



BALL CHASER

Submitted By

CHANDU JANAKIRAM (CB.EN.U4AIE20009)

PENAKA VISHNU REDDY

(CB.EN.U4AIE20048)

SVS DHANUSH (CB.EN.U4AIE20068)

.....

For the Completion of

19AIE213 – ROBOTIC OPERATING SYSTEMS AND ROBOT

SIMULATION

AIE

11th July 2021

Introduction

Chasing the white ball using hukayo camera:

Let's use the Robotics Operating system (ROS) capability and gazebo platform. To construct a basic ball tracking vehicle. To hunt down the ball, this bot employed a camera to capture frames and do image processing using open cv. The ball's characteristics, such as color, shape, and size, can be employed. goal was to create a rudimentary prototype for a bot that can detect and follow color.

Objective

- Implementing an automated wheeled robot for tracking ball in Ros and gazebo.
- The main aspect of our problem statement is to build an autonomous wheeled robot in Ros and gazebo environment for tracking down the white ball.

Software Requirements

- ROS Noetic.
- Gazebo.

Implementation and Outputs

WORKING LOGIC:

- Initially, the camera fixed on top of the robot captures the image.
- It searches for a circular shape containing white pixels in the image captured.
- If the white pixel is detected, the robot assumes it to be the ball and it starts following it.
- But if the pixels are not detected, it captures the image until it detects a white pixel.

PROCESS:

- Firstly, we create 3 packages (world, robot description, script packets).
- my_robot package defines the mobile robot as well the world that it is housed within.
- Ball_chaser package contains two nodes which are responsible for locating the position of the white ball in the camera field of view and then sending the corresponding commands to the mobile robot to follow the white ball.

C++ CODES:

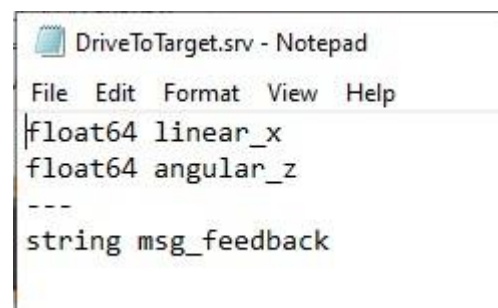
We have coded our project in C++ language, our simulation contains two C++ files one acts as Subscriber (process_image.cpp) node and other as Publisher (drive_bot.cpp) node. Our main function of the codes are as follows:

- Our subscriber subscribes the image from the gazebo and process it, determines whether the ball is present or not. Based on the placement of the ball it assigns the linear and angular values for the variables in the srv file (DriveToTarget.srv). (We will be creating a user defined service file for our implementation)
- The publisher node in simple, it publishes these values to the gazebo.

Based on the pixel value color in the image (White color = 255) the code determines whether the white ball is present or not.

CODE:

DriveToTarget.srv:



```
DriveToTarget.srv - Notepad
File Edit Format View Help
float64 linear_x
float64 angular_z
---
string msg_feedback
```

process_image.cpp:

```
process_image.cpp - Notepad
File Edit Format View Help
#include "ros/ros.h"
#include "ball_chaser/DriveToTarget.h"
#include <sensor_msgs/Image.h>

// Define a global client that can request a service
ros::ServiceClient client;

void drive_robot(float lin_x, float ang_z)
{
    ROS_INFO_STREAM("Moving robot");

    // Request motor commands
    ball_chaser::DriveToTarget srv;
    srv.request.linear_x = (float)lin_x;
    srv.request.angular_z = (float)ang_z;

    // Call the command_robot service and pass the requested motor commands
    if (!client.call(srv))
        ROS_ERROR("Failed to call service command_robot");
}

// The callback function continuously executes and read the image data
void process_image_callback(const sensor_msgs::Image img)
{
    int white_pixel = 255;
    bool found_ball = false;
    int column_index = 0;

    for (int i=0; i < img.height * img.step; i += 3)
    {
        if ((img.data[i] == 255) && (img.data[i+1] == 255) && (img.data[i+2] == 255))
        {
            column_index = i % img.step;

            if (column_index < img.step/3)
                drive_robot(0.5, 1);
            else if (column_index < (img.step/3 * 2))
                drive_robot(0.5, 0);
            else
                drive_robot(0.5, -1);
            found_ball = true;
            break;
        }
    }

    if (found_ball == false)
        drive_robot(0, 0);
}

int main(int argc, char** argv)
{
    // Initialize the process_image node and create a handle to it
    ros::init(argc, argv, "process_image");
    ros::NodeHandle n;

    // Define a client service capable of requesting services from command_robot
    client = n.serviceClient<ball_chaser::DriveToTarget>("/ball_chaser/command_robot");

    // Subscribe to /camera/rgb/image_raw topic to read the image data inside the process_image_callback function
    ros::Subscriber sub1 = n.subscribe("/camera/rgb/image_raw", 10, process_image_callback);

    // Handle ROS communication events
    ros::spin();

    return 0;
}
```

drive_bot.cpp:

```
drive_bot.cpp - Notepad
File Edit Format View Help
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "ball_chaser/DriveToTarget.h"

// Global motor command publisher
ros::Publisher motor_command_publisher;

bool handle_drive_request(ball_chaser::DriveToTarget::Request& req, ball_chaser::DriveToTarget::Response& res)
{
    ROS_INFO("DriveToTargetRequest received - linear_x:%1.2f, angular_z:%1.2f", (float)req.linear_x, (float)req.angular_z);

    // Publish motor command request
    geometry_msgs::Twist motor_command;

    motor_command.linear.x = req.linear_x;
    motor_command.angular.z = req.angular_z;

    motor_command_publisher.publish(motor_command);

    // Return a response message
    res.msg_feedback = "Motor command set - linear_x: " + std::to_string(req.linear_x) + " , angular_z: " + std::to_string(req.angular_z);
    ROS_INFO_STREAM(res.msg_feedback);

    return true;
}

int main(int argc, char** argv)
{
    // Initialize the drive_bot node and create a handle to it
    ros::init(argc, argv, "drive_bot");
    ros::NodeHandle n;

    // Inform ROS master that we will be publishing a message of type geometry_msgs::Twist on the robot actuation topic
    motor_command_publisher = n.advertise<geometry_msgs::Twist>("/cmd_vel", 10);

    // Define a drive /ball_chaser/command_robot service with a handle_drive_request callback function
    ros::ServiceServer service = n.advertiseService("/ball_chaser/command_robot", handle_drive_request);
    ROS_INFO("Ready to send drive commands");

    // Handle ROS communication events
    ros::spin();

    return 0;
}
```

LAUNCHING THE FILE:

At first, we will launch the world (ball_chaser_world.launch) and then from the world file itself we will launch the respective launch file for robot (robot_description.launch) into our gazebo environment. Command to launch the file is given below:

```
roslaunch my_robot ball_chaser_world.launch
```

ball_chaser_world.launch:

```
ball_chaser_world.launch - Notepad
File Edit Format View Help
<?xml version="1.0" encoding="UTF-8"?>

<launch>

  <!-- Robot pose -->
  <arg name="x" default="0.2"/>
  <arg name="y" default="-0.2"/>
  <arg name="z" default="0.2"/>
  <arg name="roll" default="0.75"/>
  <arg name="pitch" default="0"/>
  <arg name="yaw" default="0"/>

  <!-- Launch other relevant files-->
  <include file="$(find my_robot)/launch/robot_description.launch"/>

  <!-- World File -->
  <arg name="world_file" default="$(find my_worlds)/worlds/ballChaser.world"/>

  <!-- Launch Gazebo World -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="use_sim_time" value="true"/>
    <arg name="debug" value="false"/>
    <arg name="gui" value="true" />
    <arg name="world_name" value="$(arg world_file)"/>
  </include>

  <!-- Find my robot Description-->
  <param name="robot_description" command="$(find xacro)/xacro --inorder '$(find my_robot)/urdf/my_robot.xacro'"/>

  <!-- Spawn My Robot -->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
    args="-urdf -param robot_description -model my_robot
      -x $(arg x) -y $(arg y) -z $(arg z)
      -R $(arg roll) -P $(arg pitch) -Y $(arg yaw)"/>

  <!-- launch rviz-->
  <node name="rviz" pkg="rviz" type="rviz" respawn="false"/>

</launch>
```

robot_description.launch:

```
robot_description.launch - Notepad
File Edit Format View Help
<?xml version="1.0"?>
<launch>

  <!-- send urdf to param server -->
  <param name="robot_description" command="$(find xacro)/xacro --inorder '$(find my_robot)/urdf/my_robot.xacro'"/>

  <!-- Send fake joint values-->
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">
    <param name="use_gui" value="false"/>
  </node>

  <!-- Send robot states to tf -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" respawn="false" output="screen"/>

</launch>
```

Command to launch the nodes:

roslaunch ball_chaser ball_chaser.launch

ball_chaser.launch:

```
ball_chaser.launch - Notepad
File Edit Format View Help
<launch>

<!-- The drive_bot node -->
<node name="drive_bot" type="drive_bot" pkg="ball_chaser" output="screen">
</node>

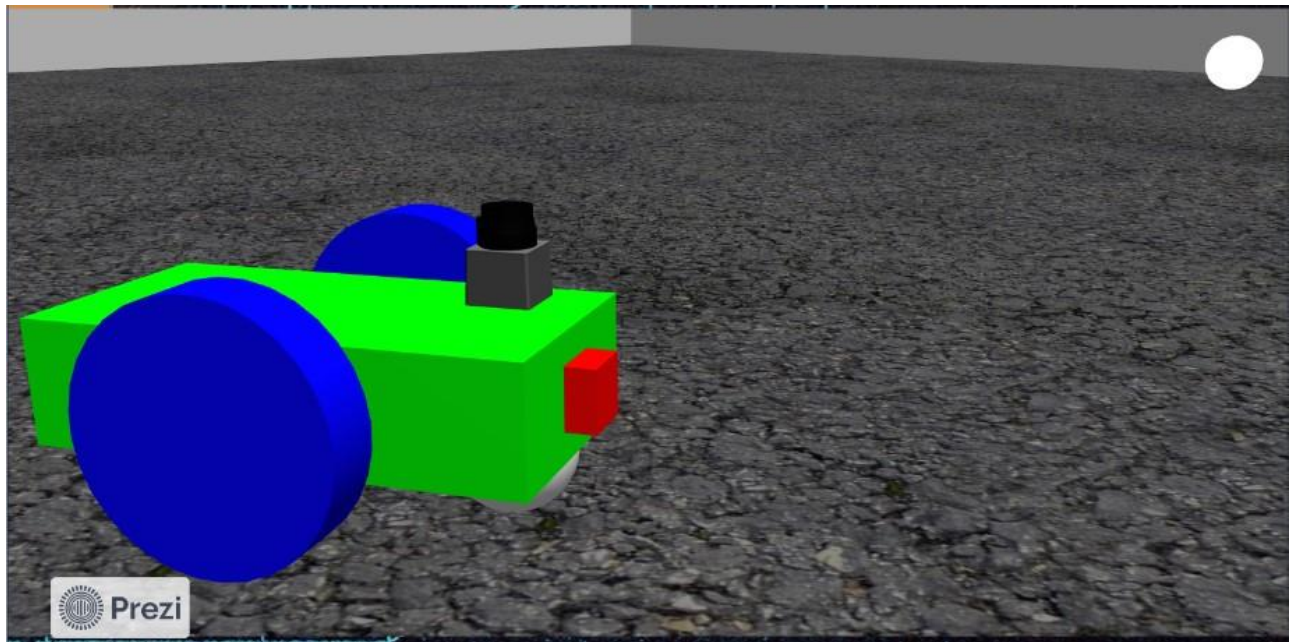
<!-- The process_image node -->
<node name="process_image" type="process_image" pkg="ball_chaser" output="screen">
</node>

</launch>
```

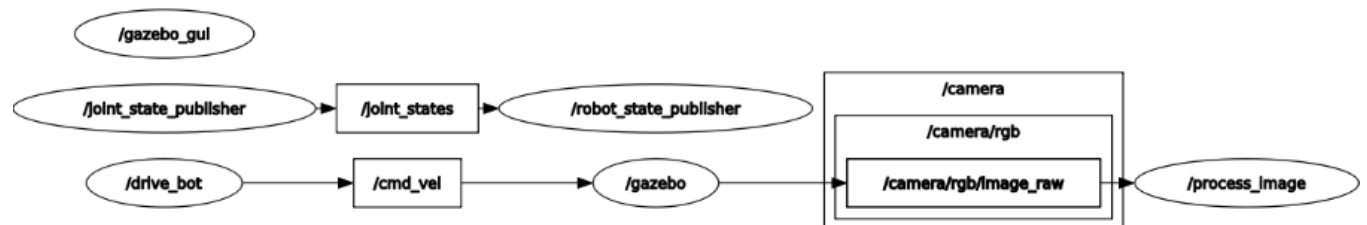
OUR WORLD:



OUR ROBOT:



RQT GRAPH:



Conclusion

Our proposed system detected the white ball and tracked it. Accurate velocity results that are both linear and angular along all axes i.e., X, Y, Z. It also shows in the terminal whether the white ball is detected; it will chase the white ball created in the gazebo platform.