manipulation in domestic environments," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*. IEEE, 2009, pp. 1–6.

[34] R. Rusu, N. Blodow, Z. Marton, and M. Beetz, "Aligning point cloud views using persistent feature histograms," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, Sept 2008, pp. 3384–3391.

[35] A. Frome, D. Huber, R. Kolluri, T. Bülow, and J. Malik, "Recognizing objects in range data using regional point descriptors," in *Computer Vision - ECCV 2004*, ser. Lecture Notes in Computer Science, T. Pajdla and J. Matas, Eds. Springer Berlin Heidelberg, 2004, vol. 3023, pp. 224–237. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24672-5_18

[36] S. Belongie, J. Malik, and J. Puzicha, "Shape matching and object recognition using shape contexts," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, no. 4, pp. 509–522, Apr 2002.

[37] F. Tombari, S. Salti, and L. Di Stefano, "Unique shape context for 3d data description," in *Proceedings of the ACM Workshop on 3D Object Retrieval*, ser. 3DOR '10. New York, NY, USA: ACM, 2010, pp. 57–62. [Online]. Available: http://doi.acm.org/10.1145/1877808.1877821

[38] Y. Guo, F. Sohel, M. Bennamoun, M. Lu, and J. Wan, "Rotational projection statistics for 3d local surface description and object recognition," *International journal of computer vision*, vol. 105, no. 1, pp. 63–86, 2013.

[39] Y. Guo, F. Sohel, M. Bennamoun, J. Wan, and M. Lu, "A novel local surface feature for 3d object recognition under clutter and occlusion," *Information Sciences*, vol. 293, pp. 196–213, 2015.

[40] M. Quigley, S. Batra, S. Gould, E. Klingbeil, Q. V. Le, A. Wellman, and A. Y. Ng, "High-accuracy 3d sensing for mobile manipulation: Improving object detection and door opening." in *ICRA*, 2009, pp. 2816–2822.

[41] M. Cazorla, D. Viejo, and C. Pomares, "Study of the sr 4000 camera," in *XI Workshop de Agentes Físicos, Valencia*, 2010.

[42] M. Imaging. (2010) Sr4000 data sheet. http://downloads.mesa-imaging.ch/dlm.php?fname=pdf/SR4000_Data_Sheet.pdf.

[43] J. Garcia and Z. Zalevsky, "Range mapping using speckle decorrelation," Oct. 7 2008, uS Patent 7,433,024.

[44] A. Shpunt, G. Medioni, D. Cohen, E. Sali, and R. Deitch, "Depth mapping based on pattern matching and stereoscopic information," Jul. 28 2010, uS Patent App. 12/844,864.

[45] Y. Arieli, B. Freedman, M. Machline, and A. Shpunt, "Depth mapping using projected patterns," Apr. 3 2012, uS Patent 8,150,142.

[46] I. Bramão, A. Reis, K. M. Petersson, and L. Faísca, "The role of color information on object recognition: A review and meta-analysis," *Acta Psychologica*, vol. 138, no. 1, pp. 244 – 253, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0001691811001338

[47] F. Tombari, S. Salti, and L. Di Stefano, "A combined texture-shape descriptor for enhanced 3d feature matching," in *Image Processing (ICIP), 2011 18th IEEE International Conference on*, Sept 2011, pp. 809–812.

[48] Occipital. (2013) Openni programmer's guide. http://com.occipital.openni.s3.amazonaws.com/OpenNI_Programmers_Guide.pdf.

[49] R. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, May 2011, pp. 1–4.

[50] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[51] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[52] NVIDIA. (2014) NVIDIA Jetson TK1 Documentation PM375 Module Specification. http://developer.download.nvidia.com/embedded/jetson/TK1/2014-03-24/JetsonTK1_ModuleSpecification_PM375_V1.01.pdf.

[53] ——. (2014) NVIDIA Jetson TK1 Software Developer Kit Quick Start Guide. http://developer.download.nvidia.com/embedded/jetson/TK1/docs/Jetson_TK1_QSG_134sq_Jun14_rev7.pdf.

[54] ——. (2014) Tegra K1 Technical Reference Manual. https://developer.nvidia.com/tegra-k1-technical-reference-manual.

[55] ——. (2011) Variable SMP (4-PLUS-1) A Multi-Core CPU Architecture for Low Power and High Performance. http://www.nvidia.com/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-Core-CPU-Architecture-for-Low-Power-and-High-Performance.pdf.

[56] ——. (2011) The Benefits of Quad Core CPUs in Mobile Devices. http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911a.pdf.

[57] ——. (2012) NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. http://www.nvidia.es/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[58] K. Khoshelham and S. O. Elberink, "Accuracy and resolution of kinect depth data for indoor mapping applications," *Sensors*, vol. 12, no. 2, pp. 1437–1454, 2012.

[59] B. Freedman, A. Shpunt, M. Machline, and Y. Arieli, "Depth mapping using projected patterns," Apr. 3 2012, uS Patent 8,150,142.

[60] T. Butkiewicz, "Low-cost coastal mapping using kinect v2 time-of-flight cameras," in *Oceans - St. John's, 2014*, Sept 2014, pp. 1–9.

[61] D. L. Lau. (2013) The Science Behind Kinect. http://lau.engineering.uky.edu/2013/11/27/the-science-behind-kinect/.

[62] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Computer Vision, 1998. Sixth International Conference on*. IEEE, 1998, pp. 839–846.

[63] A. J. Trevor, S. Gedikli, R. B. Rusu, and H. I. Christensen, "Efficient organized point cloud segmentation with connected components," *Semantic Perception Mapping and Exploration (SPME)*, 2013.

[64] R. Szeliski, *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[65] P. F. Felzenszwalb and D. P. Huttenlocher, "Efficient graph-based image segmentation," *International Journal of Computer Vision*, vol. 59, no. 2, pp. 167–181, 2004.

[66] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[67] Y. Zhong, "Intrinsic shape signatures: A shape descriptor for 3d object recognition," in *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, Sept 2009, pp. 689–696.

[68] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[69] A. Flint, A. R. Dick, and A. Van Den Hengel, "Thrift: Local 3d structure recognition." in *DICTA*, vol. 7, 2007, pp. 182–188.

[70] Z. C. Marton, R. B. Rusu, and M. Beetz, "On Fast Surface Reconstruction Methods for Large and Noisy Datasets," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 12-17 2009.

[71] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 36, 2014.

[72] H. Chen and B. Bhanu, "3d free-form object recognition in range images using local surface patches," in *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, vol. 3, Aug 2004, pp. 136–139 Vol.3.

[73] P. J. Besl and N. D. McKay, "Method for registration of 3-d shapes," in *Robotics-DL tentative*. International Society for Optics and Photonics, 1992, pp. 586–606.

[74] Y. Chen and G. Medioni, "Object modeling by registration of multiple range images," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*. IEEE, 1991, pp. 2724–2729.

[75] S. Rusinkiewicz and M. Levoy, "Efficient variants of the icp algorithm," in *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*. IEEE, 2001, pp. 145–152.

[76] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 2004, pp. 253–262.

[77] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *VLDB*, vol. 98, 1998, pp. 194–205.

[78] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, vol. 99, 1999, pp. 518–529.

[79] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 36, 2014.

[80] C. Silpa-Anan and R. Hartley, "Optimised kd-trees for fast image descriptor matching," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*. IEEE, 2008, pp. 1–8.

[81] N. Wilt, *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[82] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[83] PCL. (2013) The openni grabber framework in pcl. http://pointclouds.org/documentation/tutorials/openni_grabber.php.

[84] NVIDIA. (2009) Advanced cuda webinar: Memory optimizations. http://on-demand.gputechconf.com/gtc-express/2011/presentations/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf.

[85] O. Consortium, "Openni, the standard framework for 3d sensing."

[86] PCL. (2013) Filtering a pointcloud using a passthrough filter. http://pointclouds.org/documentation/tutorials/passthrough.php.

[87] ——. (2013) Plane model segmentation. http://pointclouds.org/documentation/tutorials/planar_segmentation.php.

[88] ——. (2013) Removing outliers using a statisticaloutlierremoval filter. http://pointclouds.org/documentation/tutorials/statistical_outlier.php.

[89] ——. (2013) Removing outliers using a conditional or radiusoutlier removal. http://pointclouds.org/documentation/tutorials/remove_outliers.php.

[90] ——. (2013) Euclidean cluster extraction. http://www.pointclouds.org/documentation/tutorials/cluster_extraction.php.

[91] ——. (2013) Downsampling a pointcloud using a voxelgrid filter. http://pointclouds.org/documentation/tutorials/voxel_grid.php.

[92] ——. (2013) How to use iterative closest point. http://pointclouds.org/documentation/tutorials/iterative_closest_point.php.

[93] Z. C. Marton, R. B. Rusu, and M. Beetz, "On Fast Surface Reconstruction Methods for Large and Noisy Datasets," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Kobe, Japan, May 12-17 2009.

[94] PCL. (2011) Fast triangulation of unordered point clouds. http://pointclouds.org/documentation/tutorials/greedy_projection.php.

[95] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," in *Proceedings of the fourth Eurographics symposium on Geometry processing*, vol. 7, 2006.

[96] V. Rabaud. (2011) Surface reconstruction. http://www.pointclouds.org/assets/files/presentations/ICCV2011-surface.pdf.

[97] PCL. (2011) Smoothing and normal estimation based on polynomial reconstruction. http://pointclouds.org/documentation/tutorials/resampling.php.

[98] J. H. P. Ono, A. C. Sementille, M. A. C. Caldeira, and J. F. Marar, "Poisson surface reconstruction with local mesh simplification."

[99] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *ACM siggraph computer graphics*, vol. 21, no. 4. ACM, 1987, pp. 163–169.

[100] S. König, S. Gumhold, and T. Dresden, "Consistent propagation of normal orientations in point clouds," 2009.