Task 12 Lab: Steering 2 - Wander and Paths

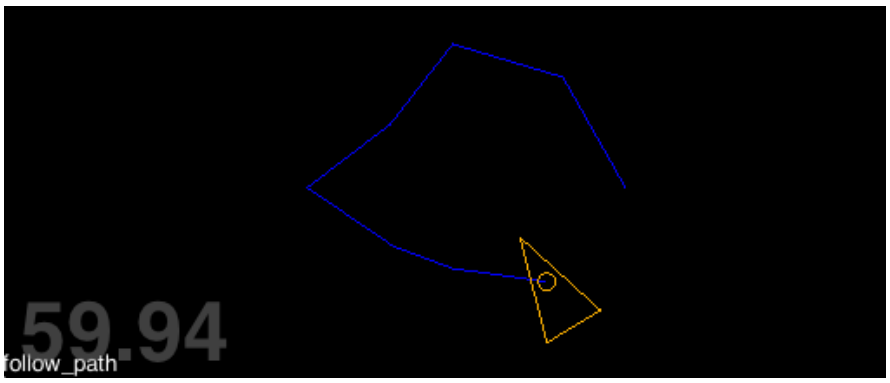
This lab will extend the code from the last lab and add the ability for agents to follow way-points in a path, and to perform a nice-looking smooth random wander. Both are useful for later work. Commit as you go - you will upload your commit history to Canvas as evidence of your progress.

Warning: Do NOT just keep adding to the code for the last lab! Copy old code to a folder for this task, commit (and push) and then make changes. You need clean code for each lab or spike in your final portfolio.

Step 1: follow_path()

The sample code for this lab can be found in the samples folder in the directory for this task. The base code is essentially the same as the previous lab with additional details for this lab. You will want to copy snippets of code that you worked out for the last lab into the code for this lab. (Don't mess up your previous lab code!)

Have a look at the new path.py file which provides a handy `Path` class. We will use the agent `seek`() and `arrive`() steering behaviours from follow each point that is contained in the path.



- In the `Path` class note the following:
  - The `create_random_path`() method creates a new path with random points, to which we can pass world related parameters (such a width and height).
  - Use `current_pt`() to find out the current point, inc`_current_pt()` to move current to the next point, and `is_finished`() to test for the end point in an open (non-loop) path.

- Modify the Agent class so that each agent will have its own path instance and will be able to follow it when in "`follow_path`" mode.
  - In the `__init__` method add the following (without the comments)

```
self.path = Path()
self.randomise_path()  # <-- Doesn't exist yet but you'll create it
self.waypoint_threshold = 0.0  # <-- Work out a value for this as you test!
```

  - Create the new `randomise_path`() so that it's ready to be called. In it, call the `path.create_random_path`() method using world-related parameters

```
cx = self.world.cx  # width
cy = self.world.cy  # height
margin = min(cx, cy) * (1/6)  # use this for padding in the next line ...
self.path.create_random_path(...)  # you have to figure out the parameters
```

  - Add a "`follow_path`" mode and modify the calculate method to use it by calling a (new) `follow_path` method.

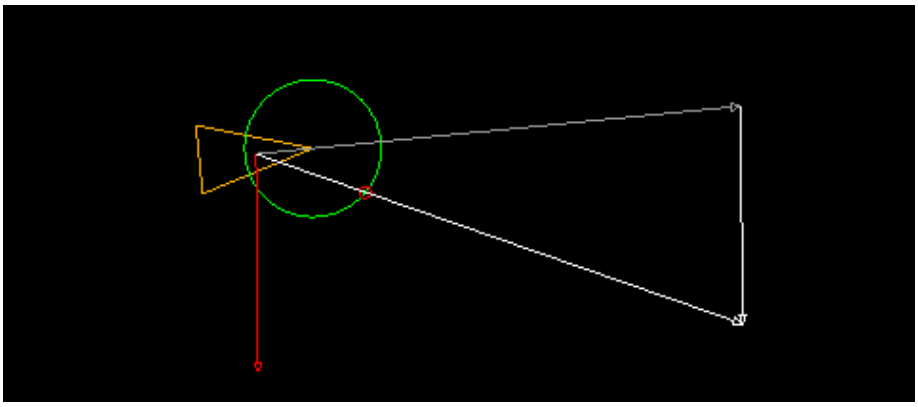
  - Add a `follow_path` method and code the following logical ideas:

```
# If heading to final point (is_finished?),
#     Return a slow down force vector (Arrive)
# Else
#   If within threshold distance of current way point, inc to next in path
#   Return a force vector to head to current point at full speed (Seek)
```

- Alter the main.py file so that you can reset the agent paths in response to a key press (say 'R'), by looping through all the agents and calling their `randomise_path`().

- Test it! Adjust the waypoint threshold distance (which will depend on if you used a squared distance value or not), adjust force and max speed values, and try different arrive speeds.

Step 2: wander()



Above is a figure that shows the green wander circle projected in front of the agent. The small red circle is the current wander target, while the grey arrow is the current velocity. A red arrow indicates the current steering force being applied to modify the current velocity so that, as a result (white arrows indicate the vector addition), the agent will head to (through) the wander target.

In order to do some smooth wandering, we first need to set up helper code. This helper code and our wander code will use several new wander-related variables. Add them to the `__init__` of the `Agent` class so that they can be used later. Also add or change max force and speed values.

```
    # 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
    self.bRadius = scale

    # Force and speed limiting code
    self.max_speed = 20.0 * scale
    self.max_force = 500.0
```

You will also need to add the following new code to the `World` class so that it can convert a single local coordinate into a world coordinate for us (because the wander code needs a bit of this ☺). Note: There is already a similar method for transforming a group of points - you could use that.

```
    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)
        world_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(world_pt)
        # done
        return world_pt
```

Okay - back to the `Agent` class and the actual `wander`() code. Because we update the jitter location based on the amount of time that has occurred, this new steering behaviour will need the `delta` time value. So, you need to alter the `Agent.update`() method so that it passes the `delta` value through to the `calculate`() method, and then the `calculate`() method can pass `delta` to the new `wander`() method.

Almost there! The wander code should look something like this. Note: You will need to import the `uniform` function from the `random` module (i.e. `from random import uniform`).

```
def wander(self, delta):
  ''' random wandering using a projected jitter circle '''
  wander_target = self.wander_target
  # this behaviour is dependent on the update rate, so this line must
  # be included when using time independent framerate.
  jitter = self.wander_jitter * delta # this time slice
  # first, add a small random vector to the target's position
  wander_target += Vector2D(uniform(-1,1) * jitter, uniform(-1,1) * jitter)
  # re-project this new vector back on to a unit circle
  wander_target.normalise()
  # increase the length of the vector to the same as the radius
  # of the wander circle
  wander_target *= self.wander_radius
  # move the target into a position wander_dist in front of the agent
  wander_dist_vector = Vector2D(self.wander_dist, 0) #also used for rendering
  target = wander_target + Vector2D(self.wander_dist, 0)
  # set the position of the Agent's debug circle to match the vectors we've created
  circle_pos = self.world.transform_point(wander_dist_vector, self.pos, self.heading, self.side,)
  self.info_wander_circle.x = circle_pos.x
  self.info_wander_circle.y = circle_pos.y
  self.info_wander_circle.radius = self.wander_radius
  # project the target into world space
  world_target = self.world.transform_point(target, self.pos, self.heading, self.side)
  #set the target debug circle position
  self.info_wander_target.x = world_target.x
  self.info_wander_target.y = world_target.y
  # and steer towards it
  return self.seek(world_target)
```

However, to get this all working right you will need to test the wander variables, perhaps weight the wander force (or total steering force), and also the add a new limit to the steering force. You want to be able to change variables while the program is running: edit-restart-test is slow!

You can add the following force-limiting code to the agent `update`() method.

```
        force = self.calculate(delta)
        force.truncate(self.max_force) # <-- new force limiting code
```

Save your work, commit and push. Upload a text file to Canvas of your commit history to indicate you are ready to present your working code to your tutor.