Grasping Assuming Symmetry

Pradnya Sushil Shinde (901013260)
Om Vinayak Gaikwad (390281097)
Omkar Bharambe (200170598)
Upasana Mahanti (418716635)
Vishrut Bohara (234400666)

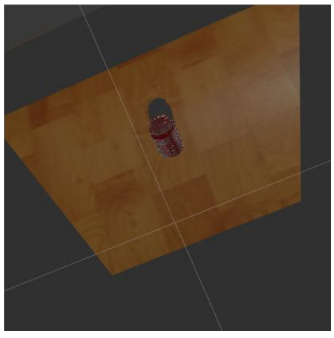
1. Introduction/Abstract

The "Grasping Assuming Symmetry" project leverages the nature of symmetry present in many objects. Using a depth camera, our approach captures real-time 3D representations of the environment, focusing on object symmetries to optimize grasping points. This integration of depth-sensing technology and algorithmic innovation offers a promising advancement in robotic manipulation, ensuring more stable and efficient interactions with a multitude of objects in real-world scenarios.

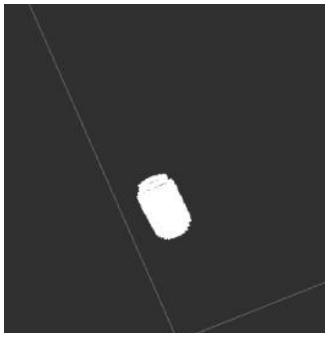
2. Implementation

Step 1: Downsampling and major plane removal

- The point cloud data is first downsampled using a voxel grid to reduce the number of points, making subsequent processes computationally efficient.
- Plane segmentation using RANSAC is applied to remove the plane from the point cloud. This ensures that only the object of interest remains in the cloud.



Input Point Cloud



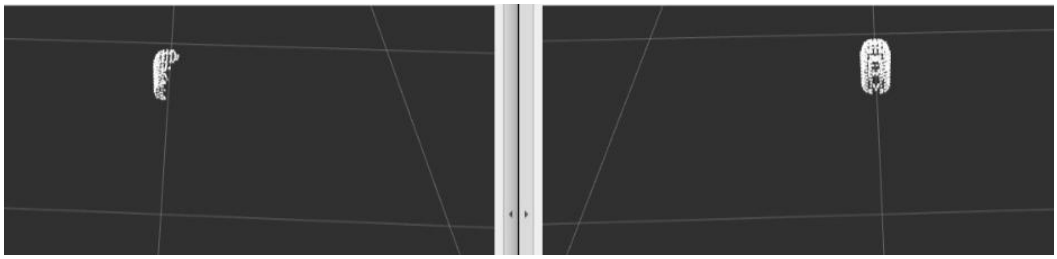
Output Point Cloud

```
//Downsampling using VoxelGrid
sor.setInputCloud(input_cloud);
sor.setLeafSize (_min_dist,_min_dist,_min_dist);
sor.filter(*filtered_cloud);
RCLCPP_INFO_STREAM(this->get_logger(), "SAMPLED PCL size " << filtered_cloud->width * filtered_cloud->height);

//Downsampled pcl::PointCloud2 -> pcl::PointXYZ
pcl::fromPCLPointCloud2(*filtered_cloud,*seg_point);

//Plane segmentation and removal
int itr = 0;
int initial_size = seg_point->size();
while (seg_point->size () > 0.2 * initial_size && itr<6)
{
    //Plane Fitting
    seg.setInputCloud (seg_point);
    seg.segment (*inliers, *coefficients);

    if (inliers->indices.size () == 0)
    {
    RCLCPP_ERROR_STREAM (this->get_logger(),"Could not estimate a planar model for the given dataset.\n");
    return;
    }

    // Extract the inliers
    extract.setInputCloud (seg_point);
    extract.setIndices (inliers);
    // Removing detected plane
    extract.setNegative (true);
    extract.filter (*seg_point);
    itr++;
}
```

Code snippet for downsampling and major plane removal

Step 2: Point Cloud Completion

- A symmetry-based approach is utilized to complete the point cloud. The best symmetry plane is determined based on visibility constraints.



Incomplete Point Cloud Complete Point Cloud

- We first create all possible planes oriented at different angles.
- We then mirror every point in the current point cloud about each of these planes.
- Once the mirrored point is obtained, we then evaluate the plane in consideration based on the condition: (distance - (distance*distance)/dist_threshold)*4/dist_threshold.
- The above condition is nothing but the visibility score metric which is calculated for all planes.

- Finally, we select the best plane which has the highest score to generate the complete point cloud.

```
void completePointCloud(pcl::PointCloud<pcl::PointXYZ>::Ptr& incomplete_points,pcl::PointCloud<pcl::PointXYZ>::Ptr& complete_points)
{
    int input_points = incomplete_points->size();
    float min_score = input_points*_visibility_score/5.0f;
    // calculate all possible planes
    std::vector<std::vector<float>> planes;
    for(float theta = _min_theta;theta<=_max_theta;theta+=(_max_theta-_min_theta)/theta_steps) { ... }
    //for all planes compute metric
    std::vector<float> best_symmetry_plane = {(_max_x+_min_x)/2.0f,(0.06f+_max_y+_min_y)/2.0f,(_max_theta+_min_theta)/2.0f,0.0};

    kd->setInputCloud(incomplete_points);
    for(int i=0;i<int(planes.size());i++){
        for(auto point: incomplete_points->points){
            if(planes[i][3]<min_score) break;
            evaluate_plane(planes[i],point);
        }
    }
    std::sort(planes.begin(), planes.end(), sortcol);
    best_symmetry_plane[0] = planes[0][0];
    best_symmetry_plane[1] = planes[0][1];
    best_symmetry_plane[2] = planes[0][2];

    //complete the pointcloud using the best plane
    *complete_points = *incomplete_points;
    for(auto point: incomplete_points->points){
        complete_points->push_back(mirror_point(point,best_symmetry_plane));
    }
}
```

Code snippet for Point Cloud completion

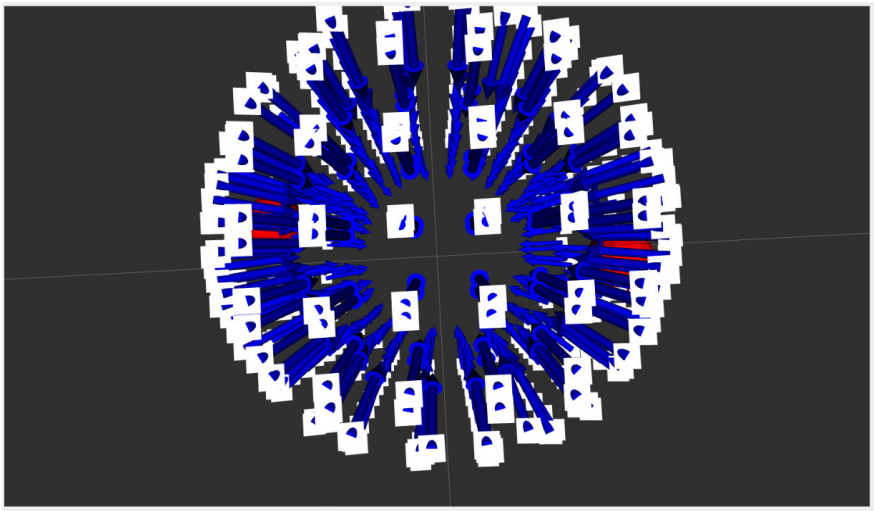
```
void evaluate_plane(std::vector<float>& plane, const pcl::PointXYZ& point){
    //Logic
    float score = 0.0;
    pcl::PointXYZ mirrored_point = mirror_point(point,plane);
    pcl::PointXYZ point_im = pcl::transformPoint(mirrored_point,transform_inv_mat);
    float xz = point_im.x/point_im.z;
    float yz = point_im.y/point_im.z;
    float dist = 1000.0f ;

    float err = 0.0f;

    int K = 1 ;
    std::vector<int> pointIdxKNNSearch(K);
    std::vector<float> pointKNNSquaredDistance(K);
    if ( kd->nearestKSearch (mirrored_point, K, pointIdxKNNSearch, pointKNNSquaredDistance) > 0 )
    {
        dist = std::sqrt(pointKNNSquaredDistance[0]) ;
    }

    if(dist <= _min_dist) return;

    if((point_im.z < _min_im_z) ||  xz > _max_im_xz + err || (yz > _max_im_yz + err) || (xz < _min_im_xz - err) || (yz < _min_im_yz - err)) {
        score = _visibility_score ;
    }
    else{
        score = std::max((dist-(dist*dist)/_dist_threshold)*4/_dist_threshold, -1.0f);
    }
    plane[3] += score;
}
```

Algorithm used to evaluate different planes to determine the best plane for symmetry.

Step 3: Normal Estimation

- Normals are calculated using the pcl::NormalEstimation class. This method estimates normals based on the local neighborhood of each point.

- KDTree is used for efficient spatial neighbor searching.
- The computed normal at a point is influenced by the positions of its neighbors. The method calculates the best fitting plane for the neighbors and determines the plane's perpendicular as the normal.
- Normals are then visualized as small arrows emanating from points in the point cloud, pointing in the direction of the normal as seen in the figure below.



Surface normals of the coke can from top view.

```
void getNormals(pcl::PointCloud<pcl::PointXYZ>::Ptr& complete_points,pcl::PointCloud<pcl::Normal>::Ptr& normals)
{
    pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> ne;
    ne.setInputCloud(complete_points);

    ne.setSearchMethod(KDtree);
    ne.setRadiusSearch(0.05);
    ne.setViewPoint(_mean.x,_mean.y,_mean.z);
    ne.compute(*normals);

    publish_marker(complete_point,normals);

}
```

Code snippet for Normal Extraction

Step 4: Grasp Detection

We have calculated the best grasp contacts for the robot when interacting with 3D point cloud data completed in the previous steps. The key steps of this process include:

1. **Input parameters:**
   This includes the completed point cloud represented using the Point Cloud Library (PCL) data structure and another point cloud containing the surface normals corresponding to

the points in the completed point cloud.

2. **Nearest Point Search:**
   A nearest point to the centroid of the completed point cloud is calculated using KDTree and the result is stored. This is done by calculating the squared distance metric of each point in the completed point cloud from the centroid (mean) of the completed point cloud.

3. **Angle calculation:**
   The function then calculates the angles between the normals of each point in the point cloud and the vectors from nearest point (contact1) to these points. It uses the dot product and vector magnitudes to calculate the angle between the normal and the vector. The function iterates through all points in complete_points and updates the current point (contact2) based on the minimum angle. The goal is to find a point that, when grasped along with contact1, results in the smallest angle between the contact normals and the grasp direction. This is based on the principle shown below:

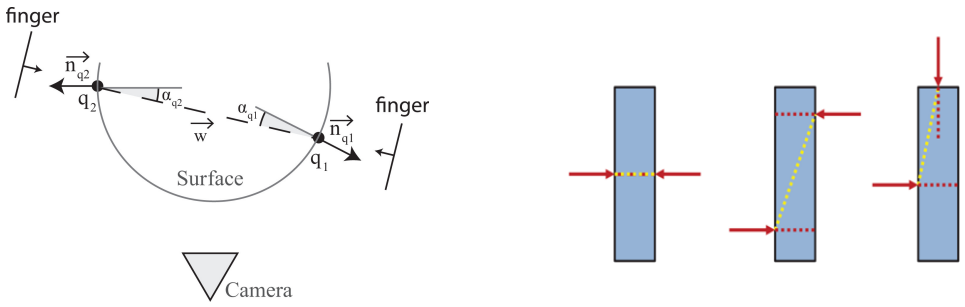


*image source: https://journals.sagepub.com/doi/10.1177/1729881419831846

   Among these, the first grasp is the most stable one as the angle between the normals at the contact points and the vector between the contact points is minimum.

4. **Best grasp:**
   Thus, we obtain the second point for grasping which will be the most stable grasping point along with the point nearest to the centroid of the completed point cloud.

Output:

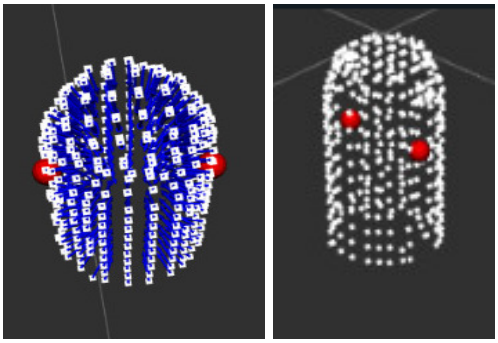


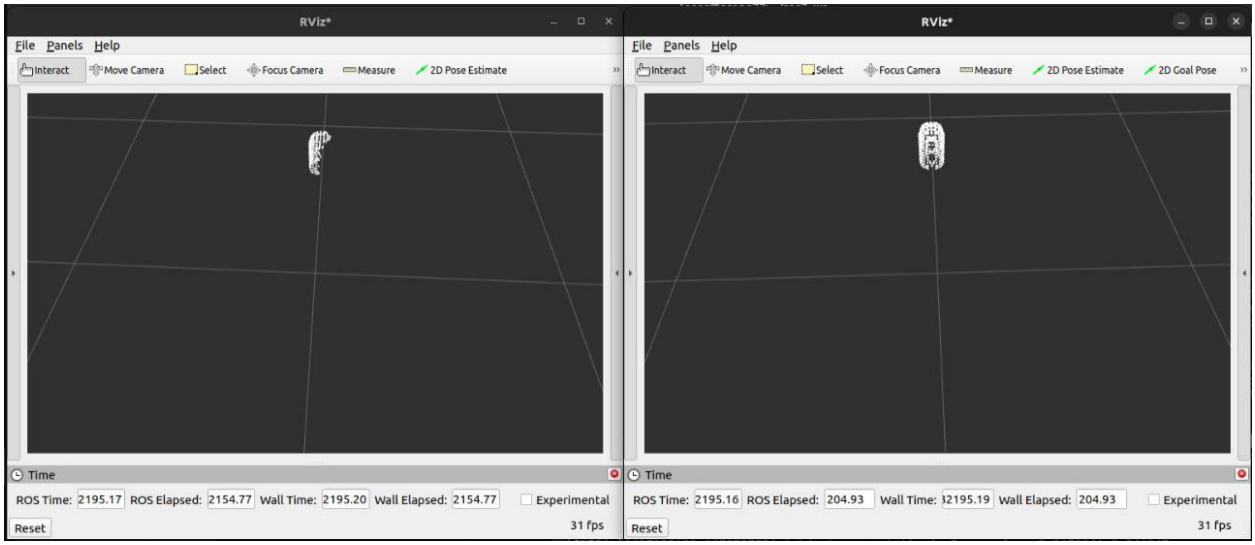
Fig - Best grasping contacts (red) have been calculated for the object ‘coke_can’.

```cpp
void findBestGrasp(pcl::PointCloud<pcl::PointXYZ>::Ptr& complete_points,pcl::PointCloud<pcl::Normal>::Ptr& normals)
{
    pcl::PointCloud<pcl::PointXYZ> contacts;
    pcl::PointXYZ contact1(0.0,0.0,0.0);
    pcl::PointXYZ contact2(1.0,1.0,1.0);
    pcl::Normal contact1_normal;

    // create a KDTree for the points
    kd->setInputCloud(complete_points);

    // get the nearest point to the _mean in KDTree using Knearest function in KDTree and save it in contact1
    int K = 1 ;
    std::vector<int> pointIdxKNNSearch(K);
    std::vector<float> pointKNNSquaredDistance(K);
    pcl::PointXYZ nearest_point;
    if ( kd->nearestKSearch (_mean, K, pointIdxKNNSearch, pointKNNSquaredDistance) > 0 )
    {
        contact1 = complete_points->points[pointIdxKNNSearch[0]];
        contact1_normal = normals->points[pointIdxKNNSearch[0]];
    }

    double min_angle = 6.24;

    for (size_t i = 0; i < complete_points->points.size(); ++i)
    {
        pcl::PointXYZ current_point = complete_points->points[i];
        pcl::Normal current_normal = normals->points[i];

        //
        pcl::PointXYZ vector_to_current_point;
        vector_to_current_point.x = current_point.x - contact1.x;
        vector_to_current_point.y = current_point.y - contact1.y;
        vector_to_current_point.z = current_point.z - contact1.z;

        // angle calculation
        double dot_product = (current_normal.normal_x * vector_to_current_point.x +
                              current_normal.normal_y * vector_to_current_point.y +
                              current_normal.normal_z * vector_to_current_point.z)*-1;

        double vector_magnitude = sqrt(vector_to_current_point.x * vector_to_current_point.x +
                                       vector_to_current_point.y * vector_to_current_point.y +
                                       vector_to_current_point.z * vector_to_current_point.z);
        // double normal_magnitude = sqrt(current_normal.normal_x * current_normal.normal_x +
        //                                current_normal.normal_y * current_normal.normal_y +
        //                                current_normal.normal_z * current_normal.normal_z);
        double angle = acos(dot_product / (vector_magnitude )); //* normal_magnitude

        // angle calculation
        dot_product = contact1_normal.normal_x * vector_to_current_point.x +
                              contact1_normal.normal_y * vector_to_current_point.y +
                              contact1_normal.normal_z * vector_to_current_point.z;
        angle += acos(dot_product / (vector_magnitude )); //* normal_magnitude

        // minimum angle checks
        if (angle < min_angle)
        {
            min_angle = angle;
            contact2 = current_point;
        }
    }

    // publish contacts
    contacts.push_back(contact1);
    contacts.push_back(contact2);
```

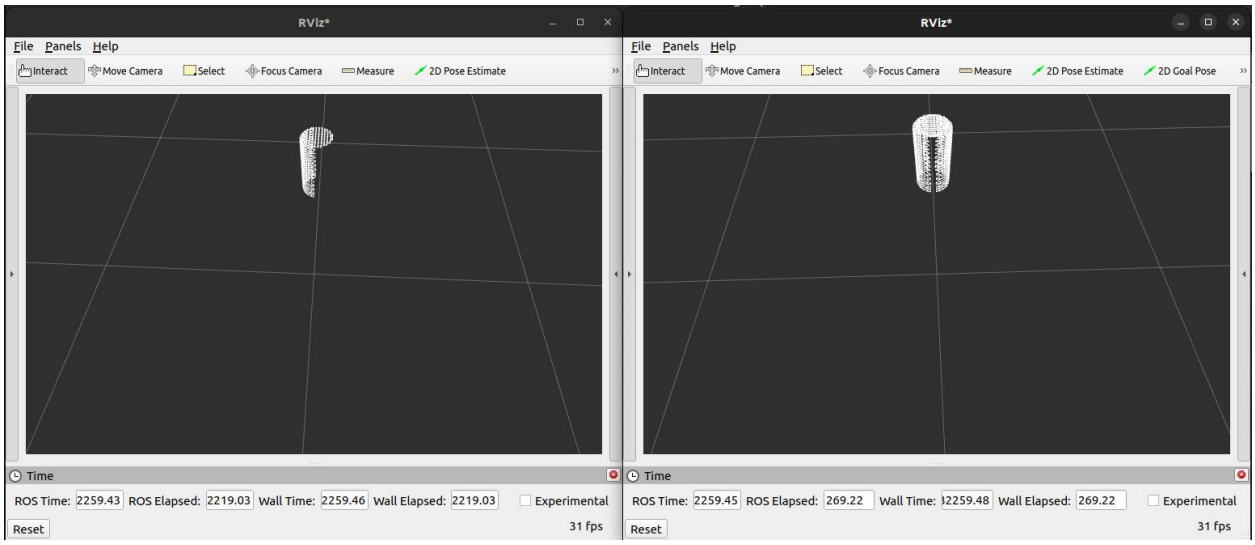
Code snippet for Normal Estimation

3. Results of Point Cloud completion

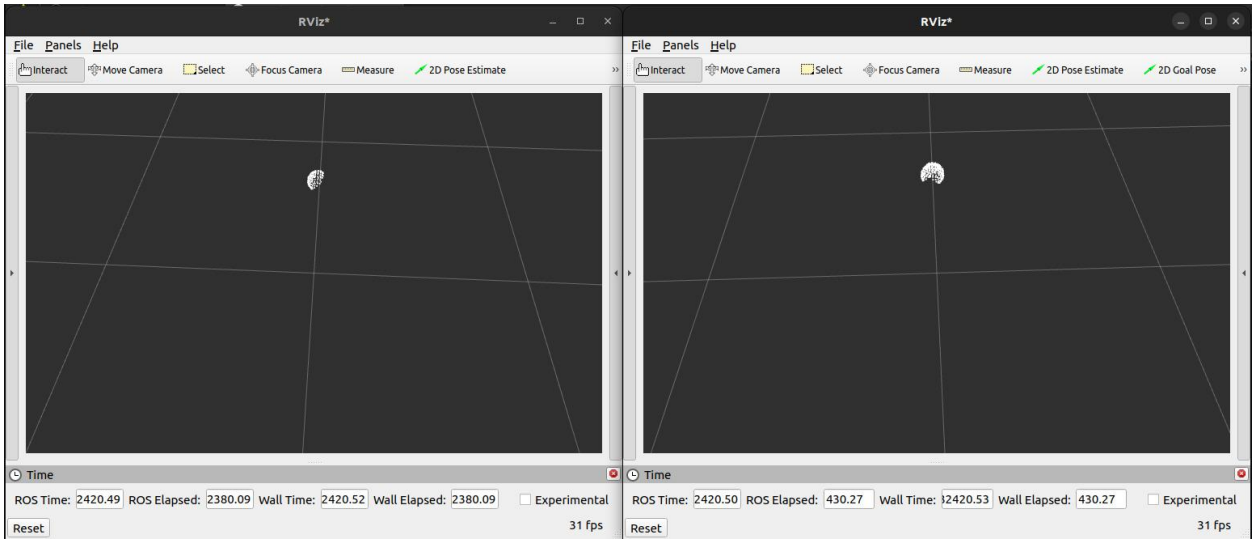
- **Case 1: Coke can**



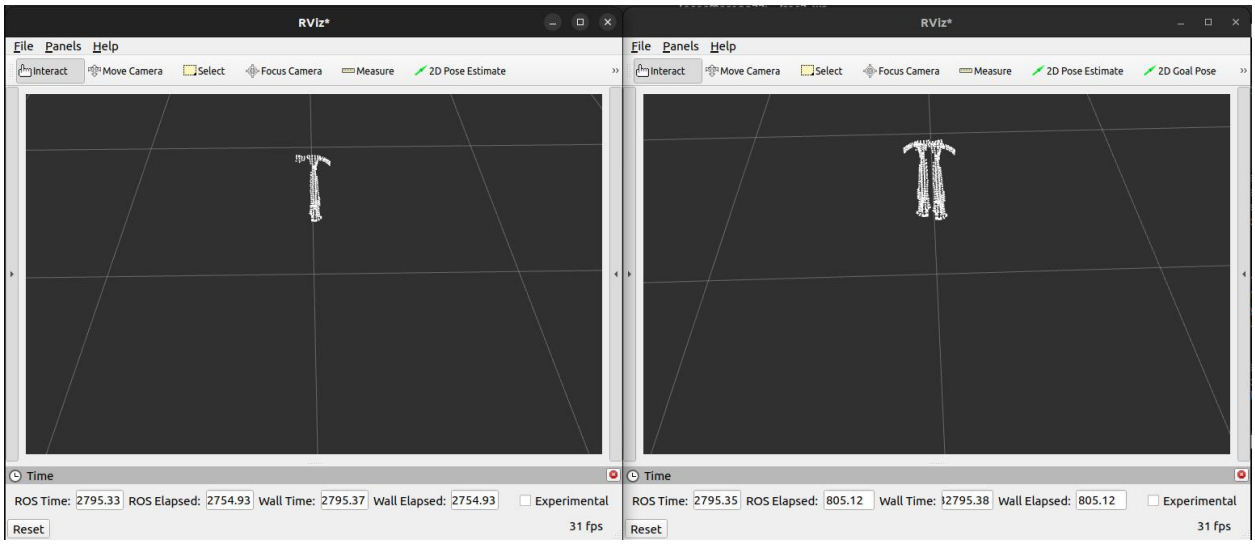
- **Case 2: Beer can**



- **Case 3: Cricket ball**



- **Case 4: Hammer** (negative case requires tuning to work properly)



4. Learning Outcome

- In conclusion, the "Grasping Assuming Symmetry" project represents a significant step forward in the realm of robotic manipulation. By harnessing the inherent symmetry of various objects and integrating depth-sensing technology with cutting-edge algorithms, we have unlocked the potential for more stable and efficient interactions with objects in real-world scenarios.
- Our project demonstrates a systematic approach, from downsampling and major plane removal to point cloud completion and normal estimation, all aimed at optimizing grasp

points. The development of grasp detection algorithms adds a practical dimension to our skills, enabling robots to interact with 3D point cloud data effectively.

- Moreover, the diverse range of objects we tested, from simple cans to more complex shapes like a cricket ball, underscores the versatility of our methodology. This project lays the foundation for enhanced robotic manipulation and real-time object interactions in the field of robotics and automation.

5. How to run the code?

- Build the code and source the workspace
- Terminal 1 : ros2 launch vbm_project_env simulation.launch.py
- Terminal 2 : ros2 run vbm_project_env sample_object
- Terminal 3 : rviz2 (to observe the point cloud by adding Topics)