CMSC 477 Project 2: Grab ‘n Go

(USING 3 LATE DAYS)

Shanay Kadakia
Eric Kim
Trevor Moore

Introduction

This project aimed to swap two stacks of legos, whose original positions are marked via colored paper pads. The size, shape, and color of the legos as well as the color of the pads could be chosen to preference. Past the basic requirements of the project, robots should also be able to recover from stacks or pads being displaced by a person during the run. The approach used here utilizes OpenCV in conjunction with the camera onboard the Robomaster EP Core for stack and pad detection rather than a custom trained YOLO model.

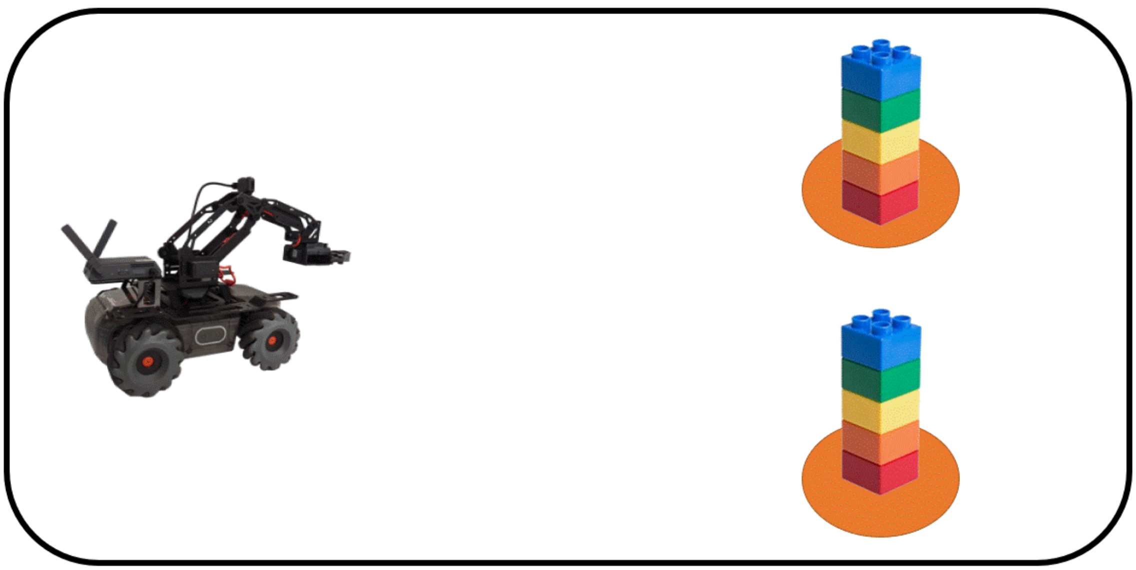
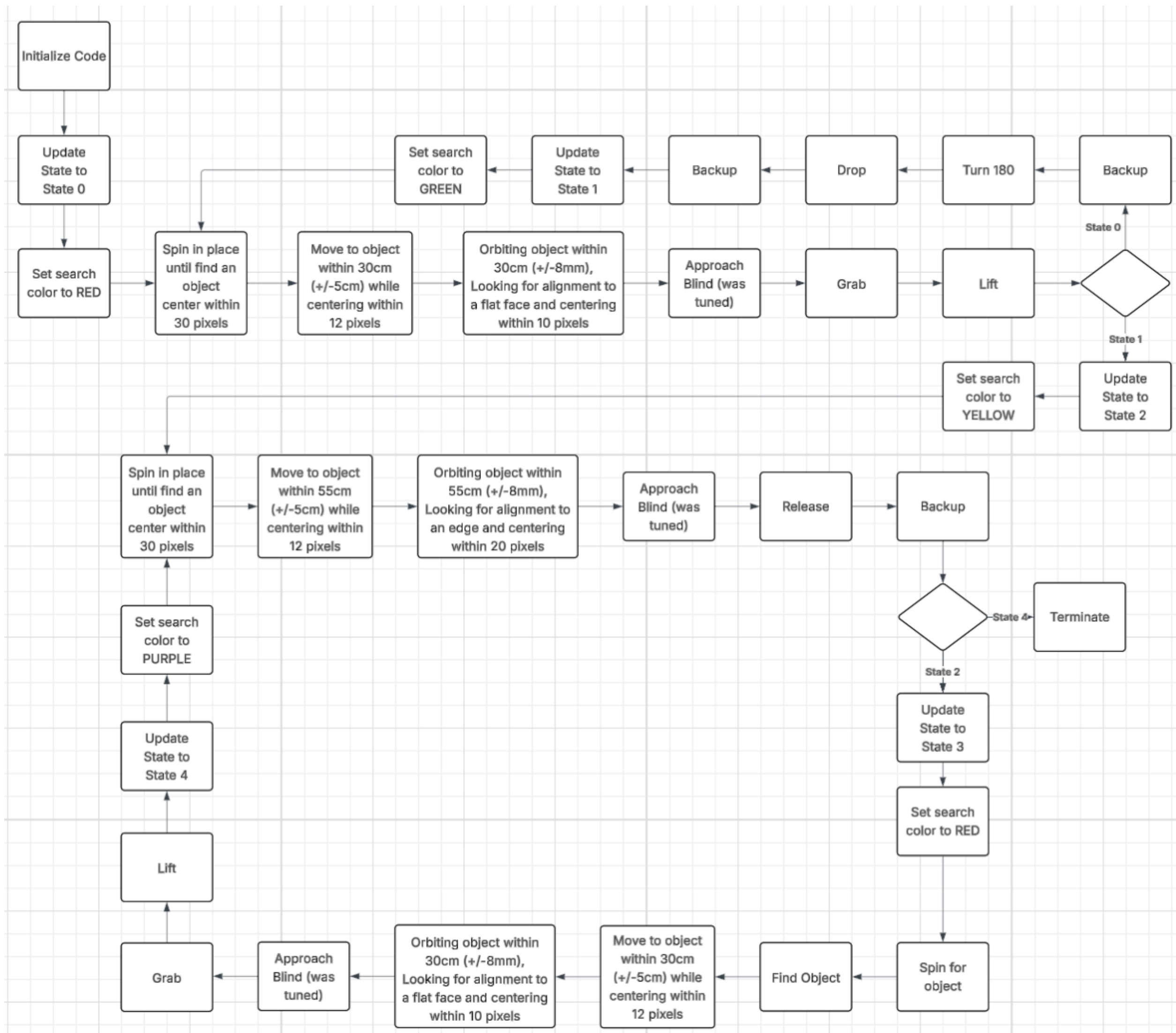


Figure 1: Animation of the project's basic goal

Our setup for the project includes two uniform colored towers, which were red and green, and two pads, purple and orange. The tower will be referred to as bricks in this report. The green brick starts at the purple pad and the red brick starts at the orange pad.

Block Diagram



Demonstration

Uninterrupted Run: https://youtu.be/tFGHIHNiYes?feature=shared

This video includes the robot swapping the two stacks without any external block or pad displacement. First, the robot sweeps to find the red block, navigates to the block, and aligns itself with one of the flat faces of the block. It then picks it up and moves it out of the way to prepare for the other block's arrival. It sweeps once again, but this time for a green block, inching closer, aligning itself, and picking it up. It then sweeps for the yellow pad, carries the green block over, and places it down. After that, it relocates the red block, picks it up, relocates the purple pad and drops it onto it.

Interrupted Run: https://youtu.be/TbZnD5IDfP4?feature=shared

This video includes the robot swapping the two stacks while both bricks and a pad are moved when the robot is not tracking them. The algorithm used is no different than the one used for the uninterrupted run, the only difference being both bricks and the purple pad being moved once the robot has interacted with each of them once. Since the robot relocates all pads and bricks before moving to them, it is unaffected by the bricks/pads moving. At 2:04 the robot orbiting the block until it aligns itself with a flat face can be seen.

Methodology

States

In order to swap the positions of the block, the robot needs to keep track of the tasks it needs to accomplish. These tasks are specific "states" for the robot, and there are five of them, not including the default state. For the five main states, the robot is always searching for a specific object based on the color mask for that object. The default state is just the robot rotating in place with a selected mask, and it will keep rotating until it finds the correctly colored object. State 0 is to find the red brick, and it will transition to the next state once it has moved the red brick a distance away from the pad. State 1 is to find the green brick, and it will proceed to the next state once it grasps and lifts the brick. State 2 is to find the yellow pad, and it will transition after lowering and releasing the gripper. State 3 is to find the red brick again, transitioning once it has gripped and lifted the red block. The final state is finding the purple pad, reaching the end once it has lowered and released the gripper.

Detection

A major part of this project was being able to detect the bricks and paper mats in order to locate their position relative to the robot. To do this we opted to use color masking because of the distinct colors of the lego bricks and paper. The general control flow was to first apply a predetermined color mask to the entire camera frame. Then generate a frame of white pixels where the object was and make everything else black. By applying Canny edge detection to this augmented image we could isolate the brick or paper. With the edges of the object we could determine the object's location in the frame. To do this we found the centroid of the shape made by the edges. This allowed us to center the robot with the object. We could also get distance to the robot since the area the edges encompassed changed relative to the distance between the robot and object. For this we calibrated an exponential line of best fit by gathering some data. Lastly, we could also determine alignment from the edges. We defined alignment as the robot facing a flat face of the brick or a straight edge of the paper. To determine whether we are aligned or not we took advantage of the fact that if a brick or paper was fully in frame the very bottom pixels would form a straight line if we were aligned and a “v” if they were not. To detect whether it was a “v” or straight line we looked at the pixels to the left and right of the very bottom edge pixel and defined a window to look within at the distance. If any white pixels were found in the window we knew we were aligned.

Motion

When the robot is greater than 30 cm from a brick or 60 cm from a pad, it will use a Proportional control loop to approach the object in “coarse” movements, using strictly translational movements. When the robot is within the mentioned distances, it performs “orbiting” movement, tracking the object in an orbit to get a better angle to grasp it or drop a block. Through thorough testing, the robot was found to improperly grip objects when the angle was too challenging. It would also improperly judge the distance of the pad at off-angles. The decision to orbit was to give more confidence to the next set of movements. The robot concludes each state with “fine” movements, which are pre-programmed velocity commands that are timing based. Once the robot gets close enough to a brick or pad, the object becomes obscured to where the detection algorithm has a hard time judging distances. It was more appropriate to turn detection off and instead move in a direct path toward the object.

Results

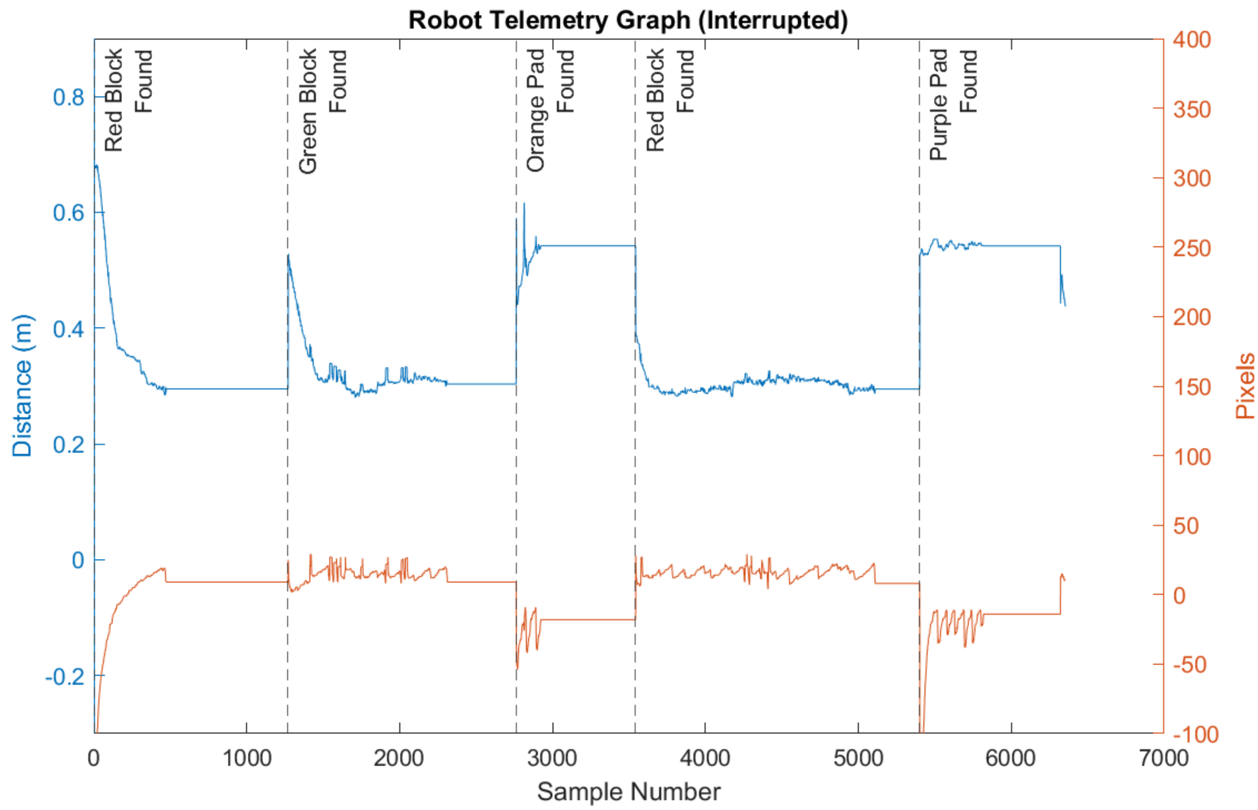


Figure 2: Distance and Pixel Measurements across Sample Number

Telemetry from the robot was gathered as it was running as shown in the videos above. The plots below show both the lateral distance (in pixels) and depth (in meters), between the robot and the centroid of the block. For lateral distance, it describes the centroid's distance from the center of the camera frame, which is 320 by 180 pixels from the corner of the camera. Figure 2 shows both distance and pixel measurements overlaid together, with the sample number as the x axis. The red block is immediately identified and the robot approaches it as the first action on this plot. The events are labeled as the plot entry number increases, but finer, pre-programmed actions are not labeled in this plot because the distances are not tracked.

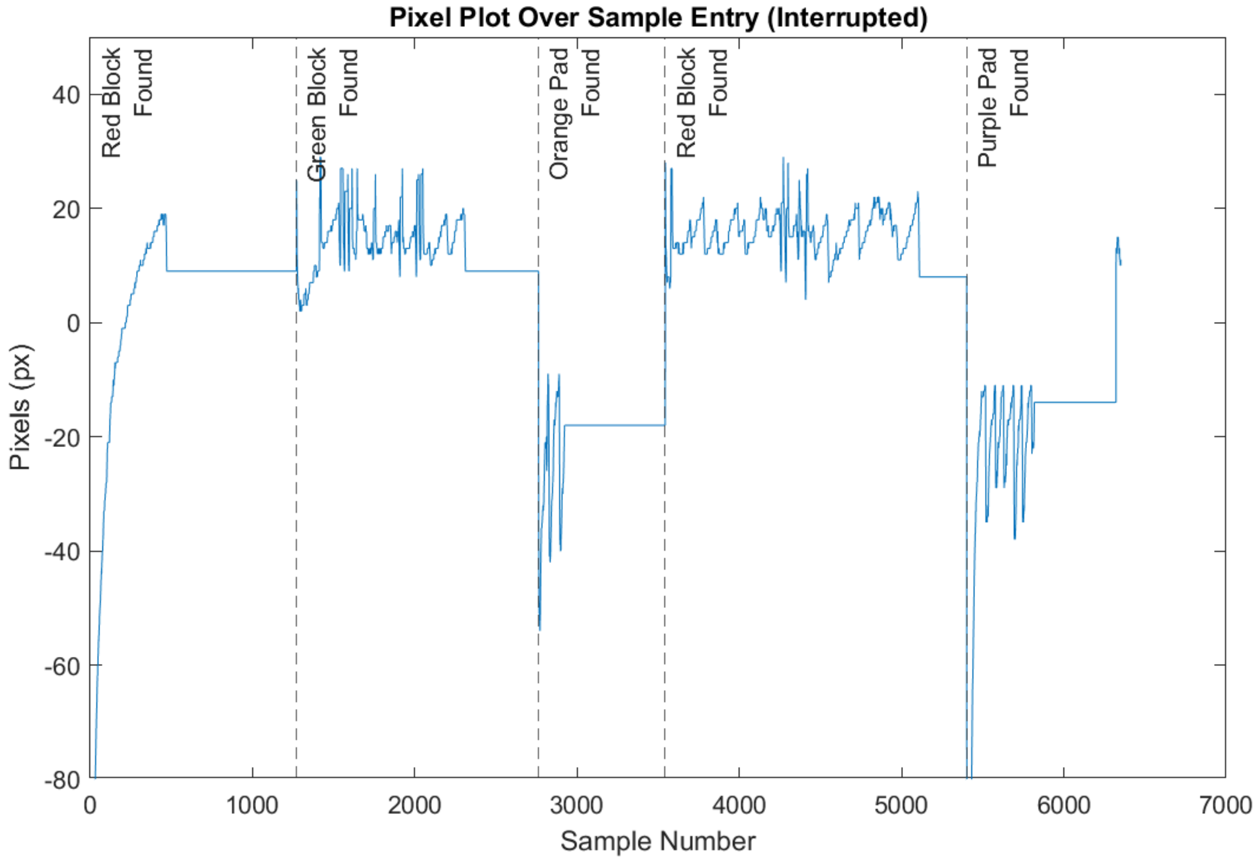


Figure 3: Pixel Width of Block Centroid from Center of Camera

Figure 3 shows the pixel plot versus sample entry. The pixel data for the camera is more limited, as the robot only captures 640 by 360 pixel images, which turns out to be adjustable. Although erratic, the pixel count tries to converge into the tolerance of acceptable pixel width for the robot's camera. Even if the pixel width is within acceptable limits, the robot checks only changes state if the distance from the block and its pixel width are within acceptable limits simultaneously.

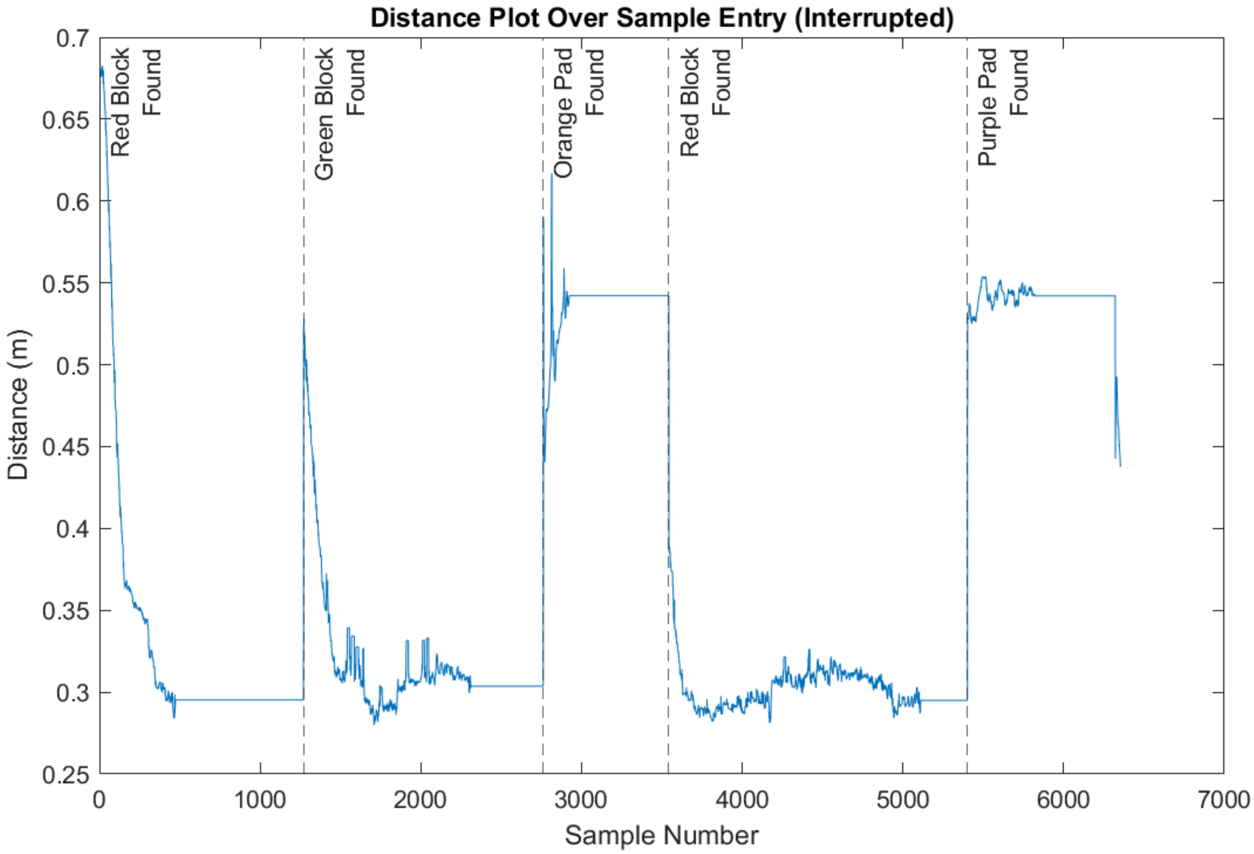


Figure 4: Depth from Camera to Object

Figure 4 shows a different plot with the distance of the camera to the block over the sample entry. Compared to the pixel plot, you can see finer data points and changes in distance due to the higher precision of distance we can get from extrapolating distance from area of the block. Each time a new object is found, there is a large change in distance over a short period of time, followed by minute movements to change its distance once the robot gets close to its programmed offset.

Conclusion

All objectives were accomplished for this project, but there are still some points of improvement to the current approach. It was possible to enhance the resolution of the camera, which was not known until after the robot runs were filmed. If the resolution was higher, better data for detection could have been collected for more accurate centroid positions, especially for fine movements when trying to grasp the block. In addition, a velocity limit would have been helpful to prevent the robot from making sudden and fast movements when tracking a far away object. Also, the orbiting movements to reorient the robot were not smooth turns and instead coarse and jittery. The smoother motion would allow for easier tracking and faster reorientation with the object.

Code Listing

detect.py

```
import cv2
import numpy as np

class Detect:
    def __init__(self):
        self.mask = None
        self.FRAME_CENTER_X = None
        self.FRAME_CENTER_Y = None
        self.object_center = None

        # Color presets (HUE: 0, 180) (SAT: 0, 255) (VAL: 0, 255)
        # self.BRICK_GREEN = [[57, 51, 0], [77, 255, 255]]
        self.BRICK_GREEN = [[59, 102, 54], [83, 255, 255]]
        self.BRICK_RED = [[-10, 120, 0], [10, 255, 255]]

        self.PAPER_PURPLE = [[123, 73, 125], [147, 255, 255]]
        # self.PAPER_ORANGE = [[5, 100, 200], [25, 255, 255]]
        self.PAPER_ORANGE = [[26, 158, 239], [31, 255, 255]]

        # Constants
        self.LOWER = 0
        self.UPPER = 1
        self.HUE = 0
        self.SAT = 1
        self.VAL = 2

        # Moving Average Content
        self.AVG_ORIENTATION = [False] * 101
        self.IDX_ORIENTATION = 0

### Isolating the Brick ###
    # Convert BGR image to HSV
    def BGRtoHSV(self, img):
        return cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
```

```
    # Create a mask with the lower and upper thresholds
    def mask_image(self, img, color):
        lower = color[self.LOWER]
        upper = color[self.UPPER]

        if lower[self.HUE] >= 0:
            mask = cv2.inRange(img, np.array(lower), np.array(upper))
        else:
            # -HUE to 0
            lower1 = np.array([179 + lower[self.HUE], lower[self.SAT],
lower[self.VAL]])
            upper1 = np.array([179, upper[self.SAT], upper[self.VAL]])

            # 0 to HUE
            lower2 = np.array([0, lower[self.SAT], lower[self.VAL]])
            upper2 = np.array([upper[self.HUE], upper[self.SAT],
upper[self.VAL]])

            # Combined Mask from -HUE to HUE
            mask = cv2.bitwise_or(cv2.inRange(img, lower1, upper1),
cv2.inRange(img, lower2, upper2))

        return mask

    def crop_image(self, img):
        mask = np.zeros((img.shape[0], img.shape[1]), dtype="uint8")

        pts = np.array([[0, img.shape[0]], [img.shape[1], img.shape[0]],
[img.shape[1], int(img.shape[0]/2)], [0, int(img.shape[0]/2)]], dtype=np.int32)
        cv2.fillConvexPoly(mask, pts, 255)

        masked = cv2.bitwise_and(img, img, mask=mask)

        return masked

    # Take a frame and apply color mask to find object
    def detect_object(self, frame, color):
        # Convert frame to HSV
        hsv = self.BGRtoHSV(frame)
```

```
        # Mask frame
        self.mask = self.mask_image(hsv, color)
        self.mask = cv2.medianBlur(self.mask, 7)

        if self.FRAME_CENTER_X is None:
            self.FRAME_CENTER_X = len(frame[0])/2
            self.FRAME_CENTER_Y = len(frame)/2

        return self.mask

    # Take in found object and find its edges
    def edges(self, object):

        edges = cv2.Canny(object, threshold1=50, threshold2=150)

        # Fill in holes
        kernel = np.ones((5, 5), np.uint8)
        edges_closed = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, kernel)

        return edges_closed

    # Remove all non brick detections
    def isolate_brick(self, edge_img):
        # ⚠️ Use raw edges (must be single-channel)
        contours, _ = cv2.findContours(edge_img, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

        # Create a black image with same shape as input edge image (converted to
BGR)
        height, width = edge_img.shape[:2]
        output_img = np.zeros((height, width, 1), dtype=np.uint8)

        if contours:
            # Find the largest contour by area
            largest_contour = max(contours, key=cv2.contourArea)

            # Draw the largest contour in green
            cv2.drawContours(output_img, [largest_contour], -1, (255), 2)
```

```
        return output_img


### Getting the center of the brick ###
    # Given edges find the center of the object
    def center(self, edges):
        white_pixels = np.column_stack(np.where(edges == 255))

        if white_pixels.size == 0:
            return None

        cyx = np.mean(white_pixels, axis=0).astype(int)

        cy, cx = cyx[0], cyx[1]

        self.object_center = (cx, cy)

        return (cx, cy)

        # Draws the center of the object on the frame

    def draw_center(self, edges):
        center = self.center(edges)

        if len(edges.shape) == 2 or edges.shape[2] == 1:
            edges_bgr = cv2.cvtColor(edges, cv2.COLOR_GRAY2BGR)
        else:
            edges_bgr = edges.copy()


        if center is not None:
            cv2.circle(edges_bgr, center, 5, (0, 0, 255), -1)

        return edges_bgr


### Getting distance to brick ###
    # Given edges find the left and right sides of the object
```

```python
def sides(self, edges, center_x, angle_thresh=75):
    lines = cv2.HoughLinesP(edges, rho=1, theta=np.pi / 180,
                            threshold=40, minLineLength=30, maxLineGap=40)

    if lines is None:
        return []

    left_group = []
    right_group = []

    # Separate vertical lines into left and right of the center
    for line in lines:
        x1, y1, x2, y2 = line[0]
        angle = np.degrees(np.arctan2(y2 - y1, x2 - x1))
        if abs(angle) > angle_thresh:
            x_avg = (x1 + x2) / 2
            if x_avg < center_x:
                left_group.append((x1, y1, x2, y2))
            else:
                right_group.append((x1, y1, x2, y2))

    # Helper function to combine group
    def combine_group(group, flip=False):
        if not group:
            return None
        x1_vals = [line[0] for line in group]
        x2_vals = [line[2] for line in group]
        y_vals = [line[1] for line in group] + [line[3] for line in group]

        x1_avg = int(np.mean(x1_vals))
        x2_avg = int(np.mean(x2_vals))
        y_min = min(y_vals)
        y_max = max(y_vals)

        if flip:
            x1_avg, x2_avg = x2_avg, x1_avg  # Flip for left line

        return (x1_avg, y_max, x2_avg, y_min)
```

```
        # Combine both sides
        combined = []
        left_line = combine_group(left_group, flip=True)
        right_line = combine_group(right_group, flip=False)

        if left_line:
            combined.append(left_line)
        if right_line:
            combined.append(right_line)

        return combined

    # Find length of a line (x1, y1) (x2, y2)
    def line_length(self, lines):
        lengths = []
        for x1, y1, x2, y2 in lines:
            length = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
            lengths.append(length)
        return lengths

    # Find distance from robot to object based on side lengths
    def distance_lines(self, lineLengths):
        avg = np.mean(np.array(lineLengths))
        # print(avg)
        distance = 2.156*np.exp(-0.01828*avg)
        return distance

    # Find distance from robot to object based on the area of the detected zone
    def distance_area_far(self, brick):
        contour, _ = cv2.findContours(brick, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
        if contour:
            A = cv2.contourArea(contour[0], oriented=False)
            output = 0.8506*np.exp(-2.033*10**-4*A)
        return output

    def distance_area_near(self, brick):
        contour, _ = cv2.findContours(brick, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
```

```
        if contour:
            A = cv2.contourArea(contour[0], oriented=False)
            output = -1.845*10**-4*A+1.401
        return output

    # Draws the center of the object on the frame
    def draw_center(self, edges):
        center = self.center(edges)

        if len(edges.shape) == 2 or edges.shape[2] == 1:
            edges_bgr = cv2.cvtColor(edges, cv2.COLOR_GRAY2BGR)
        else:
            edges_bgr = edges.copy()


### Getting orientation of the brick relative to the robot ###
    def orientation(self, img, left_thresh=15, right_thresh=15,
up_down_window=2):
        img_color = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)

        # Get coordinates of all pixels with grayscale > 200
        bright_pixels = np.column_stack(np.where(img > 200))

        if bright_pixels.size == 0:
            return False

        # Step 1: Find the pixel closest to the bottom (max Y)
        bottommost_pixel = bright_pixels[np.argmax(bright_pixels[:, 0])]
        y = bottommost_pixel[0] - 1

        # Step 2: Get all x-positions at that y-level that are also bright
        bright_xs_at_y = np.where(img[y, :] > 200)[0]

        if bright_xs_at_y.size == 0:
            return False  # Safety check

        # Step 3: Take the average x
        x = int(np.mean(bright_xs_at_y))
```

```
        height, width = img.shape[:2]

        # Vertical range
        y_start = max(y - up_down_window, 0)
        y_end = min(y + up_down_window, height - 1)

        # Left region (1x3)
        x_left = max(x - left_thresh, 0)
        left_region = img[y_start:y_end + 1, x_left:x_left + 1]

        # Right region (1x3)
        x_right = min(x + right_thresh, width - 1)
        right_region = img[y_start:y_end + 1, x_right:x_right + 1]

        # Draw red pixels on left region
        for yy in range(y_start, y_end + 1):
            img_color[yy, x_left] = (0, 0, 255)  # Red

        # Draw red pixels on right region
        for yy in range(y_start, y_end + 1):
            img_color[yy, x_right] = (0, 0, 255)  # Red

        l = np.any(left_region > 200)
        r = np.any(right_region > 200)
        has_bright_neighbor = l or r

        # Display debug image
        # cv2.imshow("Orientation Check", img_color)

        return has_bright_neighbor

    def orientation_avg(self, img, left_thresh=20, right_thresh=20,
up_down_window=1):
        orient = self.orientation(img, left_thresh, right_thresh, up_down_window)

        self.AVG_ORIENTATION[self.IDX_ORIENTATION] = orient
        self.IDX_ORIENTATION += 1
        if self.IDX_ORIENTATION >= len(self.AVG_ORIENTATION):
            self.IDX_ORIENTATION = 0
```

```python
        threshold = 2/3
        return sum(self.AVG_ORIENTATION) >= threshold * len(self.AVG_ORIENTATION)



# hsv(134.26, 77.71%, 61.57%)

def main():
    cam = cv2.VideoCapture(0)

    detector = Detect()

    while True:
        _, frame = cam.read()

        object = detector.detect_object(frame, detector.BRICK_GREEN)

        edges = detector.edges(object)

        brick = detector.isolate_brick(edges)

        aligned = detector.orientation_avg(brick)

        print(f'AVG: {aligned}')

        # Display the captured frame
        #cv2.imshow('Camera', edges)

        # Press 'q' to exit the loop
        if cv2.waitKey(1) == ord('q'):
            break

    # Release the capture and writer objects
    cam.release()
    cv2.destroyAllWindows()

if __name__ == "__main__":
```

```
    main()
```

motion.py

```
import pupil_apriltags
import cv2
import numpy as np
import time
import traceback
from queue import Empty
import time
import robomaster
from robomaster import robot
from robomaster import camera
import threading
import sys

class AprilTagDetector: # Given
    def __init__(self, family="tag36h11", threads=2, marker_size_m=0.16):
        K = np.array([[314, 0, 320], [0, 314, 180], [0, 0, 1]]) # Camera focal
length and center pixel
        marker_size_m = 0.153 # Size of the AprilTag in meters
        self.camera_params = [K[0, 0], K[1, 1], K[0, 2], K[1, 2]]
        self.marker_size_m = marker_size_m
        self.detector = pupil_apriltags.Detector(family, threads)

    def find_tags(self, frame_gray):
        detections = self.detector.detect(frame_gray, estimate_tag_pose=True,
            camera_params=self.camera_params, tag_size=self.marker_size_m)
        return detections

    def draw_detections(self, frame, detections): # Given
        for detection in detections:
            pts = detection.corners.reshape((-1, 1, 2)).astype(np.int32)

            frame = cv2.polylines(frame, [pts], isClosed=True, color=(0, 0, 255),
thickness=2)

            top_left = tuple(pts[0][0])  # First corner
            top_right = tuple(pts[1][0])  # Second corner
```

```
            bottom_right = tuple(pts[2][0])  # Third corner
            bottom_left = tuple(pts[3][0])  # Fourth corner
            cv2.line(frame, top_left, bottom_right, color=(0, 0, 255),
thickness=2)
            cv2.line(frame, top_right, bottom_left, color=(0, 0, 255),
thickness=2)

    def get_pose_apriltag_in_camera_frame(self, detection):
        R_ca = detection.pose_R
        t_ca = detection.pose_t

        roll = np.arctan2(R_ca[1][0],R_ca[0][0])
        yaw = np.arctan2(-R_ca[2][0],np.sqrt(R_ca[2][1]**2+R_ca[2][2]**2))
        pitch = np.arctan2(R_ca[2][1],R_ca[2][2])
        const = 1 #180/np.pi

        rotation = [const*roll, const*yaw, const*pitch]

        t_ca = t_ca.flatten()
        t_ca[2] = t_ca[2]*np.cos(pitch)
        t_ca[0] = t_ca[0]*np.cos(yaw)

        return t_ca, rotation

class motion:
    def __init__(self):
        robomaster.config.ROBOT_IP_STR = "192.168.50.121"

        # Robot Init
        self.ep_robot = robot.Robot()
        self.ep_robot.initialize(conn_type="sta", sn="3JKCH8800100UB")
        self.ep_arm = self.ep_robot.robotic_arm
        self.ep_gripper = self.ep_robot.gripper
        self.ep_chassis = self.ep_robot.chassis

        # Camera Init
        self.ep_camera = self.ep_robot.camera

        # State Booleans
```

```python
        self.isLost = True
        self.isGrip = False

    def gripper_close(self, power=100):
        self.ep_gripper.close(power)
        time.sleep(1)
        self.ep_gripper.pause()

    def gripper_open(self, power=100):
        self.ep_gripper.open(power)
        time.sleep(1)
        self.ep_gripper.pause()

    def arm_forward(self):
        self.ep_arm.move(x=50, y=0).wait_for_completed()

    def arm_backward(self):
        self.ep_arm.move(x=-50, y=0).wait_for_completed()

    def arm_lower(self):
        self.ep_arm.move(x=0, y=-20).wait_for_completed()

    def arm_raise(self):
        self.ep_arm.move(x=0, y=20).wait_for_completed()

    def arm_position_reader(self, sub_info):
        pos_x, pos_y = sub_info
        print("Robotic Arm: pos x:{0}, pos y:{1}".format(pos_x, pos_y))

    # lower grab raise
    def lgr(self):
        # self.arm_lower()
        time.sleep(1)
        self.gripper_close()
        time.sleep(1)
        self.gripper_close()
        time.sleep(1)
        self.arm_raise()
```

```
    # lower release raise
    def lrr(self):
        self.arm_lower()
        time.sleep(1)
        self.gripper_open()
        time.sleep(1)
        self.gripper_open()
        time.sleep(1)
        # self.arm_raise()

    def lg(self):
        self.arm_lower()

    # move backwards -> rotate 90 -> drop block -> move backwards a little -> 
rotate -90
    def move_away(self):
        self.ep_chassis.drive_speed(x=-0.3, y=0, z=0, timeout=5)
        time.sleep(2)
        self.ep_chassis.drive_speed(x=0, y=0, z=90, timeout=5)
        time.sleep(2)
        self.ep_chassis.drive_speed(x=0, y=0, z=0, timeout=5)
        time.sleep(1)
        self.lrr()
        self.ep_chassis.drive_speed(x=-0.1, y=0, z=0, timeout=5)
        time.sleep(1)
        self.ep_chassis.drive_speed(x=0, y=0, z=-90, timeout=5)
        time.sleep(2)
        self.ep_chassis.drive_speed(x=0, y=0, z=0, timeout=5)

    def scan(self):
        self.ep_chassis.drive_speed(x=0, y=0, z=30, timeout = 0.05)

    def move_to_coarse(self, TPose, isOrbiting = False, isReversed = False, Px = 
0.6, Py = 0.005, offsetX = 0.3, offsetY = 0, velz = 0):
        # AprilTag Parameters
        # Px = 2, Py = Px, Pz = 300, offsetX = 0.6, offsetY = 0

        # In robot: x = forward/back, y = left/right
        # In cmaera: x = left/right, y = forward/back
```

```
        errorX = TPose[1]-offsetX
        errorY = TPose[0]-offsetY

        velx = Px*(errorX)
        vely = Py*(errorY)

        self.ep_chassis.drive_speed(x=velx, y=vely, z=0, timeout = 0.02)
        return errorX, errorY

    def move_orbit(self, TPose, isOrbiting = False, isReversed = False, Px = 0.6,
Py = 0.005, Pz = 10, offsetX = 0.3, offsetY = 0, velz = 20):
        # AprilTag Parameters
        # Px = 2, Py = Px, Pz = 300, offsetX = 0.6, offsetY = 0

        # In robot: x = forward/back, y = left/right
        # In cmaera: x = left/right, y = forward/back

        errorX = TPose[1]-offsetX
        errorY = TPose[0]-offsetY

        velx = Px*errorX
        vely = Py*errorY
        velz = -Pz*errorY+velz

        self.ep_chassis.drive_speed(x=velx, y=vely, z=velz, timeout = 0.02)
        return errorX, errorY

    def move_block(self):
        '''
        For the initial Green block
        '''
        self.ep_chassis.drive_speed(x=0.1, y=0, z=0, timeout=10)
        time.sleep(3.1)
        self.ep_chassis.drive_speed(x=0, y=0, z=0, timeout=0.02)
        # self.ep_chassis.move(x=0.2, y=0, z=0, xy_speed = 1)
        self.lgr()
        time.sleep(1)
        self.move_away()
```

```python
        time.sleep(1)
        self.isGrip = False
        sys.exit()

    def move_to_block(self):
        '''
        For the second green and only red block
        '''
        self.ep_chassis.drive_speed(x=0.1, y=0, z=0, timeout=10)
        time.sleep(3.1)
        self.ep_chassis.drive_speed(x=0, y=0, z=0, timeout=0.02)
        # self.ep_chassis.move(x=0.2, y=0, z=0, xy_speed = 1)
        time.sleep(1)
        self.gripper_close()
        time.sleep(1)
        self.gripper_close()
        time.sleep(1)
        self.ep_arm.move(x=0, y=10).wait_for_completed()
        time.sleep(1)
        self.isGrip = False
        sys.exit()

    def move_to_pad(self):
        '''
        For placing block on pad
        '''
        self.ep_chassis.drive_speed(x=0.1, y=0, z=0, timeout=10)
        time.sleep(7.5)
        self.ep_chassis.drive_speed(x=0, y=0, z=0, timeout=0.02)
        time.sleep(1)
        # self.ep_chassis.move(x=0.2, y=0, z=0, xy_speed = 1)
        self.lrr()
        time.sleep(1)
        self.ep_chassis.drive_speed(x=-0.3, y=0, z=0, timeout=5)
        time.sleep(2)
        self.isGrip = False
        sys.exit()

    def orbit(self):
```

```
        # theta = 0
        velz = -10
        vely = 0.008

        self.ep_chassis.drive_speed(x=0, y=vely, z=velz, timeout = 0.02)
```

execution.py

```
import pupil_apriltags
import cv2
import numpy as np
import time
import traceback
from queue import Empty
import time
import robomaster
from robomaster import robot
from robomaster import camera
import os

from motion import motion
from motion import AprilTagDetector
from detect import Detect
import threading

# Perception Functions

def updateImage(img, objType):
    '''
    Used to update the images from the detector with masks and edges
    Returns the masked brick object
    '''
    mask = detector.detect_object(img, objType)
    crop_mask = detector.crop_image(mask)
    edges = detector.edges(crop_mask)
    brick = detector.isolate_brick(edges)
    center = detector.center(brick)
    brick_disp = detector.draw_center(brick)
```

```python
    return brick, center, brick_disp

def updateDistances(img, center_obj):
    '''
    Gets the distance of block from centroid
    Returns the positions (x = pixels, y = meters)
    '''
    # Get the pixel that the brick centroid is
    center_x, center_y = center_obj
    pos_x = center_x-detector.FRAME_CENTER_X
    # Get distance to brick
    pos_y = detector.distance_area_far(img)

    return (pos_x, pos_y)

def blockFound(img, center_obj, rotTol=30):
    if center_obj:
        pos = updateDistances(img, center_obj)
        if pos[0] < rotTol:
            return True
    return False

# System Functions:

def shutdown():
    '''
    Sequence for stopping the robot
    '''
    robo.ep_arm.unsub_position()
    print('Waiting for robomaster shutdown')
    robo.ep_camera.stop_video_stream()
    robo.ep_robot.close()

def initVars():
    errorX = 10
    errorY = 10

    return errorX, errorY
```

```
def open_new_file(filepath):
    '''
    Saves content to a new file, avoiding overwriting existing files.

    Args:
        filepath: The desired file path (including filename).
        content: The content to write to the file.
    '''

    _, ext = os.path.splitext(filepath)
    count = 1
    while os.path.exists(filepath):
        filepath = f"data_{count}{ext}"
        count += 1

    return filepath

def save_data(file, content):
    try:
        file.write(content)
    except Exception as e:
        print(f"An error occurred: {e}")

# Movement Functions

def canOrbit(x, y, alignment, x_tol=12, y_tol=0.05):
    '''
    Checks for the conditions to orbit around the object
    '''
    if abs(x) < x_tol and abs(y) < y_tol and not(alignment):
        return True
    return False

def canGrab(x, y, brickMask, x_tol=10, y_tol=0.008, state = None):
    alignment = detector.orientation_avg(brickMask, 15, 15)

    if state == 2 or state == 4:
        x_tol = 20.0
        y_tol = 0.008
```

```
        # alignment = True

    if abs(x) < x_tol and abs(y) < y_tol and (alignment):
        return True
    return False

def move_approach(brickMask, center_obj, errorX, errorY, state = None):
    pos = updateDistances(brickMask, center_obj)
    isAligned = detector.orientation_avg(brickMask, 15, 15)
    isOrbiting = canOrbit(errorX, errorY, isAligned)

    if state == 2 or state == 4:
        offset_dist = 0.55
        ignoreOrbit = False
    else:
        offset_dist = 0.3
        ignoreOrbit = False

    # Main Movement Logic
    if isOrbiting and not(ignoreOrbit):
        errorY, errorX = robo.move_orbit(pos, Px = 0.4, Py = 0.003)
    else:
        errorY, errorX = robo.move_to_coarse(pos, False, offsetX = offset_dist,
Px = 0.4, Py = 0.003)

    return errorX, errorY, pos[0], pos[1], isAligned

# def check_grab(brickMask, alignCount, alignMax = 10):
#     go_grab = False
#     isAligned = detector.orientation_avg(brickMask, 15, 15)

#     if isAligned:
#         alignCount += 1

#     if alignCount > alignMax:
#         go_grab = True

#     return go_grab, alignCount
```

```python
def move_grab(state):
    robo.isGrip = True
    ind = state

    if ind == 0:
        gripThread = threading.Thread(target=robo.move_block)
        gripThread.start()
    elif ind == 1:
        gripThread = threading.Thread(target=robo.move_to_block)
        gripThread.start()
    elif ind == 2:
        gripThread = threading.Thread(target=robo.move_to_pad)
        gripThread.start()
    elif ind == 3:
        gripThread = threading.Thread(target=robo.move_to_block)
        gripThread.start()
    elif ind == 4:
        gripThread = threading.Thread(target=robo.move_to_pad)
        gripThread.start()

def search_block():
    robo.ep_chassis.drive_speed(x=0, y=0, z=30, timeout = 0.05)

def state_change(state):
    '''
    The state is constructed of the following tasks:
    0 - Find Red
    1 - Find Green
    2 - Find Pad
    3 - Find Red 2
    4 - Find Pad 2
    '''


    ind = state.index(1)
    if ind < len(state):
        state[ind+1] = state[ind]
        state[ind] = 0

    return state
```

```
def color_switch(state):
    ind = state

    if ind == 0:
        return detector.BRICK_RED
    elif ind == 1:
        return detector.BRICK_GREEN
    elif ind == 2:
        return detector.PAPER_ORANGE
    elif ind == 3:
        return detector.BRICK_RED
    elif ind == 4:
        return detector.PAPER_PURPLE
    return detector.PAPER_PURPLE

# Test Functions

def test1_color():
    '''
    This is just project 0 in this project. Except it chases a brick
    '''
    robo.ep_camera.start_video_stream(display=False,
resolution=camera.STREAM_360P)

    # Initialize our variables
    errorX = 10 # Just so we don't trigger the conditional early, set arbitrarily
to 10
    errorY = 10
    count = 0
    tol = 5
    dir = False
    isAligned = True

    while True:
        try:
            img = robo.ep_camera.read_cv2_image(strategy="newest", timeout=0.5)
        except Empty:
            time.sleep(0.001)
```

```
            continue

        brick, center = updateImage(img, detector.BRICK_GREEN)

        if center:
            errorX, errorY = move_approach(brick, center, errorX, errorY)

        # Display the captured frame
        cv2.imshow('Camera', brick)

        if cv2.waitKey(1) == ord('q'):
            break

def test2_color():
    '''
    Rotates in place until a block is found (with the matching color mask)
    Uses a P control loop to approach the block, uses a pre-programmed
    fine movement state machine to approach the block until it's grabbed.
    '''
    robo.ep_camera.start_video_stream(display=False,
resolution=camera.STREAM_360P)
    robo.gripper_open()
    robo.gripper_open()
    errorX, errorY = initVars()
    state = [1]

    while True:
        try:
            img = robo.ep_camera.read_cv2_image(strategy="newest", timeout=0.5)
        except Empty:
            time.sleep(0.001)
            continue

        brick, center = updateImage(img, detector.BRICK_GREEN)

        # Top most conditional checks if the robot is still in thread
(move_to_fine)
        if not(robo.isGrip):
            if blockFound(brick, center):
```

```
            if canGrab(errorX, errorY, brick):
                move_grab(state)
            else:
                errorX, errorY = move_approach(brick, center, errorX, errorY)
        else:
            search_block()
            errorX, errorY = initVars()

    # Display the captured frame
    cv2.imshow('Camera', brick) # brick for area to distance

    if cv2.waitKey(1) == ord('q'):
        break

def test3_color():
    '''
    Picks up red block and goes to pad in a pre-programmed sequence

    Rotates in place until its target is found and then it places it away from
    its inital spot.

    Switches from a coarse motion, P-loop based, to a fine motion, pre-programmed
    state machine.
    '''
    # Detector Init
    detector = Detect()
    robo.ep_camera.start_video_stream(display=False,
resolution=camera.STREAM_360P)

    timeStart = time.time()
    errorX = 10
    errorY = 10
    xTol = 10 # Pixels
    yTol = 0.01 # Meters
    rotTol = 30

    distList = []
    xList = []
    isAligned = False
```

```python
isOrbiting = False

findRed = True
pickRed = False
findPad = False

while True:
    try:
        img = robo.ep_camera.read_cv2_image(strategy="newest", timeout=0.5)
    except Empty:
        time.sleep(0.001)
        continue

    mask = detector.detect_object(img, detector.BRICK_RED)
    if findPad:
        mask = detector.detect_object(img, detector.PAPER_ORANGE)
    edges = detector.edges(mask)
    center = detector.center(edges)

    brick = detector.isolate_brick(edges)

    if not(robo.isGrip):
        if center:
            # Output tower center pixel
            center_x, center_y = center

            pos_x = center_x-detector.FRAME_CENTER_X
            # Get position on X and average it
            xList.insert(0, pos_x)
            if len(xList) > 5:
                xList.pop()

            # This is the final average that it takes
            pos_x = sum(xList)/5

            pos_y = detector.distance_area_far(brick)
            distList.insert(0,pos_y)
            if len(distList) > 5:
                distList.pop()
```

```
                # This is the final average that it takes
                pos_y = sum(distList)/5

                isAligned = detector.orientation(brick)

                print(f"Alignment: {isAligned}, Orbit: {isOrbiting}, Pad:
{findPad}, Red: {findRed}, Distance: {pos_y}")

                if abs(pos_x) < rotTol and not(pos_y != pos_y):
                    pos = [pos_x, 0, pos_y]
                    rot = [0, 0, 0]
                    errorY, errorX = robo.move_to_coarse(pos, rot, False)
                    print("here")
                    isOrbiting = False

                    if abs(errorX) <= xTol and abs(errorY) <= yTol:
                        if isAligned:
                            robo.isGrip = True

                            if findRed:
                                findRed = False
                                pickRed = True
                                findPad = True

                            gripThread =
threading.Thread(target=robo.move_to_fine2)
                            gripThread.start()
                        else:
                            errorY, errorX = robo.move_to_coarse(pos, rot, True)
                            isOrbiting = True
                else:
                    robo.ep_chassis.drive_speed(x=0, y=0, z=30, timeout = 0.05)

            else:
                robo.ep_chassis.drive_speed(x=0, y=0, z=30, timeout = 0.05)

        # Display the captured frame
        cv2.imshow('Camera', brick)
```

```
        if cv2.waitKey(1) == ord('q'):
            break

def test3b_color():
    '''
    Picks up blocks in a pre-programmed sequence, (red to green or green to red).

    Rotates in place until its target is found and then it places it away from
    its inital spot.

    Switches from a coarse motion, P-loop based, to a fine motion, pre-programmed
    state machine.
    '''
    robo.ep_camera.start_video_stream(display=False,
resolution=camera.STREAM_360P)
    robo.gripper_open()
    robo.gripper_open()
    # robo.lgr()
    errorX, errorY = initVars()
    state = 0
    x = 0
    y = 0
    alignment = False

    path = open_new_file("Project 2")
    with open(path, 'w') as data:
        while True:
            timeStart = time.time()
            try:
                img = robo.ep_camera.read_cv2_image(strategy="newest",
timeout=0.5)
            except Empty:
                time.sleep(0.001)
                continue

            color = color_switch(state)
            brick, center, display = updateImage(img, color)
```

```
            # print(color)

            # Top most conditional checks if the robot is still in thread
(move_to_fine)
            if not(robo.isGrip):
                if blockFound(brick, center):
                    if canGrab(errorX, errorY, brick, state=state):
                        move_grab(state)
                        state += 1
                        errorX, errorY = initVars()
                    else:
                        errorX, errorY, x, y, alignment = move_approach(brick,
center, errorX, errorY, state)
                else:
                    search_block()
                    errorX, errorY = initVars()

            # Telemetry from Robot
            print(f"isAligned: {alignment}, State: {state}, Center X: {x},
Distance: {y}")
            save_data(data, f"{time.time() - timeStart}\n")
            # print(time.time() - timeStart)

            # Display the captured frame
            cv2.imshow('Camera', brick)

            if cv2.waitKey(1) == ord('q'):
                break

def detectTest():
    color = detector.PAPER_ORANGE
    robo.ep_camera.start_video_stream(display=False,
resolution=camera.STREAM_360P)

    while True:
        try:
            frame = robo.ep_camera.read_cv2_image(strategy="newest", timeout=0.5)
        except Empty:
            time.sleep(0.001)
```

```python
            continue

        # brick, center, display = updateImage(frame, color)
        # pos = updateDistances(brick, center)

        # print(f"Distance, {pos[1]}")

        # Display the captured frame
        cv2.imshow('Camera', frame)

        # Press 'q' to exit the loop
        if cv2.waitKey(1) == ord('q'):
            break

if __name__ == "__main__":
    # Robot Init
    robo = motion()
    detector = Detect()

    try:
        test3b_color()
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(traceback.format_exc())
    finally:
        shutdown()
```