Coding Example: Implement the behavior of Boids (Python approach)

In this lesson we will try to implement the Boids class using the traditional Pythonic approach and analyze it in terms of efficiency.

WE'LL COVER THE FOLLOWING

Boid Class Implementation (Python)

Since each boid is an autonomous entity with several properties such as position and velocity, it seems natural to start by writing a `Boid` class:

```
import math
import random
from vec2 import vec2

class Boid:
    def __init__(self, x=0, y=0):
        self.position = vec2(x, y)
        angle = random.uniform(0, 2*math.pi)
        self.velocity = vec2(math.cos(angle), math.sin(angle))
        self.acceleration = vec2(0, 0)
```

The `vec2` object is a very simple class that handles all common vector operations with 2 components. It will save us some writing in the main `Boid` class. Note that there are some vector packages in the Python Package Index, but that would be an overkill for such a simple example.

Boid is a difficult case for regular Python because a boid has interaction with local neighbors. Since the boids are moving so, at every step, we need to calculate the distance of one boid with every other boid that comes within the interaction radius and then sort those distances. The prototypical way of

interaction radius and then sort those distances. The prototypical way of writing the three rules is thus something like:

```
def separation(self, boids):
  count = 0
  for other in boids:
    d = (self.position - other.position).length()
    if 0 < d < desired_separation:
      count += 1
      # ...
  if count > 0:
    # ...

def alignment(self, boids): # ...
def cohesion(self, boids): # ...
```

To complete the picture, we can also create a `Flock` object:

```
class Flock:
  def __init__(self, count=150):
    self.boids = []
    for i in range(count):
      boid = Boid()
      self.boids.append(boid)

  def run(self):
    for boid in self.boids:
      boid.run(self.boids)
```

Complete Solution

Merging all the logic from above, given below is the complete implementation of Boids.

main.py

vec2.py

```
# ------------------------------------------------------------------------
# From Numpy to Python
# Copyright (2017) Nicolas P. Rougier - BSD license
# More information at https://github.com/rougier/numpy-book
# ------------------------------------------------------------------------
import math
import random
from vec2 import vec2


class Boid:
    def __init__(self, x, y):
        self.acceleration = vec2(0, 0)
        angle = random.uniform(0, 2*math.pi)
```

```
        self.velocity = vec2(math.cos(angle), math.sin(angle))
        self.position = vec2(x, y)
        self.r = 2.0

        self.max_velocity = 2
        self.max_acceleration = 0.03

    def seek(self, target):
        desired = target - self.position
        desired = desired.normalized()
        desired *= self.max_velocity
        steer = desired - self.velocity
        steer = steer.limited(self.max_acceleration)
        return steer

    # Wraparound
    def borders(self):
        x, y = self.position
        x = (x+self.width) % self.width
        y = (y+self.height) % self.height
        self.position = vec2(x,y)

    # Separation
    # Method checks for nearby boids and steers away
    def separate(self, boids):
        desired_separation = 25.0
        steer = vec2(0, 0)
        count = 0

        # For every boid in the system, check if it's too close
        for other in boids:
            d = (self.position - other.position).length()
            # If the distance is greater than 0 and less than an arbitrary
            # amount (0 when you are yourself)
            if 0 < d < desired_separation:
                # Calculate vector pointing away from neighbor
                diff = self.position - other.position
                diff = diff.normalized()
                steer += diff/d  # Weight by distance
                count += 1       # Keep track of how many

        # Average - divide by how many
        if count > 0:
            steer /= count

        # As long as the vector is greater than 0
        if steer.length() > 0:
            # Implement Reynolds: Steering = Desired - Velocity
            steer = steer.normalized()
            steer *= self.max_velocity
            steer -= self.velocity
            steer = steer.limited(self.max_acceleration)

        return steer

    # Alignment
    # For every nearby boid in the system, calculate the average velocity
    def align(self, boids):
        neighbor_dist = 50
        sum = vec2(0, 0)
        count = 0
        for other in boids:
            d = (self.position - other.position).length()
```

```
            if 0 < d < neighbor_dist:
                sum += other.velocity
                count += 1

        if count > 0:
            sum /= count
            # Implement Reynolds: Steering = Desired - Velocity
            sum = sum.normalized()
            sum *= self.max_velocity
            steer = sum - self.velocity
            steer = steer.limited(self.max_acceleration)
            return steer
        else:
            return vec2(0, 0)

    # Cohesion
    # For the average position (i.e. center) of all nearby boids, calculate
    # steering vector towards that position
    def cohesion(self, boids):
        neighbor_dist = 50
        sum = vec2(0, 0)  # Start with empty vector to accumulate all positions
        count = 0
        for other in boids:
            d = (self.position - other.position).length()
            if 0 < d < neighbor_dist:
                sum += other.position  # Add position
                count += 1
        if count > 0:
            sum /= count
            return self.seek(sum)
        else:
            return vec2(0, 0)

    def flock(self, boids):
        sep = self.separate(boids)  # Separation
        ali = self.align(boids)  # Alignment
        coh = self.cohesion(boids)  # Cohesion

        # Arbitrarily weight these forces
        sep *= 1.5
        ali *= 1.0
        coh *= 1.0

        # Add the force vectors to acceleration
        self.acceleration += sep
        self.acceleration += ali
        self.acceleration += coh

    def update(self):
        # Update velocity
        self.velocity += self.acceleration
        # Limit speed
        self.velocity = self.velocity.limited(self.max_velocity)
        self.position += self.velocity
        # Reset acceleration to 0 each cycle
        self.acceleration = vec2(0, 0)

    def run(self, boids):
        self.flock(boids)
        self.update()
        self.borders()
```

```
class Flock:
    def __init__(self, count=150, width=640, height=360):
        self.width = width
        self.height = height
        self.boids = []
        for i in range(count):
            boid = Boid(width/2, height/2)
            boid.width = width
            boid.height = height
            self.boids.append(boid)

    def run(self):
        for boid in self.boids:
            # Passing the entire list of boids to each boid individually
            boid.run(self.boids)

    def cohesion(self, boids):
        P = np.zeros((len(boids),2))
        for i, boid in enumerate(self.boids):
            P[i] = boid.cohesion(self.boids)
        return P




import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

n=50
flock = Flock(n)
P = np.zeros((n,2))

def update(*args):
    flock.run()
    for i,boid in enumerate(flock.boids):
        P[i] = boid.position
    scatter.set_offsets(P)

fig = plt.figure(figsize=(8, 5))
ax = fig.add_axes([0.0, 0.0, 1.0, 1.0], frameon=True)
scatter = ax.scatter(P[:,0], P[:,1],
                     s=30, facecolor="red", edgecolor="None", alpha=0.5)

ax.set_xlim(0,640)
ax.set_ylim(0,360)
ax.set_xticks([])
ax.set_yticks([])
Writer = animation.writers['ffmpeg']
writer = Writer(fps=15, metadata=dict(artist='Me'), bitrate=1800)
anim = animation.FuncAnimation(fig, update, interval=10)
anim.save('output/output.mp4', writer=writer)
plt.show()
```




Drawbacks

Using this approach, we can have up to 50 boids until the computation time becomes too slow for a smooth animation. As you may have guessed, we can do much better using NumPy, but let me first point out the main problem with this Python implementation.

- If you look at the code, you will certainly notice there is a lot of redundancy. More precisely, we do not exploit the fact that the *Euclidean* distance is reflexive, that is, $|x - y| = |y - x|$.
- In this naive Python implementation, each rule (function) computes n^2 distances while $\frac{n^2}{2}$ would be sufficient if properly cached.
- Furthermore, each rule re-computes every distance without caching the result for the other functions. In the end, we are computing $3n^2$ distances instead of $\frac{n^2}{2}$.

Now let's see what NumPy has to offer to give a better and more efficient solution. See you in the next lesson!