Mediapipe Gesture-AprilTag Distance Sorting

Before starting this function, you need to close the large program and APP processes. If you need to restart the large program and APP later, start them from the terminal:

```
bash ~/dofbot_pro/APP_DOFBOT_PRO/start_app.sh
```

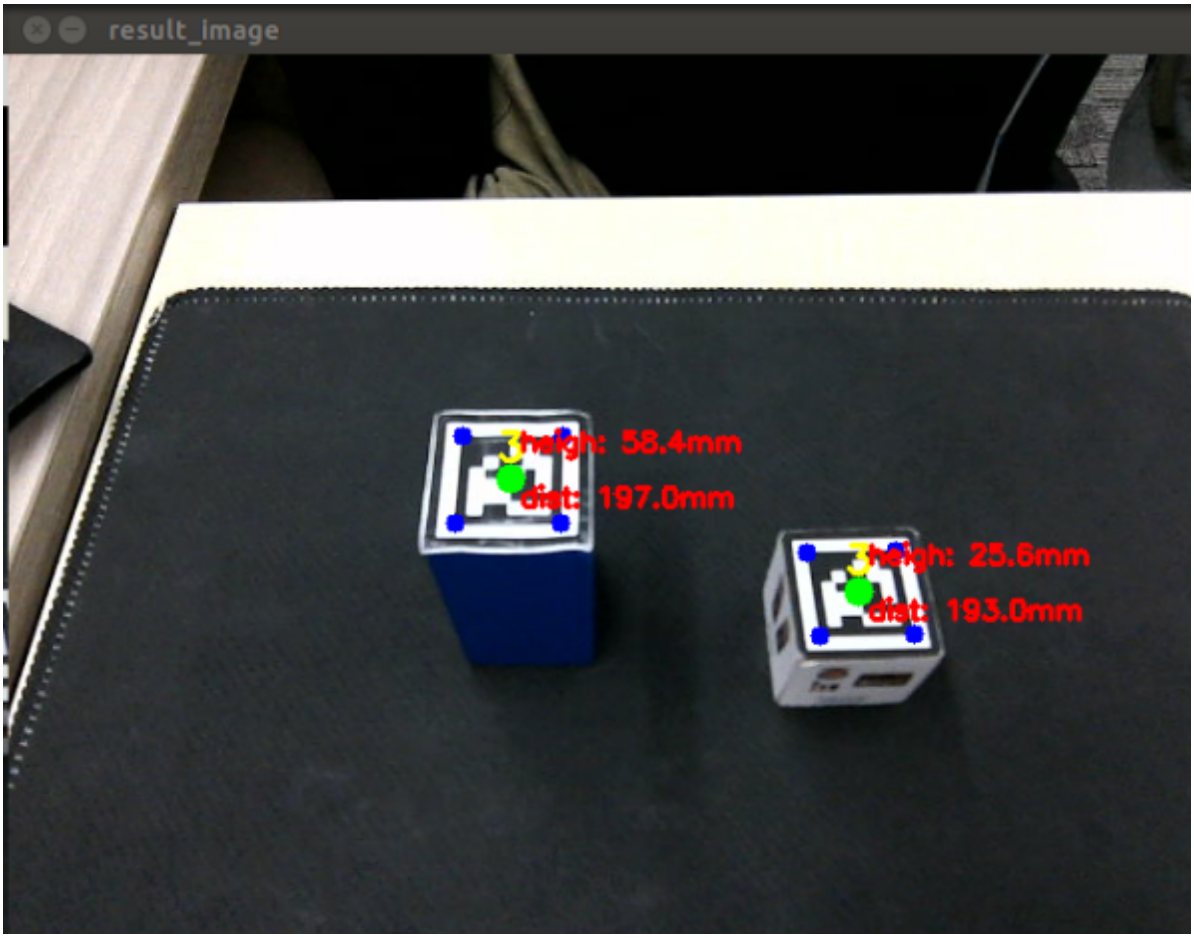
1. Function Description

After the program starts, the camera captures images and recognizes gestures. Gestures range from 1 to 5. Through the recognized gesture, the distance threshold is calculated; the robotic arm will change its posture to detect AprilTags in the image and calculate their height and distance. If any are smaller than the distance threshold, the robotic arm will lower the gripper to grasp and place them at the set position, then return to the AprilTag detection posture to continue recognition; if no AprilTag smaller than the distance threshold is detected, the robotic arm will perform a "head shaking" action group, then the robotic arm returns to the gesture recognition posture.

2. Startup and Operation

2.1. Startup Commands

```
#Start camera:
ros2 launch orbbec_camera dabai_dcw2.launch.py
#Start underlying control:
ros2 run dofbot_pro_driver arm_driver
#Start inverse kinematics program:
ros2 run dofbot_pro_info kinemarics_dofbot
#Start image conversion program:
ros2 run dofbot_pro_apriltag msgToimg
#Start AprilTag recognition program:
ros2 run dofbot_pro_apriltag apriltag_list_Dist
#Start robotic arm grasping program:
ros2 run dofbot_pro_driver grasp
#Start Mediapipe gesture recognition program:
ros2 run dofbot_pro_apriltag MediapipeGesture
```



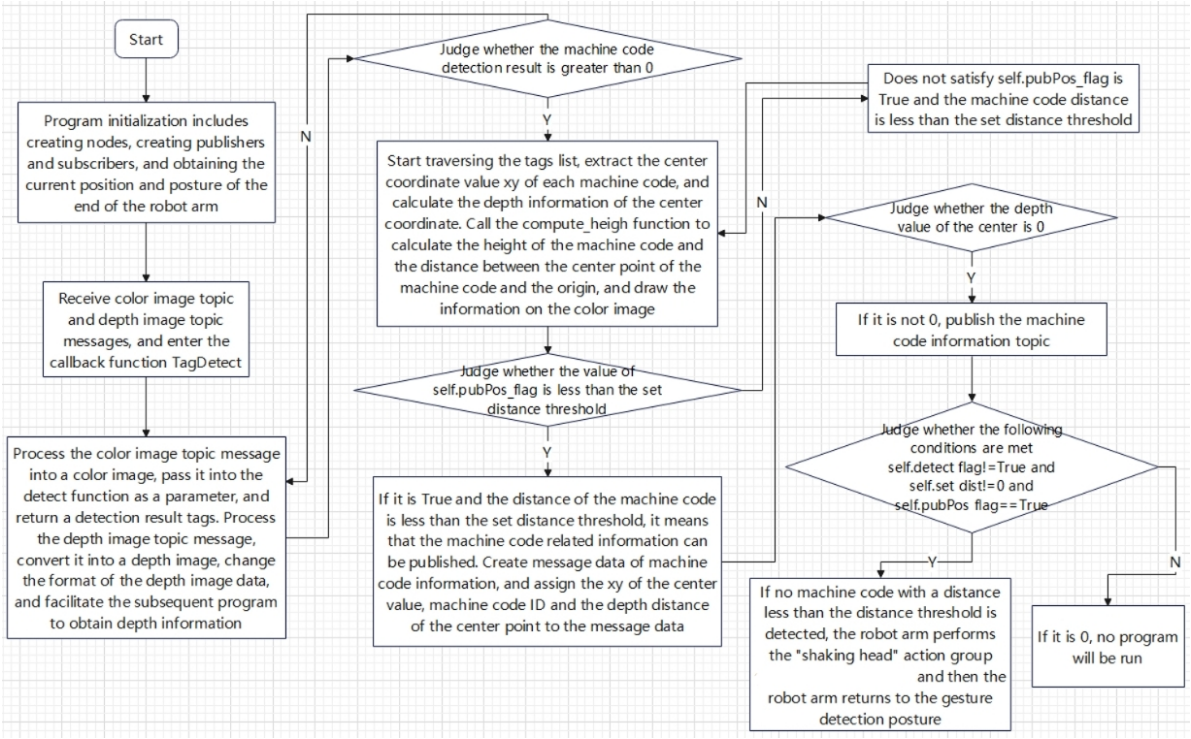
2.2. Operation

After the program starts, the robotic arm will initially present a gesture recognition posture. The recognizable gestures range from one to five. Gesture recognition waits for about 3 seconds, waiting for the AprilTag to change posture to the AprilTag detection and recognition posture. Press the spacebar to start recognition; if a AprilTag with distance smaller than the calculated distance threshold is recognized, the robotic arm will lower the gripper to grasp that AprilTag block and place it at the set position; after placement is complete, the robotic arm returns to the AprilTag recognition posture. The next recognition requires pressing the spacebar again. If no AprilTag smaller than the set distance threshold is recognized, the robotic arm will perform a "head shaking" action group, then the robotic arm returns to the gesture recognition posture.

Distance threshold calculation: 170 + gesture recognition result * 10

3. Program Flowchart

apriltag_list_Dist.py



4. Core Code Analysis

4.1. MediapipeGesture.py

Code path:

```
/home/jetson/dofbot_pro_ws/src/dofbot_pro_apriltag/dofbot_pro_apriltag/MediapipeGesture.py
```

You can refer to section 4.1 [MediapipeGesture.py] in tutorial [3D Space Sorting and Grasping\3. Mediapipe Gesture-AprilTag ID Sorting].

4.2. apriltag_list_Dist.py

Code path:

```
/home/jetson/dofbot_pro_ws/src/dofbot_pro_apriltag/dofbot_pro_apriltag/apriltag_list_Dist.py
```

Import necessary libraries

```
import cv2
import rclpy
from rclpy.node import Node
import numpy as np
from message_filters import ApproximateTimeSynchronizer, Subscriber
from sensor_msgs.msg import Image
from std_msgs.msg import Float32, Int8, Bool
from dt_apriltags import Detector
from dofbot_pro_apriltag.vutils import draw_tags
from cv_bridge import CvBridge
import cv2 as cv
from dofbot_pro_interface.srv import *
from dofbot_pro_interface.msg import ArmJoint, AprilTagInfo
import pyzbar.pyzbar as pyzbar
```

```
import time
import queue
import math
import os
encoding = ['16UC1', '32FC1']
import threading
#Import transforms3d library for handling transformations in 3D space, performing
conversions between quaternions, rotation matrices and Euler angles, supporting
3D
import transforms3d as tfs
#Import transformations for handling and calculating transformations in 3D space,
including conversions between quaternions and Euler angles
import tf_transformations as tf
from Arm_Lib import Arm_Device
```

Program parameter initialization, create publishers and subscribers

```
def __init__(self):
    rospy.init_node('apriltag_detect')
   super().__init__('apriltag_detect')

         # Initialize parameters
        self.detect_joints = [90.0, 150.0, 12.0, 20.0, 90.0, 30.0]
        self.init_joints = [90.0, 120.0, 0.0, 0.0, 90.0, 90.0]
        self.search_joints = [90.0, 120.0, 0.0, 0.0, 90.0, 30.0]
        self.Arm = Arm_Device()

        # ROS2 publishers
        self.pubGraspStatus = self.create_publisher(Bool, "grasp_done", 1)
        self.tag_info_pub = self.create_publisher(AprilTagInfo, "PosInfo", 1)
        self.pubPoint = self.create_publisher(ArmJoint, "TargetAngle", 1)

        # ROS2 subscribers (message synchronization)
        self.depth_image_sub  = Subscriber(self, Image,
"/camera/color/image_raw", qos_profile=1)
        self.rgb_image_sub = Subscriber(self, Image, "/camera/depth/image_raw",
qos_profile=1)
        self.TimeSynchronizer =
ApproximateTimeSynchronizer([self.depth_image_sub,
self.rgb_image_sub],queue_size=10,slop=0.5)
        self.TimeSynchronizer.registerCallback(self.TagDetect)

        # ROS2 other subscribers
        self.grasp_status_sub = self.create_subscription(Bool, 'grasp_done',
self.GraspStatusCallback, 1)
        self.sub_targetID = self.create_subscription(Int8, "TargetId",
self.GetTargetIDCallback, 1)

        # Initialize tools
        self.rgb_bridge = CvBridge()
        self.depth_bridge = CvBridge()
        self.pubPos_flag = False
        self.done_flag = True

        self.set_height = 40.0
        self.set_dist = 0.0
```

```
        self.detect_flag = False

        # AprilTag detector (configuration remains unchanged)
        self.at_detector = Detector(
            searchpath=['apriltags'],
            families='tag36h11',
            nthreads=8,
            quad_decimate=2.0,
            quad_sigma=0.0,
            refine_edges=1,
            decode_sharpening=0.25,
            debug=0
        )
        self.target_id = 0
        self.pr_time = time.time()
        self.cnt = 0
        self.Center_x_list = []
        self.Center_y_list = []
        self.search_joints = [90.0,150.0,12.0,20.0,90.0,30.0]

        self.CurEndPos =
[-0.006,0.116261662208,0.0911289015753,-1.04719,-0.0,0.0]
        self.camera_info_K = [477.57421875, 0.0, 319.3820495605469, 0.0,
477.55718994140625, 238.64108276367188, 0.0, 0.0, 1.0]
        self.EndToCamMat =
np.array([[1.00000000e+00,0.00000000e+00,0.00000000e+00,0.00000000e+00],
                                        [0.00000000e+00,7.96326711e-04,9.99999683e-
01,-9.90000000e-02],
                                        [0.00000000e+00,-9.99999683e-
01,7.96326711e-04,4.90000000e-02],

[0.00000000e+00,0.00000000e+00,0.00000000e+00,1.00000000e+00]])
```

Main image processing function TagDetect

```
def TagDetect(self,color_frame,depth_frame):
    #rgb_image
    # Receive the color image topic message and convert the message data into
image data
    rgb_image = self.rgb_bridge.imgmsg_to_cv2(color_frame,'rgb8')
    result_image = np.copy(rgb_image)
    #depth_image
    # Receive the depth image topic message and convert the message data into
image data
    depth_image = self.depth_bridge.imgmsg_to_cv2(depth_frame, encoding[1])
    frame = cv.resize(depth_image, (640, 480))
    depth_image_info = frame.astype(np.float32)
    # Call the detect function, passing in parameters,
    '''
    cv2.cvtColor(rgb_image, cv2.COLOR_RGB2GRAY): Converts the RGB image to a
grayscale image for tag detection.
    False: Indicates not to estimate the pose of the tag.
    None: Indicates that no camera parameters are provided, possibly only for
simple detection.
    0.025: May be the set tag size (unit is usually meters) to help the detection
algorithm determine the size of the tag.
```

```
    Returns a detection result, including the position, ID, and bounding box of
each tag.
    '''
    tags = self.at_detector.detect(cv2.cvtColor(rgb_image, cv2.COLOR_RGB2GRAY),
False, None, 0.025)
    # Sort the tags in tags, this is not a necessary step
    tags = sorted(tags, key=lambda tag: tag.tag_id) # It seems that the output
is already sorted in ascending order, so manual sorting is not required
    # Call the draw_tags function to draw information related to the recognized
machine code on the color image, including corner points, center point, and id
value
    draw_tags(result_image, tags, corners_color=(0, 0, 255), center_color=(0,
255, 0))
    key = cv2.waitKey(10)
    # Define the length of self.Center_x_list and self.Center_y_list
    self.Center_x_list = list(range(len(tags)))
    self.Center_y_list = list(range(len(tags)))
    # Wait for keyboard input, 32 means space bar is pressed. After pressing,
change the value of self.pubPos_flag to indicate that machine code related
information can be published
    if key == 32:
        self.pubPos_flag = True
    # Judge the length of tags. If it is greater than 0, it means that a machine
code has been detected and the flag for completing the grasping of the machine
code is set
    if len(tags) > 0 and self.done_flag == True:
        # Iterate through the machine codes
        for i in range(len(tags)):
            # The center xy values of the machine code are stored in the
Center_x_list and Center_y_list lists
            center_x, center_y = tags[i].center
            self.Center_x_list[i] = center_x
            self.Center_y_list[i] = center_y
            cx = center_x
            cy = center_y
            # Calculate the depth value of the center coordinate
            cz = depth_image_info[int(cy),int(cx)]/1000
            # Call the compute_heigh function to calculate the height of the
machine code. The input parameters are the center coordinates of the machine code
and the depth value of the center point. It returns a position list, where
pose[2] represents the z value, which is the height value
            pose = self.compute_heigh(cx,cy,cz)
            # Magnify the height value and convert the unit to millimeters
            heigh_detect = round(pose[2],4)*1000
            heigh = 'heigh: ' + str(heigh_detect) + 'mm'
            # Calculate the distance of the machine code from the base
coordinate system, magnify the value, and convert the unit to millimeters
            dist_detect = math.sqrt(pose[1] ** 2 + pose[0]** 2)
            dist_detect = round(dist_detect,3)*1000
            dist = 'dist: ' + str(dist_detect) + 'mm'
            # Use opencv to draw the height and distance values on the color
image
            cv.putText(result_image, heigh, (int(cx)+5, int(cy)-15),
cv.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
            cv.putText(result_image, dist, (int(cx)+5, int(cy)+15),
cv.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
            # If the detected machine code height is greater than the set height
threshold, self.pubPos_flag is True, and the set height threshold is not 0
```

```
            if heigh_detect>=self.set_height and self.set_height!=0 and
self.pubPos_flag == True:
                print("self.set_height: ",self.set_height)
                print("heigh_detect: ",heigh_detect)
                # Change the value of self.detect_flag. True means that a machine
code higher than the set threshold has been recognized
                self.detect_flag  = True
                # Assign values to the message data. The id is the machine code
id, x and y are the center values of the machine code, and z is the depth value
of the center point. Here, it is scaled down by a factor of 1000, and the unit is
meters
                tag = AprilTagInfo()
                tag.id = tags[i].tag_id
                tag.x = self.Center_x_list[i]
                tag.y = self.Center_y_list[i]
                tag.z = depth_image_info[int(tag.y),int(tag.x)]/1000
                # If the depth information is not 0, it means the data is valid,
and then publish the machine code information message
                if tag.z!=0 :
                    self.tag_info_pub.publish(tag)
                    self.pubPos_flag = False
                    self.done_flag = False
                else:
                    print("Invalid distance.")
        # If self.detect_flag is False, it means that no machine code higher
than the height threshold has been recognized, the set height threshold is not 0,
and publishing of machine code messages is enabled. If these three conditions are
met, it means that no machine code higher than the height threshold has been
recognized.
        if self.detect_flag != True and  self.set_height!=0 and
self.pubPos_flag==True:
            print("----------------------------------------")
            self.set_height!=0
            # The robotic arm performs a "head shaking" action group
            self.shake()
            #time.sleep(2)
            # Return to the gesture recognition posture to prepare for the next
gesture recognition
            self.pub_arm(self.search_joints)
            # Publish the grasp completion topic so that the gesture recognition
node program can publish the result of the next gesture recognition
            grasp_done = Bool()
            grasp_done.data = True
            self.pubGraspStatus.publish(grasp_done)
        self.pubPos_flag = False
# If no machine code is recognized after pressing the space bar, the robotic arm
will also perform the "head shaking" action group and then return to the gesture
recognition posture
    elif self.pubPos_flag == True and len(tags) == 0:
        self.shake()
        self.pub_arm(self.search_joints)
        grasp_done = Bool()
        grasp_done.data = True
        self.pubGraspStatus.publish(grasp_done)
    result_image = cv2.cvtColor(result_image, cv2.COLOR_RGB2BGR)
    cv2.imshow("result_image", result_image)
    key = cv2.waitKey(1)
```

Gesture recognition result callback function GetTargetIDCallback

```
def GetTargetIDCallback(self,msg):
    print("msg.data: ",msg.data)
    #Calculate distance threshold unit is millimeters mm, minimum is 160mm,
maximum is 200mm
    self.set_dist = 150 + msg.data*10
    #Calculate height threshold unit is millimeters mm, minimum is 40mm, maximum
is 80mm
    self.set_height = 30 + msg.data*10
    print("self.set_height: ",self.set_height)
    #After receiving message, change robotic arm posture to present AprilTag
recognition posture
    self.pub_arm(self.init_joints)
```

4.3. grasp.py

You can refer to section 4.2 [grasp.py] in tutorial [3D Space Sorting and Grasping\1. AprilTag ID Sorting].