Spike: Task 14

Title: Agent Marksmanship

Author: Sam Huffer, 101633177

Goals / Deliverables

- Basic deliverables:
  - An agent targeting simulation with:
    - An attacking agent (stationary or otherwise)
    - A moving target agent (can move between two points) that shows when it has been hit
    - Several weapons that can successfully hit the target, including:
      - Rifle (fast, accurate)
      - Rocket (slow, accurate)
      - Hand gun (fast, inaccurate)
      - Hand grenade (slow, inaccurate)
- Extensions:
  - Account for rate of fire and effect range (specifically, make a shotgun with a low rate of fire and short but deadly effective range; move the attacker closer before shooting).
  - Splash damage (projectile that explodes on the ground; attacker should take this into account when aiming).
  - A target that can avoid slow projectiles and / or move to hiding spots when attacked.

Technologies, Tools, and Resources Used

- SublimeText (for editing, executing and testing the code)
- Learning materials on Canvas (for instructions and sample code)

Tasks Undertaken

- I copied the Tactical Steering project from Task 10 into the Task 14: Agent Marksmanship folder, then cleaned up some of the existing code to be better suited for the current scenario, and started putting together a skeleton for the methods required by this task. I also added the walls from Task 11: Emergent Group Behaviour, tweaking existing methods and adding the appropriate calls in main, world, agent, etc. as needed to adjust them for the current task, and added keyboard controls for toggling the obstacles and walls on and off.
- I set up the target and shooter to start in stationary positions and for the latter to shoot the former from those stationary positions. Projectiles die on contact with the target or when they pass outside the bounds of the screen, and are pooled inside the originating agent rather than being instantiated when fired and destroyed on contact to improve program efficiency. I then implemented inaccuracy (within a specified margin of error) and tweaked the agents to show when they've been shot by going red for 0.1 seconds.
- I set up "moving back and forth" and "wander" movement types for the target, and predictive shooting for the shooter. I tried using a fancy method that uses sine and cosine functions, but that failed miserably. Currently, the shooter calculates time it'd take the projectile to get to the current / currently-calculated-future position, then predicts the target's position at that new time using $d = ut + \frac{1}{2} at^2$, and wraps position if the target is wandering. The shooter

iterates through this process until it thinks the projectile will be able to get within an acceptable distance of the target.

- I tweaked hiding spots, fleeing and hunting behaviours, and field of view updating to work with a single hunter and evader, and to fit with this task's logic for when to move and what movement to take.
- I set up explosions for rockets and hand grenades, and automatic firing, rate of fire, and effective range (while the target is evading) for all weapons, using raw rounds per second and scaled effective range values from equivalent *Halo: Combat Evolved* weapons a baseline where possible, and estimating from there how fast I'd realistically fire them if I was trying to be accurate. While doing so, I also set up the shotgun configuring it to fire a scattershot of 5 pellets at once.
- Lastly, I implemented projectile avoidance for the target when it is not stationary, adapting the obstacle avoidance code and adding a method that gets the agent to move perpendicularly to the vector it would take if moving towards the projectile.

Instructions for Operating the Code

- A: toggle the display of agents' obstacle avoidance range if it would otherwise not be displayed.
- B: toggle walls (i.e. boundaries of the simulation space) on and off
- I: toggle the display of agents' force, velocity and net desired change in position.
- N: create a new obstacle in a random but valid position.
- O: toggle obstacles and hiding spots on and off.
- P: pause or un-pause the game.
- R: reposition all obstacles in random but valid positions. Obstacles are automatically repositioned when the window changes size.
- T: scroll through target movement types (stationary, moving between two points, evading).
- W: scroll through shooter weapons.
- Escape: exit the game.

Code Snippets

```
def update(self, delta):
    ''' update vehicle position and orientation '''
    if self.hit_time is not None and (datetime.now() - self.hit_time).total_seconds() > 0.1:
        self.hit_time = None

    self.obst_detected = False
    self.sensor_obst_detected = False

    if self.moving():
        self.move(delta)

    if self.mode == 'shooter':
        if self.world.target.sub_mode == 'Evading' and self.world.obstacles_enabled and not self.see_target:
            return
        elif (datetime.now() - self.last_shot).total_seconds() > self.cooldown:
            self.last_shot = datetime.now()
            self.shoot(self.world.target)

def moving(self): ...

def move(self, delta): ...

def calculate(self, delta): ...

def calculate_target(self, delta):
    if self.sub_mode == 'Pacing' or self.sub_mode == 'Evading':
        avoid = self.avoid_projectiles(self.world.projectiles)
        if avoid.length() > 0:
            print('dodging projectile')
            return avoid

        if self.sub_mode == 'Pacing':
            return self.pace(delta)
        elif self.sub_mode == 'Evading':
            if self.world.obstacles_enabled:
                return self.hide(self.world.shooter, self.world.hiding_spots, delta)
            else:
                return self.flee(self.world.shooter.pos, delta)

    elif self.vel.length() > 0:
        return self.vel * -1
    else:
        return Vector2D(0,0)

def calculate_shooter(self, delta):
    self.update_fov(self.world.obstacles, self.world.agents)

    if self.world.target.sub_mode == 'Evading':
        if self.world.obstacles_enabled:
            return self.hunt(self.world.target, delta)
        else:
            return self.seek(self.world.target.pos)

    return Vector2D(0,0)
```

Figure 1: the code for determining how each agent should move, and if the shooter is able to shoot again.

```
def get_future_pos_with_accel(self, start_vel, accel, time):
    displacement = (start_vel * time) + (0.5 * accel * time * time) #d = ut + 0.5att
    return displacement

def shoot(self, target):
    slow_speed = 250
    fast_speed = 1000
    mod = 50
    # check if any projectiles are still pooled
    if len(self.projectile_pool) > 0:
        # get speed according to weapon type
        if self.sub_mode == 'Rocket' or self.sub_mode == 'Hand Grenade':
            p_speed = slow_speed
        else:
            p_speed = fast_speed
        print("stationary: " + str(target.pos))
        if target.sub_mode == 'Stationary':
            target_pos = target.pos.copy()
        else:
            ''' this behaviour predicts where an agent will be in time T and seeks towards that point to intercept it. ''
            to_target = target.pos - self.pos
            future_target_pos = target.pos.copy()
            dist = to_target.length()
            loop = True
            loop_count = 0
            while loop:     # loops its predictive logic several times to get progressively better predictions. If it wou
                loop_count += 1
                future_time = (future_target_pos - self.pos).length()/(p_speed)     # first loop: current target.pos
                future_target_pos = target.pos + self.get_future_pos_with_accel(target.vel, target.accel, future_time)
                if target.sub_mode == 'Evading':        # if it's not evading, it's not gonna move outside the bounds of
                    self.world.wrap_around(future_target_pos)
                vel_to_future_pos = (future_target_pos - self.pos).normalise() * p_speed
                future_self_pos = self.pos + vel_to_future_pos * future_time
                new_dist = (future_target_pos - future_self_pos).length()
                if new_dist < dist: # keep iterating?
                    dist = new_dist
                    if dist < target.radius * 0.1:      # is it's predicted pos close enough?
                        print('dist between predicted positions less than 0.1 times the targets radius')
                        loop = False
                else:
                    loop = False
            target_pos = future_target_pos
            print("predictive: " + str(target_pos))
            print("loop count: " + str(loop_count))
        if self.world.target.sub_mode == 'Evading' and self.distance(target_pos) > self.effective_range + target.radius:
            return
        if self.sub_mode == 'Shotgun':
            loop = 5    # shot gun scatters multiple projectiles
        else:
            loop = 1    # everything else fires one projectile at a time
        while loop > 0:
            # affect accuracy if using an inaccurate weapon
            if self.sub_mode == 'Hand Gun' or self.sub_mode == 'Hand Grenade' or self.sub_mode == 'Shotgun':
                target_pos.x = int(randrange(int(target_pos.x - mod), int(target_pos.x + mod)))
                target_pos.y = int(randrange(int(target_pos.y - mod), int(target_pos.y + mod)))
            # calculate velocity
            p_heading = (target_pos - self.pos).normalise()
            p_vel = p_heading * p_speed
            # set up and shoot projectile
            p = self.projectile_pool[0]
            self.projectile_pool.remove(p)
            p.vel = p_vel
            p.pos = self.pos.copy()
            p.p_type = self.sub_mode
            if self.sub_mode == 'Hand Grenade':
                p.target = target_pos   # where is the grenade gonna land?
            self.world.projectiles.append(p)    # add new projectile
            loop -= 1
```

Figure 2: the logic for what properties the projectile will have, and calculating where the shooter should aim its shot.

```
def update(self, delta):
    if not self.left_barrel and self.distance(self.shooter.pos) > self.shooter.radius + self.radius:
        self.left_barrel = True

    if not self.exploding:
        ''' update projectile position '''
        self.pos += self.vel * delta

        if self.pos.x < 0 or self.pos.x > self.world.cx or self.pos.y < 0 or self.pos.y > self.world.cy:
            self.world.destroy_projectile(self)

        collided = self.collided()

        if len(collided) > 0:
            if self.p_type == 'Rocket' or self.p_type == 'Hand Grenade':
                for collision in collided:
                    if collision in self.world.agents:
                        collision.hit_time = datetime.now()

                self.target = None
                self.exploding = True
                self.explosion_time = datetime.now()
            else:
                for collision in collided:
                    if collision in self.world.agents:
                        collision.hit_time = datetime.now()

                self.world.destroy_projectile(self)
    else:
        self.radius += 0.1 * self.scale_scalar * self.explosion_multiplier

        collided = self.collided()

        if len(collided) > 0:
            for collision in collided:
                if collision in self.world.agents:
                    collision.hit_time = datetime.now()

        if self.explosion_multiplier == 1 and (datetime.now() - self.explosion_time).total_seconds() > 1:
            self.explosion_multiplier = -5
        elif self.radius <= 0:
            self.world.destroy_projectile(self)
```

Figure 3: the projectile's update method, which handles what to do if it goes outside the screen, collides with another object, or is currently exploding.

```
def avoid_projectiles(self, projectiles):
    def avoid_projectile(self, projectile):
        to_projectile = projectile.pos - self.pos
        desired_vel = to_projectile.perp() * self.max_speed
        print('desired vel is ' + str(desired_vel))
        return (desired_vel - self.vel)

    print('checking for projectiles')

    closest_proj = None
    closest_dist = 9999999999999

    closest_proj_sns = None
    closest_dist_sns = 9999999999999

    result = Vector2D(0,0)

    for projectile in projectiles:
        dist_to_self = self.distance(projectile.pos)
        dist_to_avoid = (projectile.pos - self.sensor_pos).length()

        if dist_to_self < self.avoid_radius + projectile.radius and dist_to_self < closest_dist:
            closest_proj = projectile
            closest_dist = dist_to_self

        if dist_to_avoid < self.avoid_radius + projectile.radius and dist_to_avoid < closest_dist_sns:
            closest_proj_sns = projectile
            closest_dist_sns = dist_to_avoid

    if closest_proj is not None:
        self.obst_detected = True
        result += avoid_projectile(self, closest_proj)

    if closest_proj_sns is not None:
        self.sensor_obst_detected = True
        result += avoid_projectile(self, closest_proj_sns)

    return result
```

Figure 4: the code for the target to check if there are projectiles to dodge, and how to go about dodging them.

In-Simulation Screenshots

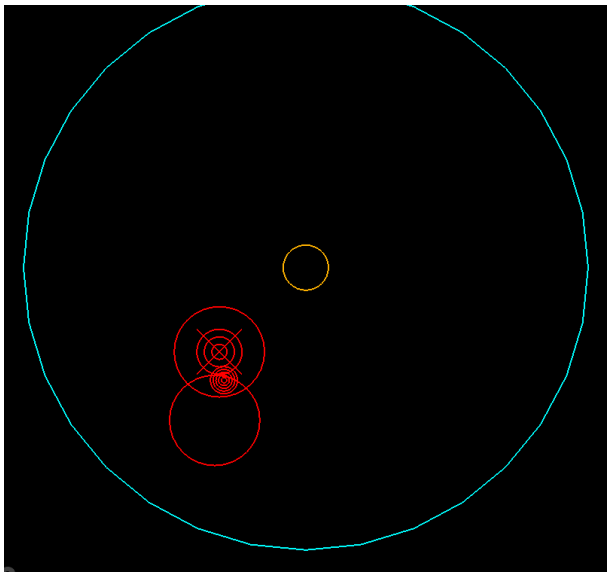

Figure 3: a grenade detonating after colliding with the target.

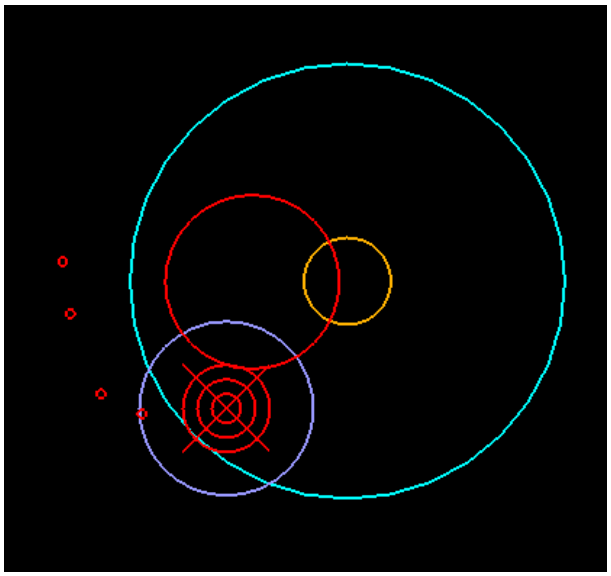

Figure 6: the shooter firing its shotgun at the target.

What I Found Out

- Setting up the different weapon / projectile types was easy enough. Setting up the movement settings and writing the predictive logic, even with the pursuit method as a basis for the latter, was the tricky bit.
- If target changes its velocity or acceleration (either their vector or magnitude), that can render the predicted position inaccurate proportionate to the change in vector or magnitude, particularly if the projectile isn't moving fast enough or shot from a close enough position for the inaccuracy to be negligible.
- Once effective range while the target was wandering was implemented, if obstacles were disabled and the shooter thought target's predicted position, after wrapping, would be out of range, the shooter would not shoot; this can look a bit weird if that position is wrapped, as it seems like the target should be fired upon as it's current position is within the weapon's range.
- With projectile avoidance, when projectiles are moving at a fast speed, the target will try to avoid them but almost certainly fail; any effort to avoid them will at most only register as a slight recoil from the projectile as if the projectile pushed the agent away. However, when projectiles are moving at a slower speed, the target will have more success trying to avoid them, doing best when the projectile would have glanced their side if it did hit. Even in an unsuccessful dodging attempt, the agent will have more visibly attempted to dodge the projectile.