# Task 9 – 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.

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

Step 1: `follow_path()`

Download the code for this lab from the unit website. 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.
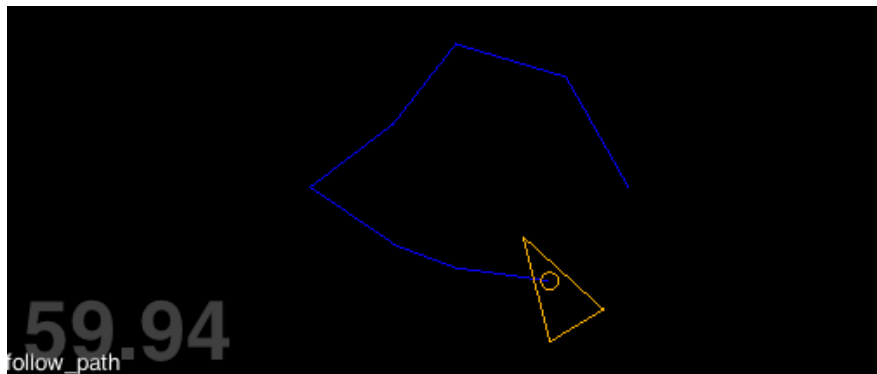
- ➢ In the `Path` class note the following:
    - ○ The `create_random_path()` method creates a new path with random points, to which we can pass world related parameters (such a width and height).
    - ○ Use `current_pt()` to find out the current point, `inc_current_pt()` to move current to the next 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)
      ```

    - ○ Modify the `render` method to call `path.render()` if currently in "`follow_path`" mode.

© Clinton Woodward, Swinburne FSET

➢ 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()`
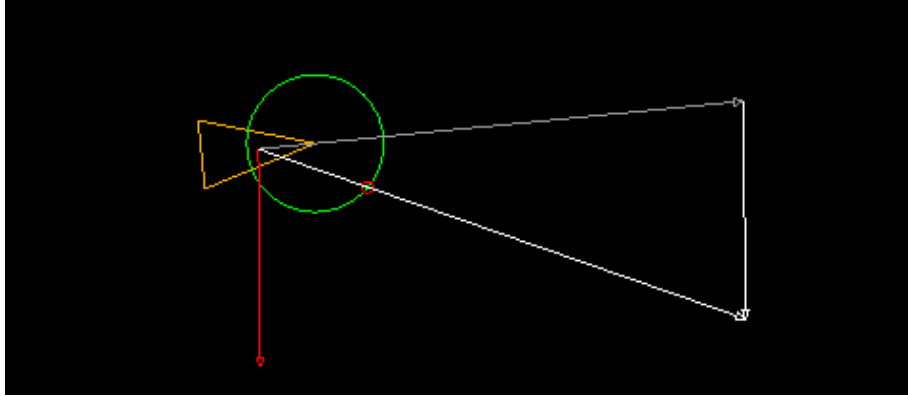
Above is a figure that shows the green wander circle projected in front of the agent. The small red circle is the current wander target, while the grey arrow is the current velocity. A red arrow indicates the current steering force being applied to modify the current velocity so that, as a result (white 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. We can start by adding drawing code to the agent to show the wander details (circle, jitter location), which then needs new wander variables (in the agent) and a new conversion routine for points from local to world space.

Add the following additional code to the `Agent render` method. *(This code should also give you some hints for the actual wander code you need to write later!)*

```
# draw wander info?
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)
```

Notice that this new code uses 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
```

© Clinton Woodward, Swinburne FSET

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.

```python
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
```

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 (eg. `from random import uniform`).

```python
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)
```

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.

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

Save your work for use in later labs or spikes.