SWINBURNE UNIVERSITY OF TECHNOLOGY

COS30002

Autonomous Steering 2

Task 9, Week 6

Matthew Coulter
S102573957

23/04/2020

Tutor: Tien Pham

Table of Contents

Step 1

Generating a random path

In the agent class, we can generate a random path for that agent by calling the create_random_path() function (in path.py) with the following parameters:

```
def randomise_path(self):
        cx = self.world.cx
        cy = self.world.cy
        margin = min(cx,cy) * 1/6
        self.path.create_random_path(5, margin, margin, cx - margin, cy - margin)
```

The margin is a margin that in the world that only allows the path to be drawn the inner square.

within the __init__ function, this is called so the agent is initialised with a new random path:

```
def __init__(self, world=None, scale=30.0, mass=1.0, mode='seek'):
        ### path to follow?
        self.path = Path()
        self.randomise_path()
        self.waypoint_threshold = 50.0
```

(Irrelevant __init__() code has been hidden)

Here we also initialise the waypoint_threshold. This is a value that determines how close the agent needs to be to a waypoint for it to be considered reached. I found that 40.0-50.0 is a good range for this value.

In order to allow the agent/s to begin pursuit, we need to update the calculate function, however this has already been done by the provided code:

```
AGENT_MODES = {
    KEY._7: 'follow_path',
}
def calculate(self):
        # calculate the current steering force
        mode = self.mode
        elif mode == 'follow_path':
            force = self.follow_path()
```

(Irrelevant calculate() code has been hidden)

Drawing the path to the screen

The world controls the render function which calls the render methods within each class. Fortunately, path already has a render method in it, hence all we have to do is modify the render function to this:

```
def render(self,color=None):
            if self.mode == 'follow_path':
                self.path.render()
```

Essentially, if the mode is set to follow, it will call the render function for the path of each agent.

Getting the agent to follow the random path

We just need to implement the follow_path code and the agent will start following it. The agent will continue to follow the path until it's current point isd the last point in the destination, this is when we would want the agent to land on the last point at 0 velocity:

```
def follow_path(self):
        if (self.path.is_finished()):
            # Arrives at the final waypoint
            return self.arrive(self.path._pts[-1],'slower')
        else:
            # Goes to the current waypoint and increments on arrival
            to_target = self.path.current_pt() - self.pos
            dist = to_target.length()
            if (dist < self.waypoint_threshold):
                self.path.inc_current_pt()
            return self.arrive(self.path.current_pt(),'slower')
```

We can easily achieve this by using the arrive function that we wrote is the previous lab. This is where the threshold is used. I originally tested to see if the positions of both the agent and the waypoint were the same. However, this is very inefficient as the agent's position is unlikely to ever be the EXACT same. Hence, we can employ the distance_length technique from the arrival function from the last lab. This works hand in hand with the threshold as once the distance is below the threshold, the waypoint will increment.

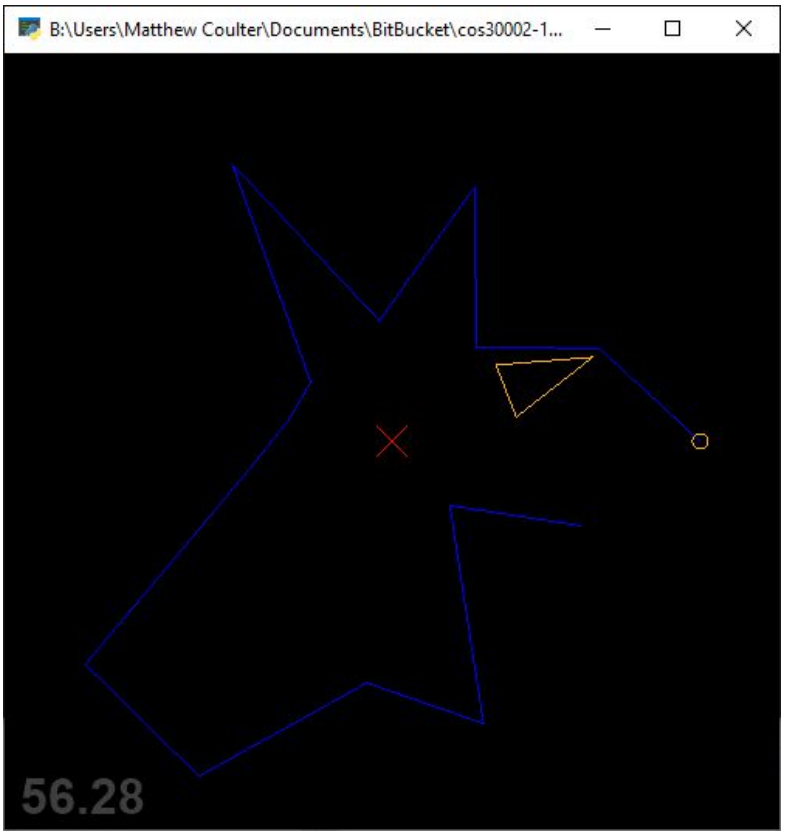
Likewise, I actually slightly modified the arrive function to use this threshold value as well.

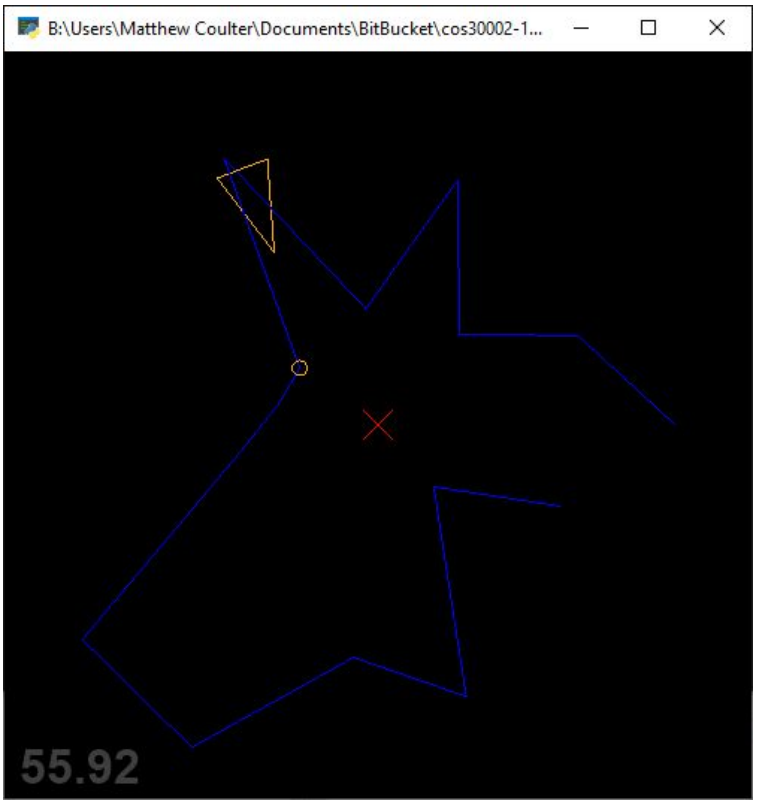
```
def arrive(self, target_pos, speed):
        if dist > self.waypoint_threshold:
```

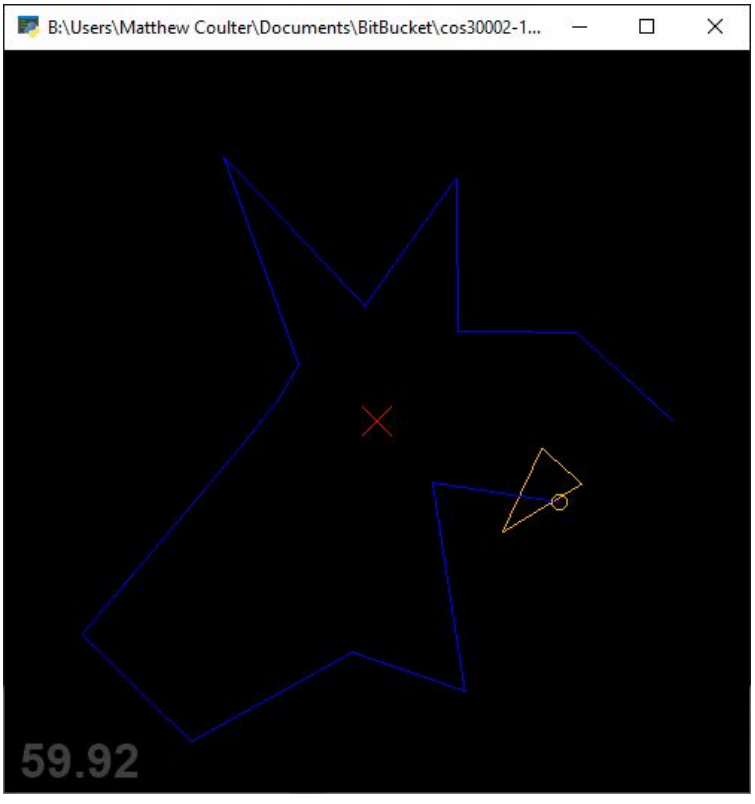
```
            # Irrelevant code...
            return (desired_vel - self.vel)
        return Vector2D(0, 0) - self.vel
```

Previously, the agent continued to try and reach the target even though the agent was obviously there already. Now, once it is in the threshold, it will come to a stop.

Screenshots of evidence:




B:\Users\Matthew Coulter\Documents\BitBucket\cos30002-1...
55.92


B:\Users\Matthew Coulter\Documents\BitBucket\cos30002-1...
59.92

Adding a reset button

When we press R, a new random path should be generated. The following code does this:

```
def on_key_press(symbol, modifiers):
    elif symbol == KEY.R:
        for agent in world.agents:
            agent.randomise_path()
```

Step 2

Drawing wander graphics to the screen

Again, we can use the render function in the asgent class to draw the wandering graphics to the screen. There is a circle and a line will be drawn. The circle essentially represents the projection circle. Essentially, the bigger the radius on it allows for a greater steering force to be applied to the agent. The smaller red circle lies on the projection circle and represents the direction the agent is steering towards.

```
if self.mode == 'wander':
            # calculate the center of the wander circle in front of the
agent
            wnd_pos = Vector2D(self.wander_dist, 0)
            wld_pos = self.world.transform_point(wnd_pos, self.pos,
self.heading, self.side)
            # draw the wander circle
            egi.green_pen()
            egi.circle(wld_pos, self.wander_radius)
            # draw the wander target (little circle on the big circle)
            egi.red_pen()
            wnd_pos = (self.wander_target + Vector2D(self.wander_dist, 0))
            wld_pos = self.world.transform_point(wnd_pos, self.pos,
self.heading, self.side)
            egi.circle(wld_pos, 3)
```

This type of steering behaviour works well as it creates a very fluent motion. WIth a balanced set of variables, the agent cannot jump from left to right to left to right etc. It needs to steer all the way from the left to the right.

Wander variables

```
# NEW WANDER INFO
        self.wander_target = Vector2D(1, 0)
        self.wander_dist = 1.0 * scale
        self.wander_radius = 1.0 * scale
        self.wander_jitter = 10.0 * scale
```

This code initialises the different variables to create the wandering behaviour.

The self.wander_target provides the vector based direction for the agent to head towards.

The wander_dist changes the placement of the circle in relation to the agent. The further away it is, the straighter the overall movement will become.

The wander_radius changes the size of the circle allowing for sharper turns when bigger and straighter movement when smaller.

The wander_jitter changes the size of each jitter that happens. A larger value will correspond to more jittered behaviour of the agent as it will move left and right more often.

Creating the wander movement

The wandering function uses the variables with the delta time variable to essentially return the vector that the agent will follow. Conveniently, the seek function is reused as the target is constantly changing.

The code is below:

```
def wander(self, delta):
        ''' random wandering using a projected jitter circle '''
        wt = self.wander_target
        # this behaviour is dependent on the update rate, so this line
must
        # be included when using time independent framerate.
        jitter_tts = self.wander_jitter * delta # this time slice
        # first, add a small random vector to the target's position
        wt += Vector2D(uniform(-1,1) * jitter_tts, uniform(-1,1) *
jitter_tts)
        # re-project this new vector back on to a unit circle
        wt.normalise()
        # increase the length of the vector to the same as the radius
        # of the wander circle
        wt *= self.wander_radius
        # move the target into a position WanderDist in front of the agent
        target = wt + Vector2D(self.wander_dist, 0)
        # project the target into world space
```

```
        wld_target = self.world.transform_point(target, self.pos,
self.heading, self.side)
        # and steer towards it
        return self.seek(wld_target)
```

Whilst there is some complex math involved, the concept is rather simple. wander is being called every time the world update function is called. The jitter uses a delta time variable to calculate the motion hence if the game were to be lagging, it is independent of the frame rate and this would not affect the fluency. The jitter circle is projected onto the projection circle through the normalisation and multiplication processes. But the most important is the calculation of wld_target as this is essentially how the agent will know what direction to go. It uses the supplied code for transform_point() which will take the wander variables and output a coordinate. The agent then seeks that position. The code for transform_point is below:

```
def transform_point(self, point, pos, forward, side):
        ''' Transform the given single point, using the provided position,
        and direction (forward and side unit vectors), to object world
space. '''
        # make a copy of the original point (so we don't trash it)
        wld_pt = point.copy()
        # create a transformation matrix to perform the operations
        mat = Matrix33()
        # rotate
        mat.rotate_by_vectors_update(forward, side)
        # and translate
        mat.translate_update(pos.x, pos.y)
        # now transform the point (in place)
        mat.transform_vector2d(wld_pt)
        # done
        return wld_pt
```