Exercise Session 2 (walkthrough)

Here is an overview of what each folder contains:

- "1 STARTING POINT"
  - A blank screen with two characters: Pacman and one ghost;
- "2 KINEMATIC SEEK"
  - A first, simple implementation of seek behaviour in a Kinematic fashion;
- "3 SEEK"
  - The steering version of the seek behaviour;
- "4 ARRIVE"
  - Seek behaviour with algorithm for slowing down at arrival;
- "5 WANDER"
  - Wander steering behaviour;
- "6 ALIGN"
  - Aligning the direction of one character to the other;
- "7 MATCH SPEED"
  - Matching the direction and velocity of one character to the other.

Make sure you navigate between folders as we go through this explanation.

1 STARTING POINT

We borrow some of the things we implemented in the 1st exercise session: the blank screen and the Pacman and Ghost characters.

For this session, we won't implement the node system: this will allow us to better visualise and understand what is going on with the different behaviours.

2 KINEMATIC SEEK

Navigate to “pacman.py”.

First, we implement two new instance variables:

- self.direction → the direction vector as seen in lectures (initialised in a random direction)
- self.velocity → the velocity vector as seen in lectures

```
18        self.direction = pygame.math.Vector2(self.speed, 0).rotate(uniform(0,360))
19        self.velocity = Vector2()
```

(The self.screen variable is for visualizing the vectors; you shouldn’t concern yourself with this)

“follow_mouse()” method

We then implement the “follow_mouse()” method. We retrieve the coordinates of the mouse on the screen using PyGame’s “get_pos()” function. (line 52). This function returns a tuple, which in Python is an immutable object, meaning that once it’s created, it can’t be modified.

We therefore create a new tuple (line 53) that has coordinates of the mouse position, plus 1. This is simply because we will use these values to normalize the vector in the next line, and because normalization involves a division, we don’t want to get an error that crashes the game.

Finally, we compute the velocity (live 54) by performing vector subtraction between position and goal (mouse position). We then normalize to turn it into a unit vector, which (as seen in lecture) is the standard way of using velocity vectors. To adjust the intensity, we can multiply this resulting vector by a number (such as 0.2, as in the code).

```
51    def follow_mouse(self):
52        mpos = pygame.mouse.get_pos()
53        mpos2 = (mpos[0]+1, mpos[1]+ 1)
54        self.velocity = (mpos2 - self.position).normalize() * 0.2
```

“update()” method

In the “update()” method, we then call the newly implemented “follow_mouse()” method to compute the velocity, and use it to update the direction and position of our Pacman.

We cap the possible speed of Pacman by checking that it is less than our defined speed, and clipping it if it isn’t, so that it doesn’t increase infinitely. Finally, we update the position.

You can run the game by opening a terminal, navigating into the directory and typing the command: “python3 run.py”

3 SEEK

In the steering context, we use velocity and acceleration.

```
19          self.vel = pygame.math.Vector2(self.speed, 0).rotate(uniform(0,360))
20          self.acc = Vector2()
```

"seek()" method

We implement the "seek()" method, which takes "target" as parameter (line 61): this is the target position that wants to be reached by the character.

When calling the method in the "update()" method (line 25) we pass the mouse position as the target.

We compute the vector for the desired direction for the character (line 62) by performing vector subtraction between the position and the target (the mouse position). Just like before, we normalize this. This is the vector that the character would want to follow if it was not affected by any steering force.

The steering force is computed by vector subtraction between where the character is actually going right now (self.vel) and where it would want to go (self.desired). This vector can be multiplied by a "STEERING_FORCE" to choose the strength of the steering. Similarly to before, we clip this value so it doesn't increase infinitely.

We finally return the steer vector.

```
61      def seek(self, target):
62          self.desired = (target-self.position).normalize() * self.speed
63          steer = (self.desired - self.vel)
64          STEERING_FORCE = 0.5
65          if steer.length() > STEERING_FORCE:
66              steer.scale_to_length(STEERING_FORCE)
67          return steer
```

"update()" method

As mentioned before, we call the "seek()" method in "update()" and pass it the mouse position as the target (line 25). The returned vector is the acceleration vector.

We then update the velocity using this acceleration vector, clipping it as usual.

And update the position of our character.

```
23      def update(self, dt):
24          # self.follow_mouse()
25          self.acc = self.seek(pygame.mouse.get_pos())
26          self.vel += self.acc
27          if self.vel.length() > self.speed:
28              self.vel.scale_to_length(self.speed)
29          self.position += self.vel
```

4 ARRIVE

We implement “seek_and_arrive()” method, which works the same way as the previously implemented “seek()” method, but makes the character progressively decelerate when entering a radius which we define (line 70).

As before, we create the desired vector (line 71). As before we want to normalize it (line 73), but before doing that we will store its length in a new variable (line 72).

We will use it to determine if we are entering the target’s radius (line 74). Once we are within the radius, we will progressively decrease the “desired” vector (line 75).

Other than this, the rest is the same as the previous “seek()” method.

```
69      def seek_and_arrive(self, target):
70          APPROACH_RADIUS = 50
71          self.desired = (target-self.position)
72          dist = self.desired.length()    #we get the distance before normalizing t
73          self.desired.normalize_ip() # ip = in place
74          if dist < APPROACH_RADIUS:
75              self.desired *= dist / APPROACH_RADIUS * self.speed
76          else:
77              self.desired *= self.speed
78          steer = (self.desired - self.vel)
79          STEERING_FORCE = 0.5
80          if steer.length() > STEERING_FORCE:
81              steer.scale_to_length(STEERING_FORCE)
82          return steer
```

5 WANDER

We will implement two different types of wandering behaviours: a naïve one, and a smarter one.

Because of this, we will define a "WANDER_TYPE" instance variable which we will use to select one of the two types of wander when running the game.

```
23          self.WANDER_TYPE = 1
```

The first wandering behaviour will spawn a random point on the screen, and set it as the target. The rate at which new points are spawned is defined with the "RAND_TARGET_TIME" variable (line 123).

In lines 126-129, we simply check whether 500 milliseconds have passed in the game loop: when this happens, a new point is spawned on the screen (in the form of a vector) and set as target (line 130).

```
122 def wander(self):
123     RAND_TARGET_TIME = 500
124     self.WANDER_TYPE = 1
125     # select random target every few sec
126     now = pygame.time.get_ticks()   #returns time since pygame.init() was called
127     if now - self.last_update > RAND_TARGET_TIME:
128         self.last_update = now
129         self.target = pygame.math.Vector2(randint(0, SCREENWIDTH), randint(0, SCREENHEIGHT))
130     return self.seek(self.target)
```

Improved wander behaviour

This is not very clever, as the movement of the character will be totally random and artificial.

For the improved version of wander, we will need two new instance variables: "ring_dist" and "ring_radius". Essentially, we want to limit the space in which the new target points are spawned to only in front of where the character is already going. This will make the wandering more natural.

To do this, we will generate a circle in front of the character at distance "ring_dist" and with radius "ring_radius". From the center of this circle we will then spawn a vector in a random direction: we will finally compute a vector from the character's position to this new vector as the target vector.

```
24          self.ring_dist = 120
25          self.ring_radius = 50
```

We compute the centre of the circle (line 133) by performing vector sum of the character's position and the orientation given by the velocity vector; we then multiply this by the afore-defined "ring_dist".

We then spawn a vector from the centre of the circle to a random direction (line 134). Finally, we set the newly found vector as the target for the seek method (line 137).

```
132 def wander_improved(self):
133     self.WANDER_TYPE = 2
134     future = self.position + self.vel.normalize() * self.ring_dist
135     target = future + pygame.math.Vector2(self.ring_radius, 0).rotate(uniform(0, 360))
136     self.displacement = target
137     return self.seek(target)
```

6 ALIGN

Here we copy the “follow_mouse” and “seek_and_arrive” methods from pacman.py to ghost.py.

The goal for this section is to make pacman copy the ghost’s direction. We do not want to update the character’s position, we only want to match the orientation: to achieve this, we comment out line 26 where the character’s position is updated.

Therefore, in pacman.py file we simply set the “self.vel” to the ghost’s velocity vector (line 25).

7 ALIGN

Now we want to also match the acceleration to match the ghost’s.

We do so in line 26.