Moratello Matteo, Obetti Christian, Romano Domenico
Master Degree in Computer Engineering
A.Y. 2019/20 Course of Autonomous Robotics

03/18/2020

Report Homework 1

TASK 1

1 Introduction

In recent years the vision has experienced a massive diffusion in the industrial sector. For this homework, we have implemented two vision algorithms, capable of returning the poses of objects of different sizes and shapes, scattered around a table.

2 Source structure and pipeline description

Source structure:

- *main.cpp*, containing the main;
- *objectDetector.h* and *objectDetector.cpp*, respectively header and implementation file for the class with auxiliary functions used and the callback function.

3 Procedure explanation

Once we received the object requested by the user from command line, we create a instance of class ObjectDetector which subscribe Apriltag topic. The ObjectDetector constructor take:

- the NodeHandle n;
- a vector of apriltag_ros::AprilTagDetection containing AprilTag detections;
- a vector of string that contains the name of the requested objects;
- an array of string containing all frames ids that will be saved in a similar variable of the class.

Furthermore, in the class we have defined the callback function that will be called at the end of the above constructor, subscribing to the topic *"tag_detections"*.
In the callback function we find the Marker ID, comparing user frame ID and the array of requested objects. Then we save the IDs in a vector of integers *index* and for every object requested by the user, print the informations about position and orientation only for objects that are on the table. For better result, we compute the RPY orientation and save all this data in an output file *"poses.txt"* and in the the class vector of AprilTagDetection *det*.
After that, we create a publisher for the topic *"/ur5/poses"* and we publish a message of poses only if there is a subscribe to Apriltag and we update the message with value of objects poses.

4 Conclusions

For this first task we didn't put much effort since the ROS environment did almost all of the job, hence we tried to make a good implementation of the code.

TASK 2

1 Source structure and pipeline description

Source structure:

- *main.cpp*, containing the main and the callback function;
- *objectDetection.h* and *objectDetection.cpp*, respectively header and implementation file for the auxiliary functions used;
- *environment_constants.h*, containing the constants used for virtual and real environment.

In this report, we will focus on the pipeline followed in *main.cpp*. Every detail relative to the procedure will be described in the source and in the next section. After the callback function is called in the main function, the ROS message will be transformed into a PointCloud object. Firstly, we filter the cloud in x, y and z axis. This allows us to reduce the number of points of the scene and avoid outliers. After that we remove the table, to enhance clustering, which is done in the next step. Then, for each cluster, we perform an object classification (criteria are discussed in the dedicated section). In the object classification step, we also compute the pose of the object, which corresponds to the centroid, and the orientation. All this data is saved in an output file *"posesPCL.txt"*.

2 Procedure explanation

The order in which the following topics are presented may actually not correspond to the order followed in the source, that's because we tried to reduce the number of iterations over the scene and each cluster. More details are available in the source.

2.1 Table removal

For this process, we followed the tutorial described in `http://pointclouds.org/documentation/tutorials/planar_segmentation.php#planar-segmentation` and also explained in class. This allowed us to get good results.

2.2 Clustering

For this process, we started from the tutorial explained in `http://pointclouds.org/documentation/tutorials/cluster_extraction.php`, that means we used the Euclidean Cluster Extraction Method. Briefly, this method iterates over each point of the scene. For each point, extract the list of neighbors (that depends on the euclidean distance), adding them recursively to a list Q of neighbor points (only if not previously added). After found all neighbors of a point, saves this list Q as a cluster.

2.3 Color detection

We picked as color of an object the one that is shared by most of the points in the cluster.
To determine the color of a point, we evaluated the RGB values of it and pick as color the one that satisfies some experimentally selected thresholds.

2.4 Object classification

The classification we decided to use can be considered somehow weak, because it is restricted to the domain of figures used for the experience:

- Cubes, that can be blue or red

- Triangular prism, which can be red or green
- Hexagonal cylinder, which can be only yellow.

As one can observe, colors can be used for a strong initial classification: the only issue is between the red hexagonal prism and the red cube. So for this case, we compute the height of the object. The height calculation was designed considering that the table is seen tilted by the camera. For the calculation of the height we have considered the lowest point of the Point Cloud of the clustered object and the lowest point of the Point Cloud of the object projected on the table, in both cases in the reference system of the camera. Height is the difference between the two points considered, based on experiments we have set a threshold at 0.9 cm, the lowest objects of this threshold are prisms, otherwise they are cubes. Figure 1 shows an example of height computation of a cube.
The reason that took us to go for this way, is that the point cloud we could work with had a bad quality and a classification based on models (such as Correspondence Grouping) was working poorly.

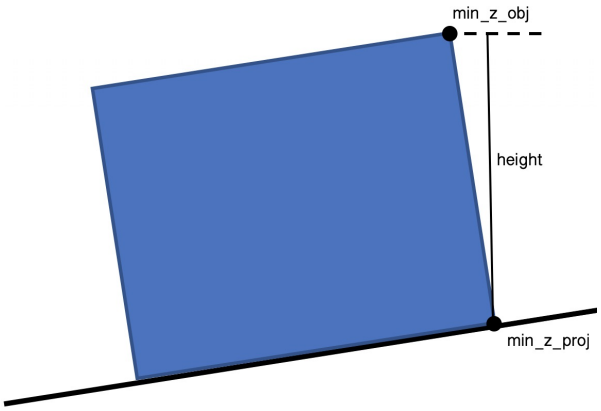

Figure 1: Height computation of a cube

2.5 Position estimation

The position corresponds to the centroid of the object, so to compute it, we simply considered each point x, y and z and did an average.

2.6 Orientation estimation

For the computation of the orientation we projected the Point Cloud of the object on the plane of the table, the equation of the plane is retrieved from the Table removal process.
In this way we can work with a flat figure, so removing some defects of the Point Cloud and making the estimation of the orientation much simpler and more robust.
To estimate the orientation of the objects, for each object we calculate a vector that corresponds to the y axis in the camera reference system. Then we compute the yaw angle between the camera's y axis and the one calculated for the object.
We assumed that the object orientation has roll and pitch components equals to zero, so that it cannot be folded in any way. The estimated orientation is expressed in quaternion.
To compute the orientation we followed the geometric properties of each classified object:

- If a cube, we needed to compute two consecutive base vertices. To do so, we take some points of the classificated cluster that share an x value close to the minimum x value and take as first vertex the point between these that has the minimum y (vertex A). Then for selecting the consecutive vertex, we select points with y value close to the maximum y and take the one between this with the minimum x (in this way we "force" it to be in the same edge of the first vertex, vertex B). After have found the two vertices, we obtain the orientation by vector subtraction.

The direction was found using the edge where are vertices A and B, indeed the vector is parallel to that edge (this also applies to the case of the triangular prism).
Figure 2 shows the vertices considered for computing vector for cubes.

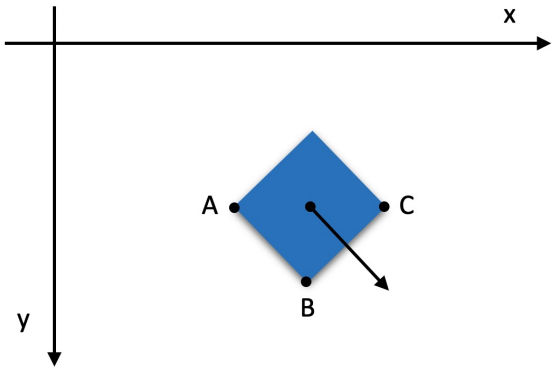

Figure 2: Vertices considered for computing vector for cubes

- If a triangular prism, we also need two consecutive vertices that are at the closest distance: we want the orientation to be parallel to the smallest edge. The procedure consist of selecting three consecutive vertices: the first and the second ones obtained with the same procedure as for the cube, the third one by picking as vertex the point with minimum y among the point that have x value close to the maximum x in the cluster. Then, we compare the distances between the consecutive edges and select as orientation the closest couple.
  Note: In the case of the prism, both in the simulated and real environment, a part of the base of the point cloud is cut. For this reason the lengths of the two sides are no longer coherent those shown in the Figure 3, but are inverted, and for this reason we choose the vector parallel to the smaller edge.
  Figure 3 shows the vertices considered for computing vector for prisms.

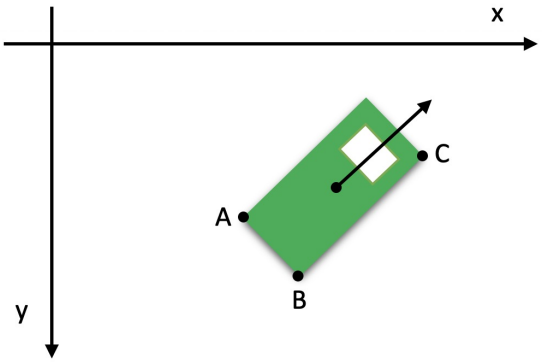

Figure 3: Vertices considered for computing vector for prisms

- If an hexagon prism, we obtain the first base vertex with the same procedure of the other solids for the first vertex, then pick as vertex the opposite one, which is the point at the maximum distance from the selected vertex. The orientation is as before the vector subtraction between them.
  Figure 4 shows the vertices considered for computing vector for hexagons.

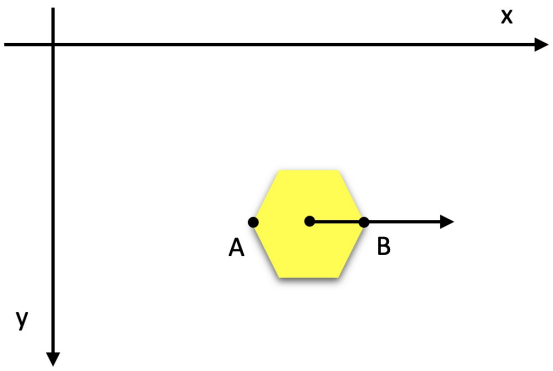

Figure 4: Vertices considered for computing vector for hexagons

3 Practical examples

In this section we'll show some results obtained in simulation to demonstrate the correctness of our algorithm and then we'll compare them with real results.
In general we have found that the quality of the Point Cloud has influenced the quality of the vertices found and consequently the orientation vector estimation.

3.1 Cube

In Figure 5 we can see the two axis y (the green one) and x (the red one) and the blue cube detected with the two vertices (red points on the same edge) and the vector found (the red line).
The data shown represent position and orientation of the cube, in particular the yaw value represents the angle between the found vector and the y axis.

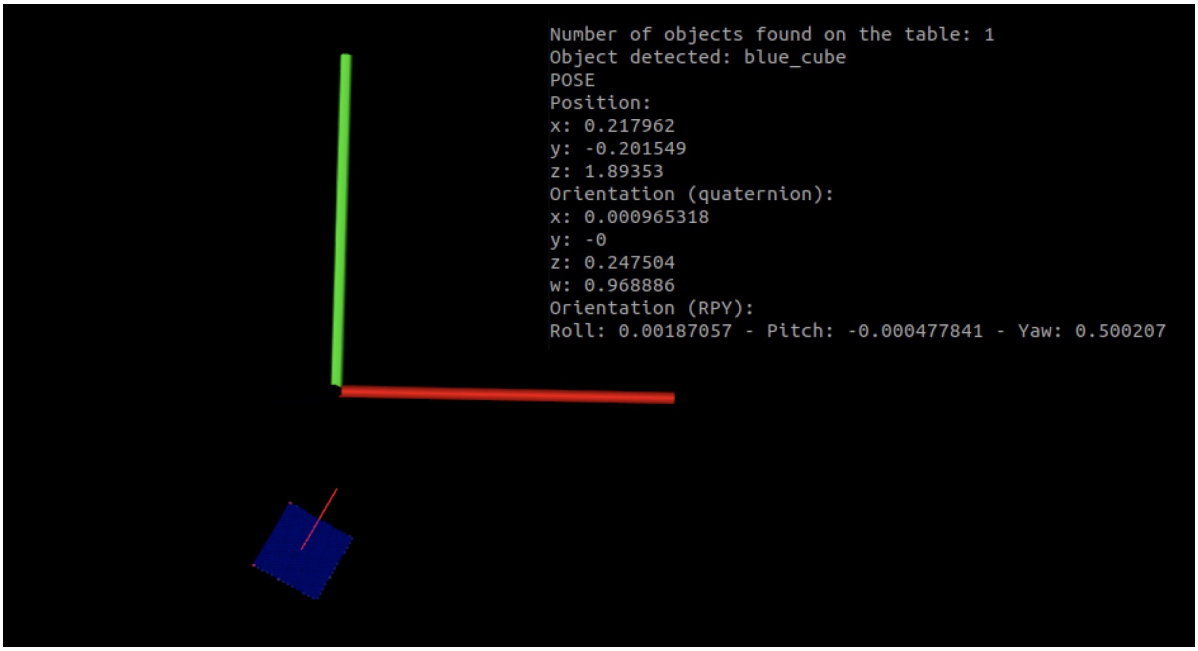

Figure 5: Result of the vector of a cube and the pose estimation in simulation.

In the real case the point cloud is not so good but we can be satisfied with the vector result. We have a set of candidates vertices (white points) while the two chosen ones are in purple.
The result is shown on Figure 6.

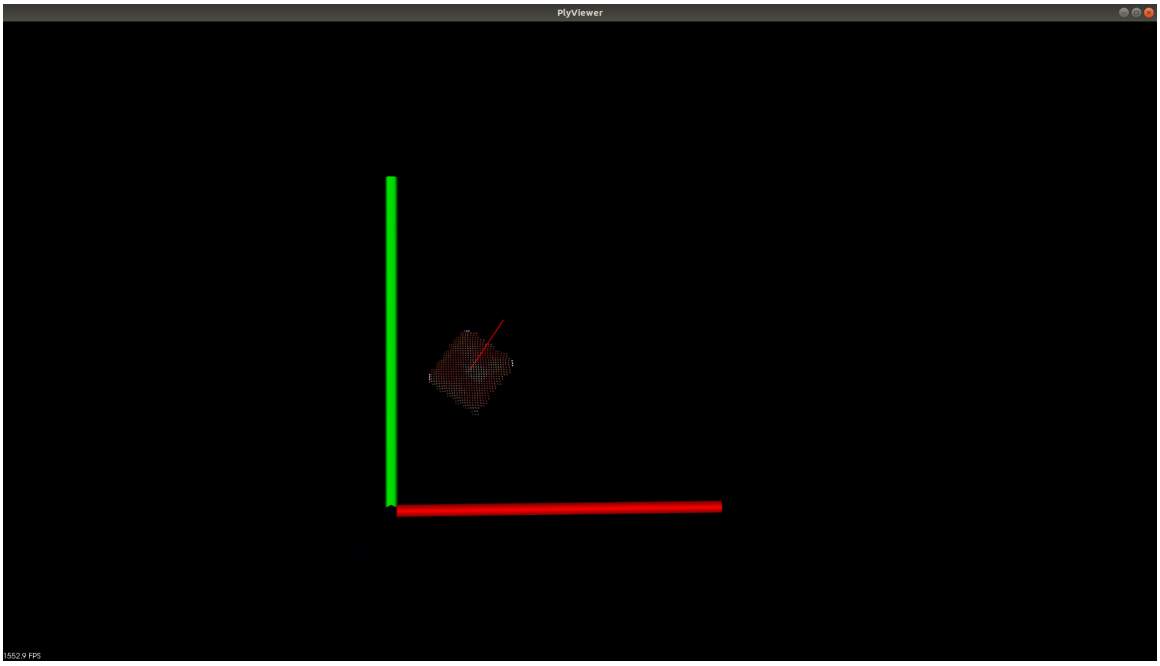

Figure 6: Result of the vector of a cube in real-world

3.2 Triangular prism

As explained before, for the triangular prism we needed three vertices for found the vector.

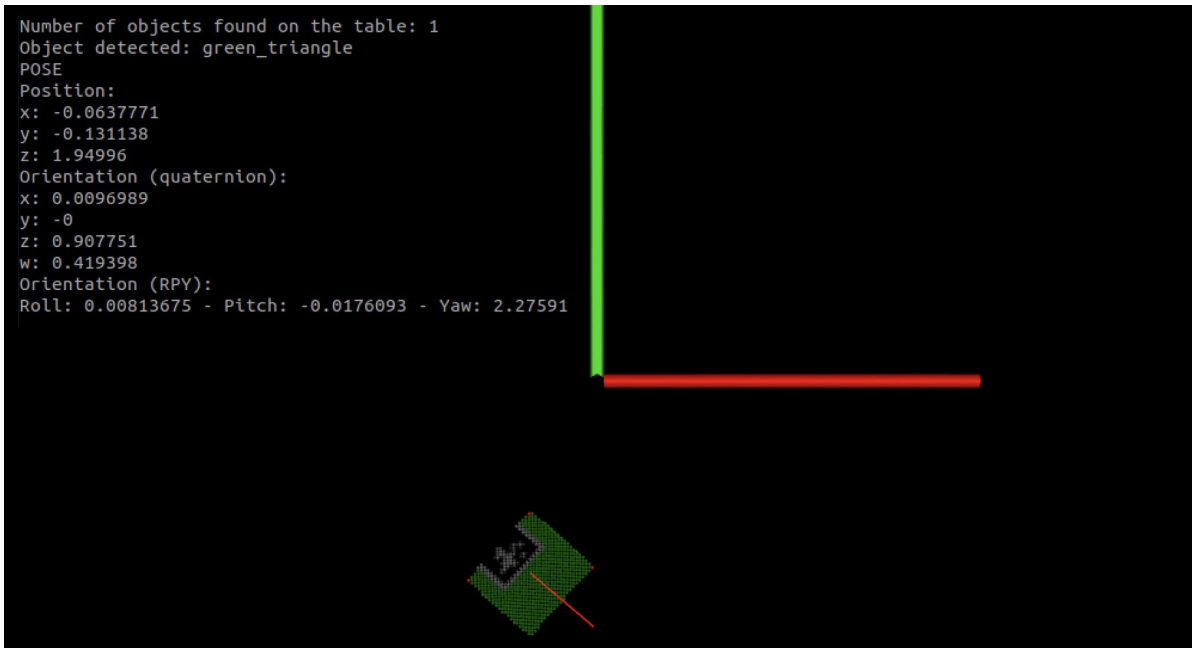

Figure 7: Result of the vector of a prism and the pose estimation in simulation

Also in this case, the result in the real environment is sufficient: there are three little purple points corresponding to the three vertices and the red vector.
Figure 8 shows the result for the real environment.

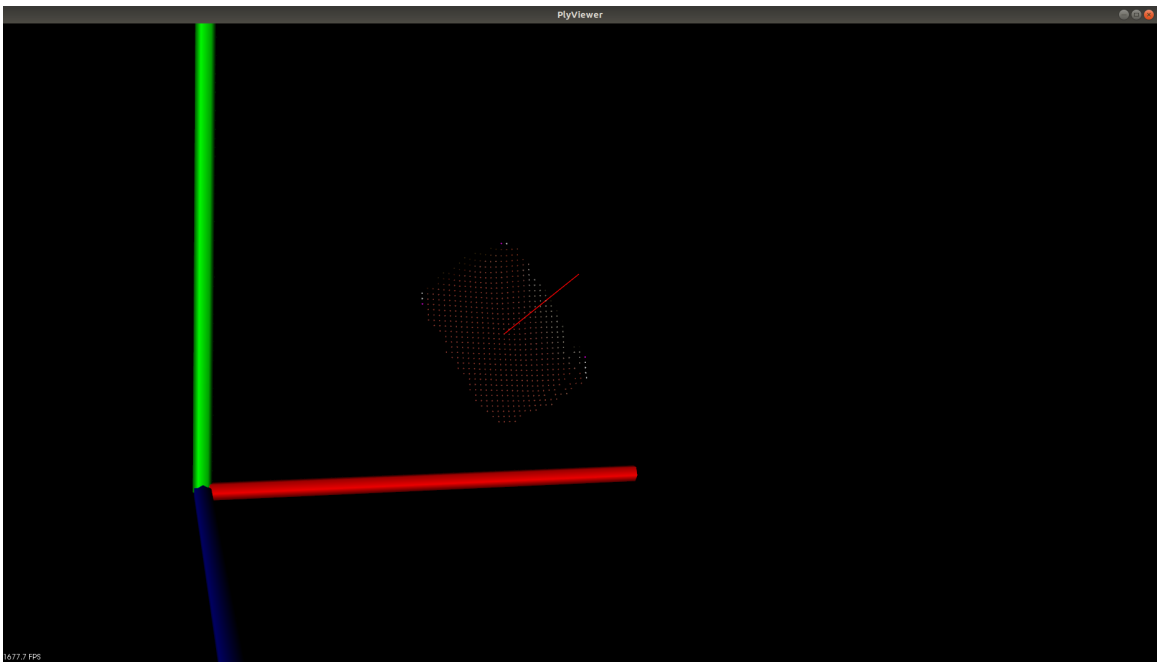

Figure 8: Result of the vector of a prism in real-world

3.3 Hexagon cylinder

Also in this final case we could make the same considerations as before.

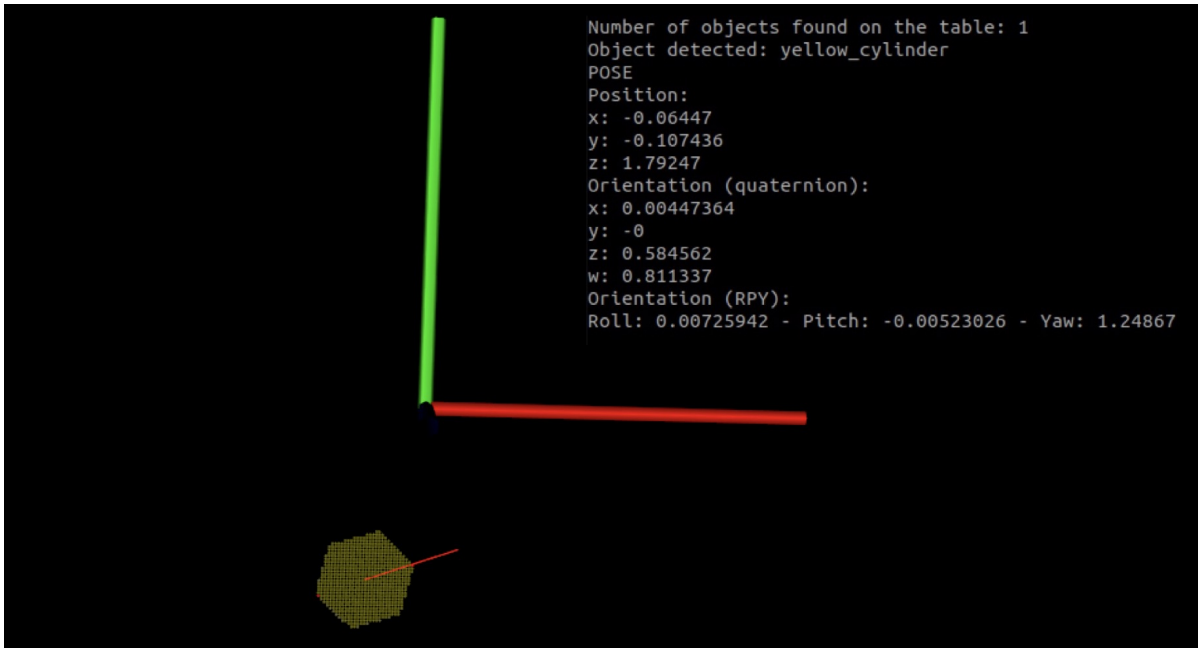

Figure 9: Result of the vector of a hexagon and the pose estimation in simulation

In real environment, we can see that the vector is not perfectly aligned with the two purple vertices but the result is overall correct. Figure 10 shows the result.

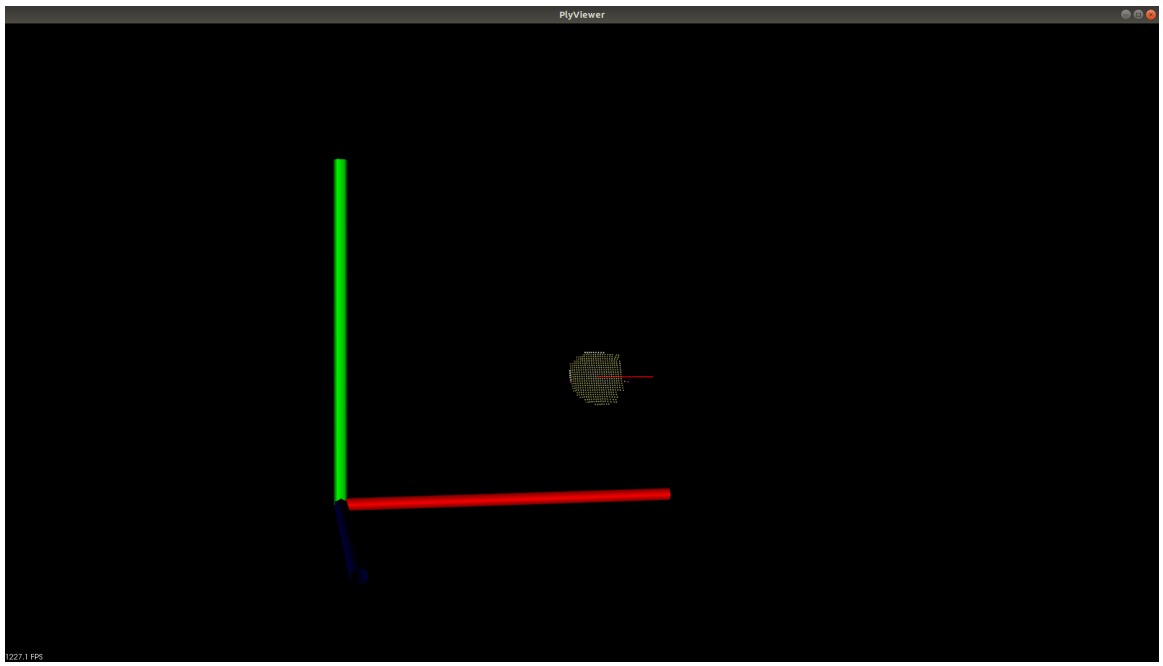

Figure 10: Result of the vector of a hexagon in real-world

4 How to run the code

4.1 Task 1

The following roslaunch command run the first task. The parameter **objects** defines the objects requested by the user.

$ roslaunch hw1 objectDetection.launch objects:=“red_cube_2 green_prism_0 yellow_cyl_0”

4.2 Task 2

The following command run the second task. The requested objects are defined in **args**[].

$ rosrun hw1 pcl red_cube blue_cube

5 Conclusions

We obtained decent results for this task: we are able to detect and classify correctly every solid in the scene.
Unfortunately the selected procedure can’t be well generalized and is domain specific. We had strong issues with the reconstructed PointCloud, because of its quality (the 3D scene was practically a 2D scene and the solid were missing the body).