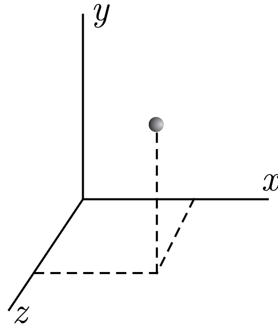
Project 12.4 Boids in space

In this project, you will generalize the two-dimensional flocking simulation that we developed in Section 12.3 to three dimensions, and visualize it using three-dimensional graphics software called VPython. For instructions on how to install VPython, visit `http://vpython.org`.

The three-dimensional coordinate system in VPython looks like this, with the positive z-axis coming out of the screen toward you.



The center of the screen is at the origin $(0, 0, 0)$.

Part 1: Generalize the `Vector` *class*

The `Pair` and `Vector` classes that we used to represent positions and velocities, respectively, are limited to representing two-dimensional quantities. For this project, you will generalize the `Vector` class that we introduced in Section 12.3 so that it can store vectors of any length. Then use this new `Vector` class in place of both `Pair` and the old `Vector` class.

The implementation of every method will need to change, except as noted below.

- The constructor should require a list or tuple parameter to initialize the vector. The length of the parameter will dictate the length of the `Vector` object. For example,

  ```
  velocity = Vector((1, 0, 0))
  ```

  will assign a three-dimensional `Vector` object representing the vector $\langle 1, 0, 0 \rangle$.
- Add a `__len__` method that returns the length of the `Vector` object.
- The `unit` and `diffAngle` methods can remain unchanged.
- You can delete the `angle` and `turn` methods, as you will no longer need them.
- The dot product of two vectors is the sum of the products of corresponding elements. For example, the dot product of $\langle 1, 2, 3 \rangle$ and $\langle 4, 5, 6 \rangle$ is $1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32$. The `dotproduct` method needs to be generalized to compute this quantity for vectors of any length.

Copyright Taylor and Francis, 2021

Part 2: Make the `World` three-dimensional

Because we were careful in Section 12.3 to make the `World` class very general, there is little to be done to extend it to three dimensions. The main difference will be that instead of enforcing boundaries on the space, you will incorporate an object like a light to which the boids are attracted. Therefore, the swarming behavior will be more similar to moths around a light than migrating birds.

- Generalize the constructor to accept a `depth` in addition to `width` and `height`. These attributes will only be used to size the VPython display and choose initial positions for the boids.
- Once you have `vpython` installed, you can import it with

  ```
  import vpython as vp
  ```

  To create a new window, which is an object belonging to the class `canvas`, do the following:

  ```
  self._scene = vp.canvas(title='Boids', width = 800, height = 800,
                          range = width,
                          background = vp.vector(0.41, 0.46, 0.91))
  ```

  In the `canvas` constructor, the `width` and `height` give the dimensions of the window and the `range` argument gives the visible range of points from the origin. The `background` color is a deep blue; feel free to change it. The `vpython` objects use their own `vp.vector` class to define positions and colors. So you will need to convert between your `Vector` class and `vp.vector` when using `vpython` objects. The following utility functions will do this.

  ```
  def vector2VP(vector):
      a, b, c = vector.get()
      return vp.vector(a, b, c)

  def VP2vector(vpvector):
      return Vector((vpvector.x, vpvector.y, vpvector.z))
  ```

  To place a yellow sphere in the center to represent the light, create a new `sphere` object like this:

  ```
  self._light = vp.sphere(scene = self._scene, color = vp.color.yellow)
  ```

- In the `stepAll` method, make the light follow the position of the mouse with

  ```
  self._light.pos = self._scene.mouse.pos
  ```

  You can change the position of any `vpython` object by changing the value of its `pos` instance variable. In the display object (`self._scene`), `mouse` refers to the mouse pointer within the VPython window.
- Finally, generalize the `_distance` function, and remove all code from the class that limits the position of an agent.

Part 3: Make a `Boid` three-dimensional

- In the constructor, initialize the position and velocity to be three-dimensional `Vector` objects. You can represent each boid with a cone, pointing in the direction of the current velocity with:

```
self._turtle = vp.cone(pos = vp.vector(x, y, z),
                       axis = vector2VP(self._velocity * 3),
                       color = vp.color.white,
                       scene = self._world._scene)
```

- The `move` method will need to be generalized to three dimensions, but you can remove all of the code that keeps the boids within a boundary. Once the new position and velocity have been computed, you can move the boid with

```
self._turtle.pos = vp.vector(newX, newY, newZ)
self._turtle.axis = vector2VP(self._velocity * 3)
```

- The `_avoid`, `_center`, and `_match` methods can remain mostly the same, except that you will need to replace instances of `Pair` with `Vector`. Also, have the `_avoid` method avoid the light in addition to avoiding other boids. When getting the position of the light, you will need the `VP2vector` function above.
- Write a new method named `_light` that returns a unit vector pointing toward the current position of the light. Incorporate this vector into your `step` method with another weight

```
LIGHT_WEIGHT = 0.3
```

The main simulation

Once you have completed the steps above, you can remove the turtle graphics setup, and simplify the main program to the following.

```
from world import *
from boid import *
import vpython as vp

WIDTH = 60
HEIGHT = 60
DEPTH = 60
NUM_MOTHS = 20

def main():
    sky = World(WIDTH, HEIGHT, DEPTH)
    for index in range(NUM_MOTHS):
        moth = Boid(sky)

    while True:
        vp.rate(25)    # 1 / 25 sec elapse between computations
        sky.stepAll()

main()
```