


Doctoral Dissertation

Doctoral Program in Computer And Control Engineering (30.th cycle)

Physical Interaction of Autonomous Robots in Complex Environments

Giorgio Toscana

* * * * * *

Supervisors

Prof. Basilio Bona, Supervisor

Ing. Marco Gaspardone, TIM

PhD. Stefano Rosa

Doctoral Examination Committee:

Prof. Silvia Maria Zanoli, Referee, Università Politecnica delle Marche

Prof. Matteo Matteucci, Referee, Politecnico di Milano

Prof. Davide Brugali, Università degli Studi di Bergamo

Prof. Marina Indri, Politecnico di Torino

Prof. Alessandro Rizzo, Politecnico di Torino

Politecnico di Torino

September 29, 2018

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Giorgio Toscana

Turin, September 29, 2018

Summary

Recent breakthroughs in the fields of computer vision and robotics are firmly changing the people perception about robots. The idea of robots that substitute humans is now turning into robots that collaborate with them. Service robotics considers robots as personal assistants. It safely places robots in domestic environments in order to facilitate humans daily life. Industrial robotics is now reconsidering its basic idea of robot as a worker. Currently, the primary method to guarantee the personnels safety in industrial environments is the installation of physical barriers around the working area of robots. The development of new technologies and new algorithms in the sensor field and in the robotic one has led to a new generation of lightweight and collaborative robots. Therefore, industrial robotics leveraged the intrinsic properties of this kind of robots to generate a robot co-worker that is able to safely coexist, collaborate and interact inside its workspace with both personnels and objects.

This Ph.D. dissertation focuses on the generation of a pipeline for fast object pose estimation and distance computation of moving objects, in both structured and unstructured environments, using RGB-D images. This pipeline outputs the command actions which let the robot complete its main task and fulfil the safety human-robot coexistence behaviour at once.

The proposed pipeline is divided into an object segmentation part, a 6 D.o.F. object pose estimation part and a real-time collision avoidance part for safe human-robot coexistence.

Firstly, the segmentation module finds candidate object clusters out of RGB-D images of clutter scenes using a graph-based image segmentation technique. This segmentation technique generates a cluster of pixels for each object found in the image.

The candidate object clusters are then fed as input to the 6 D.o.F. object pose estimation module. The latter is in charge of estimating both the translation and the orientation in 3D space of each candidate object clusters. The object pose is then employed by the robotic arm to compute a suitable grasping policy.

The last module generates a force vector field of the environment surrounding the robot, the objects and the humans. This force vector field drives the robot toward its goal while any potential collision against objects and/or humans is safely avoided.

This work has been carried out at Politecnico di Torino, in collaboration with Telecom Italia S.p.A.

Contents

List of Tables

List of Figures

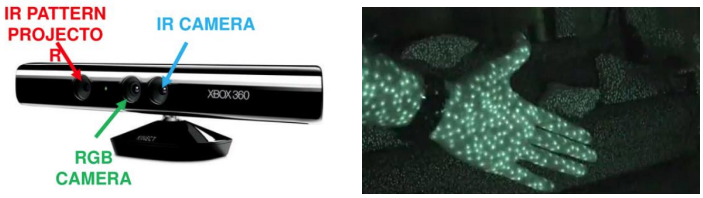
Chapter 1

Introduction

Detection and pose estimation of 3D objects is of great importance in current robotics applications, in both structured and unstructured environments, for many high level tasks such as manipulation, grasping, localization, mapping and human-robot interaction. Affordable RGB-D sensors, like the Microsoft Kinect™ [40] (Fig. 1.1a), have been of great interest to the robotics community. These sensors are able to simultaneously capture high-resolution color and *depth images* (RGB-D images) at high frame rates. Figure 1.2 shows a couple of images taken by the Kinect sensor. In particular, figure 1.2(a) shows a standard RGB color image, while figure 1.2(b) depicts the *depth image* of the same scene. The depth image is generated by the Kinect through the structured light technique. An IR projector projects on the scene a known pattern of infrared light as shown in Fig. 1.1(b). The Kinect on board IR camera captures the deformed pattern which will be analyzed by the internal Kinect software. The latter produces and time-synchronizes the depth image with the RGB one. An external calibration between the RGB and IR camera is required to have a one-to-one mapping among the pixels of the two images. A depth image pixel can be modeled as 3D vector of the form:

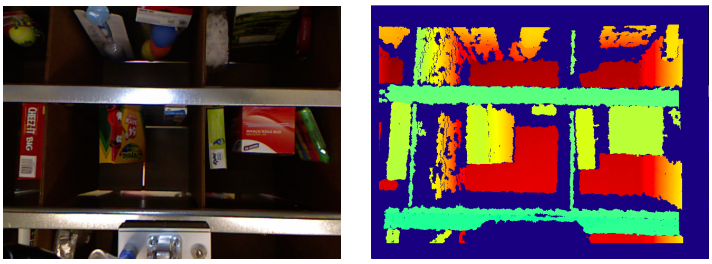
$$\mathbf{p}_{x_i} = [u_i, v_i, Z_i]^T \tag{1.1}$$

where (u_i, v_i) are the pixel coordinates in the image plane, whilst Z_i is the depth information, in millimetres, carried by the *i-th* pixel.



(a) (b)

Figure 1.1: (a) Microsoft Kinect [36]. (b) IR pattern [30]



(a) (b)

Figure 1.2: Synchronized RGB Image (a) and color coded Depth Image (b) captured by the Microsoft Kinect sensor

A depth image can be easily visualized as a RGB image. Figure 1.2(b) is thus obtained by converting the *i*-th depth image pixel into an 8-bit RGB value. Pixels far from the camera produce a color shift toward red while moving closer to the camera the pixels fade into blue color.

Usually the depth map presents some shadows in which the depth information cannot be extracted. E.g. the dark blue pixels in both figures 1.2(b) and 2.1(b) are depth map surfaces where we are not able to retrieve any depth data. For small areas, an in-paint process can be performed to reconstruct the missing depth information. Fig. 2.1(c) depicts the result of the depth in-paint process. For wider shadows the in-paint technique produces large depth estimation errors. This leads

(a) (b)

Figure 1.3: Colored PointCloud examples of different household objects

to an unusable depth image. In Chapter 2 we explain how to overcome this problem without in-painting depth images with wide shadow areas.

A depth image may also be employed to generate a *Point Cloud* of the scene captured by the camera. A Point Cloud is a set of 3D points in space that discretizes the surrounding environment. Each point belonging to a Point Cloud carries its own three-dimensional position information w.r.t. a reference frame (usually it is the reference frame of the IR camera). Besides, each point can hold other data such as its 3D color vector and its 3D normal vector.

Both figures 1.3 and 2.3(d)-(f) depict colored Point Clouds of household objects. Those Point Clouds are computed through their corresponding depth images as explained in Chapter 5.

In this work we focus on the problem of object pose estimation for robotic grasping, and in particular we are interested in the *Amazon Picking Challenge* (APC) [4] scenario. A well known approach for object pose estimation is LINEMOD [23]. It exploits both depth and color to capture the appearance and 3D shape of objects using a set of templates to represent different views of the object. Since the viewpoint of each template is known, a coarse estimate of the object pose can be obtained at the time of detection. However, pose estimates are not very precise, since each template covers a range of views around its viewpoint. 3D object models were exploited [25] in order to improve pose estimation accuracy and increase

robustness. In [59], LINEMOD is extented to be a scale-invariant patch descriptor and integrated into a regression forest, which is trained with positive samples only. Recent tests of LINEMOD showed only 32% accuracy rates in an APC-like scenario [50]. Other commonly used approaches such as tabletop from the Point Cloud Library [3] are based on a combination of coarse detection using 3D feature descriptors and fine pose estimation using ICP (Iterative Closest Point). Some of these approaches rely on object textures an are not suitable for many common texture-less objects. Moreover, all these methods are limited to objects lying on a flat tabletop, and are not robust to occlusions. In [26] the approach was extended with a voting procedure based on hashing for selecting candidate templates. In [7], random forests are trained on local features, while in [6] occlusion information is also added in the learning phase. Other commonly used approaches, such as tabletop from the Point Cloud Library [3], are based on a combination of coarse detection using 3D feature descriptors and fine pose estimation using ICP. Most of these approaches rely on object textures and are not suitable for many common texture-less objects. Moreover, these methods are limited to objects lying on a flat tabletop, and are generally not robust to occlusions. Given the difficulty to gather large annotated datasets necessary for training learning-based methods, such as Convolutional Neural Networks, and the long training times associated with learning, we propose a pipeline for detection and pose estimation of simple objects which does not rely on 2D or 3D features and does not require any training phase. First, candidate objects are segmented from the RGB-D image; then the object's pose is estimated using *Particle Swarm Optimization* (PSO). The contributions of this work are the extension of a fast graph-based segmentation algorithm [18] with the inclusion of depth information (similarly to [47]), and a detailed description of the GPU implementation of a novel quaternion-based formulation of the standard PSO equations which include the design of an objective function for pose estimation

which exploits depth information only. An optimized GPU version of the collision avoidance algorithm for a safe Human-Robot coexistence presented in [19] has been developed. A 7 Degree of Freedom (D.o.F) lightweight manipulator has been designed and built from scratch to test the proposed real-time collision avoidance scheme.

1.1 Pipeline Overview

This PhD dissertation presents the pipeline for fast object pose estimation and human-robot interaction depicted in fig. 1.5. This pipeline requires only three inputs: both the RGB and Depth image of the robot surrounding environment, plus a 3D model of the object the robot must interact with. The 3D object model can be generated from scratch by any CAD software; alternatively, it can be reconstructed by stitching together depth images of the object at different viewpoints. Figure 1.4 shows some 3D object models used in this work.

Figure 1.4: 3D Object Models Examples

The first module performs the objects segmentation on the RGB image. The depth image of the captured scene is also employed to enhance the segmentation process. The implemented segmentation executes a per pixel labelling. It outputs a set of pixels clusters. Each cluster is a candidate object found in the image. Pixels with the same color belong to the same cluster, therefore they belong to the same object.
The second module takes as input the candidate object clusters, the depth image and finally the 3D object model. This module performs both the object detection and the 6 D.o.F object pose estimation. The detection phase indicates whether a cluster is actually the cluster of the object we are looking for. The pose estimation part estimates the pose of the object instead. The pose estimation is run only on the clusters that passed the detection phase. Once the object pose is extracted, the robot can be easily programmed to grasp the selected object.
The third module is focused on the development of a safe Human-Robot coexistence. Given the depth images stream of the surrounding environment and, if necessary, the objects poses of interest objects, this modules guarantees a safe collision free robot trajectory. A safe coexistence between the robot and the human while they are performing different tasks very close to each other is thus fulfilled.

1.2 Outline

In Chapter 2 we describe the proposed graph-based object segmentation algorithm; in Chapter 3 we describe the novel quaternion formulation of the PSO equations for pose estimation. In Section 3.3 we present and discuss experimental results for the full pipeline, with an extension to articulated objects.
In Chapter 4 the GPU implementation and optimization of the proposed objects pose estimation algorithm is explained in detail.

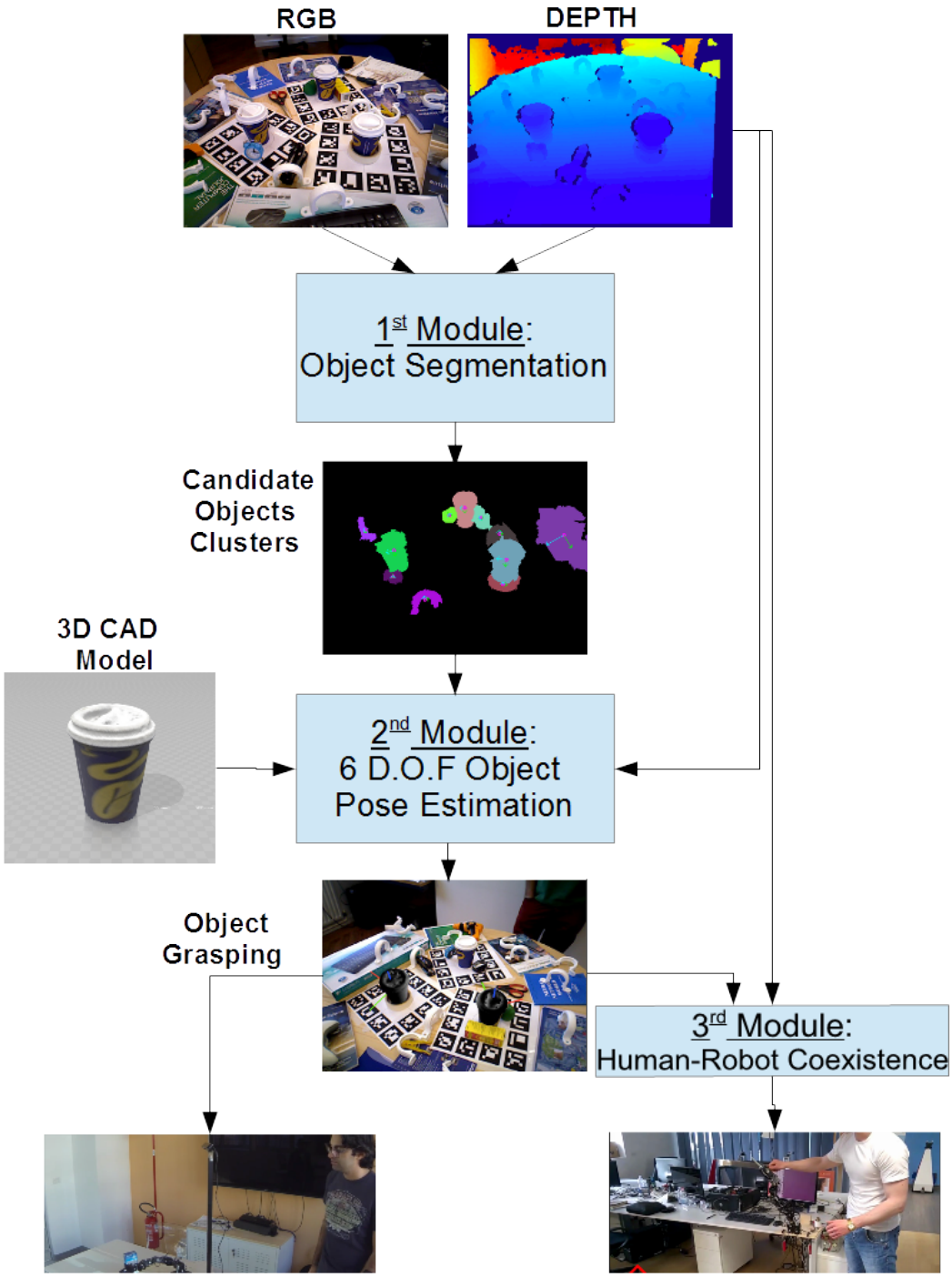


Figure 1.5: Proposed Pipeline Overview

In Chapter 5 we present and test a collision avoidance scheme for a safe Human-Robot coexistence in unstructured environments.

Finally in Chapter 6 we draw some conclusions about the work developed in this PhD dissertation.

Chapter 2

Graph-Based Object Segmentation for RGB-D Images

This Chapter deeply explains the first module of the developed pipeline for fast object pose estimation.

The image object segmentation is a technique in which given a RGB image of a scene, objects are first detected and then singled out from the image background. The proposed algorithm belongs to the class of *instance segmentation*, which consists in delineating all the object pixels corresponding to each detection. The Kinect sensor grabs a synchronized couple of images, i.e. both the RGB image and the Depth one of the surrounding environment.

The graph-based segmentation algorithm combines both images to cope with objects with large variety in appearance, from lack of texture to strong textures. A novel dynamic depth image smoothing filter is designed to filter the raw depth image without softening any relevant object edges. The State of the Art Canny edge detector is adapted and modified to extract only robust object edges out of clutter scenes (see fig. 2.6 for an example). Two non-linear cost functions are thus designed to combine color and depth cues. An unidirectional graph is then built. The graph nodes correspond to image pixels and the graph edges connect pairs of

neighbouring vertices. Each edge has a unique weight that is the outcome of the non-linear functions. The graph is then partitioned using the concept of internal and external differences between graph regions. The graph partition produces a set of pixel clusters. Each cluster represents a candidate object located in the captured image. A post-processing phase is finally run to discharge false positive clusters. The remain object clusters can then be further processed in the second module of the pipeline to obtain an estimate of their own 6 D.o.F pose (see Chapter 3).
Fig. 2.1 depicts all the steps performed by the segmentation algorithm needed to produce the final candidate objects clusters.

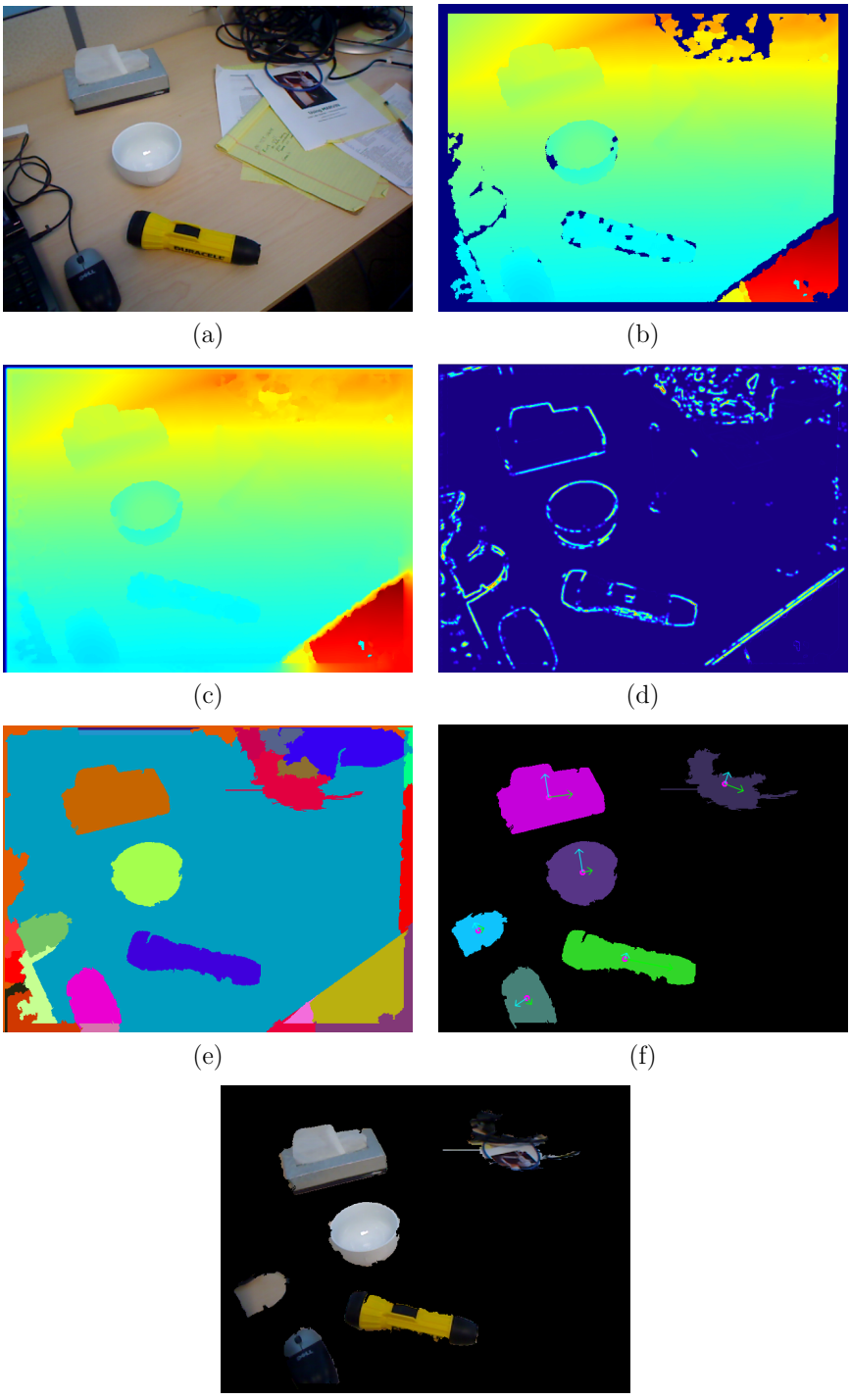


Figure 2.1: The algorithm in action. (a) RGB image; (b) depth image; (c) smoothed and inpainted depth image; (d) obtained graph weights (color coded); (e) segmentation result; (f) result after post-processing; (g) segmented objects.

2.1 Background

In [18] a graph-based segmentation algorithm has been developed based only on color cues. The segmentation algorithm proposed in this thesis extends the one in [18] and adds new feature vectors and novel non-linear weighting functions for graph edges weighting.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an undirected graph with vertices $\mathcal{V} = (v_i, \ldots, v_{Np})$ corresponding to image pixels, and edges $e_{ij} = (i, j) \in \mathcal{E}$, that connect pairs of neighboring vertices, namely v_i and v_j. Each edge e_{ij} has a corresponding weight w_{ij}, which is a measure of the similarity between v_i and v_j. Hence image segmentation reduces to partitioning of $\mathcal{G}$ in subgraphs sharing similar characteristics.

Edge weights can be computed by evaluating color or intensity difference. If the function $\Lambda : \mathbb{R}^2 \rightarrow: \mathbb{R}^5$ associates each node (pixel) v_j with the corresponding feature vector containing both node coordinates v_{i_x}, v_{i_y} and the RGB values $v_{i_r}, v_{i_g}, , v_{i_b}$, then the edges weights are computed as:

$$w_{ij} = ||\Lambda(v_i) - \Lambda(v_j)||_2, \forall (i, j) \in \mathcal{E}. \tag{2.1}$$

Let's define the *internal difference* within the region R_a as:

$$\mathcal{I}(R_a) = \max_{(i,j) \in \mathcal{E}, i,j \in R_a} w_{ij}, \tag{2.2}$$

Let's also define the *difference between two regions* R_a and R_b as:

$$\mathcal{D}(R_a, R_b) = \min_{(i,j) \in \mathcal{E}, i \in R_a, j \in R_b} w_{ij}, \tag{2.3}$$

The segmentation procedure starts by considering each pixel as a different region. Such regions are pairwise compared and two regions are merged together in a bigger

cluster if the following condition holds:

$$\mathcal{D}(R_a, R_b) \leq \min(\mathcal{I}(R_a) + \frac{\gamma}{|R_a|}, \mathcal{I}(R_b) + \frac{\gamma}{|R_b|}), \tag{2.4}$$

otherwise a boundary exist between them. In eq. (2.4) the right part represents the *minimal internal difference* between two regions R_a and R_b. Moreover, γ is a constant parameter and the operator $|\cdot|$ returns the region size in pixels.

2.2 Image cues definition

In this section the definition of the different image cues (δ) are explained in detail. Four different image cues are defined as follows:

1. $\delta_{\mathbf{hsv}}$: It is the cue matrix related to the Hue-Saturation-Value (HSV) color space.

2. $\delta_{\mathbf{depth}}$: It is the cue matrix related to the depth image space.

3. $\delta_{\mathbf{s}}$: It is the cue matrix related to the saliency image based on RGB color image.

4. $\delta_{\mathbf{bound}}$: It is the cue matrix related to the object external boundaries.

Each cue matrix (δ) can be visualized as an image map where the element $\delta_{\mathbf{i,j}}$ is the feature associated to the pixel (i, j) of both RGB and Depth input images.

2.2.1 Color Cue $\delta_{\mathbf{hsv}}$

The RGB input image is first converted into the HSV color space. The HSV color space allows a more robust separation between the color information (Hue, Saturation) and intensity (Value) in presence of objects with shadows or changes in lightness. Therefore, the metric described in [2] has been adjusted to compute

the color difference between two vertices $v_i = (v_{i_h}, v_{i_s}, v_{i_v})$ and $v_j = (v_{j_h}, v_{j_s}, v_{j_v})$, in HSV space, as follows:

$$\delta_{ij_{hsv}} = \frac{\sqrt{\delta_v^2 + \delta_s^2}}{\sqrt{k_{dv}^2 + k_{ds}^2}}, \tag{2.5}$$

where:

$$\delta_v = k_{dv}|v_{i_v} - v_{j_v}|, \delta_h = |v_{i_h} - v_{j_h}|,$$

$$\theta = \begin{cases} \delta_h, & \text{if } \delta_h < 180^\circ \\ 360^\circ - \delta_h, & \text{if } \delta_h \geq 180^\circ \end{cases}$$

$$\delta_s = k_{ds}\sqrt{v_{i_s}^2 + v_{j_s}^2 - 2v_{i_s}v_{j_s}\cos\theta}.$$

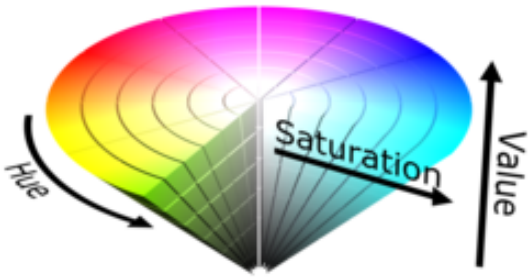
The denominator in (2.5) is introduced to normalize the color error in the range [0,1]. The parameters k_{dv}, k_{ds} are used to weight the *value* and *saturation* differences respectively. In our experiments these parameters are always kept fixed to $k_{dv} = 4.5$, $k_{ds} = 0.1$.

Figure 2.2 provides a graphical representation of the metric described in eq. 2.5. Figure 2.2a depicts the standard HSV color model as a solid cone. In contrast, fig. 2.2b visualizes all the parameters involved in eq. 2.5. The top view image of the HSV cone focuses on the Hue and Saturation components of the HSV color space. It shows two different Hue values for pixel i and j. i.e., v_{i_h} and v_{j_h}. The angular difference between these two Hue values is visualized as δ_h. The *saturation* component of two pixels is highlighted by two red segments v_{i_s} and v_{j_s}, respectively. The error distance δ_s between these two components can be found applying the law of cosines (i.e., violet segment in fig. 2.2b).

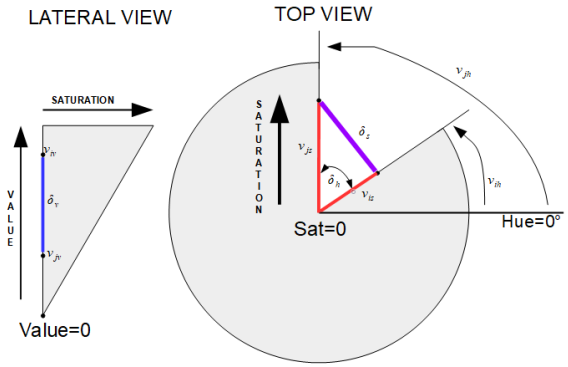
Finally, two pixels *Value* error δ_v can be visualized through the lateral view of the HSV cone model. The HSV Value components of two pixels are always parallel, hence the pixels Value distance can be computed simply as difference between v_{i_v} and v_{j_v} (i.e. blue segment).

The final Color Cue $\delta_{\mathbf{hsv}}$ is defined as a normalized distance between the pixels

saturation error and *value* error.



(a) HSV Color Cone Model



(b) Details of the HSV metric

Figure 2.2: Visualization of the HSV color metric in eq. 2.5

2.2.2 Depth Cue δ_{depth}

Since depth maps obtained from low-cost Kinect-like sensors are usually noisy and prone to quantization errors, a smoothing to the depth map is required. In [27], a depth-dependent smoothing algorithm is proposed which generates a smoothing kernel of different size for each pixel in the depth image. The area of such a kernel is based on two indicators, i.e. depth information of the pixel itself and the distance of the pixel from the object borders. The former is used to generate a wider smoothing area for pixels far from the camera, since the noise of the depth data is proportional to the distance from the sensor. The latter guarantees that the object edges are not

smoothed by the filter. The final smoothing kernel sizes are saved in a *Smoothing Area Map* $\mathcal{S}(y,x)$. The average value within a region is thus computed by means of the depth integral image $\mathcal{I}_D(y,x)$ and saved in the smoothed depth map $\mathcal{D}_s(y,x)$ as follows:

$$\begin{aligned}\mathcal{D}_s(y,x) =& \frac{1}{(2r+1)^2}\,[\,\mathcal{I}_D(y+r,x+r) - \mathcal{I}_D(y+r,x-r) \\ & -\mathcal{I}_D(y-r,x+r) + \mathcal{I}_D(y-r,x-r)\,]\,;\end{aligned} \tag{2.6}$$

where: $r = \mathcal{S}(y,x)$. The smoothing from [27] has a drawback. It introduces noise when the depth image contains shadows (depth image pixels with zero depth value); this is because (2.6) is not able to discriminate between pixels having real depth information and pixels having no depth component.The depth map $\mathcal{D}(y,x)$ is therefore employed to generate a binary image $\mathcal{B}_D(y,x)$ as follows:

$$\mathcal{B}_D(y,x) = \begin{cases} 0 & \text{if } \mathcal{D}(y,x) \neq 0 \\ 1 & \text{if } \mathcal{D}(y,x) = 0 \end{cases} \tag{2.7}$$

The binary depth map integral image $\mathcal{I}_B(y,x)$ is then used to count the number of pixels with no depth information (γ_0) inside the smoothing area of a given depth image pixel.

$$\begin{aligned}\gamma_0 =& \mathcal{I}_B(y+r,x+r) - \mathcal{I}_B(y+r,x-r) + \\ & - \mathcal{I}_B(y-r,x+r) + \mathcal{I}_B(y-r,x-r);\end{aligned} \tag{2.8}$$

where: $r = \mathcal{S}(y,x)$.

(2.6) is thus updated as shown in (2.9).

$$\begin{aligned}\mathcal{D}_s(y,x) =& \frac{1}{(2r+1)^2 - \gamma_0}\,[\,\mathcal{I}_D(y+r,x+r) + \\ & -\mathcal{I}_D(y+r,x-r) - \mathcal{I}_D(y-r,x+r) + \mathcal{I}_D(y-r,x-r)\,]\,;\end{aligned} \tag{2.9}$$

The denominator in (2.9) equals zero if and only if all depth image pixels inside the smoothing kernel are equal to zero (no depth data is available). In this case the smoothing has no meaning and $\mathcal{D}_s(y,x) = \mathcal{D}(y,x)$.
Figure 2.3 shows a comparison between the original depth smoothing algorithm and the modified one.

The depth error between two vertices $v_{i_d} = (y_{i_d}, x_{i_d})$ and $v_{j_d} = (y_{j_d}, x_{j_d})$ is then defined as:

$$\delta_{ij_{depth}} = \mathcal{D}_s(y_{i_d}, x_{i_d}) - \mathcal{D}_s(y_{j_d}, x_{j_d}) \tag{2.10}$$

$\delta_{ij_{depth}}$ is then normalized to be in the range [0,1] .

When either v_{i_d} or v_{j_d} is undefined (one of the pixels belongs to a shadow in the depth image) $\delta_{ij_{depth}}$ is set to 0, since a shadow border is not necessarily an object border. Shadows in Kinect-like depth images can be caused by several effects: occlusions, highly reflective or absorbing materials, highly skewed surfaces, thin objects, or objects placed too near to the sensor.

2.2.3 Saliency Cue $\delta_{\mathbf{sal}}$

A visual saliency map $\mathcal{V}_S(y,x)$ is also computed on the color image using the algorithm proposed in [43], which is a fast implementation of visual saliency that uses an integral image on the original scale of the image in order to obtain high quality features in real time. The algorithm is based on a single parameter for computing all the filter windows on a single integral image $\varsigma = \sigma 2^s$, where σ represents the surround and s the scale. The saliency map in [43] has been post-processed to highlight the actual object borders in the image. The saliency map $\mathcal{V}_S(y,x)$ is first normalized in the range [0,1] and then filtered through a power-law transformation as:

$$\hat{\mathcal{V}}_S(y,x) = \mathcal{V}_S(y,x)^4 \tag{2.11}$$

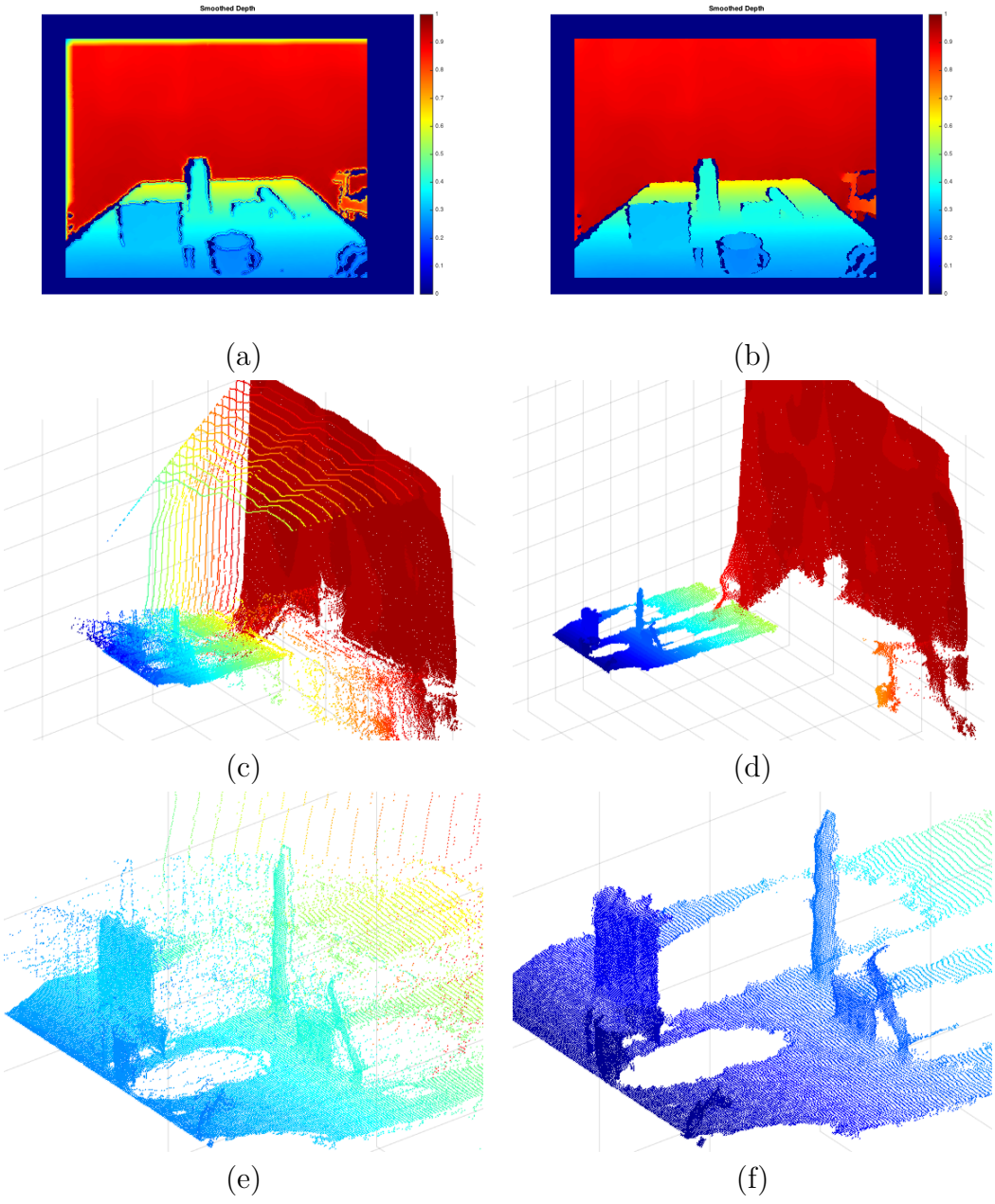


Figure 2.3: Comparison of depth smoothing algorithms in presence of missing depth values (dark blue pixels in both (a) and (b) images). Left column shows the results of the original smoothing algorithm as described in [27]. Right column depicts the depth smoothing results obtained through our own changes to the original algorithm. (a) and (b) show $\mathcal{D}_s(y, x)$. (c),(d) Show the PointCloud extracted from $\mathcal{D}_s(y, x)$. (e) and (f) show object and edge details.

This transformation lowers the values of middle gray-level pixels while keeping the high gray-level ones (pixels close to white) almost unchanged. The latter are pixels that are likely to belong to object borders. Figure 2.4 shows the transformed saliency image.

(a) (b)

Figure 2.4: (a) Original saliency image. (b) Power-law filtered saliency image

The saliency for each vertex $v_{i_s} = (y_{i_s}, x_{i_s})$ is thus defined as:

$$\delta_{i_{sal}} = \hat{\mathcal{V}}_S \left(y_{i_s}, x_{i_s} \right) \tag{2.12}$$

2.2.4 Boundary Cue $\delta_{\mathbf{bound}}$

Images with strong textures produce false-positive object borders when State of the Art edge filters are used. Fig 2.5 (a) shows a scene of a clutter environment with strong textures. Fig 2.5 (b) depicts the Canny edge filter outcome of that image. Textures and object boundaries are then no more distinguishable. [53] and [41] proposed two different approaches to obtain a good estimation of object edges in strong textured images. The first approach averages the depth gradient of an edge pixel within a small neighborhood of pixel along a candidate edge. The second approach uses a logistic function trained over examples for detecting depth boundary edges. The latter technique requires a labelled dataset of object borders.

(a) (b)

Figure 2.5: (a) Clutter scene with strong textures. (b) State of the Art Canny Edge Detector

In this thesis the state of the art Canny edge detector has been modified to allow also a depth-based edge filtering along with the RGB one. We run a Canny edge detector on the image, based on the Scharr kernels to obtain a binary edge map $\mathcal{E}_E(y, x)$. We also extract the gradient directions Θ of the edge pixels and we discretize them to one of the possible angle (namely 0°,45°,90°,135°).
For each edge pixel $\vec{e} = (y_e, x_e) \in \mathcal{E}_E$, we sample two points, one along the positive edge gradient direction and the other along the negative one and we compute the depth gradients as:

$$\begin{aligned} \rho_+ &= \mathcal{D}_s(\vec{e}) - \mathcal{D}_s(\vec{e} + \varepsilon_\rho \vec{n}) \\ \rho_- &= \mathcal{D}_s(\vec{e}) - \mathcal{D}_s(\vec{e} - \varepsilon_\rho \vec{n}) \end{aligned} \tag{2.13}$$

where $\vec{n}$ is the edge normal vector and ε_ρ indicate the pixel to pick along the edge gradient direction.
A *depth boundary* map $\mathcal{E}_B(y, x)$ is the computed as follows:

$$\mathcal{E}_B(\vec{e}) = \begin{cases} 0 & \text{if } \rho_+ < t_\rho \wedge \rho_- < t_\rho \\ 1 & \text{otherwise} \end{cases} \tag{2.14}$$

If both ρ_+, ρ_- are below a given threshold t_ρ, then the edge point $\vec{e}$ does not represent a real edge, but it is caused by texture instead; otherwise the point is a *depth boundary*. While most boundary pixels of an object correspond to depth discontinuities, the part of the object that touches the surface it is resting on does not have present depth discontinuity across it, therefore the *contact edge* pixels are filtered out from the *depth boundary* map $\mathcal{E}_B(y, x)$.
For each edge pixel $\vec{e} = (y_e, x_e) \in \mathcal{E}_E$, three points along the edge gradient direction are sampled, with the central one being the pixel $\vec{e}$. These three pixels are then projected onto the camera frame by means of the camera intrinsic matrix $\mathbf{K}$ in order to obtain the corresponding three points: $\vec{p}_e, \vec{p}_+, \vec{p}_- \in \mathbb{R}^3$.

$$\begin{aligned} \vec{p}_e &= \mathbf{K}^{-1} \left[\mathcal{D}_s(\vec{e}) \ \tilde{\mathbf{e}} \right] \\ \vec{p}_+ &= \mathbf{K}^{-1} \left[\mathcal{D}_s(\vec{e}_+) \ \tilde{\mathbf{e}}_+ \right] \\ \vec{p}_- &= \mathbf{K}^{-1} \left[\mathcal{D}_s(\vec{e}_-) \ \tilde{\mathbf{e}}_- \right] \end{aligned} \tag{2.15}$$

where:

$$\vec{e}_+ = \vec{e} + \varepsilon_e \vec{n}; \quad \vec{e}_- = \vec{e} - \varepsilon_e \vec{n}; \quad \tilde{\mathbf{e}} = [x_e, y_e, 1]^T .$$

The unit vectors $\vec{v}_{n_+}$, $\vec{v}_{n_-}$ and the angle between them θ_v are computed as follows:

$$\vec{v}_{n_+} = \frac{\vec{p}_+ - \vec{p}_e}{\|\vec{p}_+ - \vec{p}_e\|} \tag{2.16}$$

$$\vec{v}_{n_-} = \frac{\vec{p}_- - \vec{p}_e}{\|\vec{p}_- - \vec{p}_e\|} \tag{2.17}$$

$$\theta_v = \text{atan2}\left(\frac{\vec{v}_\times}{\vec{v}_{dot}} \right) \tag{2.18}$$

where: $\vec{v}_\times = \vec{v}_{n_+} \times \vec{v}_{n_-}$ and $\vec{v}_{dot} = \vec{v}_{n_+} \cdot \vec{v}_{n_-}$. For everyday objects lying on ordinary surfaces (such as tables, shelves, floors, etc.), contact edge pixels can be estimated straightforward by filtering the angle θ_v in (2.18). Common interactions between objects and holding surfaces lead to contact angles close to 90°.

A *contact boundary* map $\mathcal{E}_C(y,x)$ is the computed as follows:

$$\mathcal{E}_C(\vec{e}) = \begin{cases} 0 & \text{if } \theta_v > t_{\theta_H} \vee \ \theta_v < t_{\theta_L} \\ 1 & \text{otherwise} \end{cases} \tag{2.19}$$

where t_{θ_H} and t_{θ_L} are high and low thresholds used to cope with contact angles perturbations around the ideal value.

The contact boundary map as defined by (2.19) and (2.18). It also includes false contact edges called *internal boundaries* as shown in figure 2.6a. Since we are only interested in the objects external contours (e.g., figure 2.6b), we cancel internal edge pixels out by looking at the direction of the cross product $\vec{v}_\times$ as depicted in figure (2.7). Note that the vectors defined until now are all expressed w.r.t the camera reference frame with the Z-axis pointing outwards along the optical axis and the X-axis pointing to the right.

A contact edge pixel $\vec{i} = (y_i, x_i) \in \mathcal{E}_C(y,x)$ is estimated to be an internal boundary pixel if and only if the x component of the cross product $\vec{v}_\times$ is less than zero. (2.18) is then updated as follows:

$$\theta_v = \begin{cases} -\text{atan2}\left(\dfrac{\vec{v}_\times}{\vec{v}_{dot}}\right) & \text{if } v_{\times_x} < 0 \\ \text{atan2}\left(\dfrac{\vec{v}_\times}{\vec{v}_{dot}}\right) & \text{otherwise} \end{cases} \tag{2.20}$$

The *contact boundary* map $\mathcal{E}_C(y,x)$ definition (see 2.19) is therefore left unchanged. The *final boundary* map $\mathcal{E}_F(y,x)$ is then computed as:

$$\mathcal{E}_F(y,x) = \mathcal{E}_B(y,x) \cup \mathcal{E}_C(y,x) \tag{2.21}$$

Despite the fact that the simple texture edge filtering might be seen as limited to a small class of simple objects (i.e., prismatic ones), Figures 2.6d and 2.6f show how the algorithm is very effective for other classes of objects as well (i.e., cylindrical

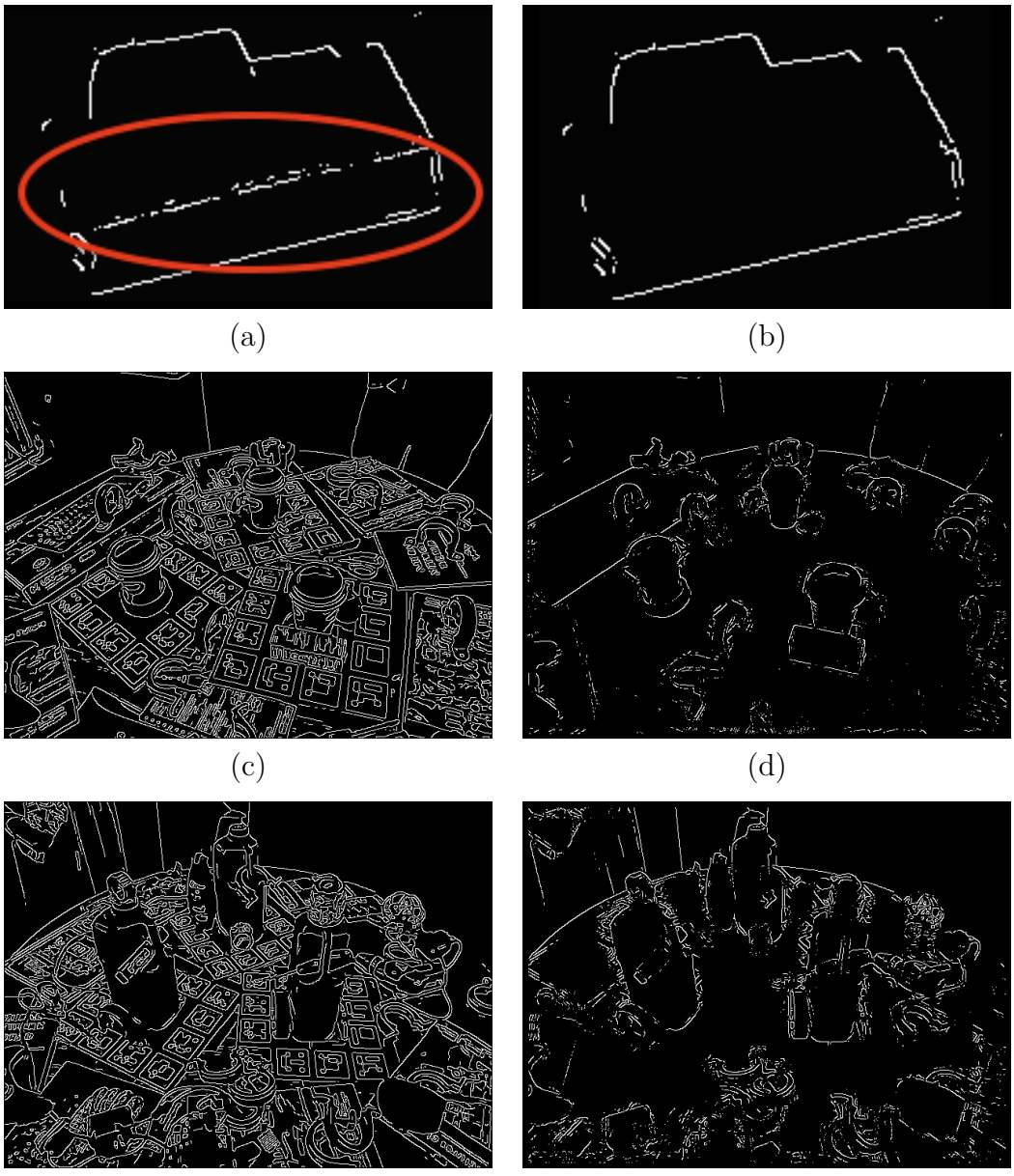


Figure 2.6: (a) Final Edge map with internal boundaries. (b) Final Edge map. (c) Canny output with no texture edge filter (Coffe Cups). (d) Coffe Cups Final Edge map. (e) Canny output with no texture edge filter (Milk Jugs). (f) Milk Jugs Final Edge map.

ones) and, in general, is able to handle complex object shapes (e.g., milk jugs).

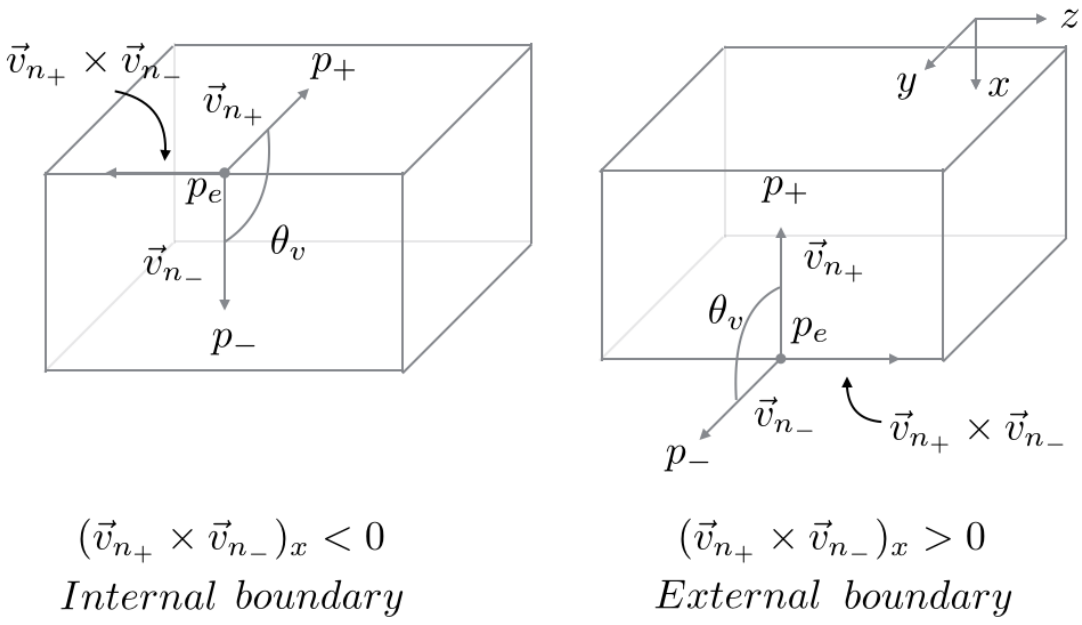


Figure 2.7: Internal boundary pixels definition.

The boundary for each vertex $v_{i_b} = (y_{i_b}, x_{i_b})$ is thus defined as:

$$\delta_{i_{bound}} = \mathcal{E}_F(y_{i_b}, x_{i_b}) \tag{2.22}$$

2.3 Weight Functions

The weight functions are used to compute a scalar weight w_{ij} for each edge e_{ij} that links the two neighboring vertices v_i and v_j . The weights are computed as a nonlinear function of all the cues (δ) introduced in this Section.

Two cost functions are proposed. The first one includes color, depth and saliency information and it is found to work best when a large number of shadows are present in the depth image. The second one includes depth, color and boundary edges and works better when full depth information is available.

In the first cost function, the difference between two vertices v_i and v_j is defined

as:

$$w_{ij} = \frac{k_y \log_2(1+\delta_{ij_{hsv}}) + k_x \log_2(1+\delta_{ij_{depth}})}{2+k_x+k_y+k_s} + \frac{\delta_{ij_{depth}}\delta_{i_{sal}}^{1+\delta_{ij_{depth}}} + \delta_{ij_{depth}}\delta_{ij_{hsv}}^{1+\delta_{ij_{depth}}} + k_s \log_2(1+\delta_{i_{sal}})}{2+k_x+k_y+k_s}, \tag{2.23}$$

where k_s, k_y , k_x are parameters for weighting in the saliency map, the color and depth difference respectively.

In the second cost function, the difference between two vertices is defined as:

$$w_{ij} = \frac{k_x \delta_{ij_{depth}} \log_2(1+\delta_{ij_{hsv}}) + k_b \delta_{i_{bound}}}{k_x + k_b}, \tag{2.24}$$

where k_b is a parameter for weighting in boundary edges, while the denominator in both (2.23) and (2.24) is needed to normalize the weights between [0,1].

We use base-2 logarithms since all the cues used as input to the weight functions (δ) are in the range [0,1]; the dynamic range of the cues is thus left unchanged. Moreover, logarithmic functions map a narrow range of low cue values into a wider range of output levels while compressing higher values. This property tends to create edge weights that are spread within their own dynamic range rather than generating quasi-binary weights maps.

(2.23) is composed by a logarithmic term which handles a single independent variable (i.e., $k \log_2(1+\delta)$) and a coupled term which relates two variables (i.e., $\delta_{depth}\delta_{hsv}^{1+\delta_{depth}}$). The former controls the effect of each input information independently through the parameters k. It plays a fundamental role when no depth data are available, since the coupled terms equal zero. Fig. 2.8a shows how an increase or decrease of k would slide the red squares upward or downward respectively The latter, instead, biases the weights assignment by introducing depth information. The lower the depth variation is, the lower is the contribute of color and saliency cues; this happens in highly textured objects where two pixels, belonging to the same object surface, generate small depth gradient but large color (or saliency) difference. For large depth gradients the weight shows an exponential trend and

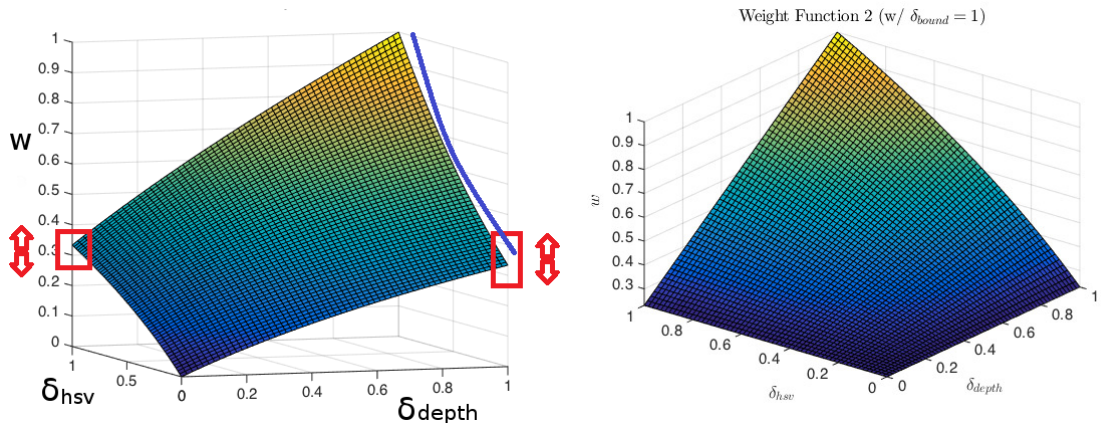
reaches maximum together with the color (or saliency) difference (see the blue curve in 2.8a). The exponential shape of w for medium-high depth gradients is needed to mitigate the linear contribute of δ_{depth}. This situation may arise in presence of concave objects (i.e., ceramic bowls or horseshoe-like objects) where medium-high variations of the depth gradient do not necessarily mean that the corresponding two graph vertices belong to different objects. In this case, in fact, the function generates lower edge weights in presence of low and medium visual cue differences.

The cost function in (2.24) is used when little or no shadows are present in the depth map. We fill the small gaps in the depth image by in-painting. Depth maps with large shadows are not in-painted since the reconstruction error would generate noise and false object boundaries. In this case, the depth map is not modified and the cost function in (2.23) is used instead. The second weight function has a coupled term that trades the information of δ_{hsv} and δ_{depth} like the first weight function but without any saliency data. The idea of this terms is the same defined in (2.23) but we noticed how the logarithmic term here works better than the exponential one. The second term adds a bias term k_b when the vertex v_i is a boundary pixel $p_i = (y_i, x_i) \in \mathcal{E}_F$.
Fig 2.9 shows the weight maps of the two cost function in (2.23) and (2.24) for a given input image respectively.

The graph is partitioned using Disjoint-set Forests. At the first iteration each node represents a distinct region R_i. Regions are iteratively merged based on (2.4). The final result is a set of regions $\mathcal{R}$.

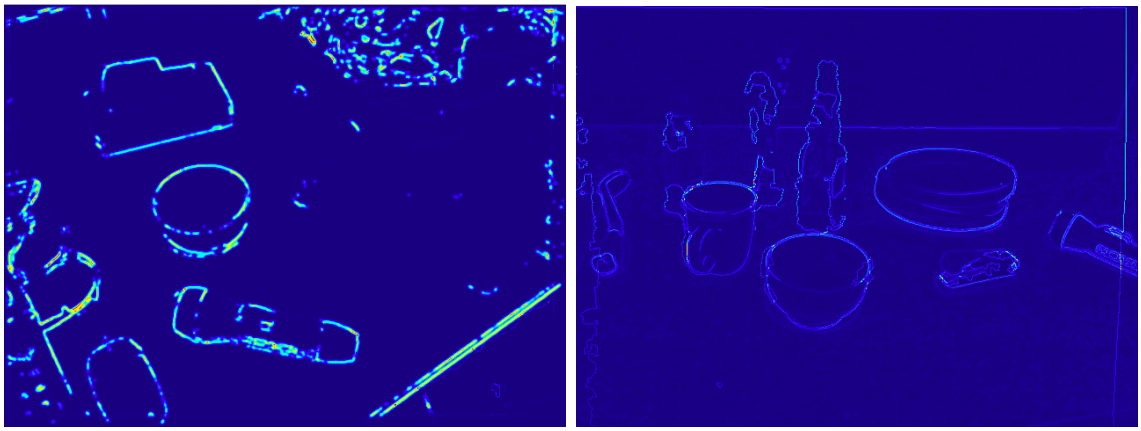
2.4 Post-Processing

In order to discard false positives, such as regions that belong to the background, some rejections steps are required on the set $\mathcal{R}$.



(a) (b)

Figure 2.8: (a) Cost function as defined in (2.23) without the saliency variable (δ_{sal}) for visualization purposes. (b) Cost function as defined in (2.24) when $\delta_{bound} = 1$ (i.e., the pixel under the v_{i-th} vertex is a boundary one)



(a) (b)

Figure 2.9: Example weight maps w: (a) outcome of weight function in eq. (2.23). (b) outcome of weight function in eq. (2.24)

Principal component analysis is performed on each region to estimate the principal components $\vec{x_1}$, $\vec{x_2}$, the relative eigenvalues λ_1, λ_2 and its eccentricity ε.

$$\varepsilon = \sqrt{1 - \frac{\lambda_2}{\lambda_1}} \tag{2.25}$$

The eccentricity provides a rough estimation about the shape of the region $\mathcal{R}$. High

eccentricity values, hence high differences between the two eigenvalues, describe a malformed cluster due to errors in the segmentation step or objects that are actually wide (e.g. the table surface in fig. 2.1 which is correctly filtered out by this step.) Fig. 2.1 (f) shows the outcome of the PCA region ($\mathcal{R}$) discard step. Each cluster has been overlaid with both its own centroid (red circle) and their principal component unit vectors $\vec{x_1}$, $\vec{x_2}$ (blue and green arrows respectively). The magnitude of $\vec{x_1}$, $\vec{x_2}$ is driven by the eigenvalues λ_1 and λ_2 respectively. If either λ_1, λ_2 or ε are over given thresholds, the region is discarded. The threshold can be roughly estimated if the classes of objects to be found are known in advance.

We add two more rejection steps when dealing with difficult lightning conditions and poor depth maps (see Section 2.5.1). When not using in-painting of the depth image, regions whose pixels with no valid depth data are greater than 30% of total region size are also discarded, as this may lead to the failure of the robot grasping policies defined thereafter. Finally, dark regions can be discarded too. A 32 bins histogram of the brightness component of the region is computed. If 30% of region pixels fall within the first three bins of the histogram (i.e., pixels values in the range [0,24]), the region is discarded. Since we are interested in grasping, it is also possible to discard regions which are out of reach for a robotic arm, being too distant to the camera frame.

2.5 Experimental Results

We tested our approach on three public datasets of RGB-D scenes [50], [51] and [59]. We also compare the results with the one proposed in [41] and show a qualitative comparison with the original algorithm [18] and [2]. For each image of every dataset, objects have been manually labelled by delineating the pixels inside the object boundary. If the segmented objects overlap more than 70% with the corresponding object pixels, we consider the object as successfully segmented, as in

[47].

The software has been developed using the OpenCV library in C++ under Linux and runs on CPU. The source code is available [1]. All frames are 640×480 and the average processing time per image was 0.6 s on a standard PC with a 2.3Ghz CPU (single thread). Figure 2.10 shows two results on different datasets, while Figure 2.11 shows a comparison between different approaches. The full video of the segmentation algorithm results on different publicly available datasets can be watched at the following link: Graph Segmentation Results.

2.5.1 Rutgers APC RGB-D Dataset

This dataset ([50]) has been created specifically for the Amazon Picking Challenge and is composed of different runs, each one containing a series of RGB images and corresponding depth images, acquired using a Asus XTion sensor in different positions and with a variable number of objects on shelves. For each position, four consecutive images are provided.

The dataset is particularly challenging due to low lightning and heavy presence of shadows and missing areas in the depth image. We assembled three runs from the dataset with increasing average number of objects in each image. We tested our approach on a subset of runs. We used the first weight function, due to the large number of shadows. Parameters have following values: $\gamma = 5$, $k_x = 1.05$, $k_y = 1.5$, $k_s = 0.5$. Results are reported in Table 2.1.

2.5.2 RGB-D Scenes Dataset

The RGB-D Scenes Dataset consists of 8 scenes annotated with objects that belong to the RGB-D Object Dataset. (bowls, caps, cereal boxes, coffee mugs, and soda cans). Each scene is a single video sequence consisting of multiple RGB-D

[1] https://github.com/rrg-polito/graph-canny-segm

Table 2.1: Results for the Rutgers APC Dataset.

	No. of objects	% of objects detected
Run_1	80	87.9%
Run_2	120	92.3%
Run_3	121	75.6%

frames. The objects are visible from different viewpoints and distances and may be partially or completely occluded. We compare the results of the proposed algorithm with the one from [41]. We tested the approach on subset of six objects and on three different scenes. We used the second weight function and set the parameters to the following values: $\gamma = 0.0016$, $k_x = 7.5$, $k_b = 0.66$. Results are shown in Table 2.2. It should be noted that our approach does not rely on knowledge of the camera pose and is thus more general, at the cost of lower accuracy for some objects, while attaining 100% accuracy for other objects. Results are comparable to [41], though the metric we use is more strict (in [41] an overlap of 50% is considered as a good detection).

Table 2.2: Results for the RGB-D Dataset. Inside parentheses, the results from [41] are reported for comparison.

	% of objects detected					
	Soda can	Coffee mug	Cap	Bowl	Flashlight	Cereal box
Table_1	90.6% (100%)	100% (83.6%)	80.1% (93.6%)	85.5% (90.3%)	98.1% (98.1%)	72% (97.8%)
Desk_1	100% (93.7%)	100% (92.5%)	74.2% (100%)	-	-	-
Kitchen_small_1	98.6% (74.8%)	100% (70.1%)	86.5% (97.3%)	100% (90%)	100% (88.5%)	77.6% (84.4%)

2.5.3 Multiple-instance dataset

In [59], 6 objects are captured under varying viewpoint with lots of background clutter, scale and pose changes, and in particular foreground occlusions and multi-instance representation (three instances of the same object are present in each frame as well as other objects and clutter). We tested the approach on a subset of scenes. Parameters for the second weight function are as follows: $\gamma = 0.001$, $k_x = 1.2$, $k_b = 0.05$. Results are shown in Table 2.3.

Table 2.3: Results for the multiple instance dataset.

	No. of objects	% of objects detected
Milk	2589	66.6%
Coffee_Cup	2127	87.2%
Shampoo	2118	99.6%
Camera	894	96.3%

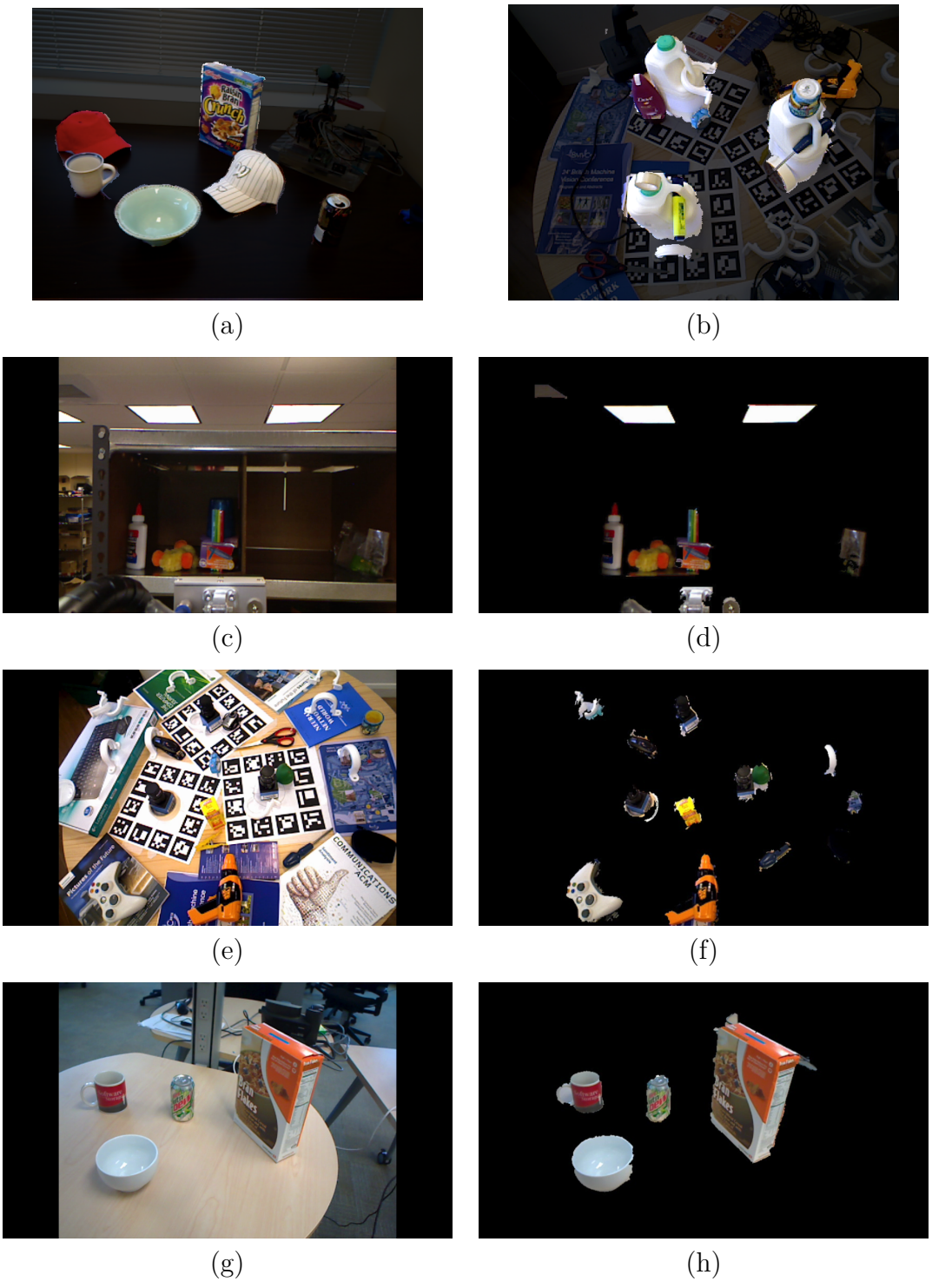


Figure 2.10: Examples of the results on the [41] dataset (a) and [59] dataset (b). Dataset [50] in (c) and (d). Dataset [59] in (e) and (f). Dataset [41] in (g) and (h). Segmented objects are highlighted.

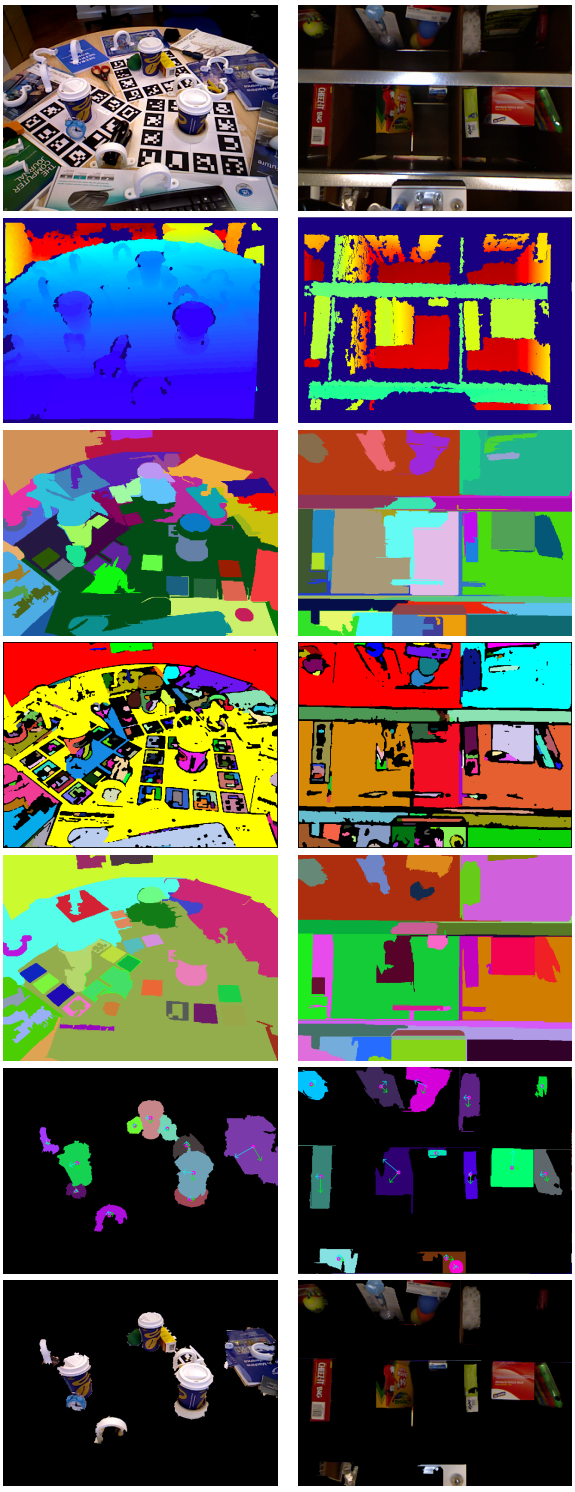


Figure 2.11: Comparison between different approaches. First and second row: original images; third row: [18]; fourth row: [2]; fifth row: [47]; sixth and seventh row: our approach.

Chapter 3

Quaternion-based Particle Swarm Optimization for Object Pose Estimation From RGB-D Images

In Chapter 2, the segmentation phase produces a set of regions $\mathcal{R}$ that encloses the actual objects present in the input image. Each region is composed by a collection of pixels all belonging to the same object. All objects information returned by the segmentation phase are related to the image reference frame, i.e. in the two-dimensional domain.

Robot-Object interaction such as grasping, collision avoidance, navigation requires a good knowledge of the objects pose in 3D space. An object in 3D space can be uniquely defined by a translation component and an orientation one. The former characterise the object position along the three axes component $\mathbf{t} = [x, y, z]^T$ w.r.t. to a given reference frame. The orientation part is used to uniquely defines the object attitude in 3D space. Body attitude in 3D space can be embodied in different orientation representations (e.g., Euler Angles, Axis-Angle, Rotation Matrix, Unit Quaternions, etc..). Body orientation can always be expressed by three parameters, irrespective of the chosen body attitude representation.

In this Chapter is presented a novel algorithm for 6DoF object pose estimation based on 2D information only. This algorithm takes as input the set of object regions $\mathcal{R}$ resulting from the segmentation step in Chapter 2. Then, the estimation procedure returns the pose w.r.t the camera frame for each cluster in $\mathcal{R}$.

3.1 Background

Particle Swarm Optimization (PSO) [31] is an heuristic technique inspired by the swarming or collaborative behavior of biological populations. It is useful for exploring the search space of a problem to find the settings or parameters required to maximize a particular objective. A set of candidate solutions (particles) $\mathbf{p}_i(t) = (\mathbf{x}_i(t), \mathbf{v}_i(t))$, where $\mathbf{x}_i$ and $\mathbf{v}_i$ are the position and velocity of particle i at time t, is maintained in the search space. The algorithm consists of three steps, which are repeated until some stopping condition is met: first, the fitness of each particle is evaluated, then individual and global best fitnesses and positions are updated; finally velocity and position are updated for each particle. In the update phase the velocity is computed as follows:

$$\mathbf{v}_i(t+1) = w\mathbf{v}_i(t) + c_1 r_1[\mathbf{x}_{pbest}(t) - \mathbf{x}_i(t)] + c_2 r_2[\mathbf{x}_{gbest}(t) - \mathbf{x}_i(t)], \tag{3.1}$$

where w, c_1, c_2 (with $0 \leq w \leq 1.2$, $0 \leq c_1 \leq 2$, $0 \leq c_2 \leq 2$) are tunable parameters; r_1, r_2 are random values. $\mathbf{x}_{pbest}(t)$ is the best candidate solution for the particle $\mathbf{p}_i$ at time t and $\mathbf{x}_{gbest}(t)$ is the global best candidate solution at time t. The particle position is then computed as:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1). \tag{3.2}$$

3.2 Pose estimation

To estimate the 6DoF object pose we use a quaternion-based formulation of the standard PSO equations (3.1) and (3.2). We use unitary quaternions to describe the orientation of an object in 3D space since they are *gimbal-lock* free and they have a well-defined interpolation formula (SLERP) [56]. Gimbal-lock would produce wrong results when the fitness function of a particle is computed; moreover, other rotation formalisms would require the conversion to and from rotation matrix form at each step. Unitary quaternions, however, turn the optimization problem into a constrained one. We design the new PSO equations such that the explored orientations are always represented by unit-length quaternions. This means that every particle, in every time instant, holds a valid object pose hypothesis. For now on, when we talk about quaternions we refer to unit-length quaternions. Both quaternions $\mathbf{q} = [\, q_0, \vec{q}\,]$ and -$\mathbf{q} \in \mathbb{S}^3$ define the same orientation. To overcome this ambiguity we cast the quaternion to the northern hemisphere of $\mathbb{S}^3$, i.e., we ensure that the scalar part of a quaternion is always positive or equal to zero ($q_0 \geq 0$) .

3.2.1 Angular velocity and orientation update

The standard velocity update equation (3.1) describes a weighted sum of three vectors in Euclidean space.

The current linear velocity of an object is expressed by the vector $w\mathbf{v}_i(t)$. The object *cognitive* linear velocity is given by vector $c_1 r_1 \left[\mathbf{x}_{pbest}(t) - \mathbf{x}_i(t)\right]$ and the *social* linear velocity acting on an object is $c_2 r_2 \left[\mathbf{x}_{gbest}(t) - \mathbf{x}_i(t)\right]$. This, along with (3.2), is used to optimize the position component of the object pose. In (3.1) velocities are computed as the difference between two position vectors. Subtraction has no meaning for unit-length quaternions so a new equation must be derived to obtain the object's angular velocity based on the current and best orientations of an object. The goal is to obtain both the cognitive and social angular velocity effecting

an object, through the *quaternion inverse displacement*. Figure 3.1 shows a graphical visualization of the SLERP interpolation on the $\mathbb{S}^3$ sphere.

The *Spherical Linear Interpolation* (SLERP) is a linear interpolation (LERP) performed on a surface of a unit sphere. This is a fundamental property when we are dealing with unit-quaternions interpolation. The SLERP, thus, ensures that each intermediate quaternion along any interpolation path between the starting- and ending- quaternions is always a unit-length one. Hence the SLERP guarantees that any quaternion along the interpolation path represents a valid rotation in 3D space.

Let $\mathbf{p}_0,\ \mathbf{p}_1 \in \mathbb{R}^2$ be the starting- and ending- points respectively and $t \in \mathbb{R}$ with $0 \leq t \leq 1$, the LERP is defined as:

$$\text{LERP}(\mathbf{p}_0, \mathbf{p}_1, t) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1 \tag{3.3}$$

The LERP generates a straight line connecting $\mathbf{p}_0$ and $\mathbf{p}_1$. On the contrary, the SLERP generates points along the great circle arc on the surface of the unit sphere as described in eq. 3.4

$$\text{SLERP}(\mathbf{p}_0, \mathbf{p}_1, t) = \frac{\sin(1-t)\Omega}{\sin\Omega}\mathbf{p}_0 + \frac{\sin(t)\Omega}{\sin\Omega}\mathbf{p}_1 \tag{3.4}$$

where Ω is the angle between $\mathbf{p}_0$ and $\mathbf{p}_1$.

Let $\mathbf{q}_0,\ \mathbf{q}_1 \in \mathbb{S}^3$ and $t \in \mathbb{R}$ with $0 \leq t \leq 1$, the SLERP and its derivative are defined as:

$$Slerp\,(\mathbf{q}_0, \mathbf{q}_1, t) = \mathbf{q}(t) = (\mathbf{q}_1 \star \mathbf{q}_0^*)^t \star \mathbf{q}_0 \tag{3.5}$$

$$\begin{aligned}\frac{\mathrm{d}Slerp}{\mathrm{d}t} = \dot{\mathbf{q}}(t) =& \text{Log}\,(\mathbf{q}_1 \star \mathbf{q}_0^*)\,(\mathbf{q}_1 \star \mathbf{q}_0^*)^t \star \mathbf{q}_0 = \\ =& \text{Log}\,(\mathbf{q}_1 \star \mathbf{q}_0^*) \star \mathbf{q}(t),\end{aligned} \tag{3.6}$$

where the superscript $*$ is the quaternion conjugate operator, the symbol $\star$ defines the quaternion product and the Log operator is the *logarithmic map*. From the quaternion kinematics we can write the derivative of $\mathbf{q}$ in t as:

$$\dot{\mathbf{q}}(t) = \frac{1}{2}\mathbf{q}(t) \star \boldsymbol{\omega}(t), \tag{3.7}$$

where $\boldsymbol{\omega}(t)$ is the instantaneous angular velocity vector acting on the object. In (3.7), $\boldsymbol{\omega}(t)$ is the augmented angular velocity with scalar part equal to zero, i.e., $\boldsymbol{\omega}(t) = [0,\ \omega_x,\ \omega_y,\ \omega_z]^T$. The instantaneous angular velocity needed to rotate the object from the initial orientation ($\mathbf{q}_0$) to the final one ($\mathbf{q}_1$) is obtained combining (3.7) and (3.6):

$$\boldsymbol{\omega}(t) = 2\text{Log}\left(\mathbf{q}_1 \star \mathbf{q}_0^*\right) \tag{3.8}$$

Eq. (3.8) shows how the angular velocity remains constant throughout the quaternion interpolation since its value only depends on the *quaterion error* ($\mathbf{q}_1 \star \mathbf{q}_0^*$). Moreover, we are dealing with quaternions belonging only to the northern hemisphere of $\mathbb{S}^3$. This aspect ensures that the SLERP represents the shortest arc between $\mathbf{q}_0$, $\mathbf{q}_1$. Hence, the obtained angular velocity is the optimal one. The logarithmic map for a unit-length quaternion reduces to:

$$\text{Log}(\mathbf{q}) = \left[0, \frac{\vec{q}}{\parallel \vec{q} \parallel} \arccos(q_0)\right] \tag{3.9}$$

Eq. (3.8) can now be rewritten as:

$$\widetilde{\boldsymbol{\omega}} = 2\frac{\vec{\tilde{q}}}{\parallel \vec{\tilde{q}} \parallel} \arccos(\tilde{q}_0) \tag{3.10}$$

where $\tilde{\mathbf{q}} = \mathbf{q}_1 \star \mathbf{q}_0^* = [\ \tilde{q}_0, \vec{\tilde{q}}\ ]$. The angular velocity update equation for the i-th

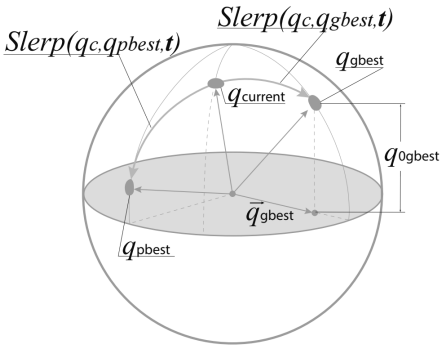


Figure 3.1: Visualization of SLERP interpolation in the angular velocity update equation on the S^3 hypersphere (Projection of 3-sphere into 3D space for visualization purposes).

particle is formulated as follows:

$$\begin{aligned} \boldsymbol{\omega}_i\left(t+1\right) = w\boldsymbol{\omega}_i(t) + \\ c_1 r_1 \left[2\mathrm{Log}\left(\mathbf{q}_{pbest_i}(t) \star \mathbf{q}^*_{current_i}(t)\right)\right] + \\ c_2 r_2 \left[2\mathrm{Log}\left(\mathbf{q}_{gbest}(t) \star \mathbf{q}^*_{current_i}(t)\right)\right] \end{aligned} \tag{3.11}$$

The orientation of the i-th particle is then updated by means of the discrete form of the quaternion kinematics:

$$\mathbf{q}_i\left(t+1\right) = \cos\left(\psi(t)\right)\mathbf{q}_i(t) + \frac{1}{2}\frac{\sin(\psi(t))}{\psi(t)}\mathbf{q}_i(t) \star \boldsymbol{\omega}_i(t+1)T_c, \psi(t) = \left\|\boldsymbol{\omega}_i(t+1)\right\|_2 \frac{T_c}{2} \tag{3.12}$$

T_c represents the integration time of the discrete time quaternion kinematics. In this work T_c just collapses to a tunable parameter as we are dealing with iteration steps (t) rather than with the true definition of time. This new parameter could be employed to scale the total angular velocity obtained in (3.11). In this way, T_c can control the amount of perturbation to apply to the current object orientation.

3.2.2 Objective function

Each particles' object pose hypothesis must be checked against a fitness function to estimate how close that particle is from the true pose of the real object. The algorithm takes as inputs a set of regions $\mathcal{R}$ generated by the segmentation algorithm described in Chapter 2; each one is a cluster of pixels that represents an object. The depth map of that cluster is also extracted and it is the only source of information used in the PSO algorithm. Each particle renders its pose hypothesis against the depth map of the cluster.

Let p_j be the j-th PSO particle. Let $\tilde{\mathbf{q}}_j$ and $\tilde{\mathbf{t}}_j$ be the unit quaternion orientation and translation vector hypothesis of p_j respectively.

Let $r_k \in \mathcal{R}$ be the k-th pixels cluster extracted in Chapter 2. The cluster r_k is a cluster of depth pixels, i.e. the i-th pixel of r_k is defined by the following vector:

$$r_{k_i} = \begin{bmatrix} u_i \\ v_i \\ Z_i \end{bmatrix} \tag{3.13}$$

where (u_i, v_i) are the i-th pixel coordinates and Z_i is the i-th pixel depth datum in millimeter defined w.r.t. camera reference frame.

Every 3D object model is composed by a number N_h of faces (or triangles). Every triangle is defined by three vertices modeled as 3D vectors (e.g. vertex 1 of h-th model triangle $\mathbf{p}_{h_1} = [X_{h_1}, Y_{h_1}, Z_{h_1}]^T$). For each triangle h can be defined a matrix of vertices as follows:

$$\mathbf{p}_{\text{face}_h} = \begin{bmatrix} X_{h_1} & X_{h_2} & X_{h_3} \\ Y_{h_1} & Y_{h_2} & Y_{h_3} \\ Z_{h_1} & Z_{h_3} & Z_{h_3} \end{bmatrix} \tag{3.14}$$

The rendering of a 3D object model (e.g., fig.1.4) into a 2D depth image is performed by the j-th particle as in eq. 3.15.

$$\mathbf{Y}_j = \bigcup_{h=1}^{N_h} \mathbf{b}_{jh} = \mathcal{F}_{\text{edge}} \left[\mathcal{K}\mathcal{T}\left(\tilde{\mathbf{q}}_j, \tilde{\mathbf{t}}_j\right) \mathbf{p}_{\text{face}_h} \right] \tag{3.15}$$

where: $\mathcal{K}$ is the intrinsic depth camera matrix defined in 5.2. $\mathcal{T}\left(\tilde{\mathbf{q}}_j, \tilde{\mathbf{t}}_j\right)$ is the homogeneous transformation matrix built upon the j-th particle's object pose hypothesis. It converts all the triangles vertices defined in the object model reference frame to the depth camera reference frame. $\mathcal{F}_{\text{edge}}$ is the rendering function that fills with depth values all the pixels inside the 2D projected model face. $\mathbf{b}_{jh}$ is the h-th 2D filled projected triangle on the image plane (see Chapter 4). Finally, $\mathbf{Y}_j$ is the final 2D rendered pixel cluster of the given object model based on the object pose hypothesis of the j-th particle.

The comparison is thus performed between each rendered cluster $\mathbf{Y}_j$ and the real segmented cluster r_k.

The fitness value of the j-th particle is thus computed as follows:

$$\Phi_j = \frac{\alpha}{N_{R_j}} \sum_{i=1}^{N_{R_j}} \left(z_{K_i} - z_{R_{ij}} \right)^2 + \beta \, \frac{\mu_j + \kappa_j}{2} \tag{3.16}$$

where: N_{R_j} is the number of pixels of the depth map rendered by the j-th particle, $z_{R_{ij}}$ is the depth value of the pixel i rendered by the j-th particle, while z_{K_i} is the corresponding depth value of the cluster at pixel i. α and β are two constant parameters used to weight the two terms of the fitness function. In the fitness function, a second term along with the depth error one is needed. Depth error alone might generate ambiguity, leading to wrong pose estimation in some special cases (e.g., a box could fit one of its smaller sides against its largest size that is visible in the segmented cluster, giving a depth error close to zero even if the particle's pose is wrong). The term μ_j models the percentage of cluster pixels that

are not covered by the rendered 3D model of particle j:

$$\mu_j = \frac{N_{CW_j}}{N_{PC}} \in [0,1]\,. \quad \text{If } \mu_j = \begin{cases} 0 & \text{perfect match} \\ 1 & \text{the rendered object is outside the cluster} \end{cases}$$

where N_{CW_j} is the cluster's area (in pixels) that is not covered by any pixel of the rendered object of the particle j and N_{PC} is the area of the segmented cluster. The condition $\mu_j = 0$ could also hold when the rendered object has a larger area than the cluster and it is covering the entire cluster. Hence, the term κ_j is added to compensate for this problem.

κ_j is the complement of μ_j i.e., it gives the percentage of rendered pixels of particle j that are not covered by valid depth values in the cluster depth map:

$$\kappa_j = \frac{N_{RW}}{N_{R_j}} \in [0,1]\,. \quad \text{If } \kappa_j = \begin{cases} 0 & \text{perfect match} \\ 1 & \text{the rendered object is outside the cluster} \end{cases}$$

where N_{RW} is the rendered object's number of pixels which do not correspond to pixels of the segmented cluster.

Figure 3.2 shows four cases where the depth error would be low, but the estimate pose would be incorrect. Figures 3.2a and 3.2d also show the effect of μ_j and κ_j. If we consider Figure 3.2a, the object model is rendered completely inside the object cluster due to a wrong pose estimation, but κ_j equals zero. The index μ_j instead hold into account that a part of the cluster is not covered by the rendered object. In Figure 3.2d the rendered object is larger than the cluster, but since all the pixels in the cluster are covered by the rendered object, μ_j is zero. Instead, κ_j assumes an high value, taking into account how many rendered pixels fall outside the cluster. Figures 3.2b and 3.2c show intermediate situations.

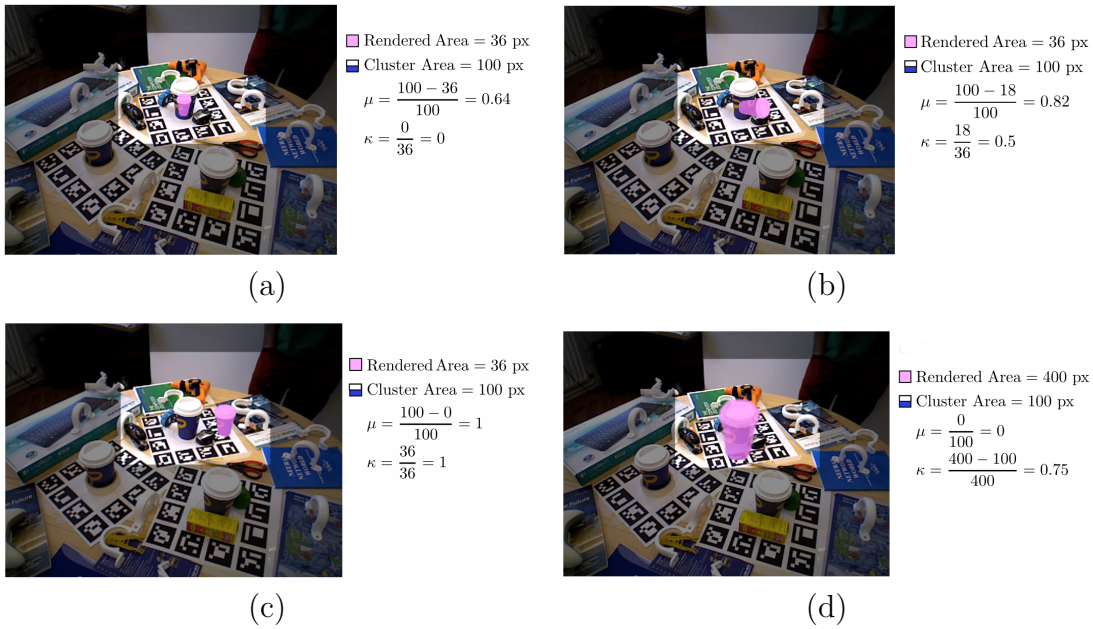


Figure 3.2: (a) The rendered particle is inside the object cluster. (b) The rendered particle is only partially inside the object cluster. (c) The rendered particle is outside the object cluster (d) The rendered particle covers the object cluster.

3.2.3 PSO initialization

The PSO requires an initialization step in which different object pose hypotheses are assigned to each particle. The segmentation phase provides a rough approximation of the 3D centroid of a cluster. The latter is not necessarily the 3D centroid of the real object as errors in the clustering step might lead to either over-segmentation (e.g., an object is split in two or more clusters), or under-segmentation (e.g., a cluster does not enclose the whole object; thus object borders or even small parts of an object are missing). However, the 3D centroid of a cluster ($\bar{\mathbf{c}}$) can be exploited to generate the position component of the j-th particle ($\mathbf{t}_j$) as follows:

$$\mathbf{t}_j = \hat{\mathbf{t}}_{lo} + \left(\hat{\mathbf{t}}_{hi} - \hat{\mathbf{t}}_{lo}\right) \boldsymbol{\delta} \tag{3.17}$$

$$\mathbf{v}_j = -\tilde{\mathbf{v}} + 2\tilde{\mathbf{v}}\boldsymbol{\delta} \tag{3.18}$$

$$\hat{\mathbf{t}}_{lo} = \bar{\mathbf{c}} - \tilde{\mathbf{t}};, \hat{\mathbf{t}}_{hi} = \bar{\mathbf{c}} + \tilde{\mathbf{t}};$$

where: $\boldsymbol{\delta} \backsim \mathcal{U}(0,1)$; $\tilde{\mathbf{t}}$ is a constant relative position vector used to define the search space domain of the translation component of the object pose. Eq. (3.18) assigns a linear velocity, between the constant values $\pm\tilde{\mathbf{v}}$, to the particle j.

The segmentation step cannot generate an estimate of the object orientation, so the object attitude initialization and optimization are performed on the whole surface of the northern hemisphere of $\mathbb{S}^3$. Let $\mathbf{q}_{init} \in \mathbb{S}^3$ be a constant attitude quaternion lying on the surface of the northern hemisphere of $\mathbb{S}^3$, the initial orientation of the j-th particle is generated as follows:

$$\mathbf{q}_j = \cos(\psi)\,\mathbf{q}_{init} + \frac{1}{2}\frac{\sin(\psi)}{\psi}\mathbf{q}_{init} \star \widehat{\boldsymbol{\omega}}_j T_c \tag{3.19}$$

$$\widehat{\boldsymbol{\omega}}_j = \widetilde{\boldsymbol{\omega}}_{lo} + (\widetilde{\boldsymbol{\omega}}_{hi} - \widetilde{\boldsymbol{\omega}}_{lo})\,\boldsymbol{\delta} \tag{3.20}$$

$$\psi = \|\widehat{\boldsymbol{\omega}}_j\|_2 \frac{T_c}{2}$$

where Eq. (3.20) assigns a random angular velocity to the particle j. The dynamic range of the initial angular velocity is limited by the values $\widetilde{\boldsymbol{\omega}}_{lo}$ and $\widetilde{\boldsymbol{\omega}}_{hi}$. The larger the difference $(\widetilde{\boldsymbol{\omega}}_{hi} - \widetilde{\boldsymbol{\omega}}_{lo})$, the greater will be the perturbation of the initial attitude quaternion $\mathbf{q}_{init}$. Experiments show that the choice of $\mathbf{q}_{init}$ has no influence on the convergence of the PSO as long as a wide dynamic range of the initial angular velocity is provided. This result corroborates the fact that our algorithm converges to the actual object pose without any prior knowledge about the object attitude.

We also experimentally determined that the final fitness value of the best particle can be used to discriminate correctly detected objects from false positives. This is necessary, since the segmentation part inevitably produces a number of false positive regions.

3.3 Experimental results

The software has been developed using the CUDA and the OpenCV library in C++ under Linux. In our experiments, The algorithm was tested on a workstation equipped with a Tesla K40 GPU, while the segmentation part runs on CPU. The source code for Q-PSO is available online [1].

The video at this link: Particle's Trajectory, shows the trajectory of a particle toward the search space. The particle, at each time step, renders the 3D CAD model of a shampoo bottle onto the 2D depth image of the candidate cluster regions $\mathcal{R}$. The fitness score of that rendering is thus computed and the particle current object pose hypothesis is updated. Hence, a new particle rendering process is performed. Note how the particle tends to converge toward the optimal pose estimation of the shampoo bottle. Initially, the particle explores new object poses; accordingly the object poses are quite far from each other between two consecutive steps. As the iteration steps increase, the particle velocity slows down and a finer object pose estimation begins.

3.3.1 Complete pipeline evaluation

We tested the complete algorithm on two public datasets for 3D pose estimation [23] and [59]. In our experiments all the 3D models were decimated to 3072 faces, which offers the best time performance with our GPU setup. This is a tradeoff, since subsampling the 3D object will have a slight effect on the accuracy of pose estimation.

[25] contains 15 registered video sequences, each with a texture-less 3D object surrounded by clutter. Results are shown in Table 3.1. We compare the results with ground truth using the metric from [25]. The proposed apporach performs

[1] https://github.com/morpheus1820/Q-PSO

slightly better for four objects (phone, duck, eggbox and glue), comparable to [59] for one object (driller), and with lower accuracy for one object (bench vise). The reason for that is discussed afterwards and in Figure 3.4.

In [59], 6 objects are captured under varying viewpoint with lots of background clutter, scale and pose changes, and in particular foreground occlusions and multi-instance representation (three instances of the same object are present in each frame as well as other objects and clutter). We tested the approach on a subset of objects and scenes. Results are shown in Table 3.2. We compare the results with the metric from [25]. We see comparable accuracy for one object (coffee cup) and lower accuracy for one object (juice carton), while our method outperforms the others on two objects (shampoo and milk). We used the following fixed parameters on all the experiments: $\gamma = 0.001, k_x = 1.2, k_b = 0.05, \alpha = 1, \beta = 0.05, T_c = 1, \widetilde{\boldsymbol{\omega}}_{lo} = [0, -10, -10, -10]^T$, $\widetilde{\boldsymbol{\omega}}_{hi}$ = [0,10,10,10]^T, $c_1 = c_2 = 1, w = 0.3, \tilde{\mathbf{t}}$ = [0.3,0.3,0.3], $\tilde{\mathbf{v}}$ = [3,3,3].

The segmentation part runs on CPU and the average processing time per image is 0.4s; the pose estimation part runs on GPU. In our experiments we used 1024 particles and run 10 PSO iterations for each segmented cluster and the total time is 85ms for each cluster. We use global topology for the PSO. By comparison, [25] requires a training stage of 17-50s for each object and 119ms for detecting an object, but under some position and rotation constraints (0-90° for tilt, ±45° for in-plane rotation, 65-115cm for scaling); our approach operates on the whole north hemysphere of $\mathbb{S}^3$.

Figure 3.3 shows some examples of the results on different datasets. In Figure 3.4 we also show some failed pose estimation cases and discuss the probable causes. We found out that failure in pose estimation is due to bad segmentation results most of the time; in one case, however, as shown on the left column of Figure 3.4, we can see how dowsampling the 3D CAD model of a particularly complex object can affect the computation of the fitness function and lead to wrong pose

estimation.

Figure 3.3: Examples of the approach on different images. First column: RGB image; second column: segmented objects; third column: rendered objects model superimposed.

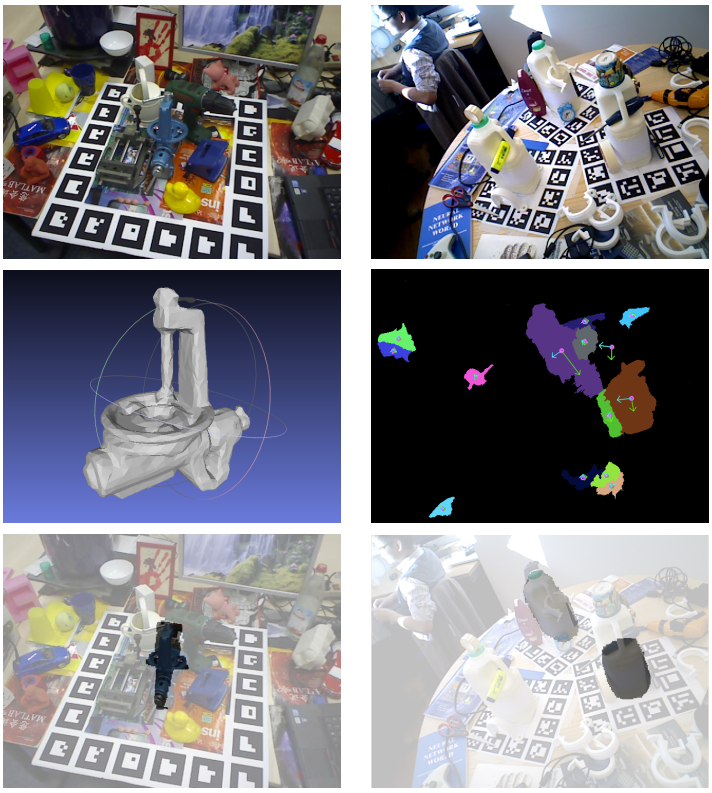


Figure 3.4: Examples of failed detections. First column: wrong estimated pose probably due to the complex model and its over-simplification after subsampling; second column: a case with a true positive (top object), a false positive (left object) and a wrong pose estimation (right object) both due to bad segmentation.

Approach	[23]	[59]	Our Approach
Sequence			
Bench Vise	0.85	0.96	0.72
Driller	0.69	0.9	0.9
Phone	0.56	0.73	0.98
Duck	0.58	0.91	0.97
Eggbox	0.86	0.74	0.95
Glue	0.44	0.68	0.8

Table 3.1: Comparison between different approaches on the [25] dataset.

3.3.2 Extension to articulated objects

An interesting property of the PSO formulation is that it can be easily adapted to articulated objects. We define an articulated object as an object composed by an

Approach	[23]	[59]	Our Approach
Sequence			
Coffee Cup	0.82	0.88	0.85
Shampoo	0.63	0.76	0.9
Juice Carton	0.49	0.87	0.4
Milk	0.18	0.39	0.67

Table 3.2: Comparison between different approaches on the [59] dataset.

arbitrary number of parts $\mathcal{P} = (P_0, \ldots, P_i)$, where $\mathcal{P}_0$ is the main part, connected by a set of joints $\mathcal{J} = (j_0, \ldots, j_i)$. Each joint can can be either prismatic or revolute. The state vector for each particle is augmented with the position and velocity of each joint. The velocity v^{art} for each joint j is computed as:

$$\begin{aligned} v_i^{art_j}(t+1) = w v_i^{art_j}(t) + c_1 r_1 [x_{pbest}^{art_j}(t) - x_i^{art_j}(t)] + \\ c_2 r_2 [x_{gbest}^{art_j}(t) - x_i^{art_j}(t)]. \end{aligned} \tag{3.21}$$

In Figure 3.5 we show an example of the PSO running on two articulated objects (laptop and cabinet) composed by two parts joined by a revolute joint and a prismatic joint respectively, from the Articulated Object Challenge dataset [39].

Figure 3.5: Examples of the algorithm running on 2-parts articulated objects.

3.3.3 Robustness of pose estimation to segmentation inaccuracy

We also evaluate the effect of the segmentation accuracy on the pose estimation performances. Degradation of the cluster from the ground truth is measure using the image dissimilarity index presented in [8]. We show five levels of degradation, from 0% (perfect contours) to 50%. In Table 3.3 we report the effect of noise affecting the segmented cluster on the PSO performance for the same objects as Table 3.2. We report the average pose estimation accuracy over 10 runs for each object using the same metric as [25]. In Figure 3.6 we also show some visual results on the articulated dataset. It can be seen that the PSO is unaffected up until 30% of degradation, after which the pose estimation consistently fails. This conforms with what we expect, as the noise on the segmented cluster borders has a direct effect on the computation of the second part of the fitness function. Current learning-based approaches like [23] and [59] do not rely on a segmentation step, but are still affected by the same issues that can cause bad segmentation, such as presence of strong textures or lack of textures, and noisy depth images.

Degradation	0%	15%	30%	45%	50%
Object					
Coffee Cup	0.8	0.8	0.8	0.3	0.3
Shampoo	0.9	0.9	0.7	0.1	0
Juice Carton	0.5	0.5	0.5	0	0
Milk	0.6	0.5	0.5	0.2	0.1

Table 3.3: Effect of segmentation noise on the performances of pose estimation..

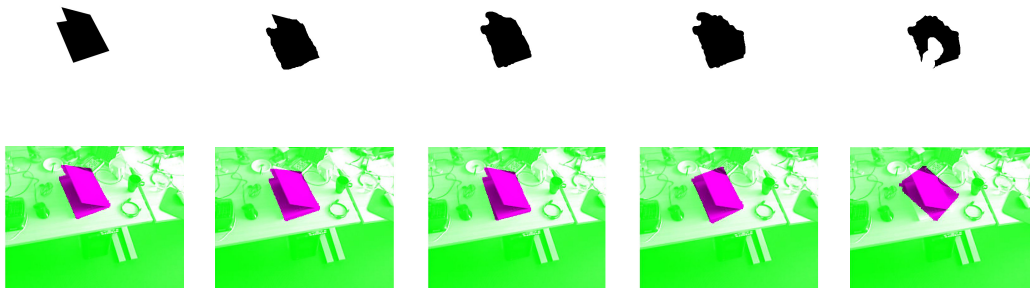


Figure 3.6: Effect of segmentation noise on the performances of pose estimation. First row: degraded segmentation masks (0% to 50%); second row: estimated poses.

Chapter 4

GPU implemetation and optimization of the Q-PSO

4.1 Introduction

The PSO algorithm belongs to the class of parallel metaheuristic. Every PSO particle can evolve and test its current fitness to solve the optimization problem concurrently. This peculiarity is well suited to exploit the huge parallel computation capacity of modern GPUs (e.g. , [62], [5], [44]).

A number of GPU implementations of PSO have been proposed (e.g. , [62], [5], [44]). They all present their own GPU implementation of the standard PSO algorithm considering different types of particles' topology. Final results and comparisons with the other methods are then carried out through the most common benchmark functions (e.g., Rosenbrock, Schwefel, Rastrigin and Griewank functions).

Our optimized GPU algorithm differs widely from the above cited ones. The evaluation of the particles' fitness score must be preceded by a parallel rendering phase in which each particle renders the 3D object model through its own pose hypothesis. The rendering data are thus gathered by the fitness function that updates the quality of the particles' object pose hypothesis in a concurrent way.

Hence, the standard PSO algorithm must be rearranged with new data structures to guarantee the maximum utilization of the GPU resources. New optimization techniques are thus carried out in order to develop an efficient quaternion-based PSO implementation:

Algorithm 1 shows the main algorithmic flow of the GPU implementation. All functions are CUDA kernels. Each GPU thread is uniquely allocated to a particle, unless otherwise stated.

4.2 Variables definition

Let DIM be the dimensions of the search space and $NPART$ be the swarm size. We create the following one-dimensional arrays:

- ***d_pso_pose***$[DIM * NPART]$: particles current pose.
- ***d_pso_vel***$[DIM * NPART]$: particles current velocity.
- ***d_pso_pose_b***$[DIM * NPART]$: particles best pose.
- ***d_randGen***$[H * NPART]$: random float numbers, where $H >> DIM$.
- ***d_randIdx***$[NPART]$: it is used to keep track of the index each thread has within the ***d_randGen*** vector.
- ***d_depth_buffer***$[R * C * NPART]$: particles depth buffer, where R is the depth image rows and C is the depth image columns.
- ***d_AABB***$[NPART * 64 * 4]$: Partial Axis-Aligned Bounding Box (AABB) of the rendered 3D objects by the particles.
- ***d_finalAABB***$[NPART * 32]$: Final AABB of the rendered 3D objects by the particles.

- ***d_obj_model***$[3 * NUM\_VERTICES]$: Array of the 3D object model vertices.
- ***d_depth_kinect***$[R * C]$: Post-segmentation Depth Image of the scene captured by the Kinect sensor.
- ***d_pso_fit_error***$[NPART]$: Fitness score of each particle in a given loop iteration.
- ***d_pso_personal_best_fit***$[NPART]$: Particles' personal best fitness score.
- ***d_solution_best_fit***[1]: Global Best Fitness Score
- ***d_solution_best_pose***$[DIM]$: Global Best Object Pose in Global Topology
- **warpSize**=32: Constant CUDA *Warp* size.

4.3 PSO Initialization on GPU

Algorithm 3 highlights the main steps of the Q-PSO initialization process present inside the ***initAllParticles()*** CUDA kernel. Let the number of particles (NPART) be equal to 1024. This kernel is launched with 8 blocks of 128 threads each in order to allocate 1 thread per particle. Firstly, each particle initializes the local variable *s_randIdx* that it will use to fetch, from global memory, the random numbers ($\boldsymbol{\delta}$) required by Eqs (3.17),(3.18),(3.19),(3.20). The layout of the d_randGen array depicted in Fig 4.1 has been designed to guarantee a coalesced warp access to global memory by indexing the array as ***d_randGen***$[tIdx+($***s_randIdx***$++)$*NPART]. We decided to fill an array of dimension $[H * NPART]$ in CPU with random numbers and then upload it only once in device global memory at start-up. In our tests, we noticed how this approach along with the coalesced memory access is much faster than using the GPU random number generator offered by the CUDA

library. Moreover, it guarantees backwards compatibility with older CUDA versions.

The particles then compute their own indices in agreement with the array layout specified in Fig 4.3. Therefore each parallel thread initializes its particle current pose, its particle personal best pose and its particle current velocity in a coalesced fashion.

Finally each particle stores to device global memory its own current random number index for the next iteration.

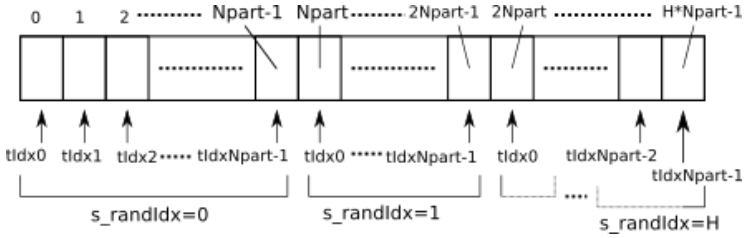


Figure 4.1: ***d_randGen*** array layout.

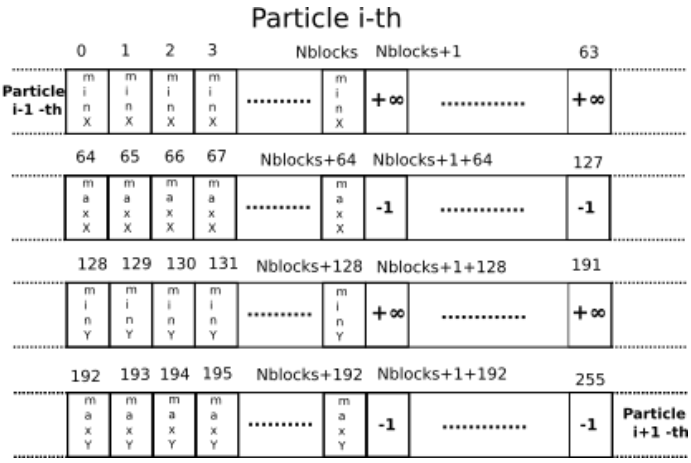


Figure 4.2: Layout of ***d_AABB*** for the i-th particle.

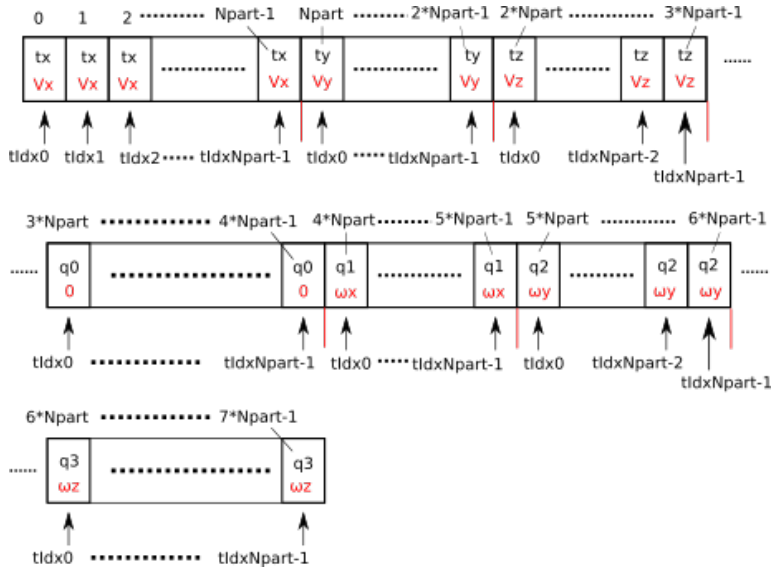


Figure 4.3: ***d_pso_pose***, ***d_pso_pose_b*** (black array elements), ***d_pso_vel*** (red array elements) arrays' layout to guarantee the coalesced memory access

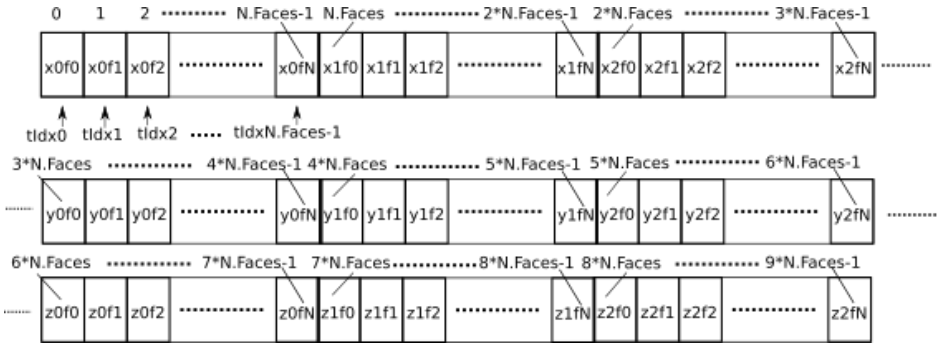


Figure 4.4: ***d_obj_model***: array layout of the object model vertices to guarantee the coalesced memory access

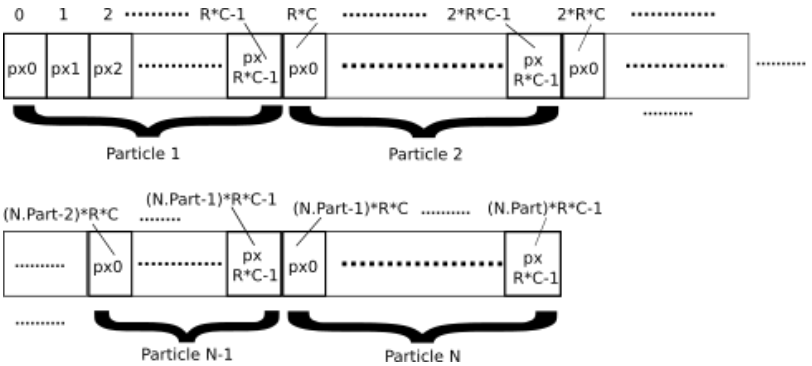


Figure 4.5: ***d_depth_buffer***: array layout of the depth buffer used by each particle to render its own 3D object model.

4.4 Rendering of the particles pose hypotesis

The particles move through the search space trying to minimize a given fitness function. This function must be designed in a way that is able to discriminate a good object pose hypothesis from a wrong one. A given object pose hypothesis can be quantified if and only if a particle utilizes it to render the 3D object model against the 2D object cluster. A 2D object cluster can be obtained by any segmentation algorithm that works on RGB-D images, as the one we proposed in this Ph.D. dissertation. In computer graphics, the most used pipeline to render a 3D object model is OpenGL. Unfortunately the OpenGL architecture does not fit well with our needs of both high computational speed and full particles parallelism. Actually, OpenGL would have broken the desired particles parallelism and would have slowed down the optimization process. The OpenGL pipeline cannot be called inside any CUDA kernel. Therefore, the only workaround would have been to load back to CPU all the particles' pose. Moreover, parallel OpenGL instances cannot be instantiated. This would have led to a single OpenGL instance designed to render each particle sequentially onto NPART different depth buffers. Different depth buffers are required to avoid any depth value overwriting. Ultimately, a copy of the CPU depth buffers should have been uploaded to device global memory to continue the GPU algorithm flow.
In the light of foregoing, we designed our own software rendering pipeline directly in GPU. This solves the hardware rendering limitations ensuring parallel rendering among the particles and no memory copy between CPU and GPU at each iteration.

The rendering phase is intrinsically a parallel process, as multiple mesh triangles can be drawn at once onto the depth buffer. We leverage this property to build an optimized version of the **edge function**(see [46]) that ensures high GPU occupancy. The goal is to design a kernel that is able to render each particle's object pose hypothesis independently from the others and simultaneously it must

process each particle mesh triangles concurrently. This two levels of parallelism is achieved using the CUDA streams. If multiple kernels are launched on a single CUDA stream, they will be executed sequentially. On the contrary, if multiple kernels are launched on different CUDA streams they will run, if possible, concurrently. The approach is shown in Algorithm 2. The kernel ***renderAllParticlesPoseHp()*** is launched with one thread per triangle mesh to obtain the first level of parallelism. NSTREAMS are instantiated and launched to achieve the second level of parallelism, i.e. the concurrent particles. GPU hardware limitations forced us to set the numbers of streams to 16 instead of NPART. A complete particle parallelism cannot be accomplished. Our experiments showed that opening more than 16 streams does not reduce anymore the time of the rendering phase.

Algorithm 4 outlines the main steps of the parallel rendering process. Let the object mesh faces be equal to 3072. The ***renderAllParticlesPoseHp*** kernel is launched with ***Nblocks***=6 blocks of ***NthsPerBlock***=512 threads each; i.e., a thread per object triangle. This kernel is also designated to find the axis-aligned bounding box (AABB) of the rendered object onto the depth buffer. The AABB keeps track of the area onto the depth buffer which has been written by a particle. This technique will narrow the search area when the fitness function is computed. We optimized the AABB calculus inside this kernel by exploiting the shared memory properties. Each block instantiates its own shared memory and only the threads belonging to that block can access it. To store the top-left and the bottom-right corners coordinates of a AABB are needed four values. We thus instantiate four shared memory arrays of dimensions 512 (i.e. the numbers of threads per block) and we initialize them (Alg. 4, lines 4 to 8).
Lines 9 to 14, in Alg. 4, project the 3D mesh triangle of index *tIdx* onto 2D pixels coordinate. Lines 15 to 18, in Alg. 4, then compute the 2D triangle's AABB. Lines 19 to 23, in Alg. 4, save, in an univocal array position, the current coordinates of the triangle AABB. So far, each block holds a subset of triangles' AABB, as tough

the final AABB was fragmented in 6 groups of 512 smaller chunks. Hence, line 24 performs a *Parallel Min Max Reduction* optimized to run in shared memory. The outcome of this function is to join all the 512 smaller AABBs per block into a single AABB per block. At this stage, each block owns only a partial AABB. The OR-operation of the 6 AABBs would build the final AABB up. Unfortunately, the final AABB of the particle rendering cannot be computed inside this kernel as shared memory access is local to blocks. To avoid any atomic operations that will serialize the threads, we decided to define in device global memory a temporary array of partial results ***d_AABB*** (see Fig. 4.2) where each particle can save in a coalesced way its 6 partial AABBs (lines 26 to 30). The total length of ***d_AABB*** is NPART*64*4. Each particle has 64*4 elements reserved. 4 are to store the two rectangle corners coordinates. 64 are twice the CUDA *warp* size.Even if $64 - Nblocks$ elements are not used at all, this choice lets the optimized parallel reduction run as fast as possible.
Finally, each triangle is drawn onto the depth buffer concurrently (line 31). The layout of the depth buffer *d_depth_buffer* is depicted in Fig. 4.5.

4.5 Reconstruction of the final AABB

The ***ComputeAABBFinalReductionKernel*** kernel is assigned to reconstruct the final AABBs of the renderings performed by all the particles. This function takes as input the *d_AABB* array. The layout of this array, as already explained, has been designed to optimize and speed-up the process of forming the final AABB. We reserved $64 * 4$ elements per particle. 64 elements per particle might be thought of as a waste of memory since only 6 (i.e., *Nblocks*) out of 64 values are needed. A multiple number of the CUDA Warp Size is actually required to perform the fastest parallel reduction in GPU. The number 64 turns out to be the smallest number that is a multiple of the Warp Size (i.e., 32).

We thus launched this kernel with 1 block per particle and $64 * 4 = 256$ threads per block. 256 threads are 8 warps, this means that we can assign 2 warps to load in shared memory each coordinate of the partial AABBs. This choice ensures both a bank conflict free write in shared memory and a coalesced read from global memory. Lines from 15 to 30 in Alg. 5 outline the read and write procedure.
The block locality property of the shared memories are no more a problem in this kernel as each block corresponds to a PSO particle. Therefore, each particle can execute the OR-operation of its partial AABBs directly in shared memory to obtain its final AABB. The *MinMaxAABBReduction* function in line 31 computes this OR-operation. Finally, the particles' final AABB is stored in device global memory.

4.6 Fitness function on GPU

Each particle must now quantify the goodness of its own object pose hypothesis. The fitness function introduced in Eq. 3.16 gathers information from the rendering step, such as the pixels depth value (z_{K_i}, $z_{R_{ij}}$) and the rendering coverage indices (i.e., μ_j, κ_j).
The particles' rendering depth values $z_{R_{ij}}$ and the object cluster depth values z_{K_i} can be read respectively from the *d_depth_buffer* and the *d_depth_kinect* arrays. In contrast, the rendering coverage indices must be computed just before the evaluation of the fitness function. We leverage the AABBs computed in the previous kernel to reduce the number of accesses in global memory and thus speed-up the computations.
The ***ComputeAllParticlesFitness*** kernel is launched in the same way of the rendering kernel. We open 16 streams and run the kernel with 1 block of 1024 threads. This configuration leaves available 1024 threads per particle. We designed this approach to generate two levels of parallelism in this phase too. The particles

process 1024 pixels at once while they evaluate their own object pose hypothesis concurrently. The AABBs are not of fixed area; moreover, we do not know the number of rendered pixels inside the AABB until we count them. Dynamic memory allocation is a very expensive operation in GPU that would have greatly slowed down the computation of the fitness scores. We thus create four circular buffers in shared memory (Lines 6 to 9 in Alg. 6) where we store: the partial results of the pixels depth error (*sError[]*), the partial indices value (*sμ[]*, sκ[]) and the partial number of rendered pixels of that particle (*sNrenderedPoints*[]).

Firstly, we load in shared memory the final AABB of each particle. The array d_finalAABB is of dimension [NPART * 32] even if the number of elements necessary to define a AABB is only four. We reserve 32 elements as the CUDA warp size to allow a coalesced warp access in device global memory (Lines from 11 to 15 in Alg. 6).

The definition of the rendering coverage indices force us to find the minimum AABB that encloses both the particle rendering and the segmented cluster object. An OR-operation is then performed between the rendering AABB and the cluster one (lines from 24 to 27 in Alg. 6).

Secondly, we launch a *grid-stride loop* to access 1024 pixels at once and we loop *NumIters* times. *NumIters* is the number of loops needed by a block of 1024 threads to access all the pixels in the AABB only once. In line 45 we fill the circular buffers with partial results obtained from the current 1024 pixels.

The partial results are then summed up through a parallel sum reduction over the circular buffers (line 46). Finally, the first active thread computes Eq. (3.16) and writes the resulting fitness score of the particle *pIdx* in the *d_pso_fit_error* array in global memory.

4.7 Updating personal and global best on GPU

The ***ComputeAllParticleFitness*** kernel fills the *d_pso_fit_error* array with the current fitness scores of the particles. The ***updatePersonalAndGlobalBestAllParticles*** kernel takes as input this array and test whether the particles have improved their personal fitness or not. Moreover, this kernel tests whether some particles have exceeded the global best fitness score up to that iteration. The personal best update is a complete parallel process. In contrast, the global update in a PSO with global topology is intrinsically a serial process where an atomic read and write must be performed on the global best variable to avoid particles race condition. We designed this kernel such that no atomic operations are executed.
Before launching the ***updatePersonalAndGlobalBestAllParticles*** kernel, we run a *parallel minimum reduction* on the *d_pso_fit_error* array. The outcomes of this command are both the best fitness score (*d_result_min_fit*) and the ID of the best particle *best_particle_Idx*, in that iteration. These temporary results are passed to the kernel. The kernel is then launched with 1 thread per particle. Therefore, each particle fetches its current fitness score and personal best score. It tests whether its current score is strictly minor of its personal best. If this condition falls true, the particle updates its personal best and copies its current pose as its best pose.
The update of the global best follows a slightly different approach. A single check is required to determinate if the set of particles owns a particle with a lesser fitness score. The first active thread is then elected to check the fitness score of the best particle of that iteration (*best_particle_Idx*) against the global best score. If the strictly minor condition holds true, the global best fitness is set to the fitness of the best particle. The pose of the best solution is then updated with the pose of the best particle. Alg. 7 highlights the above steps.

4.8 Updating the particles pose and velocity

The kernel ***computeNewPoseAndVelAllParticles*** is used to propagate the particles through the search space. In this kernel the update Eqs. (3.1), (3.2),(3.11) and (3.12) are employed. These equations can be safely computed concurrently. Hence, we run this kernel with 1 thread per particle. Each particle calculates its indices to access the device global memory in a coalesced manner (Lines 4 to 10, Alg. 8). In line 11, in Alg. 8, the particles store in local memory the current random index. The actual particles propagation is performed starting from line 12 to 25 of Alg. 8. At the end, the particles save their current random index for the next iteration (line 26 in Alg. 8).

Chapter 5

Human-Robot collision avoidance for a safe coexistence

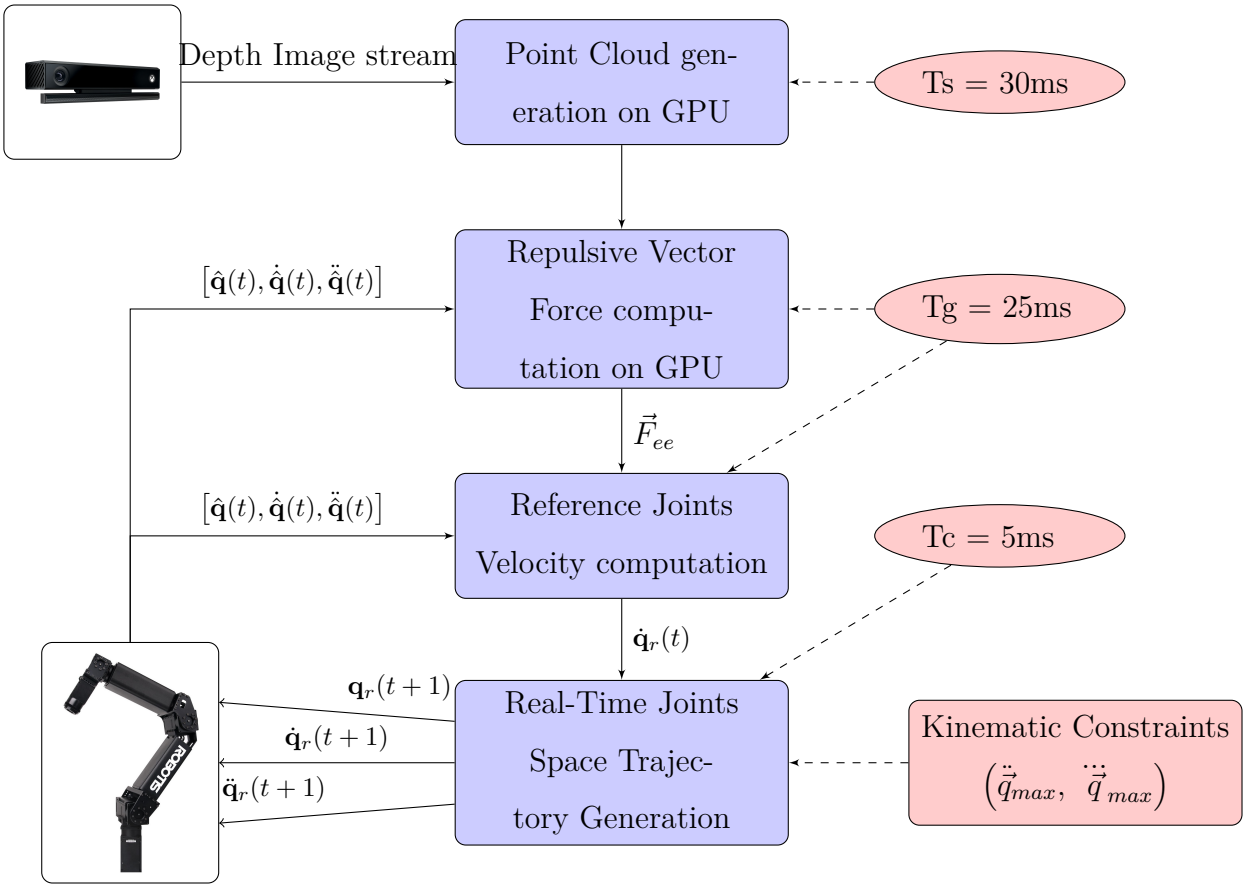
In the last few years, in the robotics community has grown the idea about bringing closer humans and robots. Human-Robot collaboration could be the right union to improve the skills of both the parties involved. In industrial robotics applications a robot co-worker can simplify the personnels job; e.g. the robot speeds up some repetitive or heavy tasks while the operator is performing the peculiar ones. They both work on the same machine, they share the same working area but they execute different tasks in parallel. This may yield to an increase of the productivity for that company.

The human-robot collaboration paradigm is based on the concept of the robot workspace sharing with no physical guarding barriers between human and robot. This inevitably leads to some human safety related issues. The main goal for robotics engineers is therefore the generation of a *virtual environment* where the human-robot coexistence is guaranteed to be safe. The first step toward a safe coexistence is the employment of a new generation of collaborative and lightweight manipulators. KUKA and Universal Robots with their own LWR and UR products respectively, are the leaders in the field of collaborative robotics. These robots

have been designed to be lightweight and precise. They have redundant joints encoders to guarantee a safe robot stop in presence of failure. KUKA has also joint torque sensors at each joint and harmonic drive that introduces joint elasticity. New impedance control schemes have been proposed with or without torque/force sensors to handle and react to any accidental contacts against both the environment and the human (see [38] and [37]).

This Chapter is focused on the design of a collision avoidance scheme for human-robot coexistence. For this porpoise, we built our own lightweight manipulator. The designed robot has 7DoF in order to be redundant for any task assigned to it. To validate our algorithm, we first assign to the robot a task that must be fulfilled (e.g. the robot is programmed to execute a given trajectory). Then, we stress any collision that may occur in a real scenario of human-robot collaboration. Collision avoidance against both people and objects are tested.

5.1 Overview of the proposed control scheme



The above block diagram highlights the main steps of the proposed collision avoidance algorithm. The Microsoft Kinect is the only exteroceptive sensor employed in this dissertation. The Kinect captures a new depth frame of the surrounding environment at a frequency of 30Hz. The monitored environment covers the robot workspace and any object and/or person that may interfere with the robotic arm. The Kinect sensor is placed at a horizontal distance of 2 meters and at a height of 1.2 meters w.r.t. the robot base frame. Once a new depth frame is ready, the *point cloud generation* block converts the depth image in an organized Point Cloud. The computed point cloud is then moved to the *repulsive vector force generation* block. The latter reads the actual joint state of the robot (i.e. $\left[\hat{\mathbf{q}}(t), \dot{\hat{\mathbf{q}}}(t), \ddot{\hat{\mathbf{q}}}(t)\right]$) and compute a force vector for each point in the point cloud that

may interfere with the main robot task. The outcome of this block is a cumulative vector force $\vec{F}_{ee}$ that must be applied to the robot end-effector in order to modify the current robot trajectory in case any imminent collision has been detected. The *reference joints velocity computation block* takes as input the cumulative force vector and generates the reference joints velocities ($\dot{\mathbf{q}}_r(t)$) required to execute the desired collision-avoidance trajectory. Both the *reference joints velocity computation* block and the *repulsive vector force generation* one have a sampling time of 25ms. Finally, a real-time, 7-*th* order *joints space trajectory generation* block is needed to respect the robot kinematic motion constraints $\left(\vec{\ddot{q}}_{max},\ \vec{\dddot{q}}_{max}\right)$ and avoid jerky movements of the robotic arm. This block generates a new trajectory point $[\mathbf{q}_r(t+1),\ \dot{\mathbf{q}}_r(t+1),\ \ddot{\mathbf{q}}_r(t+1)]$ which is sent to the robot controller every 5ms.
The following chapter sections will explain each block in detail.

5.2 Point Cloud generation on GPU

The depth image captured by the Kinect sensor is a 2D image where each pixel encodes the distance between the camera frame and the object seen by that pixel. Hereafter the *i-th* depth image pixel can be expressed by the following three coordinates:

$$\mathbf{p}_{x_i} = [u_i, v_i, Z_i]^T \tag{5.1}$$

where (u_i, v_i) are the pixel coordinates in the image plane, whilst Z_i is the depth information, in millimetres, carried by the *i-th* pixel.
Basically, a *point cloud* is a collection of 3D points (e.g. $\mathbf{p}_{c_i} = [X_i, Y_i, Z_i]^T$) used to discretize any object in the 3D space. This block is asked to convert a depth image to a point cloud.
The Kinect depth sensor is simply modelled as pin-hole camera. The latter is characterized by an intrinsic camera matrix $\mathcal{K}$ required to project a pixel from the

2D image reference frame to the camera 3D reference frame and vice versa.

$$\mathcal{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

where f_x and f_y are the focal lengths of the camera; c_x and c_y are the coordinates in pixel of the center of the image plane. Hence, eq. 5.3 projects a depth image pixel $\mathbf{p}_{x_i}$ into the 3D camera reference frame.

$$\mathbf{p}_{cam_i} = \mathcal{K}^{-1}\mathbf{p}_{x_i} \tag{5.3}$$

The extrinsic camera matrix $\mathcal{T}_{base}^{cam}$ defines the rigid transformation between the camera frame and the robot base frame.
To obtain the point cloud of the scene in robot base frame the rigid transformation in eq. 5.4 must be applied.

$$\mathbf{p}_{base} = (\mathcal{T}_{base}^{cam})^{-1}\,\mathbf{p}_{cam} \tag{5.4}$$

We coded the above rigid transformations in GPU since each pixel can be processed independently from each other speeding up the execution time of this block.

5.2.1 Extrinsic camera calibration

The extrinsic camera matrix $\mathcal{T}_{base}^{cam}$ defines the rigid transformation between the camera frame and the robot base frame.
This matrix must be known in advance. In the presented control scheme, $\mathcal{T}_{base}^{cam}$ is a static transformation since no relative motion occurs between the camera frame and the robot base frame. Hence, an offline extrinsic calibration is needed to estimate the extrinsic camera matrix $\mathcal{T}_{base}^{cam}$.

Beforehand, a 9×7 chessboard is laid down on a flat table. Thereafter, we ensured that each corner of the chessboard is both framed by the camera and is reachable by the robot end-effector. We then found the 63 corner pixels $\mathbf{p}_{x_i}$ in the captured depth image. The projection $\mathbf{p}_{cam_i}$ of the depth image corner pixels $\mathbf{p}_{x_i}$ into the 3D camera reference frame is performed through the eq. 5.4.
Finally, The end-effector of the robot arm is placed on each chessboard corner and the 63 end-effector position vectors $\mathbf{p}_{ee_i}$ w.r.t the robot base frame are sampled by means of the robot forward kinematic function.

$$\mathbf{p}_{ee_i} = \mathtt{FK}(\mathbf{q}) \tag{5.5}$$

where $\mathbf{q}$ is the current joints positions vector. The problem of finding the optimal rotation matrix and translation vector can be solved through the Singular Value Decomposition (SVD) approach.
Let $\mathbf{t}_{base}^{cam}$ and $\mathbf{R}_{base}^{cam}$ the translation vector and the rotation matrix of $\mathcal{T}_{base}^{cam}$ respectively. The following cost function must be minimized:

$$\sum_{i=1}^{63} \| \left(\mathbf{R}_{base}^{cam}\mathbf{p}_{ee_i} + \mathbf{t}_{base}^{cam}\right) - \mathbf{p}_{cam_i} \|^2 \tag{5.6}$$

Firstly, we compute the centered vector of both point sets as follows:

$$\mathbf{x}_i = \mathbf{p}_{ee_i} - \bar{\mathbf{p}}_{ee} \quad , \quad \mathbf{y}_i = \mathbf{p}_{cam_i} - \bar{\mathbf{p}}_{cam} \tag{5.7}$$

where: $\bar{\mathbf{p}}_{ee}$ and $\bar{\mathbf{p}}_{cam}$ are the centroid of both points sets respetively.
The 3×3 covariance matrix S is computed as follows:

$$S = \mathrm{X}\mathrm{Y}^T \tag{5.8}$$

where X and Y are the 3×63 matrices that have $\mathbf{x}_i$ and $\mathbf{y}_i$ as their columns, respectively.
The SVD returns the covariance matrix as a product of three matrices: $S = U\Sigma V^T$.
Hence, the optimal rotation is given as:

$$\mathbf{R}_{base}^{cam} = VU^T \tag{5.9}$$

The optimal translation is thus:

$$\mathbf{t}_{base}^{cam} = \bar{\mathbf{p}}_{cam} - \mathbf{R}_{base}^{cam}\bar{\mathbf{p}}_{ee} \tag{5.10}$$

5.3 Repulsive vector force computation on GPU

We rearrange the equation in [19] to compute on GPU the repulsive vector force for each point in the point cloud.
Beforehand, we define a maximum human-robot interaction workspace as a 3D sphere centred in the robot base frame and radius $\rho = 1.1m$. Points outside this sphere do not contribute in the force vector field computation. Moreover, we cancel from the depth image the manipulator itself. This ensures that the robot is not considered as an obstacle. In the robot design phase, we created the 3D CAD model of the manipulator. We render this 3D CAD model onto the image plane using the actual pose of the robotic arm. The depth image pixels affected by the rendering process are not considered for the point cloud generation.
From now on, all the vectors are defined w.r.t. the robot base reference frame.
Firstly, the distance between the point obstacle $\mathbf{p}_o$ and the current end-effector position is computed as:

$$\mathbf{D}(\mathbf{ee}, \mathbf{o}) = \mathbf{p}_{ee} - \mathbf{p}_o \tag{5.11}$$

where $\mathbf{p}_{ee}$ is the current position of the robot end-effector given by the robot forward kinematic function as shown in eq. 5.12.

$$\mathbf{p}_{ee}(t) = \texttt{FK}(\hat{\mathbf{q}}(t)) \tag{5.12}$$

The point obstacle $\mathbf{p}_o$ generates a repulsive force vector on the robot end-effector as shown in eq. 5.13.

$$\mathbf{V}_r(\mathbf{ee}, \mathbf{o}) = v(\mathbf{ee}, \mathbf{o}) \frac{\mathbf{D}(\mathbf{ee}, \mathbf{o})}{\|\mathbf{D}(\mathbf{ee}, \mathbf{o})\|} \tag{5.13}$$

The direction of the vector force is equal to the one of the $\mathbf{D}(\mathbf{ee}, \mathbf{o})$ vector, but the magnitude is formulated as in eq. 5.14 (see [19]).

$$v(\mathbf{ee}, \mathbf{o}) = \frac{V_{max}}{1 + e^{(\|\mathbf{D}(\mathbf{ee},\mathbf{o})\|(2/\rho)-1)\alpha}} \tag{5.14}$$

The force magnitude is zero outside the sphere with radius ρ. The maximum value of the force (V_{max}) is reached when the distance $\|\mathbf{D}(\mathbf{ee}, \mathbf{o})\| = 0$. As the distance $\|\mathbf{D}(\mathbf{ee}, \mathbf{o})\|$ approaches ρ, the magnitude converges smoothly to zero as depicted in fig. 5.1. The parameter α defines the sharpness of the exponential function. A parallel reduction sum is then performed in GPU to sum up all the point force vectors. The cumulative force vector is thus given by the eq. 5.15.

$$\mathbf{V}_{r_{tot}}(\mathbf{ee}) = \sum_{\mathbf{p}_o \in \mathbf{S}} \mathbf{V}_r(\mathbf{ee}, \mathbf{o}) \tag{5.15}$$

The final force vector $\vec{F}_{ee}$ that must be applied to the robot end-effector is formulated in eq.5.16.

$$\vec{F}_{ee}(\mathbf{ee}) = v(\mathbf{ee}, \mathbf{o}_{min}) \frac{\mathbf{V}_{r_{tot}}(\mathbf{ee})}{\|\mathbf{V}_{r_{tot}}(\mathbf{ee})\|} \tag{5.16}$$

The direction of $\vec{F}_{ee}(\mathbf{ee})$ is given by the direction of the total sum vector $\mathbf{V}_{r_{tot}}(\mathbf{ee})$. The magnitude of $\vec{F}_{ee}(\mathbf{ee})$ is computed as in eq. 5.14 using only the the closest

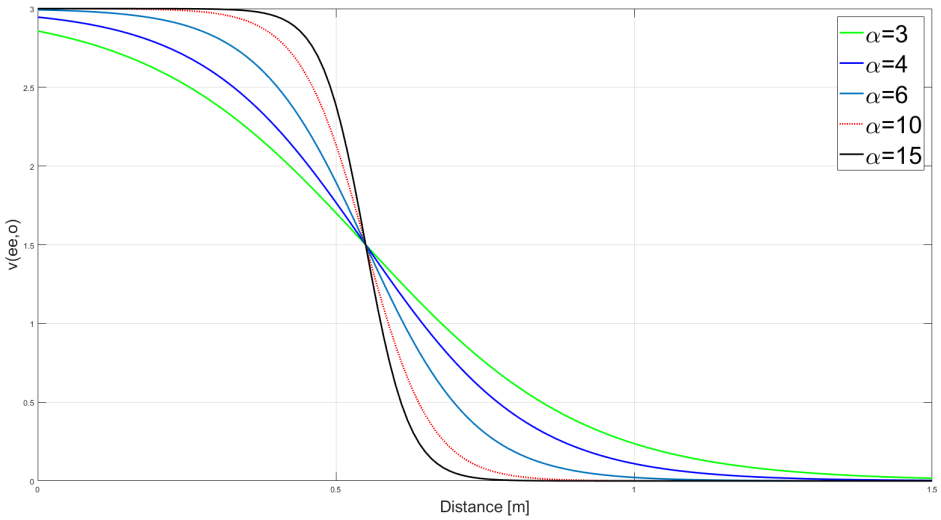


Figure 5.1: Magnitude of the vector field for different values of α, $V_{max} = 3m/s$, $\rho = 1.1m$

obstacle $\mathbf{p}_{o_{min}}$ to the robot end-effector.

5.4 Reference Joints Velocity Computation

The total force vector $\vec{F}_{ee}(\mathbf{ee})$ is a three-dimensional vector in robot base frame that tries to push the robot away from imminent collision against objects or people. In order to control the robot along the new obstacle-free trajectory, we need to converter the end-effector external force into a reference joints trajectory.

Given the desired force vector at the end-effector, this block generates a reference joints velocity vector ($\dot{\mathbf{q}}_r(t)$) at each sampling time Tg=25ms.

Beforehand, the force vector $\vec{F}_{ee}(\mathbf{ee})$ is simply converted into a velocity vector $\mathbf{v}_{ee}(\mathbf{ee})$ as follows:

$$\mathbf{v}_{ee}(\mathbf{ee}) \doteq \vec{F}_{ee}(\mathbf{ee}) \tag{5.17}$$

Once the desired end-effector velocity is computed, the inversion of differential kinematics can be carried out.

The robotic arm built for this work has $n = 7$ active joints. This means that the

robot is always kinematically redundant for any assigned task. In particular, for a task that requires positioning and orienting the end-effector in 3D space ($m = 6$), the degree of redundancy is equal to $n - m = 1$. In this thesis, we are interested in positioning the robot end-effector in 3D space ($m = 3$). The degree of redundancy is thus equal to $n - m = 4$. We leverage the properties of redundant robots to solve the obstacle avoidance problem efficiently. Kinematically redundant robots produce infinite solutions to the inverse kinematics problem. Moreover, robot's joints internal displacement can be generated without affection the task variables. This **self-motion** is employed to optimize the behaviour of the robot by managing some additional constraints. The redundancy resolution is based on the *null-space* method. The differential kinematics equation for a manipulator is formulated in eq. 5.18 (see [55]).

$$\dot{\mathbf{p}}(t) = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}(t) \tag{5.18}$$

For a 7DoF manipulator and a positioning task in 3D space, $\mathbf{J}(\mathbf{q})$ is the 3×7 Jacobian matrix, $\dot{\mathbf{q}}(t)$ is the 7×1 joints velocities vector and, finally, $\dot{\mathbf{p}}(t)$ is the 3×1 end-effector linear velocities vector.

The general solution of eq. 5.18 using the null-space approach is given by eq. 5.19

$$\dot{\mathbf{q}}(t) = \mathbf{J}^{\dagger}(\mathbf{q})\dot{\mathbf{p}}(t) + \left(\mathbf{I} - \mathbf{J}^{\dagger}(\mathbf{q})\mathbf{J}(\mathbf{q})\right)\dot{\mathbf{q}}_0(t) \tag{5.19}$$

where: $\mathbf{J}^{\dagger}$ is the *damped pseudo-inverse* defined as follows:

$$\mathbf{J}^{\dagger}(\mathbf{q}) = \mathbf{J}^{T}(\mathbf{q})\left(\mathbf{J}(\mathbf{q})\mathbf{J}^{T}(\mathbf{q}) + \rho^2\mathbf{I}\right)^{-1} \tag{5.20}$$

$\mathbf{I}$ is the 7×7 identity matrix and ρ is the *damping factor*. The *damped pseudo-inverse* has been introduced to mitigate the effect of any kinematic singularities along the real-time generated trajectory.

Eq. 5.19 is composed by two terms. The first term $\mathbf{J}^{\dagger}(\mathbf{q})\dot{\mathbf{p}}(t)$ fulfils the constraint

5.18 by minimizing the velocity norm cost function in eq. 5.21.

$$g(\dot{\mathbf{q}}) = \frac{1}{2}\dot{\mathbf{q}}^T\dot{\mathbf{q}} \tag{5.21}$$

In contrast, the second term $\left(\mathbf{I} - \mathbf{J}^{\dagger}\mathbf{J}\right)\dot{\mathbf{q}}_0$ is the homogeneous solution of eq. 5.18. It tries to satisfy a secondary constraint by projecting an arbitrary joints velocity $\dot{\mathbf{q}}_0$ in the null space of $\mathbf{J}$. The 7×7 matrix $\left(\mathbf{I} - \mathbf{J}^{\dagger}\mathbf{J}\right)$ is thus the projection matrix that projects $\dot{\mathbf{q}}_0$ in the null space of $\mathbf{J}$ without breaking the constraint 5.18.
Therefore, the general solution in eq. 5.19 allows the manipulator self-motion. If $\dot{\mathbf{p}}(t) = 0$, it can be chosen a proper joints velocity $\dot{\mathbf{q}}_0$ in order to reconfigure the robotic arm without affecting neither the end-effector position nor the end-effector orientation.
The kinematic constraint in 5.18 has higher priority than the additional constraint imposed by $\dot{\mathbf{q}}_0$. We thus assign the highest priority to the collision avoidance task and the lower priority task is left to the trajectory tracking.
The reference joints velocities are then computed as follows:

$$\dot{\mathbf{q}}_r(t) = \mathbf{J}^{\dagger}(\mathbf{q})\mathbf{v}_{ee}(\mathbf{ee}) + \left(\mathbf{I} - \mathbf{J}^{\dagger}(\mathbf{q})\mathbf{J}(\mathbf{q})\right)\dot{\mathbf{q}}_0(t) \tag{5.22}$$

where, the null space velocity $\dot{\mathbf{q}}_0(t)$ is designed in 5.23 to let the robot track a given trajectory imposed during a common working cycle (e.g. the robot is asked to pick some objects from point A and place them to point B).

$$\dot{\mathbf{q}}_0(t) = K_p(\mathbf{q}^*(t) - \hat{\mathbf{q}}(t)) + K_d(\dot{\mathbf{q}}^*(t) - \dot{\hat{\mathbf{q}}}(t)) \tag{5.23}$$

In eq. 5.23 the pair $(\mathbf{q}^*(t),\ \dot{\mathbf{q}}^*(t))$ is the desired robot trajectory, the pair $\left(\hat{\mathbf{q}}(t),\ \dot{\hat{\mathbf{q}}}(t)\right)$ is the current robot trajectory and both K_p and K_d are constant gains.

5.5 Joints Space Trajectory Generation

This block generates an on-line smooth trajectory for the robot joints given both the robot kinematic constraints and the target joints velocity vector $\dot{\mathbf{q}}_r(t)$. The latter is updated every Tg=25ms but the trajectory generator produces a new trajectory point $[\mathbf{q}_r(t+1),\ \dot{\mathbf{q}}_r(t+1),\ \ddot{\mathbf{q}}_r(t+1)]$ every Tc=5ms.
We used the on-line trajectory generator presented in [34]. Figure 5.2 depicts the

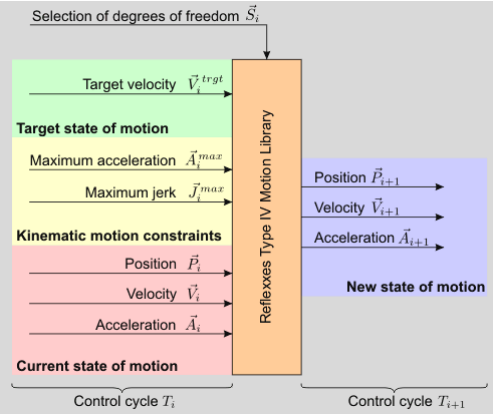


Figure 5.2: Input and output values of the velocity-based On-Line Trajectory Generation algorithm in [34]

black-box scheme of the trajectory generator. The target velocity usually varies at each sampling time Tg. This block guarantees a smooth velocity trajectory interpolation between the ongoing trajectory and the new one. This is mandatory in order to have a differentiable acceleration profile and a continuous jerk profile. The new state of motion is directly fed to the robot controller that is designated to control the axes motion.

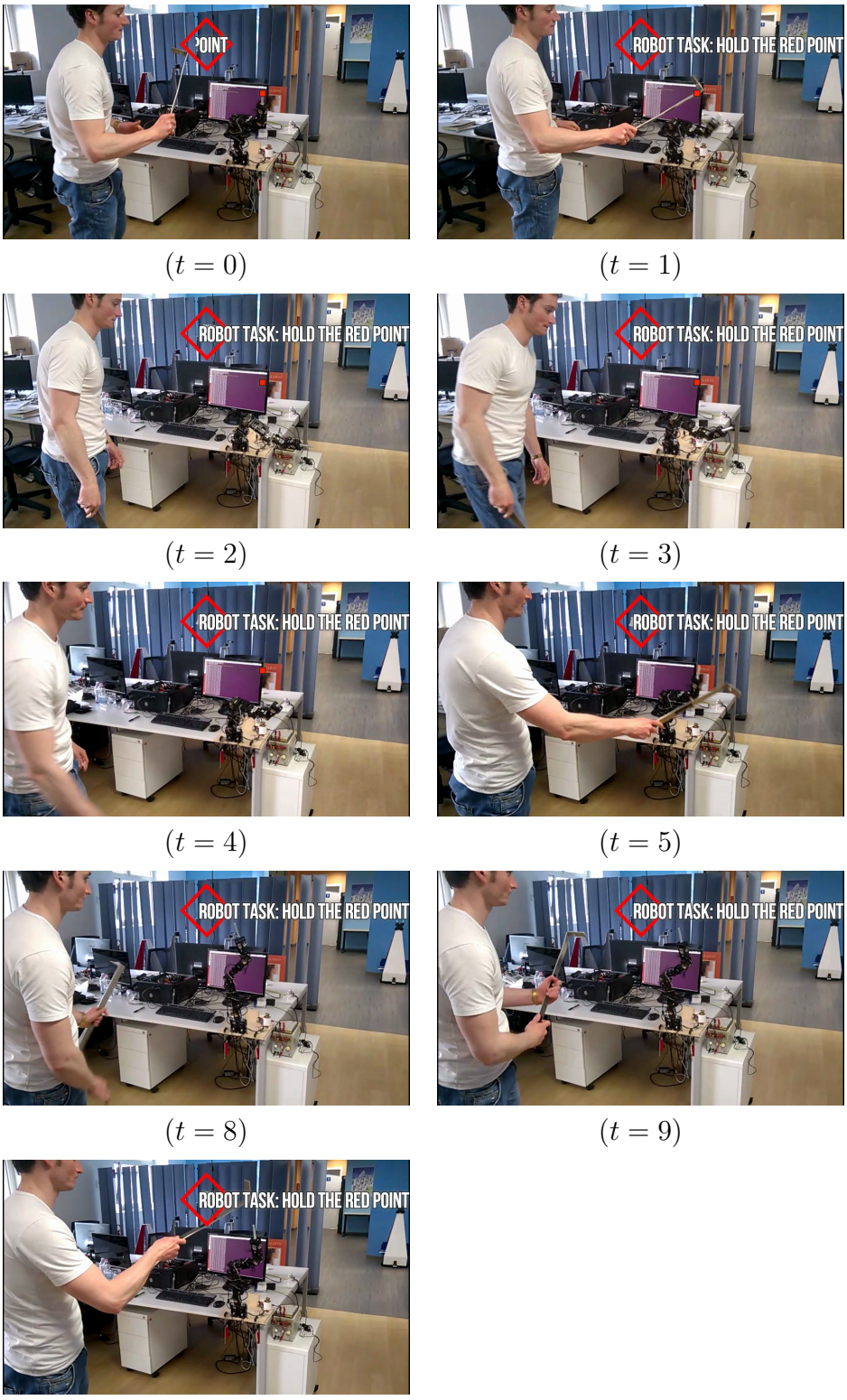
5.6 Collision Avoidance Test on a Real Lightweight Manipulator

In this section we use the robot we built in our laboratory to test the collision avoidance algorithm presented in this Chapter. We encourage to watch the full video of the test at the following link HRC:Collision Avoidance to fully appreciate the algorithm in action.

The robot base has been fixed on a common desk of our lab. We assign to the robot a positioning task. The null space joints velocity $\dot{\mathbf{q}}_0(t)$ in eq. 5.23 is thus computed based on the desired trajectory expressed in eq. 5.24.

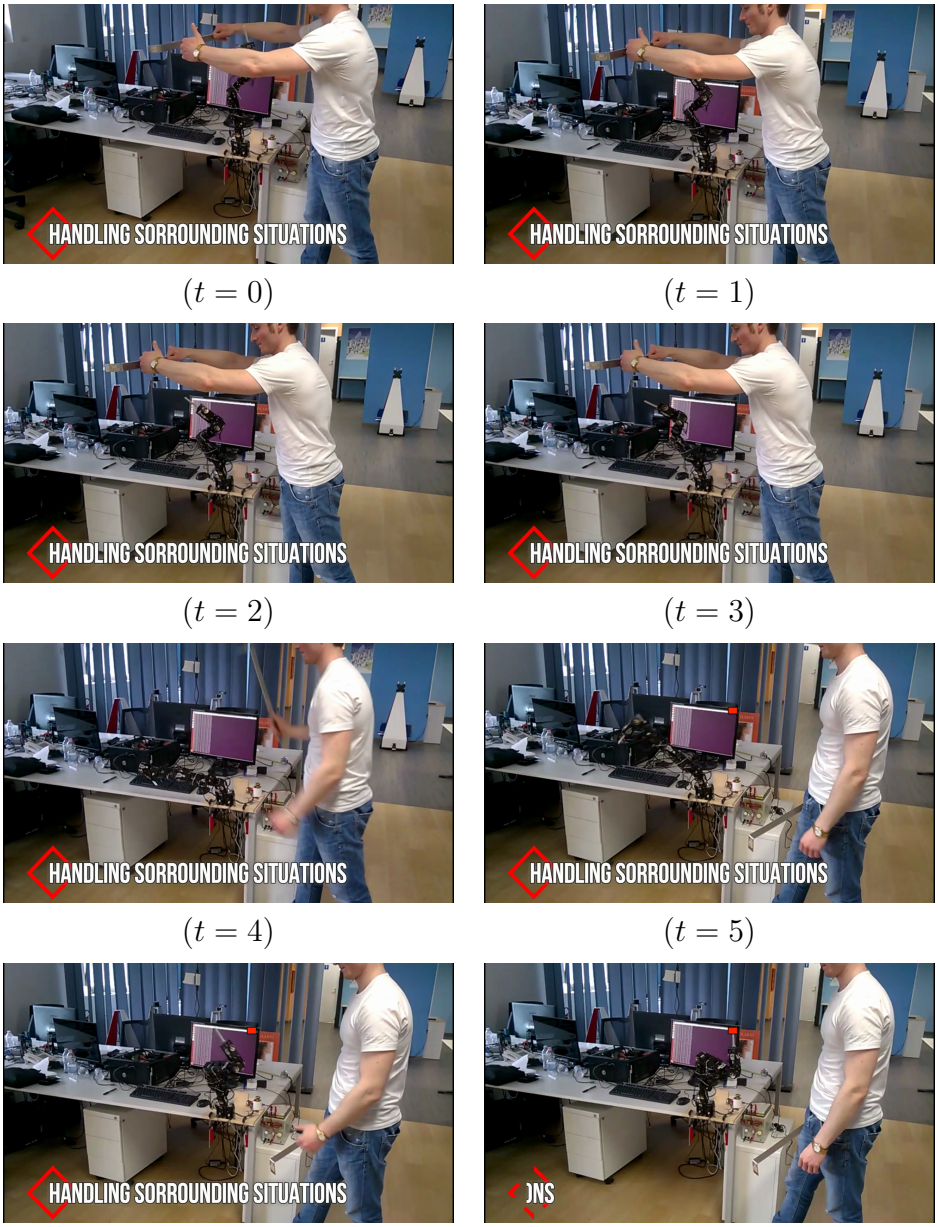
$$\begin{cases} \mathbf{q}^*(t) = \mathbf{q}^* = \left[-\frac{\pi}{6},\ 0,\ \frac{\pi}{2.5}, -\frac{\pi}{3},\ 0, -\frac{\pi}{3},\ 0\right]^T \\ \dot{\mathbf{q}}^*(t) = \dot{\mathbf{q}}^* = \mathbf{0} \end{cases} \tag{5.24}$$

The end-effector desired position in the operational space can be visualized in the video and in both figures 5.3 and 5.4 as a red square. A person is asked to perturb the robot positioning task with voluntary gestures that could yield to human-robot collisions.



$(t = 0)$ $(t = 1)$ $(t = 2)$ $(t = 3)$ $(t = 4)$ $(t = 5)$ $(t = 8)$ $(t = 9)$ $(t = 10)$

Figure 5.3: A person perturbs the positioning task of the robot. He is trying to generate a human-robot collision



$(t = 0)$ $(t = 1)$

$(t = 2)$ $(t = 3)$

$(t = 4)$ $(t = 5)$

$(t = 7)$ $(t = 8)$

Figure 5.4: A person surrounds the robot while it is performing the positioning task

Chapter 6

Conclusion

This Ph.D. dissertation is focussed on novel and optimized algorithms developed by the author during a three year Ph.D. path in Politecnico di Torino with the collaboration of Telecom Italia S.p.A.

We presented a novel pipeline for segmenting simple tabletop-like objects from RGB-D images and estimating their 3D pose that, contrary to many current approaches, does not rely on machine learning methods and it is robust to changes in object appearance, lack of textures and heavy textures. The pipeline is composed by a object segmentation algorithm, a pose estimation part and finally a collision avoidance algorithm for a safe human-robot coexistence in both structured and unstructured environment.

In Chapter 2 the segmentation algorithm is presented. Candidate object regions are first retrieved using a fast graph-based image segmentation algorithm that integrates color and depth information and is able to work with texture-less objects as well as heavy textured ones. We designed a modified Canny edge detector that integrates depth information in order to find robust edges in RGB-D images. We proposed a novel depth smoothing filter with dynamic smoothing kernels that also work with depth shadows. Actual object borders are correctly preserved. Two non-linear weighting functions are designed in order to generate the graph edge weights.

These functions take as inputs both color and depth image cues. A post-processing phase is used to discard most false positives. Future work, related to the segmentation part will be devoted to parallelization of the graph creation and partitioning phases and to investigate different partitioning strategies.

In Chapter 3, we presented QPSO, a Particle Swarm Optimization algorithm with a novel quaternion-based kinematics formulation. The QPSO is run on the candidate object regions, using a 3D model of the object. The fitness function is based on depth information only, keeping the algorithm robust to textureless or heavy textured objects. Moreover, we argue that the final fitness value can be used to further discard false positive regions from the segmentation part. This assigns to the QPSO an intrinsically object detection property that can be exploited in parallel with the pose estimation part.

We also show how QPSO can be easily extended in order to estimate the pose of articulated objects with prismatic or revolute joints, by simply adding the joint values in the PSO state.

The segmentation part runs in linearithmic time. The PSO exploits the parallelization potential offered by GPUs and it is able to run 10 iterations at more than 10 fps on a current GPU. Chapter 3 is devoted to the optimized GPU code of the QPSO. A novel GPU code is thus designed to efficiently solve this constrained optimization problem. Future work will be devoted to the full extension of the algorithm to articulated objects with arbitrary number of links and experiments with different segmentation algorithms.

Finally, Chapter 5 explains how humans can safely work close to the robots. An important aspect of human-robot coexistence is the real-time generation of collision free robot trajectories. A novel GPU-optimized collision-avoidance algorithm is tested on a real lightweight manipulator. The robot is asked to fulfill a given task while the human is trying to perturb it in order to generate a human-robot collision.

Future work will be dedicated to the design of robot reaction strategies in presence of human-robot collisions. Moreover, contact force estimation along the robot body without any force-torque sensor is a growing research topic. This work could be the starting point for the development of a safe Human-Robot interaction.

Appendix A

Pseudo-code of the GPU implementation

Algorithm 1: Main steps of the multi-instance Q-PSO GPU implementation.

Input: Depth map of all the object clusters ($\mathcal{R}$) found in the scene
Input: 3D object model to search for
Result: Multi-instance object pose

```
1  object_best_poses = ∅;
2  for cluster = 1 to | R | do
3      initAllParticles<<<,>>>();
4      for step = 1 to maxSteps do
5          renderAllParticlesPoseHp<<<,>>>();
6          computeAllParticlesFitness<<<,>>>();
7          updatePersonalAndGlobalBestAllParticles<<<,>>>();
8          computeNewPoseAndVelAllParticles<<<,>>>();
9      end
10     if best_fitness > th then
           /* the object does not belong to the same class of the 3D
              model                                                  */
11         continue;
12     else
           /* load the best object pose back to CPU and save it     */
13         object_best_poses.push_back(best_pose);
14     end
15 end
```

Algorithm 2: Parallel rendering on multiple streams

Input: d_depth_buffer, d_AABB, d_pso_pose, particleIdx, offset_buffer
Input: 3D object model to render: d_obj_model
Result: d_depth_buffer, d_AABB

```
1  offset_buffer = 0;
2  particleIdx = 0;
3  streams[NSTREAMS] = init(NSTREAMS);
   /* Nblocks × NthsPerBlock >= 3D Object Mesh Faces                    */
4  Nblocks = 6;
5  NthsPerBlock = 512;
6  for i = 0 to NPART/NSTREAMS do
7      for Sidx = 0 to NSTREAMS do
8          renderAllParticlesPoseHp<<<
             Nblocks,NthsPerBlock,streams[Sidx]>>>
9
             (d_obj_model,d_pso_pose,particleIdx,offset_buffer,d_depth_buffer,d_AABB);

10         offset_buffer += R*C;
11         ++particleIdx;
12     end
13 end
```

Algorithm 3: Q-PSO GPU Initialization

Input: d_randGen, d_pso_pose, d_pso_vel, d_pso_pose_b, d_randIdx
Result: d_pso_pose, d_pso_vel, d_pso_pose_b are initialized

```
/* One Thread per PSO particle                                          */
1  tIdx = blockDim.x * blockIdx.x + threadIdx.x;
/* Local variable to access the d_randGen array in a coalesced
   fashion                                                              */
2  s_randIdx=0;
/* Compute the arrays' indices to fullfil the coalescent access
   requirement                                                          */
3  idx_tx = tIdx + tx_*NPART;
4  idx_ty = tIdx + ty_*NPART;
5  idx_tz = tIdx + tz_*NPART;
6  idx_q0 = tIdx + q0_*NPART;
7  idx_q1 = tIdx + q1_*NPART;
8  idx_q2 = tIdx + q2_*NPART;
9  idx_q3 = tIdx + q3_*NPART;
/* Initialize the particles' Position                                   */
10 d_pso_pose[idx_tx] = Eq. (3.17);
11 d_pso_pose[idx_ty] = Eq. (3.17);
12 d_pso_pose[idx_tz] = Eq. (3.17);
/* Initialize the particles' Orientation                                */
13 d_pso_pose[idx_q0] = Eq. (3.19);
14 d_pso_pose[idx_q1] = Eq. (3.19);
15 d_pso_pose[idx_q2] = Eq. (3.19);
16 d_pso_pose[idx_q3] = Eq. (3.19);
/* Initialize the particles' Linear Velocity                            */
17 d_pso_vel[idx_tx] = Eq. (3.18);
18 d_pso_vel[idx_ty] = Eq. (3.18);
19 d_pso_vel[idx_tz] = Eq. (3.18);
/* Initialize the particles' Angular Velocity                           */
20 d_pso_vel[idx_q0] = Eq. (3.20);
21 d_pso_vel[idx_q1] = Eq. (3.20);
22 d_pso_vel[idx_q2] = Eq. (3.20);
23 d_pso_vel[idx_q3] = Eq. (3.20);
/* Copy Results to Personal Best                                        */
24 d_pso_pose_b[idx_tx] = d_pso_pose[idx_tx];
25 d_pso_pose_b[idx_ty] = d_pso_pose[idx_ty];
26 d_pso_pose_b[idx_tz] = d_pso_pose[idx_tz];
27 d_pso_pose_b[idx_q0] = d_pso_pose[idx_q0];
28 d_pso_pose_b[idx_q1] = d_pso_pose[idx_q1];
29 d_pso_pose_b[idx_q2] = d_pso_pose[idx_q2];
30 d_pso_pose_b[idx_q3] = d_pso_pose[idx_q3];
/* where δ in Eqs. (3.17),(3.18),(3.19),(3.20) =
   d_randGen[tIdx+(s_randIdx++)*NPART]                                  */
/* keep track of the indices                                            */
31 d_randIdx[tIdx] = s_randIdx;
```

Algorithm 4: GPU Rendering Kernel

```
   Input: d_depth_buffer, d_AABB, d_pso_pose, particleIdx, offset_buffer
   Input: 3D object model to render: d_obj_model
   Result: d_depth_buffer, d_AABB
 1 renderAllParticlesPoseHp( ... )
 2 {
   /* one thread per triangle (Mesh Face).  tIdx = Global Mesh Face Index   */
 3 tIdx = blockDim.x * blockIdx.x + threadIdx.x;
   /* Shared Memories declaration and initialization.                       */
   /* threadIdx.x = Local (within a block) Mesh Face Index)                 */
 4 __shared__ sMinX[NthsPerBlock];     sMinX[threadIdx.x] = maxAABB;
 5 __shared__ sMaxX[NthsPerBlock];     sMaxX[threadIdx.x] = minAABB;
 6 __shared__ sMinY[NthsPerBlock];     sMinY[threadIdx.x] = maxAABB;
 7 __shared__ sMaxY[NthsPerBlock];     sMaxY[threadIdx.x] = minAABB;
 8 __syncthreads();
 9 pso_pose_vec[7] = fetch_particle_pose_hypothesis(d_pso_pose,particleIdx);
10 float4 triangle[3] = fetch_mesh_face(d_obj_model,tIdx);
   /* Compute the normalized pixels coordinate (UV) of the projected triangle
      vertices                                                              */
11 float4 UVz[3];
   /* first vertex of the face tIdx                                         */
12 projectModelPointsToPixels(pso_pose_vec,triangle[0],UVz[0]);
   /* second vertex of the face tIdx                                        */
13 projectModelPointsToPixels(pso_pose_vec,triangle[1],UVz[1]);
   /* third vertex of the face tIdx                                         */
14 projectModelPointsToPixels(pso_pose_vec,triangle[2],UVz[2]);
   /* AABB of the projected triangle                                        */
15 minx = (min(min(UVz[0].X, UVz[1].X), UVz[2].X);
16 maxx = (max(max(UVz[0].X, UVz[1].X), UVz[2].X);
17 miny = (min(min(UVz[0].Y, UVz[1].Y), UVz[2].Y);
18 maxy = (max(max(UVz[0].Y, UVz[1].Y), UVz[2].Y);
   /* fill the shared memories with the t.l.  and b.r.  corners of the AABB of
      each rendered face                                                    */
19 sMinX[threadIdx.x] = minx;
20 sMaxX[threadIdx.x] = maxx;
21 sMinY[threadIdx.x] = miny;
22 sMaxY[threadIdx.x] = maxy;
23 __syncthreads();
24 MinMaxAABBReduction(sMinX,sMaxX,sMinY,sMaxY);
   /* The partial results of each block is stored in sMXXX[0]               */
   /* Only the first active thread of each block stores its block partial
      results to global memory                                              */
25 _8warpSize_ = 8*warpSize;
26 if isFirstActiveThread(tIdx) then
27 |   d_AABB[particleIdx*_8warpSize_ + blockIdx.x] = sMinX[0];
28 |   d_AABB[particleIdx*_8warpSize_ + blockIdx.x + 64] = sMaxX[0];
29 |   d_AABB[particleIdx*_8warpSize_ + blockIdx.x + 128] = sMinY[0];
30 |   d_AABB[particleIdx*_8warpSize_ + blockIdx.x + 192] = sMaxY[0];
   /* Parallel Rendering of the projected triangle                          */
31 renderFixedPointEdgeFcn(d_depth_buffer,minx,maxx,miny,maxy,offset_buffer);
32 }
```

Algorithm 5: Parallel Final Reduction for computing the Final AABB of each particle

```
   Input: d_AABB, d_finalAABB
   Result: d_finalAABB
1  ComputeAABBFinalReductionKernel( ... )
2  {
   /* 1 Block per particle with 256 (8 Warps) threads per block                    */
3  tIdx = blockDim.x * blockIdx.x + threadIdx.x;
4  globalWarpIdx = tIdx/32;// Global Warp ID
5  blockWarpIdx = threadIdx.x/32;// Warp ID within a Block
6  laneIdx = tIdx & 0x1F; // [0-31] Global Thread ID within a Warp
7  blocklaneIdx = threadIdx.x & 0x1F; // [0-31] Thread ID within a Warp in the Block
8  pIdx = blockIdx.x;// particle ID
9  _2warpSize_=2*warpSize;
10 _8warpSize_=8*warpSize;
11 ___shared___ sMinX[_2warpSize_];
12 ___shared___ sMaxX[_2warpSize_];
13 ___shared___ sMinY[_2warpSize_];
14 ___shared___ sMaxY[_2warpSize_];
15 tab:challenge /* Parallel and Coalesced copy of the AABB partial results from global
      to shared memory                                                            */
16 if blockWarpIdx<2 then
   |  /* chosen warp 0 and 1 to handle sMinX                                      */
17 |  currentLaneIdx = blocklaneIdx+blockWarpIdx*_warpSize_; // [0-63]
18 |  sMinX[currentLaneIdx] = d_AABB[pIdx*_8warpSize_ + currentLaneIdx];
19 if blockWarpIdx>=2 && blockWarpIdx<4 then
   |  /* chosen warp 2 and 3 to handle sMaxX                                      */
20 |  currentLaneIdx = blocklaneIdx+blockWarpIdx*_warpSize_; // [64-127]
21 |  idx = blocklaneIdx+((blockWarpIdx-1)/2)*_warpSize_;
22 |  sMaxX[idx] = d_AABB[pIdx*_8warpSize_ + currentLaneIdx];
23 if blockWarpIdx>=4 && blockWarpIdx<6 then
   |  /* chosen warp 4 and 5 to handle sMinY                                      */
24 |  currentLaneIdx = blocklaneIdx+blockWarpIdx*_warpSize_; // [128-191]
25 |  idx = blocklaneIdx+((blockWarpIdx-3)/2)*_warpSize_;
26 |  sMinY[idx] = d_AABB[pIdx*_8warpSize_ + currentLaneIdx];
27 if blockWarpIdx>=6 && blockWarpIdx<8 then
   |  /* chosen warp 6 and 7 to handle sMaxY                                      */
28 |  currentLaneIdx = blocklaneIdx+blockWarpIdx*_warpSize_; // [192-255]
29 |  idx = blocklaneIdx+((blockWarpIdx-5)/2)*_warpSize_;
30 |  sMaxY[idx] = d_AABB[pIdx*_8warpSize_ + currentLaneIdx];
31 __syncthreads();
   /* Min and Max Reduction over the shared memories                               */
32 MinMaxAABBReduction(sMinX,sMaxX,sMinY,sMaxY);
   /* Each Particle now has 4 values i.e.  the AABB of its own rendered 3D model  */
   /* Let the first active thread of that Block fill the final AABB array         */
33 if isFirstActiveThread(threadIdx.x) then
34 |  d_finalAABB[pIdx*_warpSize_ + 0] = sMinX[0];
35 |  d_finalAABB[pIdx*_warpSize_ + 1] = sMaxX[0];
36 |  d_finalAABB[pIdx*_warpSize_ + 2] = sMinY[0];
37 |  d_finalAABB[pIdx*_warpSize_ + 3] = sMaxY[0];
38 }
```

Algorithm 6: Particles' fitness computation

```
   Input: d_depth_buffer, offset_buffer, particleIdx, d_finalAABB, d_pso_fit_error, d_depth_kinect
   Result: d_pso_fit_error
1  ComputeAllParticlesFitness( ... )
2  {
   /* 1 Block of 1024 threads per particle (we test 1024 pixels at time)                    */
   /* Global thread index                                                                   */
3  tIdx = blockDim.x * blockIdx.x + threadIdx.x;
4  totalNumThreads = blockDim.x * gridDim.x;
5  ___shared___ sMinMaxXY[warpSize];
6  ___shared___ sError[2048];// 2*1024 threads
7  ___shared___ sμ[2048];// mu
8  ___shared___ sκ[2048];// kappa
9  ___shared___ sNrenderedPoints[64];// 32 warps within 1024 threads x2
10 initSharedMemories();
   /* Chosen the first warp to load the particle pIdx's AABB in coalesced fashion           */
11 if tIdx<warpSize then
12 |    MinMaxXY[tIdx] = d_finalAABB[pIdx*warpSize + tIdx];
13 |    sNrenderedPoints[tIdx] = 0;
14 |    sNrenderedPoints[tIdx+32] = 0;
15 ___syncthreads();
   /* Save in the local registers the particle's AABB                                       */
16 minXr=MinMaxXY[0];
17 maxXr=MinMaxXY[1];
18 minYr=MinMaxXY[2];
19 maxYr=MinMaxXY[3];
   /* Save in the local registers the AABB of the segmented cluster                         */
20 minXs = constant.cAABBminX;
21 maxXs = constant.cAABBmaxX;
22 minYs = constant.cAABBminY;
23 maxYs = constant.cAABBmaxY;
   /* OR of the 2 AABBs                                                                     */
24 minX = min(minXr,minXs);
25 maxX = max(maxXr,maxXs);
26 minY = min(minYr,minYs);
27 maxY = max(maxYr,maxYs);
28 W_bar = maxX-minX;// AABB Width
29 H_bar = maxY-minY;// AABB Height
30 Area = W_bar*H_bar;
31 NumIters = ceil( Area / totalNumThreads );
32 for i = 0 : NumIters do
   |    /* Get the absolute pixels coord.  inside the final AABB                           */
33 |    tIdxLoop = tIdx + i*totalNumThreads;
34 |    x_bar = tIdxLoop % W_bar;
35 |    y_bar = fix(tIdxLoop / W_bar) ;
36 |    pxX = minX + x_bar;
37 |    pxY = minY + y_bar;
38 |    if pxX && pxY inside AABB then
   |    |    /* Fetch the depth value of that pixel from the rendering                    */
39 |    |    pxIdxRendered = pxY*C + pxX + offset_buffer;
40 |    |    z_rendering = d_depth_buffer[pxIdxRendered];
   |    |    /* Zero the depth buffer                                                     */
41 |    |    d_depth_buffer[pxIdxRendered]=0;
   |    |    /* Fetch the depth value of that pixel in the segmented scene              */
42 |    |    pxIdxKinect = pxY*C + pxX;
43 |    |    z_kinect = d_depth_kinect[pxIdxKinect];
   |    |    /* Shared memory arrays are treated as circular arrays of dimensions 2048  */
44 |    |    tIdxLoopCirc = tIdxLoop & 2047;
   |    |    /* Fill these circular buffers with partial results                         */
45 |    |    fill_partial_results(tIdxLoopCirc, z_rendering, z_kinect, sError[], sμ[], sκ[], sNrenderedPoints[]);
46 end
   /* A Parallel Sum Reduction computes and stores the final results in the first element of each circular buffer */
47 SumReduction(sError[],sNrenderedPoints[],sμ[],sκ[],tIdx);
   /* Store the final fitness for each particle to global memory                            */
48 if isFirstActiveThread(tIdx) then
49 |    d_pso_fit_error[pIdx] = Eq. (3.16);
50 }
```

Algorithm 7: Update particles' personal and global best fitness

```
   Input: d_pso_fit_error, d_pso_personal_best_fit, d_pso_pose, d_pso_pose_b, d_solution_best_fit,
          d_solution_best_pose, d_result_min_fit, best_particle_Idx
   Result: d_pso_pose_b, d_solution_best_fit, d_solution_best_pose
 1 updatePersonalAndGlobalBestAllParticle( ... )
 2 {
   /* 1 Thread per Particle                                                          */
 3 tIdx = blockDim.x * blockIdx.x + threadIdx.x;
   /* Fetch the particle tIdx's current and personal best fitness score              */
 4 personal_current_fit = d_pso_fit_error[tIdx];
 5 personal_best_fit = d_pso_personal_best_fit[tIdx];
   /* Update the particle's personal best fit                                        */
 6 if personal_current_fit < personal_best_fit then
 7     d_pso_personal_best_fit[tIdx] = personal_current_fit;
       /* Update the particle's personal best pose                                   */
       /* Compute the indices for a coalesced access                                 */
 8     idx_tx = tIdx + tx*NPART;
 9     idx_ty = tIdx + ty*NPART;
10     idx_tz = tIdx + tz*NPART;
11     idx_q0 = tIdx + q0*NPART;
12     idx_q1 = tIdx + q1*NPART;
13     idx_q2 = tIdx + q2*NPART;
14     idx_q3 = tIdx + q3*NPART;
15     d_pso_pos_b[idx_tx] = d_pso_pose[idx_tx];
16     d_pso_pos_b[idx_ty] = d_pso_pose[idx_ty];
17     d_pso_pos_b[idx_tz] = d_pso_pose[idx_tz];
18     d_pso_pos_b[idx_q0] = d_pso_pose[idx_q0];
19     d_pso_pos_b[idx_q1] = d_pso_pose[idx_q1];
20     d_pso_pos_b[idx_q2] = d_pso_pose[idx_q2];
21     d_pso_pos_b[idx_q3] = d_pso_pose[idx_q3];
   /* Update the Global Best                                                         */
22 if isFirstActiveThread(tIdx) then
       /* Minimum fitness score in this very iteration                               */
23     min_fit = d_result_min_fit;
       /* Best Particle ID in this very iteration                                    */
24     pIdx = best_particle_Idx;
       /* Best Fitness score until this very iteration                               */
25     solution_best_fit = d_solution_best_fit;
26     if min_fit < solution_best_fit then
           /* Update the global best fitness score through the best particle pIdx    */
27         d_solution_best_fit = min_fit;
           /* Update the global best pose through the best particle pIdx             */
28         idx_tx = pIdx + tx*NPART;
29         idx_ty = pIdx + ty*NPART;
30         idx_tz = pIdx + tz*NPART;
31         idx_q0 = pIdx + q0*NPART;
32         idx_q1 = pIdx + q1*NPART;
33         idx_q2 = pIdx + q2*NPART;
34         idx_q3 = pIdx + q3*NPART;
35         d_solution_best_pose[tx] = d_pso_pose[idx_tx];
36         d_solution_best_pose[ty] = d_pso_pose[idx_ty];
37         d_solution_best_pose[tz] = d_pso_pose[idx_tz];
38         d_solution_best_pose[q0] = d_pso_pose[idx_q0];
39         d_solution_best_pose[q1] = d_pso_pose[idx_q1];
40         d_solution_best_pose[q2] = d_pso_pose[idx_q2];
41         d_solution_best_pose[q3] = d_pso_pose[idx_q3];
42 }
```

Algorithm 8: Compute both the new particles' pose and velocity

Input: d_pso_pose, d_pso_vel, d_pso_pose_b, d_solution_best_pose, d_randIdx

Result: d_pso_pose, d_pso_vel, d_randIdx

```
1  computeNewPoseAndVelAllParticles( ... )
2  {
   /* 1 Thread per Particle                                          */
3  tIdx = blockDim.x * blockIdx.x + threadIdx.x;
   /* Compute the arrays' indices to fullfill the coalescent access
      requirement                                                    */
4  idx_tx = tIdx + tx_*NPART;
5  idx_ty = tIdx + ty_*NPART;
6  idx_tz = tIdx + tz_*NPART;
7  idx_q0 = tIdx + q0_*NPART;
8  idx_q1 = tIdx + q1_*NPART;
9  idx_q2 = tIdx + q2_*NPART;
10 idx_q3 = tIdx + q3_*NPART;
   /* Read Random vector indices to generate (r1, r2)                */
11 s_randIdx = d_randIdx[tIdx];
   /* Update the particles' Linear Velocity                          */
12 d_pso_vel[idx_tx] = Eq. (3.1);
13 d_pso_vel[idx_ty] = Eq. (3.1);
14 d_pso_vel[idx_tz] = Eq. (3.1);
   /* Update the particles' Position                                 */
15 d_pso_pose[idx_tx] = Eq. (3.2);
16 d_pso_pose[idx_ty] = Eq. (3.2);
17 d_pso_pose[idx_tz] = Eq. (3.2);
   /* Update the particles' Angular Velocity                         */
18 d_pso_vel[idx_q0] = Eq. (3.11);
19 d_pso_vel[idx_q1] = Eq. (3.11);
20 d_pso_vel[idx_q2] = Eq. (3.11);
21 d_pso_vel[idx_q3] = Eq. (3.11);
   /* Update the particles' Orientation                              */
22 d_pso_pose[idx_q0] = Eq. (??);
23 d_pso_pose[idx_q1] = Eq. (??);
24 d_pso_pose[idx_q2] = Eq. (??);
25 d_pso_pose[idx_q3] = Eq. (??);
   /* where r1, r2 in Eqs.(3.1),(3.11) =
      d_randGen[tIdx+(s_randIdx++)*NPART]                            */
   /* keep track of the indices                                      */
26 d_randIdx[tIdx] = s_randIdx;
27 }
```

Bibliography

[1] W. Abd-Almageed, M. Hussein, and M. Abdelkader. "Real-Time Human Detection and Tracking from Mobile Vehicles". In: *Intelligent Transportation Systems Conference, 2007. ITSC 2007. IEEE*. 2007, pp. 149–154. DOI: 10.1109/ITSC.2007.4357721.

[2] A. Abramov et al. "Depth-supported real-time video segmentation with the Kinect". In: *Applications of Computer Vision (WACV), 2012 IEEE Workshop on*. Jan. 2012, pp. 457–464. DOI: 10.1109/WACV.2012.6163000.

[3] Aitor Aldoma et al. "Pattern Recognition: Joint 34th DAGM and 36th OAGM Symposium, Graz, Austria, August 28-31, 2012. Proceedings". In: ed. by Axel Pinz et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Chap. OUR-CVFH – Oriented, Unique and Repeatable Clustered Viewpoint Feature Histogram for Object Recognition and 6DOF Pose Estimation, pp. 113–122. ISBN: 978-3-642-32717-9. DOI: 10.1007\/978-3-642-32717-9_12.

[4] *Amazon Picking Challenge*. Website. http://amazonpickingchallenge.org. 2016.

[5] Carmelo Bastos-Filho, Débora Nascimento, and Marcos Oliveira Junior. *Running particle swarm optimization on graphic processing units*. INTECH Open Access Publisher, 2011.

[6] Ujwal Bonde, Vijay Badrinarayanan, and Roberto Cipolla. "Robust instance recognition in presence of occlusion and clutter". In: *European Conference on Computer Vision*. Springer. 2014, pp. 520–535.

[7] Eric Brachmann et al. "Learning 6d object pose estimation using 3d object coordinates". In: *European Conference on Computer Vision*. Springer. 2014, pp. 536–551.

[8] Stefano Carpin, Andreas Birk, and Viktoras Jucikas. "On map merging". In: *Robotics and Autonomous Systems* 53.1 (2005), pp. 1–14. ISSN: 0921-8890. DOI: http://dx.doi.org/10.1016/j.robot.2005.07.001. URL: http://www.sciencedirect.com/science/article/pii/S0921889005001041.

[9] Andrea Censi. "An ICP variant using a point-to-line metric". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Pasadena, CA, May 2008. DOI: 10.1109/ROBOT.2008.4543181. URL: http://purl.org/censi/2007/plicp.

[10] Changhyun Choi and Henrik I Christensen. "3D pose estimation of daily objects using an RGB-D camera". In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE. 2012, pp. 3342–3349.

[11] *CUDA grid-stride loop*. Website. https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/. 2016.

[12] *CUDA Shared Memory*. Website. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared. 2016.

[13] *CUDA SIMT*. Website. http://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture. 2016.

[14] *CUDA Streams*. Website. http://docs.nvidia.com/cuda/cuda-c-programming-guide/#streams. 2016.

[15] *CUDA SUM REDUCTION*. Presentation. http://developer.download.nvidia.com/compute/cu
Beta/x86_website/projects/reduction/doc/reduction.pdf. 2016.

[16] *CUDA Warp*. Website. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html#th
hierarchy. 2016.

[17] *cuRAND*. Website. http://docs.nvidia.com/cuda/curand/. 2016.

[18] Pedro F Felzenszwalb and Daniel P Huttenlocher. "Efficient graph-based image segmentation". In: *International Journal of Computer Vision* 59.2 (2004), pp. 167–181.

[19] Fabrizio Flacco et al. "A depth space approach to human-robot collision avoidance". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE. 2012, pp. 338–345.

[20] Saurabh Gupta et al. "Learning rich features from RGB-D images for object detection and segmentation". In: *Computer Vision–ECCV 2014*. Springer, 2014, pp. 345–360.

[21] Swastik Gupta, Pablo Arbelaez, and Jagannath Malik. "Perceptual organization and recognition of indoor scenes from RGB-D images". In: *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*. IEEE. 2013, pp. 564–571.

[22] Bharath Hariharan et al. "Simultaneous detection and segmentation". In: *Computer Vision–ECCV 2014*. Springer, 2014, pp. 297–312.

[23] S. Hinterstoisser S. Holzer et al. "Multimodal Templates for Real-Time Detection of Texture-less Objects in Heavily Cluttered Scenes". In: (2011).

[24] Stefan Hinterstoisser et al. "Gradient response maps for real-time detection of textureless objects". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 34.5 (2012), pp. 876–888.

[25] Stefan Hinterstoisser et al. "Model based training, detection and pose estimation of texture-less 3d objects in heavily cluttered scenes". In: *Computer Vision–ACCV 2012*. Springer, 2012, pp. 548–562.

[26] T. Hodaň et al. "Detection and fine 3D pose estimation of texture-less objects in RGB-D images". In: *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. Sept. 2015, pp. 4421–4428. DOI: 10.1109/IROS.2015.7354005.

[27] S. Holzer et al. "Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images". In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. Oct. 2012, pp. 2684–2689. DOI: 10.1109/IROS.2012.6385999.

[28] B.K.P. Horn. "Closed-form solution of absolute orientation using unit quaternions". In: *J. Opt. Soc. Amer.* 4.4 (1987), pp. 629–642.

[29] R.A. Horn and C.R. Johnson. *Matrix Analysis*. UK: Cambridge University Press, 1985.

[30] *Joseph L. Flatley, Visualized: Kinect*. Website. https://www.engadget.com/2010/11/08/visualiz kinect-night-vision-lots-and-lots-and-lots-of-do/. 2010.

[31] James Kennedy. "Particle swarm optimization". In: *Encyclopedia of Machine Learning*. Springer, 2010, pp. 760–766.

[32] Byung-soo Kim, Shili Xu, and Silvio Savarese. "Accurate localization of 3D objects from RGB-D data using segmentation hypotheses". In: *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*. IEEE. 2013, pp. 3182–3189.

[33] Eunyoung Kim and Gerard Medioni. "3D object recognition in range images using visibility context". In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2011, pp. 3800–3807.

[34] Torsten Kröger. *On-Line Trajectory Generation in Robotic Systems: Basic Concepts for Instantaneous Reactions to Unforeseen (Sensor) Events*. Vol. 58. Springer, 2010.

[35] Kevin Lai et al. "A large-scale hierarchical multi-view rgb-d object dataset". In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE. 2011, pp. 1817–1824.

[36] *M. De Cecco, Robotics and Sensor Fusion*. Website. https://www.slideserve.com/dalmar/micro kinect. 2016.

[37] Emanuele Magrini, Fabrizio Flacco, and Alessandro De Luca. "Control of generalized contact motion and force in physical human-robot interaction". In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE. 2015, pp. 2298–2304.

[38] Emanuele Magrini, Fabrizio Flacco, and Alessandro De Luca. "Estimation of contact forces using a virtual force sensor". In: *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE. 2014, pp. 2126–2133.

[39] Frank Michel et al. "Pose Estimation of Kinematic Chain Instances via Object Coordinate Regression". In: *Proceedings of the British Machine Vision Conference 2015, BMVC 2015, Swansea, UK, September 7-10, 2015*. 2015, pp. 181.1–181.11. DOI: 10.5244/C.29.181. URL: http://dx.doi.org/10.5244/C.29.181.

[40] *Microsoft Kinect*. Website. https://developer.microsoft.com/en-us/windows/kinect. 2011.

[41] Ajay K Mishra, Ashish Shrivastava, and Yiannis Aloimonos. "Segmenting "simple" objects using RGB-D". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE. 2012, pp. 4406–4413.

[42] Ajay Mishra, Yiannis Aloimonos, and Cheong Loong Fah. "Active segmentation with fixation". In: *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE. 2009, pp. 468–475.

[43] Sebastian Montabone and Alvaro Soto. "Human detection using a mobile platform and novel features derived from a visual saliency mechanism". In: *Image and Vision Computing* 28.3 (2010), pp. 391–402.

[44] L de P Veronese and Renato A Krohling. "Swarm's flight: accelerating the particles using C-CUDA". In: *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*. IEEE. 2009, pp. 3264–3270.

[45] Karl Pauwels et al. "Real-time object pose recognition and tracking with an imprecisely calibrated moving RGB-D camera". In: *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE. 2014, pp. 2733–2740.

[46] Juan Pineda. "A parallel algorithm for polygon rasterization". In: *ACM SIGGRAPH Computer Graphics*. Vol. 22. 4. ACM. 1988, pp. 17–20.

[47] Deepak Rao et al. "Grasping novel objects with depth segmentation". In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. IEEE. 2010, pp. 2578–2585.

[48] Carl Yuheng Ren and Ian Reid. "A unified energy minimization framework for model fitting in depth". In: *Computer Vision–ECCV 2012. Workshops and Demonstrations*. Springer. 2012, pp. 72–82.

[49] Xiaofeng Ren, Liefeng Bo, and Dieter Fox. "Rgb-(d) scene labeling: Features and algorithms". In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE. 2012, pp. 2759–2766.

[50] Colin Rennie et al. "A Dataset for Improved RGBD-based Object Detection and Pose Estimation for Warehouse Pick-and-Place". In: *CoRR* abs/1509.01277 (2016).

[51] *RGB-D Object Dataset*. Website. http://rgbd-dataset.cs.washington.edu. 2016.

[52] S. Rosa and G. Toscana. "Fast Feature-Less Quaternion-based Particle Swarm Optimization for Object Pose Estimation From RGB-D Images". In: *Proceedings of the British Machine Vision Conference (BMVC), 2016*. 2016.

[53] Henrik Schäfer, Frank Lenzen, and Christoph S Garbe. "Depth and Intensity Based Edge Detection in Time-of-Flight Images." In: *3DV*. 2013, pp. 111–118.

[54] Max Schwarz, Hannes Schulz, and Sven Behnke. "RGB-D object recognition and pose estimation based on pre-trained convolutional neural network features". In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE. 2015, pp. 1329–1335.

[55] Lorenzo Sciavicco and Bruno Siciliano. *Modelling and control of robot manipulators*. Springer Science & Business Media, 2012.

[56] Ken Shoemake. "Animating rotation with quaternion curves". In: *ACM SIGGRAPH computer graphics*. Vol. 19. 3. ACM. 1985, pp. 245–254.

[57] Bruno Siciliano and Oussama Khatib, eds. *Springer Handbook of Robotics*. Berlin, Heidelberg: Springer, 2008. ISBN: 978-3-540-23957-4. URL: http://dx.doi.org/10.1007%5C/978-3-540-30301-5.

[58] Joan Sola. "Quaternion kinematics for the error-state KF". In: (2017).

[59] Alykhan Tejani et al. "Latent-class hough forests for 3D object detection and pose estimation". In: *Computer Vision–ECCV 2014*. Springer, 2014, pp. 462–477.

[60] Giorgio Toscana and Stefano Rosa. "Fast Graph-Based Object Segmentation for RGB-D Images". In: *CoRR* abs/1605.03746 (2016).

[61] Jiaolong Yang, Hongdong Li, and Yunde Jia. "Go-icp: Solving 3d registration efficiently and globally optimally". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2013, pp. 1457–1464.

[62] You Zhou and Ying Tan. "GPU-based parallel particle swarm optimization". In: *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*. IEEE. 2009, pp. 1493–1500.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.