Homework 4+5: Minesweeper

`Minesweeper` was a doozy to program, so let's get started with our Node definition.

Subscriptions

```
class Minesweeper(Node):
    def __init__(self):
        super().__init__('minesweeper')
        self.video_feed = self.create_subscription(
            Image,
            'camera/color/image_raw',
            self.video_callback,
            qos.qos_profile_sensor_data)

        self.hazard_subscription = self.create_subscription(
            HazardDetectionVector,
            'qbert/hazard_detection',
            self.hazard_callback,
            qos.qos_profile_sensor_data)

        self.april_subscription = self.create_subscription(
            AprilTagDetectionArray,
            '/detections',
            self.april_callback,
            qos.qos_profile_sensor_data)
```

Our `Minesweeper` node has three subscriptions. However, `hazard_subscription` is not used.

Member Variables

```
        self.bridge = CvBridge()
        self.fourcc = cv2.VideoWriter_fourcc(*'XVID') # Specify video codec
        self.out = cv2.VideoWriter('output.avi', self.fourcc, 30.0, (640, 480)) # Creat

        self.boomed = 0
        self.booming = False

        self.timer_stopper = None
        self.forward_timer = None

        self.heading_home = False
        self.last_center = None
```

`Minesweeper` has several important member variables. `bridge`, `fourcc`, and `out` are all used to facilitate the processing of video input from `qbert`.

`boomed` is used to track the number of mines boomed, and `booming` is a boolean we use to track whether `qbert` is in **close proximity** to a mine (more details later).

`forward_timer`, and `timer_stopper` are two timers we reused from `Wanderer`, which tell `qbert` to go forward for a set amount of time.

`heading_home` is set to `True` after 4 mines are boomed.

`last_center` is a list `[x, y]` we use to track the last coordinates of a spotted centroid (Mine or AprilTag).

`video_callback`

```
    def video_callback(self, msg):
        lower = (29, 100, 6)
        upper = (64, 255, 255)
```

`video_callback` is largely ripped from the **OpenCV ball tracking demo** given in the class slides.

`lower` and `upper` are tuples which define our HSV detection bounds, and they are notable because our group never found out the optimal setting for these tuples in order to eliminate false positive mine detections.

```
        if not self.heading_home:
            if center:
                self.last_center = center
            self.tracking_callback()
```

This last part of `video_callback` begins our tracking logic. Given that `qbert` has not bumped 4 mines, we will set the centroid which we are tracking to any `center` (mine) which our ball tracking program has collected. Then we call our `tracking_callback`.

`tracking_callback`

```
    def tracking_callback(self):
        tracking_twist = Twist()
        if not self.heading_home:
            if self.last_center:
                if self.last_center[0] < 225:
                    tracking_twist.angular.z = .1
                elif self.last_center[0] > 375:
                    tracking_twist.angular.z = -.1
                else:
                    tracking_twist.linear.x = .075
                if self.last_center[1] >= 350:
                    if not self.booming:
                        self.forward_timer = self.create_timer(.1, self.forward_callbacl
                        self.timer_stopper = self.create_timer(3, self.destroy_forward)
                        self.booming = True
                        print("booming")
            else:
                tracking_twist.angular.z = 0.1
            if self.boomed >= 4:
                self.heading_home = True
                self.last_center = None
```

We begin our `tracking_callback` by instantiating a blank `Twist()` and implicitly checking if we are tracking AprilTags or mines with

```
if not self.heading_home
```

Through some light tuning, we established that as long as `qbert` travels towards a mine within the bounds `x > 225` and `x < 375`, it will succesfully home into range for our **booming** logic to takeover.

```
            if self.last_center[1] >= 350:
                if not self.booming:
                    self.forward_timer = self.create_timer(.1, self.forward_callback)
                    self.timer_stopper = self.create_timer(3, self.destroy_forward)
                    self.booming = True
                    print("booming")
```

We check if the `y` value of our last tracked center is above a certain threshhold before beginning our **booming** logic.

```
  def forward_callback(self):
        forward = Twist()
        forward.linear.x = .1
        self.publisher.publish(forward)

    def destroy_forward(self):
        self.destroy_timer(self.forward_timer)
        self.destroy_timer(self.timer_stopper)
        self.booming = False
        self.boomed += 1
        print("boomed")
        self.last_center = None
        self.publisher.publish(Twist())
```

`forward_timer` is a timer which propogates a `linear.x` velocity of `.1`, and is terminated by `destroy_forward`. `destroy_forward` contains the logic to increment the mines boomed, and implicitly tells `qbert` to look for a new mine.

```
        if not self.last_center:
            tracking_twist.angular.z = 0.005
```

This is the last part of `tracking_callback`. When `qbert` either cannot see a mine/AprilTag or has finished booming a mine, it will begin to spin slowly (avoiding blur and compensating for low framerate), seeking a new mine.

Homing onto AprilTags

```
        else:
            if self.last_center:
                if self.last_center[0] < 175:
                    tracking_twist.angular.z = .1
                elif self.last_center[0] > 250:
                    tracking_twist.angular.z = -.1
                else:
                    tracking_twist.linear.x = .075
                if self.last_center[1] >= 350:
                    if not self.booming:
                        self.forward_timer = self.create_timer(.1, self.forward_callbacl
                        self.timer_stopper = self.create_timer(3, self.destroy_forward)
                        self.booming = True
                        print("booming")
```

Recall that `if not self.heading_home` implicitly allows `qbert` to check whether it is tracking AprilTags or mines. This else block contains the same logic `qbert` uses to home onto mines, but with more narrow bounds on the `x` axis.

How are we actually detecting AprilTags?

src/qbert_py_minesweeper/package.xml

```
<exec_depend>apriltag_msgs</exec_depend>
```

We are using the predefined AprilTag detector node given in the slides to process AprilTags sighted in `qbert`'s camera. We run this node externally on another terminal instance, and this node publishes to the topic `/detections`. Earlier, we subscribed to `/detections` in our `april_subscription`. Here's how we are using these messages.

```
def april_callback(self, detections):
    if self.heading_home:
        if(detections.detections):
            if(detections.detections[0].id == 8):
                x = detections.detections[0].centre.x
                y = detections.detections[0].centre.y
                self.last_center = [x, y]
                print(f"here: {self.last_center}")
            elif self.heading_home:
                self.last_center = None
        self.tracking_callback()
```

We only allow `april_callback` to run if `qbert` has boomed 4 or more mines. Then, we are simply checking if `qbert` sees an AprilTag, and if the `id` is 8. If so, we set the last spotted centroid to the AprilTag's `(x,y)` coordinate, and call `tracking_callback`

This finalizes `Minesweeper`.

Issues

As mentioned before, our chosen HSV bounds were allowing false positive centroid detections for mines, which was debilitating our program. Additionally, `Minesweeper` is not robust to visually obstructive obstacles, nor does it truly tap into more elegant concepts such as odometry, SLAM, or using CV for navigation.

In the future

We wanted to implement `Wanderer` into `Minesweeper`, which would allow for better mine seeking.