Assignment 3

สมาชิกกลุ่ม LOCALHOST

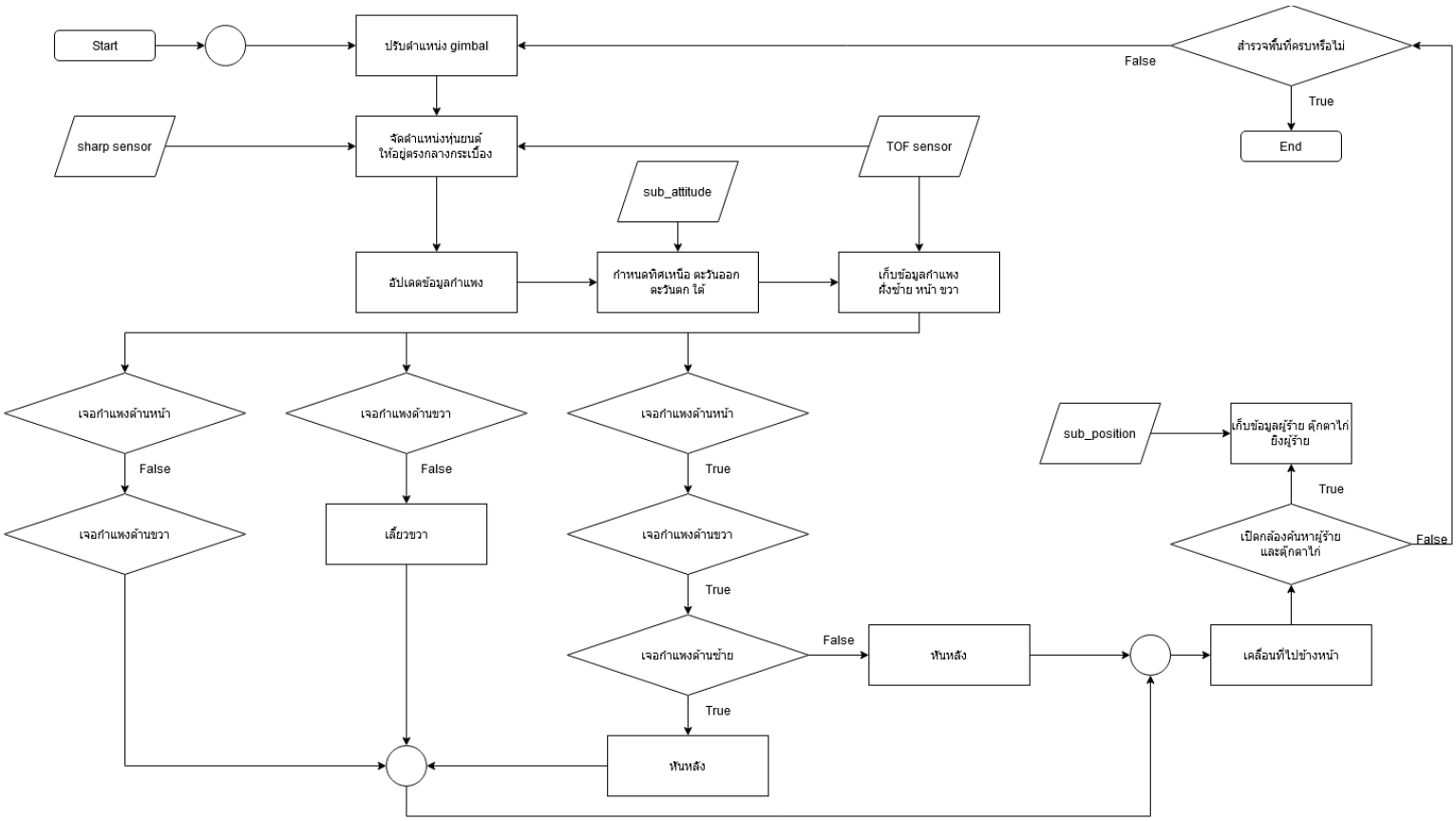
1. 6610110425 คีตศิลป์ คงสี
2. 6610110034 คุณานนต์ หนูแสง
3. 6610110327 สิรวิชญ์ น้อยผา
4. 6610110341 สุธินันท์ รองพล

บทนำ (Introduction)

งาน Assignment 3 ครั้งนี้เป็นการช่วยตัวประกันในเขาวงกตขนาด 6*6 (กระเบื้องในห้อง R300 1 แผ่นมีขนาด 60 เซนติเมตร) การที่จะบรรลุเป้าหมายนั้นจะเริ่มต้นให้หุ่น Robomaster เดินสำรวจเขาวงกตและช่วยตัวประกันด้วยการยิง ผู้ร้ายจะแสดงโดยอะคริลิครูปเป้าคนสูงประมาณ 15-20 ซม. กว้างประมาณ 5-10 เซนติเมตร ตัวประกันแสดงโดยไก่สีเหลือง

Logic & algorithm

ผู้รับผิดชอบ: นายสุธินันท์ รองพล 6610110341



รูปที่ 1 Flowchart Logic การทำงานของหุ่นยนต์

อธิบายการทำงาน

1.หุ่นยนต์ทำการปรับตำแหน่ง gimbal (recenter)

2.จัดตำแหน่งหุ่นยนต์ให้อยู่ตำแหน่งกลางกระเบื้อง

3.อัปเดตข้อมูลกำแพงและพิกัดการเคลื่อนที่

4.กำทิศทางเริ่มต้นของหุ่น ทิศเหนือ ทิศใต้ ทิศตะวันออก ทิศตะวันตก

5.ใช้ TOF ตรวจสอบกำแพงด้านซ้าย ด้านหน้า ด้านหลัง โดยให้ gimbal แต่ละฝั่ง

- หากเจอกำแพงด้านไหนจะให้ค่าเป็น True และให้ค่า False ในเส้นทางที่ไปได้

6.นำค่าที่ได้มาเข้าเงื่อนไข เพื่อทำการเคลื่อนที่หุ่น

- ไม่เจอกำแพงด้านหน้า และ ไม่เจอกำแพงด้านขวา

  ให้เคลื่อนที่ตรงไป

- ไม่เจอกำแพงด้านขวา

  ให้เลี้ยวขวา

- เจอกำแพงทั้งด้านซ้าย ด้านหน้า ด้านขวา

  หันหลังกลับ

- เจอกำแพงด้านด้านหน้า และ เจอกำแพงด้านขวา

  ให้เลี้ยวซ้าย

6.เคลื่อนที่ไปข้างหน้า 1 ครั้ง โปรแกรมทำการเปิดกล้องเพื่อหา ผู้ร้าย และ ไก่

-ในกรณีที่เจอผู้ร้ายและไก่จะทำการยิงผู้ร้ายและเก็บตำแหน่งข้อมูล และทำการเคลื่อนที่ไปข้างหน้าระยะทางประมาณ 57 cm.

7.ตรวจสอบหุ่นยนต์ว่าสำรวจพื้นที่ครบแล้วหรือไม่

- True: เสร็จสิ้นภารกิจ
- False: เริ่มต้นกระบวนการใหม่ตั้งแต่ข้อ 1

ปัญหาที่เจอในขณะรันโปรแกรม

การติดต่อระหว่างหุ่นยนต์และคอมพิวเตอร์ ไม่มีการตอบกลับภายในเวลาที่กำหนด

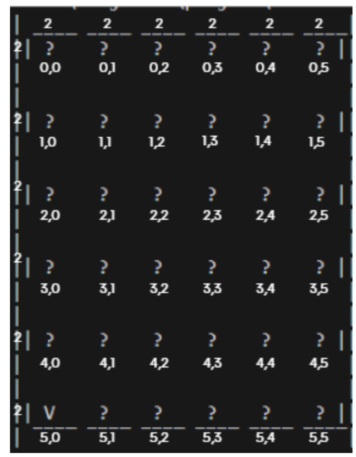
```
2024-10-08 22:02:51,390 ERROR client.py:163 Client: send_sync_msg wait msg receiver:2202,
cmdset:0x3f, cmdid:0xf0 timeout!
```

Localization & map

ผู้รับผิดชอบ: 6610110034 คุณานนต์ หนูแสง

การระบุตำแหน่งของหุ่นยนต์จะเริ่มจากการกำหนด grid แทนพื้นที่ในเขาวงกตซึ่งจะใช้ list ในการเก็บข้อมูลซึ่งจะแบ่งได้เป็น [North(front), West(left), East(right), South(back), Visited, Chick(C), Antagonist(A)] ตามลำดับ

```
grid = [
[[2,2,0,0,0,0,0],[2,0,0,0,0,0,0],[2,0,0,0,0,0,0],[2,0,0,0,0,0,0],[2,0,0,0,0,0,0],[2,0,2,0,0,0,0]],
[[0,2,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,2,0,0,0,0]],
[[0,2,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,2,0,0,0,0]],
[[0,2,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,2,0,0,0,0]],
[[0,2,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,2,0,0,0,0]],
[[0,2,0,2,0,0,0],[0,0,0,2,0,0,0],[0,0,0,2,0,0,0],[0,0,0,2,0,0,0],[0,0,0,2,0,0,0],[0,0,2,2,0,0,0]]
]
```



รูปที่ 2 ภาพแสดง grid ของเขาวงกตขนาด 6 x 6

- เลข **2** แทนตำแหน่งที่มีกำแพงในเขาวงกต
- เลข **1** หมายถึงตำแหน่งที่หุ่นยนต์เคยไปแล้ว หรือสามารถใช้แทนการตรวจพบลูกไก่หรือผู้ร้ายในกรณีการสำรวจ
- เลข **0** หมายถึงตำแหน่งที่หุ่นยนต์ยังไม่เคยไปสำรวจ หรือยังไม่พบตำแหน่งของลูกไก่และผู้ร้าย

ในการอัปเดตแผนที่ (grid) ของหุ่นยนต์ในเขาวงกต:

- เริ่มจากตำแหน่งปัจจุบันของหุ่นยนต์ (current_position) ซึ่งเป็นพิกัด x, y เช่น (0, 0) หมายถึงจุดเริ่มต้นในเขาวงกต
- ฟังก์ชัน update_wall จะอัปเดตข้อมูลกำแพงของตำแหน่งปัจจุบันตามทิศทางที่หุ่นยนต์กำลังหันอยู่ โดยใช้การบันทึกใน grid[x][y] สำหรับกำแพงแต่ละด้าน:
    - [0] กำแพงด้านหน้า

- [1] กำแพงด้านซ้าย
- [2] กำแพงด้านขวา
- [3] กำแพงด้านหลัง

ตัวอย่างเช่น:

1. เมื่อหุ่นยนต์เคลื่อนที่ไปข้างหน้า (move_forward):
    - จะบันทึกว่ามีกำแพงด้านขวาที่ตำแหน่งปัจจุบัน
    - อัปเดตตำแหน่งให้เลื่อนหน้าไปยังพิกัดใหม่
2. เมื่อหุ่นยนต์เลี้ยวขวา (turn right):
    - จะบันทึกว่าหุ่นยนต์เคยเยี่ยมชมตำแหน่งด้านขวา และอัปเดตตำแหน่งให้เลื่อนไปทางขวา
3. การเลี้ยวกลับ (turn back):
    - บันทึกกำแพงทุกด้านก่อนที่จะอัปเดตตำแหน่งให้เลื่อนไปข้างหลัง
4. การเลี้ยวซ้าย (turn left):
    - บันทึกกำแพงด้านหน้าและขวา และอัปเดตตำแหน่งให้เลื่อนไปทางซ้าย

นอกจากนี้ หุ่นยนต์ยังใช้เซนเซอร์ ToF เพื่อตรวจสอบกำแพงด้านหน้า ซ้าย และขวา ซึ่งจะบันทึกข้อมูลลงในแผนที่ตามทิศทางที่หุ่นยนต์กำลังหันหน้าอยู่โดยใช้ค่ามุม yaw ในการระบุตำแหน่งทิศทาง

การกำหนดทิศทางของหุ่นยนต์ Robomaster ใช้การวัดมุม yaw เพื่อระบุทิศทางที่หุ่นยนต์หันหน้าอยู่:

- ทิศเหนือ (N): เมื่อมุม yaw อยู่ในช่วง -45 ถึง 45 องศา
- ทิศตะวันออก (E): เมื่อมุม yaw อยู่ในช่วง 45 ถึง 135 องศา
- ทิศใต้ (S): เมื่อมุม yaw อยู่ในช่วง 135 ถึง 180 องศา หรือ -135 ถึง -180 องศา
- ทิศตะวันตก (W): เมื่อมุม yaw อยู่ในช่วง -135 ถึง -45 องศา

ข้อมูลทิศทางที่ได้จะถูกใช้ในการตัดสินใจอัปเดตแผนที่ (grid) ซึ่งแต่ละตำแหน่งจะมี 5 ช่องข้อมูล ได้แก่:

1. ข้อมูลว่ามีกำแพงหรือไม่ในแต่ละด้าน (หน้า, ซ้าย, ขวา, หลัง)
2. ข้อมูลว่าหุ่นยนต์เคยเยี่ยมชมตำแหน่งนั้นแล้วหรือไม่

การอัปเดตตำแหน่งและแผนที่ขึ้นอยู่กับสถานะปัจจุบันของหุ่นยนต์ (robo_status_now) และการกระทำ (status_logic) เช่น เคลื่อนที่ไปข้างหน้า หมุนขวา หมุนซ้าย หรือถอยหลัง การกระทำแต่ละอย่างจะส่งผลต่อการปรับตำแหน่งของหุ่นยนต์ใน grid และบันทึกว่าหุ่นยนต์เคยผ่านตำแหน่งใดบ้าง

การใช้ประโยชน์จากตำแหน่งของหุ่นที่เคยมาและตำแหน่งของกำแพงในเขาวงกต

เมื่อหุ่นยนต์สำรวจเขาวงกตและเราเก็บข้อมูลตำแหน่งที่หุ่นเคยไปและตำแหน่งของกำแพงไว้แล้ว เราสามารถลดการใช้เวลาในการตรวจสอบกำแพงด้วยเซนเซอร์ TOF ได้ วิธีการคือสร้างตัวแปรชื่อว่า visit_counts เพื่อบันทึกจำนวนครั้งที่หุ่นยนต์เคยไปยังแต่ละตำแหน่งในเขาวงกต

หากตำแหน่งใดเคยถูกเยี่ยมมากกว่า 2 ครั้งจะไม่ใช้ TOF เซนเซอร์ตรวจสอบกำแพงในทิศทางต่าง ๆ แต่จะอ้างอิงข้อมูลตำแหน่งกำแพงที่ถูกเก็บไว้ในตาราง Grid ที่ถูกอัปเดตตำแหน่งของหุ่นยนต์และตำแหน่งของกำแพงแทน อย่างไรก็ตามหากหุ่นยนต์ไม่เคยไปยังตำแหน่งนั้นมาก่อนเลย หุ่นจะใช้ TOF เซนเซอร์เพื่อตรวจสอบกำแพงทางซ้าย หน้า และขวา โดยตำแหน่งกำแพงที่ถูกตรวจพบจะถูกบันทึกในการเช็คครั้งแรกที่หุ่นยนต์เข้าไปในตำแหน่งนั้น

ตำแหน่งปัจจุบันของหุ่นยนต์จะถูกเก็บไว้ ซึ่งแบ่งออกเป็นแถว (row) และคอลัมน์ (col) ตามตำแหน่งในเขาวงกต นอกจากนี้ยังมีตัวแปร counts ที่บันทึกจำนวนครั้งที่หุ่นยนต์เคยไปยังตำแหน่งนั้นๆ

ตัวอย่างการทำงานของฟังก์ชันเริ่มต้น:

สมมติว่าตำแหน่งเริ่มต้นของหุ่นคือ row = 5 และ col = 2 และหุ่นยนต์หันหน้าไปทางทิศเหนือ (เช็คด้วย status เป็น 'N') เมื่อหุ่นเดินไปข้างหน้า ระบบจะอัปเดตตำแหน่งถัดไปโดยบันทึกว่าหุ่นยนต์เคยไปยังตำแหน่งนั้นแล้ว หลังจากนั้น หุ่นยนต์จะเคลื่อนไปยังตำแหน่งใหม่โดยอัปเดต current_po จาก [5,2] เป็น [4,2] ซึ่งหมายความว่าหุ่นได้ย้ายไปข้างหน้า 1 ช่อง

ฟังก์ชันตรวจสอบตำแหน่งของกำแพง

ทำหน้าที่ตรวจสอบตำแหน่งกำแพงด้านหน้าซ้ายและขวาของหุ่นยนต์ตามทิศทางที่หุ่นกำลังหัน (N, E, S, W) โดยดูจากค่าในตาราง grid ว่ามีกำแพงหรือไม่ (2 หมายถึงมีกำแพง) ซึ่งเกี่ยวข้องกันกับในส่วนของฟังก์ชัน logic จะทำงาน 2 กรณี:

1. ถ้าหุ่นยนต์เคยไปที่ตำแหน่งนั้นน้อยกว่า 2 ครั้ง จะใช้ TOF เซนเซอร์ตรวจสอบกำแพงรอบตัว (ซ้าย, หน้า, ขวา) แล้วบันทึกค่ากำแพงลงใน grid
2. ถ้าหุ่นยนต์เคยไปที่ตำแหน่งนั้นแล้ว 2 ครั้ง จะดึงข้อมูลกำแพงที่บันทึกไว้ใน grid มาใช้แทนการเช็คด้วยเซนเซอร์เพื่อลดการตรวจสอบซ้ำ

หลักการนี้ช่วยให้หุ่นยนต์ทำงานได้เร็วขึ้นเมื่อเคยไปตำแหน่งเดิมแล้ว

การ Localization ไก่ และ ผู้ร้าย และ หา Shortest path

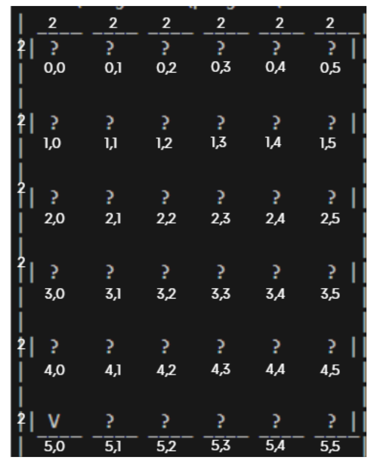
ผู้รับผิดชอบ: 6610110327 สิรวิชญ์ น้อยผา

Localization ไก่ และ ผู้ร้าย

[North(front), West(left), East(right), South(back), Visited, Chick(C), Antagonist(A)] ตามลำดับ

```
grid = [

[[2,2,0,0,0,0,0],[2,0,0,0,0,0,0],[2,0,0,0,0,0,0],[2,0,0,0,0,0,0],[2,0,0,0,0,0,0],[2,0,2,0,0,0,0]],
[[0,2,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,2,0,0,0,0]],
[[0,2,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,2,0,0,0,0]],
[[0,2,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,2,0,0,0,0]],
[[0,2,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,0],[0,0,2,0,0,0,0]],
[[0,2,0,2,0,0,0],[0,0,0,2,0,0,0],[0,0,0,2,0,0,0],[0,0,0,2,0,0,0],[0,0,0,2,0,0,0],[0,0,2,2,0,0,0]]
]
```



จาก ค่าแผนที่ของ grid

- grid[x][y][5] = จะเป็นสถานะของไก่ในตำแหน่งนั้น (1 หมายถึง จะหมายถึงไก่อยู่ในตำแหน่งนั้น)
- grid[x][y][6] = จะเป็นสถานะของผู้ร้ายในตำแหน่งนั้น (1 หมายถึง จะหมายถึงผู้ร้ายอยู่ในตำแหน่งนั้น)

ในการ localize ของไก่และผู้ร้าย

การ localize ของไก่ จะมีการสร้าง list_chick ไว้เพื่อเก็บตำแหน่งของไก่เมื่อ detect เจอเมื่อหุ่นยนต์ detect เจอไก่ ค่าสถานะของตัวแปร found_chick จะเป็น True จากนั้น จะทำการอัพเดท grid[x][y][5] = 1 เพื่ออัพเดทการเจอไก่ แล้วอัพเดทค่า current_position โดยค่า current_position นี้จะเป็นตำแหน่งที่เจอไก่เก็บพิกัดเป็น [x, y] แล้วนำตำแหน่งที่เจอนี้จัดเก็บเข้าไปใน list_chick จากนั้น บันทึกค่า found_chick เป็น False เพื่อใช้ในการตรวจสอบตำแหน่งของไก่ในตำแหน่งต่อไป

การ localize ของผู้ร้าย จะมีการสร้าง list_acrylic ไว้เพื่อเก็บตำแหน่งของผู้ร้ายเมื่อ detect เจอ โดยการเก็บตำแหน่งจะเกิดขึ้นเมื่อหุ่นยนต์ทำการยิงใส่ผู้ร้าย โดยจะมีตัวแปร check เก็บการตรวจจับผู้ร้ายว่าเจอหรือไม่ (ถ้าเจอเป็น True ไม่เจอเป็น False) เมื่อค่า check เป็น True ก็จะทำการอัพเดท grid[x][y][6] = 1 แล้วอัพเดทค่า current_position โดยค่า current_position นี้จะเป็นตำแหน่งที่เจอผู้ร้ายเก็บพิกัดเป็น [x, y] แล้วนำตำแหน่งที่เจอนี้จัดเก็บเข้าไปใน list_acrylic จากนั้น บันทึกค่า check เป็น False เพื่อใช้ในการตรวจสอบตำแหน่งของผู้ร้ายในตำแหน่งต่อไป

Shortest path

ในการหา shortest path ที่จะ detect ไก่ และ ผู้ร้าย จะใช้ข้อมูลตำแหน่งจากการเดินรอบแรกโดยจะ save ไฟล์ออกมาเป็น file csv 3 ไฟล์ คือ list_path จะใช้เก็บทิศทางของหุ่นยนต์ (N, W, E, S) และ เก็บค่าพิกัดที่หุ่นยนต์เดินทางไป (x, y) list_acrylic เก็บตำแหน่งของผู้ร้ายเป็นพิกัด (x, y) และ list_chicken เก็บตำแหน่งของไก่เป็นพิกัด (x ,y) โดยการเก็บพิกัด (x, y) มาจากค่า current_position

ต่อมาก็จะเป็นการโหลดไฟล์ list_travel, list_acrylic และ list_chicken ที่เป็นไฟล์ csv มาใช้ ในการเอาค่าตำแหน่งจากไฟล์เหล่านี้มาใช้ในการหา shortest_path โดยใช้ Dijktra’s algorithm ซ่างเป็นอัลกอริทึมที่ใช้แผนผังเส้นทางที่สั้นที่สุดโดยเริ่มจากโหนดต้นทาง โดยแผนผังต้นทางที่สั้นที่สุดคือต้นไม้ที่เชื่อมต่อโหนดทั้งหมดในกราฟกลับไปยังต้นทางที่มีคุณสมบัติที่ความยาวของเส้นทางจากโหนดนึงไปโหนดนึงน้อยที่สุด เพื่อหาเส้นทางที่สั้นที่สุดจากตำแหน่งเริ่มต้น (start position) ไปยังเป้าหมาย (chickens และ acrylics) ทั้งหมด

ในการใช้ Dijktra’s algorithm จะสร้างฟังก์ชัน dijkstra_all_targets() เพื่อหาทางไปยังเป้าหมายทั้งหมด (chickens และ acrylics) โดยกำหนดพารามิเตอร์เป็น

- **start**: ตำแหน่งเริ่มต้นของหุ่นยนต์
- **chicken_positions, acrylic_positions**: ตำแหน่งของ chickens และ acrylics ที่หุ่นยนต์ต้องค้นหา
- **graph**: โครงสร้างกราฟที่สร้างจากข้อมูลการเดินทาง ซึ่งแต่ละ node จะเชื่อมต่อกับ neighbors ด้วย Manhattan distance

ต่อมาจะสร้าง ฟังก์ชัน reconstruct_path() ใช้ในการสร้างเส้นทางที่ย้อนกลับจากตำแหน่งเป้าหมายสุดท้ายกลับไปยังตำแหน่งเริ่มต้น เพื่อให้ได้เส้นทางการเดินทางที่ถูกต้อง

ฟังก์ชัน create_graph() ใช้สร้างกราฟจากข้อมูลการเดินทางในรูปแบบของ adjacency list โดยคำนวณน้ำหนักระหว่าง node แต่ละตำแหน่งด้วย Manhattan distance ซึ่งฟังก์ชันนี้จะใช้ร่วมกับ dijkstra_all_targets()

เมื่อได้เส้นทางที่สั้นที่สุดก็จะเอาข้อมูลตำแหน่งที่ต้องเคลื่อนที่มาหาทิศทางการเคลื่อนที่เพื่อนำมาใช้ในการสร้าง logic การเดินละการ detect ไก่และผู้ร้าย โดยจะมี ฟังก์ชัน get_directions_from_path() โดยฟังก์ชันนี้จะดึงทิศทาง (direction) ที่หุ่นยนต์ต้องใช้จากข้อมูลการเดินทาง โดยตรวจสอบตำแหน่งที่อยู่ในเส้นทางสั้นที่สุดที่ได้จาก Dijkstra's algorithm

ในส่วนของการเคลื่อนที่ของหุ่นใช้ ทิศตะวันออก (E ; East) ทิศตะวันตก (W ; West) ทิศเหนือ (N ; North) และทิศใต้ (S ; South) เพื่อมาใช้ในการกำหนดเส้นทางการเดินในครั้งต่อไป

ตัวอย่างข้อมูลเส้นทาง

Direction	N	N	N	E	E
Travel	(5, 2)	(4, 2)	(3, 2)	(3, 3)	(3, 4)

จุดเริ่มต้นของหุ่นยนต์คือพิกัด (5, 2) โดยการเดินครั้งแรกหุ่นยนต์จะหันหน้าไปทางทิศเหนือ (N) และจะเดินหน้าต่อไปโดยไม่เปลี่ยนทิศทาง ในทุกการเดินแต่ละครั้ง หุ่นยนต์จะเคลื่อนที่ไปข้างหน้าในระยะประมาณ 57 ซม. พร้อมกับเปิดกล้องเพื่อตรวจสอบผู้ร้ายและไก่

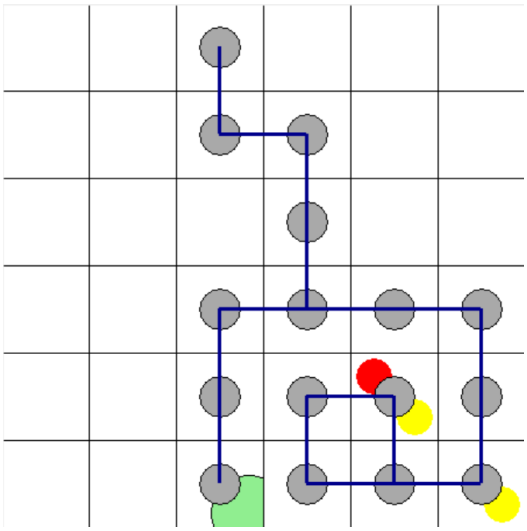
หากมีการเปลี่ยนทิศทาง หุ่นยนต์จะทำตามหลักดังนี้:

ฝั่งซ้ายของทิศเหนือ (N) คือทิศตะวันตก (W) ฝั่งขวาคือทิศตะวันออก (E)

ตัวอย่างเช่น หากการเคลื่อนที่ครั้งถัดไปกำหนดให้หันไปทางทิศตะวันออก (E) หุ่นยนต์จะเลี้ยวขวาและเดินหน้าตรงไปทางทิศนั้น แต่ถ้าทิศถัดไปคือทิศตะวันตก (W) หุ่นยนต์จะเลี้ยวซ้ายและเดินต่อไปข้างหน้า

ทุกครั้งที่มีการเดินหน้า หุ่นยนต์จะหันไปตามทิศที่กำหนดแล้วเดินตรงไป โดยหากต้องเลี้ยว หุ่นยนต์จะเลี้ยวซ้ายหรือขวาตามทิศทางที่ระบุ

ตัวอย่าง เส้นทางการเดิน



รูปที่ 3 ภาพแสดงเส้นทางการเดินของหุ่นยนต์

list_path.csv

```
direction,travel
N,"(5, 2)"
N,"(4, 2)"
N,"(3, 2)"
E,"(3, 3)"
E,"(3, 4)"
E,"(3, 5)"
S,"(4, 5)"
S,"(5, 5)"
W,"(5, 4)"
N,"(4, 4)"
W,"(4, 3)"
S,"(5, 3)"
E,"(5, 4)"
E,"(5, 5)"
N,"(4, 5)"
```

N,"(3, 5)"

W,"(3, 4)"

W,"(3, 3)"

N,"(2, 3)"

N,"(1, 3)"

W,"(1, 2)"

N,"(0, 2)"

list_chicken.csv

chicken

"(5, 5)"

"(4, 4)"

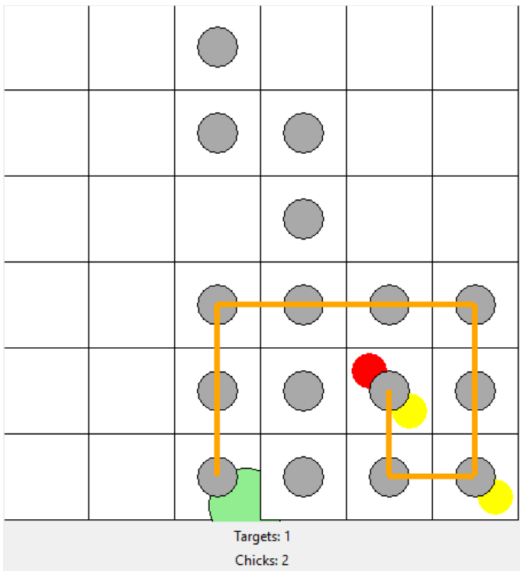
list_acrylic.csv

acrylic

"(4, 4)"

```
Start Position: (5, 2)
Shortest Path: [(5, 2), (4, 2), (3, 2), (3, 3), (3, 4), (3, 5), (4, 5), (5, 5), (5, 4), (4, 4)]
Total Distance to Target: 9
Directions for Shortest Path: ['N', 'N', 'N', 'E', 'E', 'E', 'S', 'S', 'W']
```

รูปที่ 4 ภาพแสดงข้อมูล shortest path

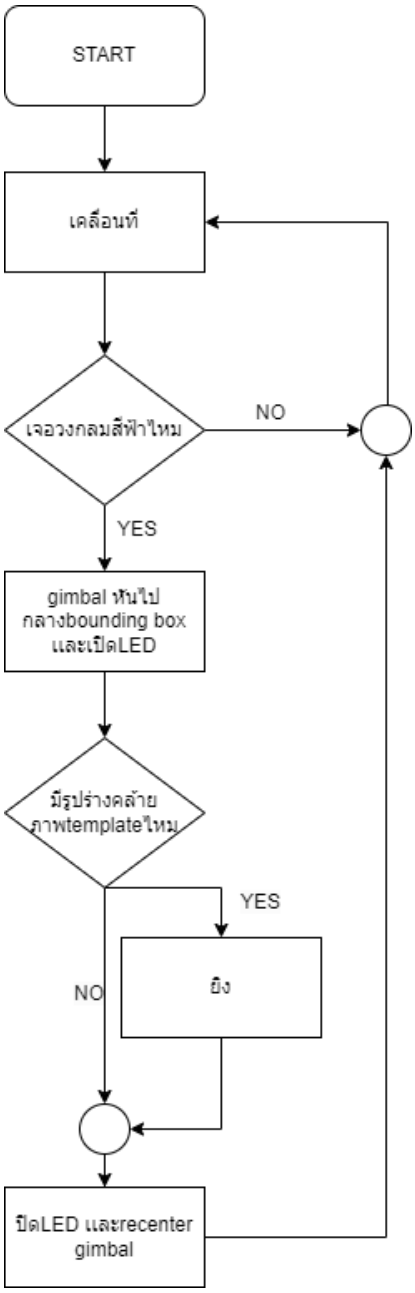


รูปที่ 5 ภาพแสดงเส้นทางการเดินแบบ shortest path

ค้นหาผู้ร้าย(อะคิลิกใสรูปเป้าคน) แล้วยิง

ผู้รับผิดชอบ: 6610110425 นายคีตศิลป์ คงสี

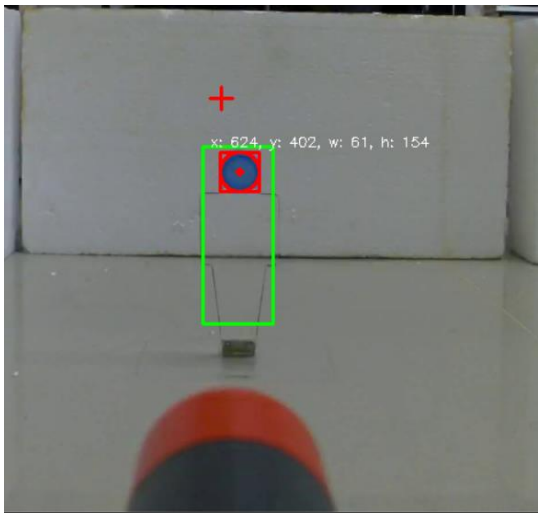
Flowchart



รูปที่ 6 flowchart logicการตรวจจับผู้ร้าย

- เมื่อเคลื่อนจนจบstatus_logicปัจจุบัน
- ให้ตรวจสอบว่าเจอวงกลมสีฟ้าหรือไม่
    - เจอ -> gimbalหันไปกลางbounding box เปิดLED และเทียบว่าเป้าหมายมีรูปร่างคล้ายกับภาพtemplateที่เตรียมไว้ไหม
        - คล้าย -> ยิง
        - ไม่คล้าย -> ไปขั้นตอนต่อไป
    - ไม่เจอ -> ไปขั้นตอนต่อไป
- ปิดLED และrecenter gimbal
- เคลื่อนที่status_logicต่อไป

ตรวจจับหัวผู้ราย(วงกลมสีฟ้า) และวาดbounding box



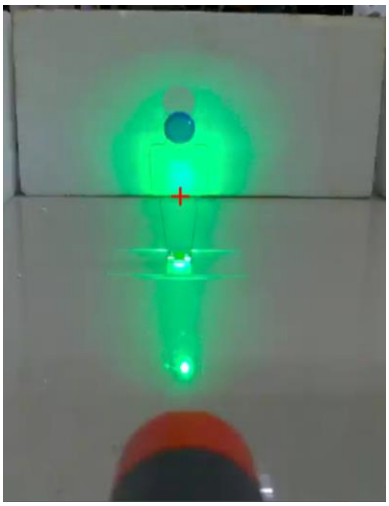
รูปที่ 7 แสดงการตรวจจับวงกลมสีฟ้า และวาดbounding box

ตรวจจับวงกลมสีฟ้า(หัวผู้ร้าย)ในภาพ จากนั้นทำการวาดbounding boxในพื้นที่ที่คาดการณ์ว่ามีตัวคนร้าย เพื่อตรวจสอบในขั้นตอนถัดไป

หลักการทำงาน

- แปลงภาพจากรูปแบบสี BGR เป็น HSV ซึ่งช่วยในการแยกสีฟ้าออกจากสีอื่นๆ ได้ง่ายขึ้น
- กำหนดช่วงค่าของสีฟ้าในระบบสี HSV โดยใช้ค่าต่ำสุดและค่าสูงสุดที่กำหนด ใช้cv2.inRange เพื่อสร้าง maskที่เก็บเฉพาะพื้นที่ที่มีสีฟ้าอยู่ในช่วงที่กำหนด
- ใช้mask เพื่อตัดส่วนที่ไม่ใช่สีฟ้าออกจากภาพ แปลงภาพที่ได้เป็นGrayscale ใช้การเบลอภาพด้วย Gaussian Blur เพื่อลดสัญญาณรบกวนและทำให้การตรวจจับวงกลมมีความแม่นยำมากขึ้น
- ใช้cv2.HoughCircles เพื่อค้นหาวงกลมในภาพที่ผ่านการเบลอแล้ว
- วงกลมที่ตรวจพบ จะตัดส่วนของmaskที่อยู่รอบๆ วงกลมนั้นมาเพื่อตรวจสอบขอบเขต หา contour ของพื้นที่ที่เป็นmask และคำนวณความยืดหยุ่น (perimeter) และพื้นที่ (area) ของ contour นั้น
- คำนวณความกลม (circularity) เพื่อยืนยันว่าคล้ายกับวงกลมจริงๆ หากค่าความกลมเกินเกณฑ์ที่กำหนด (มากกว่า 0.67) จะถือว่าวงกลมนั้นเป็นวงกลมสีฟ้าที่ถูกต้อง
- วาดวงกลมที่ตรวจพบลงบนภาพผลลัพธ์ด้วยสีแดง
- วาดกรอบสี่เหลี่ยมรอบๆ พื้นที่ที่คาดการณ์ว่ามีตัวผู้รายอยู่

หันgimbalไปตรงกลางbounding box และเปิดLED



รูปที่ 8 หันgimbalไปตรงกลางbounding box

ในขั้นตอนก่อนหน้า เราได้วาดbounding box รูปสี่เหลี่ยมในพื้นที่ที่คาดการณ์ว่าจะมีส่วนตัวของอะคิลิกใส โดยในขั้นตอนนี้เราจะ หันgimbalไปตรงกลางbounding boxที่ได้วาดขึ้น

หลักการทำงาน

- ตรวจจับวัตถุและสร้าง Marker:
  - เมื่อวัตถุถูกตรวจพบ จะสร้างMarker ซึ่งเก็บข้อมูลตำแหน่งและขนาดของวัตถุที่ตรวจจับได้ (x,y,w,h)
- การคำนวณerror:
  - คำนวณความคลาดเคลื่อนในแกน x และ y โดยเปรียบเทียบตำแหน่งของจุดศูนย์กลางของวัตถุกับจุดกึ่งกลางของภาพเฟรมที่ได้รับจากกล้อง (center_x, center_y)
  - ค่าความคลาดเคลื่อนในแกน x (err_x) และแกน y (err_y) จะถูกนำมาใช้ในการคำนวณความเร็วในการหันของ gimbal
- ใช้ PID Controller สำหรับ Gimbal:

	K_c	T_I	T_D
P	$K_u/2$		
PI	$K_u/2.2$	$P_u/1.2$	
PID	$K_u/1.7$	$P_u/2$	$P_u/8$

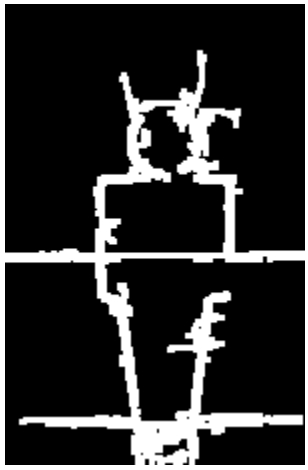
รูปที่ 9 การปรับPID Controller แบบ Ziegler-Nichols closed loop

- หาค่าKu(กำลังขยายของสัดส่วน) โดยใช้ค่าที่ทำให้เกิดการสั่นอย่างต่อเนื่องไม่ลดทอน เมื่อใช้แค่ค่าPu ไม่ใช้I, D และหาค่าPu(คาบของการสั่น)
- นำค่าKu และPuที่ได้มาปรับPID Controller แบบ Ziegler-Nichols closed loop
- ในการคำนวณความเร็วในการหัน (speed_x, speed_y), ค่าerror (err_x, err_y) จะถูกคูณด้วยค่าPID เพื่อกำหนดความเร็วในการหันของแกน x และ y ของ gimbal

- การส่งคำสั่งความเร็วไปยัง Gimbal:
  - ความเร็วที่คำนวณได้ (speed_x, speed_y) จะถูกส่งไปยัง gimbal ผ่านฟังก์ชัน drive_speed ซึ่งจะเป็นการปรับความเร็วในทิศทางแนวตั้ง (pitch) และแนวนอน (yaw) เพื่อให้ gimbal หันไปยังตำแหน่งที่วัตถุอยู่ในจุดกึ่งกลางของภาพ
- การเก็บข้อมูลและปรับปรุงการควบคุม:
  - error เวลา และความเร็วในการหันที่คำนวณได้จะถูกเก็บไว้ในlist เพื่อใช้ในการคำนวณความเร็วในการหันรอบต่อไป
  - มีการตรวจสอบ เพื่อเปิดLED เมื่อหันgimbalไปกลาง bounding box แล้ว

ตรวจรูปร่างของเป้าหมาย

หาขอบของอะคิลิกใส



รูปที่ 10 ตัวอย่างภาพOutput จากการหาขอบของอะคิลิกใส

หลักการทำงาน

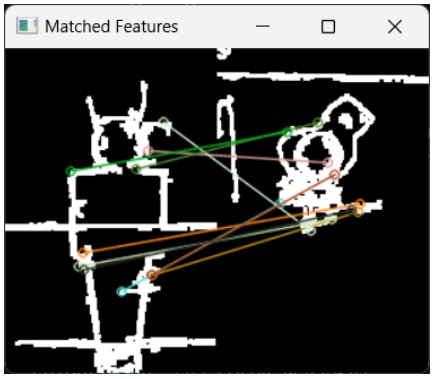
- อ่านภาพ แล้วทำการตัดส่วนของภาพให้เหลือเฉพาะพื้นที่บริเวณกลางภาพ
- แยกภาพสีออกเป็นสามช่องสีหลักคือ สีน้ำเงิน, สีเขียว, และสีแดง โดยใช้cv2.split
- ปรับcontrast ของช่องสีสีน้ำเงินโดยใช้ cv2.convertScaleAbs ซึ่งจะเพิ่มความเข้มของสีตามค่าพารามิเตอร์ alpha และเพิ่มค่าความสว่างตาม beta โดยคำนวณตามสูตร abs(b×α+β) โดย b คือภาพต้นฉบับ
- ใช้ cv2.Canny เพื่อทำการตรวจจับขอบของภาพที่ผ่านการปรับcontrast แล้ว โดยใช้เกณฑ์ในการตรวจจับขอบ ระหว่าง 40 และ 160
- สร้างkernel เป็นเมทริกซ์ของเลข 1 ขนาด(3,3) ซึ่งใช้ในการขยายขอบ แล้วใช้cv2.dilate เพื่อขยายขอบที่ตรวจพบ
- บันทึกภาพที่ผ่านการประมวลผล โดยใช้ cv2.imwrite

จับคู่คุณลักษณะระหว่างภาพต้นแบบ กับภาพที่ต้องการตรวจจับ

หลักการทำงาน

- อ่านภาพต้นแบบ และ ภาพที่ต้องการตรวจจับ
- ใช้cv2.ORB_create() สร้างobject ORB (Oriented FAST and Rotated BRIEF) ซึ่งเป็นalgorithm สำหรับการตรวจจับคุณลักษณะ (keypoints) และการสร้างคำอธิบาย (descriptors) ของภาพ ซึ่งมีความเร็วและประสิทธิภาพสูง เหมาะสำหรับการประยุกต์ใช้งานแบบเรียลไทม์
- ใช้ orb.detectAndCompute() เพื่อหา
    - Keypoints: จุดสำคัญในภาพที่มีความเด่นชัด เช่น มุม, ขอบ, หรือจุดที่มีการเปลี่ยนแปลงของความสว่างอย่างรวดเร็ว
    - Descriptors: ลักษณะเฉพาะของแต่ละ keypoint ซึ่งใช้ในการเปรียบเทียบและจับคู่ keypoints ระหว่างภาพสองภาพ
- ใช้BFMatcher ในการจับคู่ descriptors ระหว่างสองภาพโดยการเปรียบเทียบแต่ละ descriptor ในภาพต้นแบบกับทุก descriptor ในภาพที่ต้องการตรวจจับ โดยมีพารามิเตอร์
    - cv2.NORM_HAMMING: เป็นวิธีการวัดระยะทางระหว่าง descriptors โดยใช้ Hamming Distance ซึ่งเหมาะสมกับ descriptors แบบไบนารี
    - crossCheck=True: เพื่อให้แน่ใจว่าการจับคู่เป็นแบบสองทาง (mutual))
- ใช้ Matcher ที่สร้างขึ้นเพื่อจับคู่ descriptors ระหว่างภาพต้นแบบและภาพที่ต้องการตรวจจับ
- จัดเรียงการจับคู่ตามระยะทางจากน้อยไปมาก ซึ่งหมายความว่าการจับคู่ที่ดีที่สุด (ระยะทางน้อยที่สุด) จะอยู่ตัวแรกของlist
- นับจำนวนการจับคู่ที่มีระยะน้อยกว่า 50 เพื่อใช้เป็นตัววัดความคล้ายกันระหว่างสองภาพ และprint คะแนนความคล้ายออกมา
- ถ้าคะแนนความคล้ายกัน (similarity_score) มากกว่า21 จะเรียกใช้ ep_blaster.fire(times=1) เพื่อยิง

รูปที่ 11 แสดงคะแนนความคล้ายของภาพอะคิลิกใสที่ใช้ทดลอง



รูปที่ 12 แสดงคะแนนความคล้ายของภาพที่ไม่ใช่อะคิลิกใสที่ใช้ทดลอง

หลักการทำงาน ORB (Oriented FAST and Rotated BRIEF)

- Feature Detection: ใช้ FAST (Features from Accelerated Segment Test) ในการตรวจจับจุดเด่น (keypoints) ในภาพ โดยalgorithm จะพิจารณาความเข้มของpixelรอบจุดที่สนใจ และเลือกจุดที่มีความเปลี่ยนแปลงสูง
- Orientation Assignment: ORB คำนวณการหมุนของ keypoints โดยใช้การวิเคราะห์ความเปรียบต่างของpixelรอบ keypoints ซึ่งจะช่วยให้สามารถบันทึกการหมุนของจุดเด่นในภาพได้อย่างแม่นยำ
- Feature Description: ใช้ BRIEF (Binary Robust Independent Elementary Features) ในการสร้างตัวอธิบาย (descriptor) ที่แสดงลักษณะเฉพาะของ keypoints โดยใช้ค่าบิตไบนารีที่สามารถเปรียบเทียบได้ง่าย
- Descriptor Rotation: ORB ปรับ BRIEF descriptor ให้สอดคล้องกับการหมุนของ keypoints ที่ตรวจพบ เพื่อให้ descriptor สามารถทนต่อการเปลี่ยนแปลงในการหมุนได้ดีขึ้น
- Feature Matching: หลังจากที่ได้ descriptor แล้ว จะใช้เทคนิคการเปรียบเทียบตัว descriptor ระหว่างภาพ เพื่อจับคู่ keypoints ที่คล้ายกัน

การตรวจจับตุ๊กตาลูกไก่

ผู้รับผิดชอบ: 6610110425 นายคีตศิลป์ คงสี

- อ่านภาพจากกล้อง แล้วครอบภาพ
  - ep_camera.read_cv2_image() : ดึงภาพจากกล้องของหุ่นยนต์
  - strategy="newest" : ดึงภาพที่ใหม่ที่สุดจากกล้อง
  - timeout=0.5 : รอรับข้อมูลจากกล้องโดยให้เวลาสูงสุด 0.5 วินาที
  - ครอบภาพให้เห็นเพียงบริเวณในเขาวงกต ไม่ให้เห็นกำแพง หรือวัตถุภายนอกเขาวงกตที่มีโทนสีเดียวกัน
- แปลงภาพจาก BGR เป็น HSV เพื่อใช้ในการวิเคราะห์สี
  - ใช้ cv2.cvtColor เพื่อแปลงภาพจาก RGB เป็น HSV (Hue, Saturation, Value)
  - การใช้พื้นที่สี HSV ช่วยให้การตรวจจับสีในภาพทำได้ง่ายและแม่นยำกว่าแบบ RGB เนื่องจากแบบ HSV สามารถแยกแยะความแตกต่างของสีโดยอิงตามเฉดสี (Hue) ได้ดีกว่าการรวมสามแม่สีอย่าง RGB
- กำหนดช่วงสีที่ต้องการตรวจจับ
  - lower_chicken ระบุค่าสี HSV ต่ำสุดที่ต้องการตรวจจับ (Hue: 33, Saturation: 150, Value: 100)
  - upper_chicken ระบุค่าสี HSV สูงสุด (Hue: 38, Saturation: 255, Value: 255)

- สร้าง Mask
  - cv2.inRange() : สร้าง mask ซึ่งเป็นการกรองภาพที่อยู่ในช่วงสีที่กำหนด หากสีในภาพ hsv_frame อยู่ในช่วงที่กำหนด lower_chicken ถึง upper_chicken จะได้ค่าสี 255 (สีขาวใน mask) แต่ถ้าสีไม่ได้อยู่ในช่วงที่กำหนดจะได้ค่าสีเป็น 0 (สีดำใน mask)
- ใช้ cv2.findContours() ในการตรวจหาขอบเขตของวัตถุในภาพ

  - mask_chicken และ mask_bottle: เป็นภาพ Mask ที่ได้จากการแปลงภาพต้นฉบับเป็นสี HSV แล้วทำการ threshold (ใช้ cv2.inRange()) เพื่อให้ภาพเป็นสีขาวดำ โดยส่วนที่ต้องการตรวจจับวัตถุ (ตุ๊กตาไก่หรือขวด) จะเป็นสีขาว (255) และส่วนที่เหลือจะเป็นสีดำ (0)
  - cv2.RETR_EXTERNAL: เป็นพารามิเตอร์ที่ใช้กำหนดวิธีการค้นหา contours โดย RETR_EXTERNAL จะดึงเฉพาะ contours ที่อยู่นอกสุดของวัตถุที่ถูกตรวจจับ (จะไม่สนใจขอบเขตภายในที่ซ้อนกัน)
  - cv2.CHAIN_APPROX_SIMPLE: เป็นพารามิเตอร์ที่ใช้กำหนดวิธีการเก็บข้อมูล contours โดย CHAIN_APPROX_SIMPLE จะลดจำนวนจุดที่ใช้เก็บ contours ให้น้อยลง ถ้าจุดเหล่านั้นสามารถแทนที่ด้วย

เส้นตรง เช่น ในเส้นตรงจะเก็บแค่จุดหัวและจุดท้าย ไม่เก็บจุดทุกจุดในเส้นนั้น เพื่อประหยัดหน่วยความจำและประมวลผลได้เร็วขึ้น

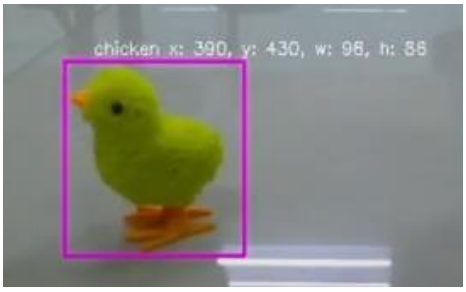
- เลือก contours ที่ใหญ่ที่สุด

  - max(): หา contour ที่มีพื้นที่มากที่สุดในลิสต์ของ contours
  - key=cv2.contourArea: พารามิเตอร์นี้บอกให้ฟังก์ชัน max() ทำการเปรียบเทียบพื้นที่ของแต่ละ contour โดยใช้ฟังก์ชัน cv2.contourArea() เพื่อหาพื้นที่ของแต่ละ contour

- ใช้ cv2.boundingRect() ในการหากรอบสี่เหลี่ยมรอบ contours ที่ใหญ่ที่สุด
  - cv2.boundingRect(contour): ใช้ในการคำนวณกรอบสี่เหลี่ยมผืนผ้าล้อมรอบ contour ที่ถูกส่งเข้ามา โดยฟังก์ชันจะคำนวณจากพิกัดขอบของ contour นั้น ๆ
  - ผลลัพธ์ที่ได้คือ (x, y, w, h):

    - x และ y คือพิกัดมุมซ้ายบนของกรอบสี่เหลี่ยมที่ล้อมรอบ contour
    - w คือความกว้างของกรอบสี่เหลี่ยม (width)
    - h คือความสูงของกรอบสี่เหลี่ยม (height)

- เทียบอัตราส่วน และวาดกรอบสี่เหลี่ยมรอบตุ๊กตาไก่
  - หากอัตราส่วนของกรอบสี่เหลี่ยมรอบ contours ที่ใหญ่ที่สุด(กว้าง:สูง) อยู่ในช่วงที่กำหนด แสดงว่าเจอตุ๊กตาลูกไก่ และวาดกรอบสี่เหลี่ยมรอบตุ๊กตาไก่



รูปที่13 วาดbounding boxรอบตุ๊กตาลูกไก่

ผลการทดลอง(สนามassignment 3 รอบที่ 1)

เส้นทางที่สำรวจ	shortest path