Jaco Robotic Arm

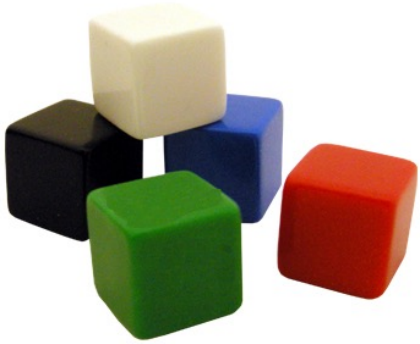
Victor Choudhary

Overview

The objective of the project is to build a autonomous robotic system to play a game (tic-tac-toe in our case) with human on other end with the help of Jaco Robotic Arm and Depth camera.

Requirements of the project are :-

- Jaco Robotic Arm
- Depth Camera
- Cubes of 2 different colours
- Tic Tac Toe board
- Robot Operating System (R.O.S) working computer with ubuntu.



ROS

Project Working & Master Node

Overview of working

Overall working of the project is divided into following 4 parts :-

- Master Node
- Vision Node
- Jaco Movement Nodes
  - Jaco arm movement node
  - Jaco finger movement node
- Tic Tac Toe Game playing engine

Major working is being handled by the master node. All other nodes are being used by master to fulfil there particular functioning. Vision node is to system like what eyes are to humans. It provides information about the current scenario of the game to the master node. Jaco Movement Node takes control information from master and does the job of moving the arm & fingers. Game engine is responsible for the strategy to be used by the master in order to direct control commands to jaco arm.

Master Node

Master node serves as master controller of the overall system. It tracks the progress of the game and is used for decision making i.e. what should be done & when should be done.

The functioning of the master node is divided in 2 major functions :-

- main ( )
- get_cubes ( )

The main() function is entry point into the system i.e. whenever the nodes are initiated, the system first approaches this function. Initially it simply initiates the ROS system and calls function define_locations() (this function puts the hardcoded locations into the main memory), init_matrix() (this function is used to initialise the matrix used for tic tac toe mapping).
Finally it calls get_cubes() function.

```
int main(int argc,char** argv) {
    ros::init(argc,argv,"game_master");
        ……….
        ……….
    return 1;
}
```

The function get_cubes(int argc, char** argv) takes the system parameters provide to main function as parameters.
get_cubes() function simply subscribes to the "/camera/depth_registered/points" topic and calls work() function from the vision_node.
It the published the point cloud message to "transformed_output".

```
void get_cubes(int argc,char** argv){
    ros::init(argc,argv,"matrix_transform");
        ……….
        ……….
    ros::spin();
}
```

Finally there are few variable parameters which are being introduced in master node, such as location, position, etc. which are being used by supporting nodes for communcation with each other.

Jaco Arm Movement

Overview

The package used for handling the arm movement is JACO_ROS, it provides a ROS interface for Kinova Robotics Jaco robotic manipulator arm. Thus providing access to kinova JACO C++ hardware API through ROS.
The JACO API is exposed to ROS using a combination of *actionlib* (which are used for sending trajector commands to the arm), *services* (for instant control such as homing the arm, stopping the arm) and *published topics* (i.e. joint feedbacks).

Arm can be controlled via 2 mechanisms :-

- Angular Commands
- Cartesian co-ordinates

In our project we have used the *cartesian coordinate system* for commaning the arm. The cartesian control is accomplished via actionserver and published as seperate topics.

The following is the message format for Cartesian Control :-

- **`float64 position.x`** `i.e.(end effector distance)`
- **`float64 position.y`** `i.e.(end effector distance)`
- **`float64 position.z`** `i.e.(end effector distance)`
- **`float64 orientation.x`** `i.e. (end effector quaternion)`
- **`float64 orientation.y`** `i.e. (end effector quaternion)`
- **`float64 orientation.z`** `i.e. (end effector quaternion)`
- **`float64 oreintation.w`** `i.e. (end effector quaternion)`

The arm has maximum reach of about 0.9 m (90 cm), so the "position" range is about +/-0.9 m for all three dimenstions.
Three wrist joints are capable of continuous rotation, & therefore capable of being commanded up to +/-174.5 radian.

Published topics for Cartesian Position :-

- **`/jaco_arm_driver/out/tool_position`**

Action Server topics :-

- **`/jaco_arm_driver/arm_pose/arm_pose`**

Arm Movement Node :-

There are 3 major functions responsible for arm movement :-

- move_it ( )
- navigate_to ( )
- move_finger ( )

move_it(int cell_location) is located in master.cpp file. It serves as single overall function for moving the arm from one location to another, moving the fingers for dropping and picking the cubes. It requires 1 parameter i.e. location of the board cell where the cube is to be moved.

It first takes the *arm to the home position* by using the homing service provided by jaco_ros package.

```
ros::ServiceClient home_client = n.serviceClient
     <jaco_msgs::HomeArm>("/jaco_arm_driver/in/
     home_arm");
jaco_msgs::HomeArm home_srv;
home_client.call(home_srv);
```

After this it gives command to open the each finger to 30 degree. Then it call the navigate_to() function in order to take the arm to the desired coordinates. After the arm is naviated to the desired location, it commands the fingers to close i.e. sends them 45,45,30 degree orientation. In this way the cube is picked, again

naviagte_to() is used to take the arm to cell location. And the cube is dropped over the cell location by opening the fingers.
Finally the arm is taken to home position by calling the home_client.call() method.

navigate_to(position pos) is the function used called by move_it() function is serves as a abstract function to move the arm from one coordinate to another coordinate. The functioning of navigate_to() function is based on the use of action library api.
It first defines a client for action lib.

```
typedef actionlib:: SimpleActionClient
        <jaco_msgs::ArmPoseAction> Client;
```

Then subscribe the client to the action topic for jaco_ros.

```
Client client("/jaco_arm_driver/arm_pose/
                arm_pose",true);
client.waitForServer();
```

Rest of the function is based on setting the coordinate for target location. and finally calling the sendGoal() function of client.

```
client.sendGoal(goal);
```

move_finger(double angle1, double angle2, double angle3) is the function called by move_it() in order to move the fingers of the arm. The functioing of this function is similar to navigate_to() function as described above. First we create a actionlib client, then provide goal coordinates and finally call sendGoal() function.
The only major difference arise in the format of goal.

```
jaco_msgs::SetFingersPositionGoal goal;
goal.fingers.finger1 = angle1;
goal.fingers.finger2 = angle2;
goal.fingers.finger3 = angle3;
client.sendGoal(goal);
```

Vision

Overview

The role of vision is to provide feedback to master about the location of the cubes on the game board. Major functioning is controlled by *Point Cloud Library (PCL)* for point cloud processing. The PCL framework contains numerous state-of-the art algorithms including filterin, feature estimation, surface recognition & segmentation.
The point cloud processing library is provided by ROS in the name of ***PCL Package,*** it is a standalone C++ library for processing point cloud data.

Vision Node :-

There are 2 major functions responsible for giving the vision node functionality :-

- work() : responsible for coordination between extraction & movement
- colored_segmentation()

int colored_segmentation(…) method is the major function which actually determines the cube location. It takes sensor_msgs::PointCloud2ConstPtr& data as parameter & returns the location of the newly discovered cube.
The function of colored_segmentation() is little complicated. It basically functions in 3 stages :-

- Filtering the Image data
  In this part, the major role is to filter the point cloud data and throw away the unnecessary data. i.e. we pass the point cloud data through 2 filtering processes. The first filtering is done for x-direction data, removing data beyond a certain limit. And the next similar filtering is done in y-direction.

pcl::PassThrough<pcl::PointXYZRGB> pass, pass2;

```
pass.setInputCloud(cloud);
pass.setFilterFieldName("x");
pass.setFilterLimits(-0.2,0.08);
pass.filter(*cloud_filtered);

pass2.setInputCloud(cloud_filtered);
pass2.setFilterFieldName("y");
pass2.setFilterLimits(-0.15,0.1);

pass2.filter(*cloud_final);
```

- Segmentation
  In this part, we segment the point cloud data according the location & color statistics of each point.
  It is done simply by creating a pcl::SACSegmentation <pcl::PointXYZRGB> object and then setting various parameters and finally calling the segment() function on this object.

```
pcl::SACSegmentation<pcl::PointXYZRGB> seg;
seg.setOptimizeCoefficients(true);
seg.setModelType(pcl::SACMODEL_PLANE);
seg.setMethodType(pcl::SAC_RANSAC);
seg.setMaxIterations (1000);
seg.setDistanceThreshold(0.01);

seg.setInputCloud(cloud_final);
seg.segment(*inliers,*coefficients);
```

- Extract Indices
  In this part, we extract the segments one by one and finally get the actuall cube segments in the form of clusters.
  The following code are the major components of this process.

```
// create filtering object
pcl::ExtractIndices<pcl::PointXYZRGB> extract;
```

```
    // while 30% of cloud is still there
    while(cloud_final->points.size() > 0.1 *
          nr_points){
       // segment largest planar components from
the remaining cloud
         seg.setInputCloud(cloud_final);
           ……….
           ……….
           ……….
       }
```

The following code is used to formulate a RegionGrowingRGB object and then finally extract the clusters.

```
    pcl::RegionGrowingRGB<pcl::PointXYZRGB> reg;
    reg.setInputCloud(cloud_final);
    reg.setIndices(indices);
    reg.setSearchMethod(tree);
    reg.setDistanceThreshold(10);
    reg.setPointColorThreshold(6);
    reg.setRegionColorThreshold(5);
    reg.setMinClusterSize(600);

    pcl::PointCloud<pcl::PointXYZRGB>::Ptr
         colored_cloud = reg.getColoredCloud();

    std::vector<pcl::PointIndices> clusters;
    reg.extract(clusters);
```

Game Engine

Game engine for Tic-Tac-Toe is defined in *game.h* file.
The game engine gives a abstract functionality for game by the following functions :-

- start()
  `void tictactoe_game::start( tictactoe_player const player)` is the function used for starting the game. It simply initializes the game to initial state.
- lookup_strategy()
  `std::set<tictactoe_status> tictactoe_game :: lookup_strategy()` is the function used for looking up the further startegy for the game.
- lookup_move()
  `tictactoe_status tictactoe_game::lookup_move()` is the function used for looking up the further move.
- move() for player
  `bool tictactoe_game::move(tictactoe_cell const cell, tictactoe_player const player)` is the function used for getting the placing the move for player. It takes 2 parameters i.e. cell & player.
- move() for player
  `tictactoe_cell tictactoe_game::move(tictactoe_player const player)` is the function used for getting the cell at which the robot would move the cube.

For increasing the efficiency of the game engine, we have precomputed the strategical moves for both x and o players. These can be found in *strategy_x.h* & *strategy_o.h*