SWINBURNE UNIVERSITY OF TECHNOLOGY

COS30002

Soldier on Patrol

Task 15, Week 8
Spike: Spike_No15

Matthew Coulter
S102573957

07/05/2020

Tutor: Tien Pham

Table of Contents

Table of Contents **2**

Goals / Deliverables 3

Technologies, Tools, and Resources used: 3

Tasks undertaken: 4

Copy the path class over 4

Copy the path class over from previous task 4

Initialise a path variable in world 4

Put this randomise path in world class 4

Assign the agent path to world path 4

Add the follow path function into agent: 4

Creating the enemy 5

Here is the code for the enemy: 5

implementing attack system 6

The new weapon class should look like this: 6

The updated projectile class should look like this: 8

High level "patrol" and "attack" modes 9

High level “Shooting” and “reloading” modes when attacking 10

Updating the modes based on conditions 10

Extension: incorporating separation forces 11

What we found out: 12

Multiple agents in patrol mode fly around the saem path with distance: 12

When we spawn an enemy in, their mode updates to attack: 13

The enemy dies when it has no health and the agent begin patrolling again: 14

When an agent has no ammo left, it will begin to hide behind other dead bodies: 15

(Problem) Agent may get stuck shooting a dead body as it is inline with an alive enemy 16

Goals / Deliverables

The aim of this task is to implement a finite state machine (FSM) into our agent simulation. For this task, I will be implementing a basic FSM of two states: attacking and patrolling. The finite state machine works by having a condition/s that will update the state of an object. In this case, when there are enemies that exist, the state of the agent will be updated to attack it. Once the enemy is killed, the state will be returned back to patrol.

Technologies, Tools, and Resources used:

If you wish to reproduce my work, here is a list of all the required tools and resources that you will need:

- Visual Studio
- Python language pack
- This spike report
- lab09 code
- Git bash or sourcetree (if you wish to access my code)
    - repo: ‘git clone https://mattcoulter7@bitbucket.org/mattcoulter7/cos30002-102573957.git’

Tasks undertaken:

Copy the path class over

From our task 10 tactical steering, we are going to copy the path class over related functions in agent as the agents will follow the path when they are patrolling. However, instead of having an individual path for each agent, I think it is better to have a path in the world that the agent's path is assigned to, this way they will fly around together in a more realistic scenario.

1. **Copy the path class over from previous task**
2. **Initialise a path variable in world**

```
        # Path for all agents

        self.path = Path(looped = True)

        self.randomise_path()
```

3. **Put this randomise path in world class**

```
    def randomise_path(self):

        cx = self.cx

        cy = self.cy

        margin = min(cx,cy) * 1/6

        self.path.create_random_path(10, margin, margin, cx - margin, cy -
margin,looped = True)
```

4. **Assign the agent path to world path**

```
        # Path to follow
        self.path = world.path
        self.waypoint_threshold = 100.0
```

5. **Add the follow path function into agent:**

```
    def follow_path(self):
        if (self.path.is_finished()):
            # Arrives at the final waypoint
            return self.arrive(self.path._pts[-1])
        else:
            # Goes to the current waypoint and increments on arrival
            to_target = self.path.current_pt() - self.pos
            dist = to_target.length()
            if (dist < self.waypoint_threshold):
                self.path.inc_current_pt()
            return self.arrive(self.path.current_pt())
```

Creating the enemy

The enemy will be a stagnant object looking like this: 
As their health gets lower, their colour will update to indicate their health is getting lower. A cross will be drawn through them once they are dead.

Here is the code for the enemy:

```
from vector2d import Point2D, Vector2D
from graphics import egi
from random import randrange

class Enemy(object):
    """description of class"""
    def __init__(self,world = None,pos = None,scale = 30.0):
        self.world = world
        if pos is not None: self.pos = pos
        else: self.pos = Vector2D(randrange(world.cx),
randrange(world.cy))
        self.scale = scale
        self.alive = True
        self.max_health = randrange(50,200)
        self.health = self.max_health

    def render(self):
        # Get color relative to health → Yellow...Orange...Red
        egi.set_pen_color(color = (1.0, self.health/self.max_health, 0.0,
1))
        # Draw Body
        egi.circle(self.pos,self.scale)
        # Draw Arms
        leftarm = Vector2D(self.pos.x - self.scale,self.pos.y)
        rightarm = Vector2D(self.pos.x + self.scale,self.pos.y)
        egi.circle(leftarm,self.scale / 3)
        egi.circle(rightarm,self.scale / 3)
        # Draw cross to indicate death
        if not self.alive:
            egi.cross(self.pos,self.scale/2)

    def update(self,delta):
        if self.alive:
            if self.health <= 0:
                self.alive = False

    def get_shot(self,damage):
        self.health -= damage
```

Implementing attack system

We have borrowed a lot of the code from our last task to implement the shooting mechanisms. Copy the weapon and projectile class over into your project. However, because we are shooting the enemy instead of agents, we will need to update both the classes. Also, I have added the reloading and rate of fire in as we will have a mode for our agent when it is reloading.

The new weapon class should look like this:

```
from projectile import Projectile
from point2d import Point2D
from graphics import egi, KEY
from queue import Queue
from random import randrange

class Weapon(object):
    """description of class"""
    def __init__(self,agent = None,world = None):
        self.max_ammo = 10
        self.projectiles = []
        self.projectiles_queue = Queue(maxsize = self.max_ammo)
        self.agent = agent
        self.world = world
        self.color = 'WHITE'
        self.shape = [
            Point2D(0.0,  0.0),
            Point2D( 1.0,  0.0),
            Point2D( 1.0, -0.25),
            Point2D( 0.25, -0.25),
            Point2D( 0.25, -0.5),
            Point2D( 0.0, -0.5),
            Point2D( 0.0, 0.0)
        ]
        self.proj_speed = 2000.0
        self.accuracy = 2.0
        self.fire_rate = 20
        self.fire_rate_tmr = self.fire_rate
        self.cooling_down = False # time for reloading and rate of fire
        self.remaining_ammo = self.max_ammo
        self.reloading = False
        self.reloading_time = 300
        self.reloading_tmr = self.reloading_time
        self.initialise_queue(self.projectiles_queue)

    def update(self,delta):
        # Update position of all projectiles
```

```
        for proj in self.projectiles:
            proj.update(delta)

        # Reload when no ammo left
        if self.remaining_ammo == 0:
            self.reloading = True

        # Reload timer
        if self.reloading:
            self.reloading_tmr -= 1
        if self.reloading_tmr == 0:
            self.reloading = False
            self.reloading_tmr = self.reloading_time
            self.remaining_ammo = self.max_ammo
        # Cool down timer between shots
        if self.cooling_down:
            self.fire_rate_tmr -= 1
        if self.fire_rate_tmr == 0:
            self.cooling_down = False
            self.fire_rate_tmr = self.fire_rate
```

The update function is updating the fire rate timer preventing the weapon from shooting any faster. It also updates the reload time. This is called within the update function of the agent.

```
    def render(self):
        '''Renders gun to screen'''
        egi.set_pen_color(name=self.color)
        pts = self.world.transform_points(self.shape, self.agent.pos,
self.agent.heading, self.agent.side, self.agent.scale)
        egi.closed_shape(pts)
        for proj in self.projectiles:
            proj.render()

    def shoot(self):
        '''Moves obj from queue into list'''
        # Shoot if queue still has obj to move
        if not self.projectiles_queue.empty() and not self.cooling_down
and not self.reloading:
            # Get obj at front of queue
            proj = self.projectiles_queue.get()
            # Prepare for shooting
            proj.calculate()
            # Put obj in list so it is updated
            self.projectiles.append(proj)
            self.cooling_down = True
            self.remaining_ammo -= 1
```

```
    def initialise_queue(self,queue):
        '''Fills up a queue to its max size'''
        for i in range(queue.maxsize):
            queue.put(Projectile(self.world,self))
```

The attacking should only happen in attack mode. Hence in agent's update function, call shoot when it should shoot and mode is attacking:

```
       if self.mode == 'attack' and self.should_shoot():
            self.weapon.shoot()
```

The updated projectile class should look like this:

```
from graphics import egi
from vector2d import Vector2D
from playsound import playsound

class Projectile(object):
    """description of class"""
    def __init__(self,world,weapon):
        self.world = world
        self.weapon = weapon
        self.pos = Vector2D()
        self.vel = Vector2D()
        self.max_speed = None
        self.damage = 30

    def update(self,delta):
        if self.intersect_edge():
            self.recycle()
        enemy_hit = self.intersect_enemy()
        if enemy_hit:
            enemy_hit.get_shot(self.damage)
            self.recycle()
        # check for limits of new velocity
        self.vel.normalise()
        self.vel *= self.max_speed
        # update position
        self.pos += self.vel * delta
        # Recycle self back into queue if outside of map

    def render(self):
        egi.circle(self.pos,5)

#-----------------------------------------------------------------------

    def intersect_edge(self):
        '''check if projectile goes out of the map'''
```

```
        if self.pos.x < 0:
            return True
        elif self.pos.x > self.world.cx:
            return True
        elif self.pos.y < 0:
            return True
        elif self.pos.y > self.world.cy:
            return True
        return False

    def intersect_enemy(self):
        '''check if projectile hits another agent'''
        for enemy in self.world.enemies:
            to_enemy = enemy.pos - self.pos
            dist = to_enemy.length()
            if dist < enemy.scale:
                return enemy
        return False

    def recycle(self):
        '''remove projectile from list and put it at the end of the queue
for reuse'''
        if self in self.weapon.projectiles:
            self.weapon.projectiles.remove(self)
            self.weapon.projectiles_queue.put(self)

    def calculate(self):
        '''prepare to be put back onto the screen'''
        self.max_speed = self.weapon.proj_speed
        self.pos = self.weapon.agent.pos.copy()
        self.vel = self.weapon.agent.vel.copy()
```

High level "patrol" and "attack" modes

The patrol mode will simply be following the path. Rewrite your calculate() function to look like this:

```
    def calculate(self,delta):
        # calculate the current steering force
        mode = self.mode
        force = Vector2D(0,0)
        if mode == 'patrol':
            force = self.follow_path()
        else:
            force = Vector2D()
        self.force = force
        return force
```

Essentially, when the mode is set to patrol, it will follow the path we assigned earlier.

High level “Shooting” and “reloading” modes when attacking

First, we should add our attacking mode into the calculate function:

```
        elif mode == 'attack':
            force = self.attack()
```

However, when it is attacking, it will need to reload when there is not enough ammunition left in the magazine. Because this takes time, we have implemented the hiding mechanism from task 10.

```
        elif mode == 'hide':
            force = self.hide()
```

Instead of having dedicated hiding objects, we will use the dead bodies as hiding objects! add this hide function to do so:

```
    def hide(self):
        '''Goes to nearest dead body and hides behind it'''
        alive = list(filter(lambda e: e.alive, self.world.enemies))
        enemy = min(alive, key = lambda e: (e.pos - self.pos).length() *
e.health)
        dead = list(filter(lambda e: e.alive == False,
self.world.enemies))
        if dead:
            closest = min(dead, key = lambda e: (e.pos -
self.pos).length())
            # Uses angle to calculate best spot behind the hiding object
            position = enemy.pos - closest.pos
            position.normalise()
            hiding_point = closest.pos - closest.scale * 2 * position
            return self.arrive(hiding_point)
        return Vector2D()
```

Updating the modes based on conditions

If you try to run your code, nothing will change because we haven't implemented anything that will update the conditions causing the calculate function to return a different result. Hence, from the update function, we will call another function called check_state which will update the mode if necessary. It is nice to have it in a separate location as it will make it easier when adding further modes to the game.

```
    def check_state(self):
        ''' check if state should be updated '''
        alive_enemies = list(filter(lambda e: e.alive == True,
self.world.enemies))
        if alive_enemies:
            if self.weapon.reloading:
                self.mode = 'hide'
            else:
                self.mode = 'attack'
        else:
            self.mode = 'patrol'
```

It is imperative the check_state is called at the beginning of the update function so there are no delays with forces:

```
        ''' Check if state needs to be updated '''
        self.check_state()
        self.weapon.update(delta)
```

Spawning in an enemy:

World will contain another list for the enemies. Will append another enemy to that list to spawn them in:

```
def on_mouse_press(x, y, button, modifiers):
    if button == 1:  # left
        world.enemies.append(Enemy(world,pos = Vector2D(x,y)))
```

```
def on_key_press(symbol, modifiers):
    elif symbol == KEY.SPACE:
        world.enemies.append(Enemy(world))
```

Extension: incorporating separation forces

There was a problem with al;l the agents following the same path that needed to be addressed. They all eventually overlap as the saem forces directional forces are being applied to all of them. Hence, We should borrow the separation sum force from task 11. Here is the code for that:

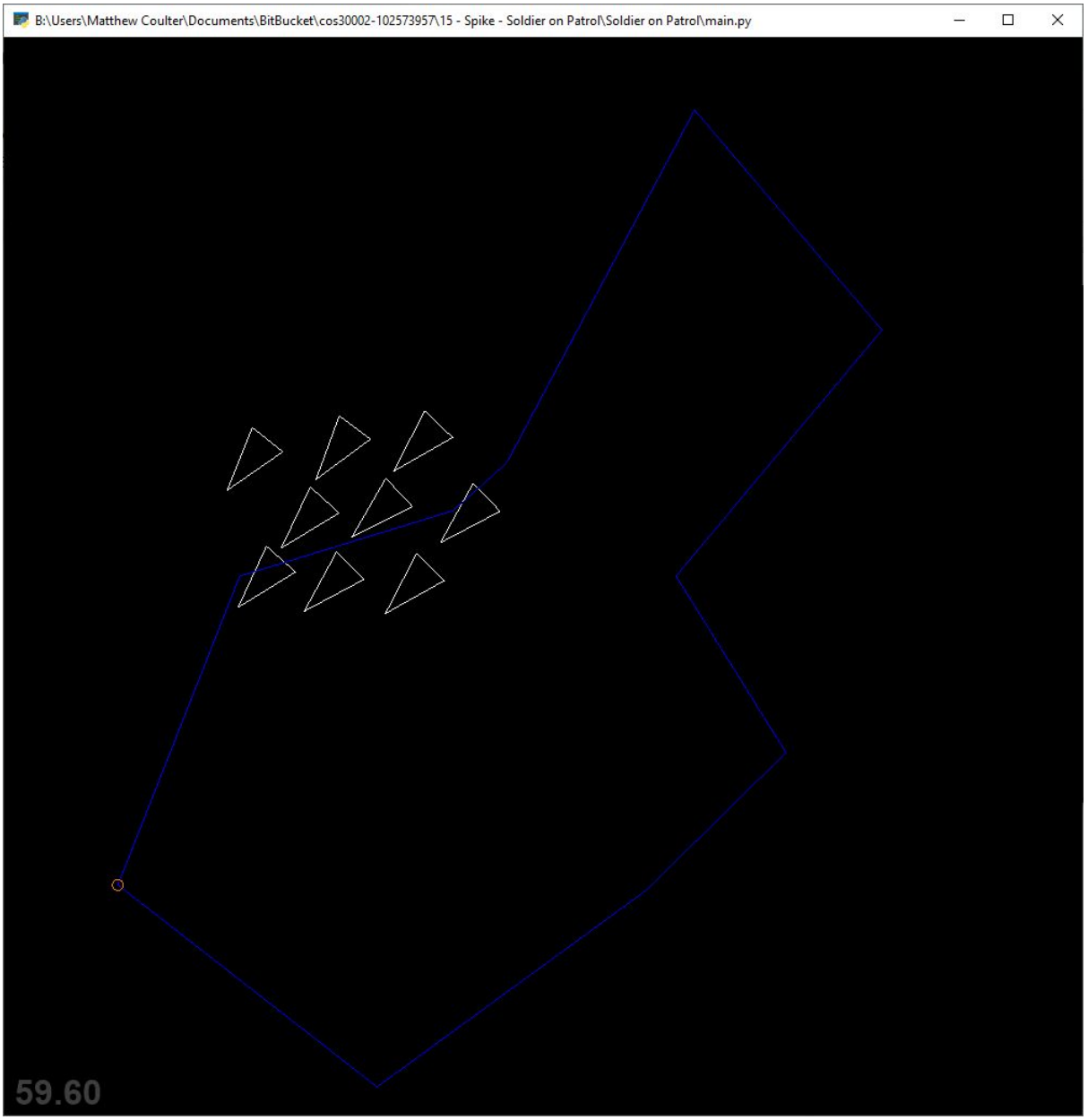
```
   def separate(self):
        # Moves away from nearby agent
        closeby = list(filter(lambda a: a is not self and (a.pos -
self.pos).length() < self.separation,self.world.agents))
        if (closeby):
            closest_agent_pos = min(closeby,key = lambda a: (a.pos -
self.pos).length()).pos
            target = (2 * self.pos - closest_agent_pos)
            to_target = target - self.pos
            length = to_target.copy().length()
            to_target.normalise()
            to_target *= self.separation - length
            return to_target
        return Vector2D()
```

I actually made a few changes to this function.

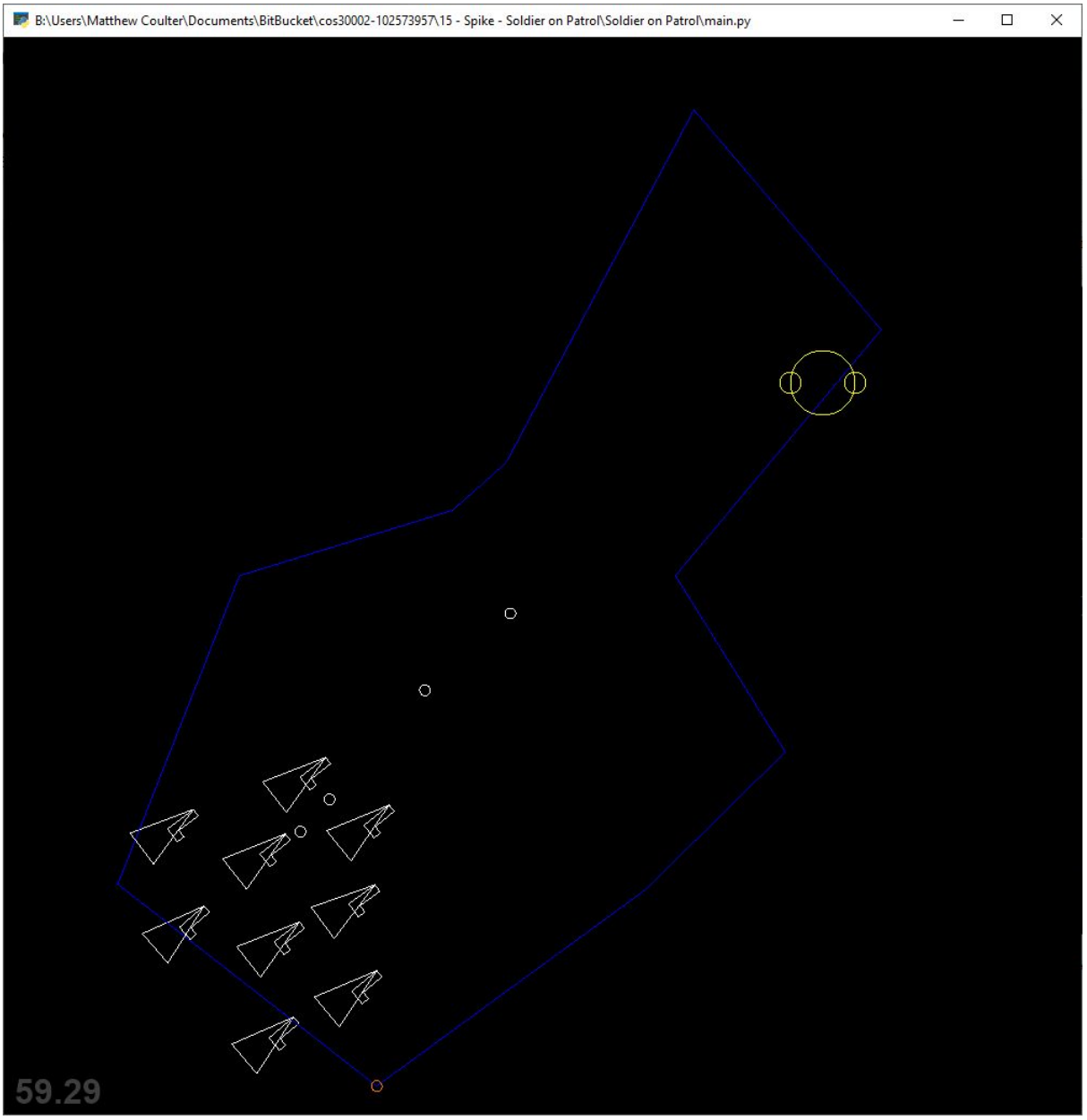
1. Previously, when the agents were closer together, the direction was right but the force was only the distance between them. What it should have been is the separation constant - that distance so when they are closer together it will pull them farther away.
2. I removed the whole closest() function because it can be done much simpler with lambda functions.

What we found out:

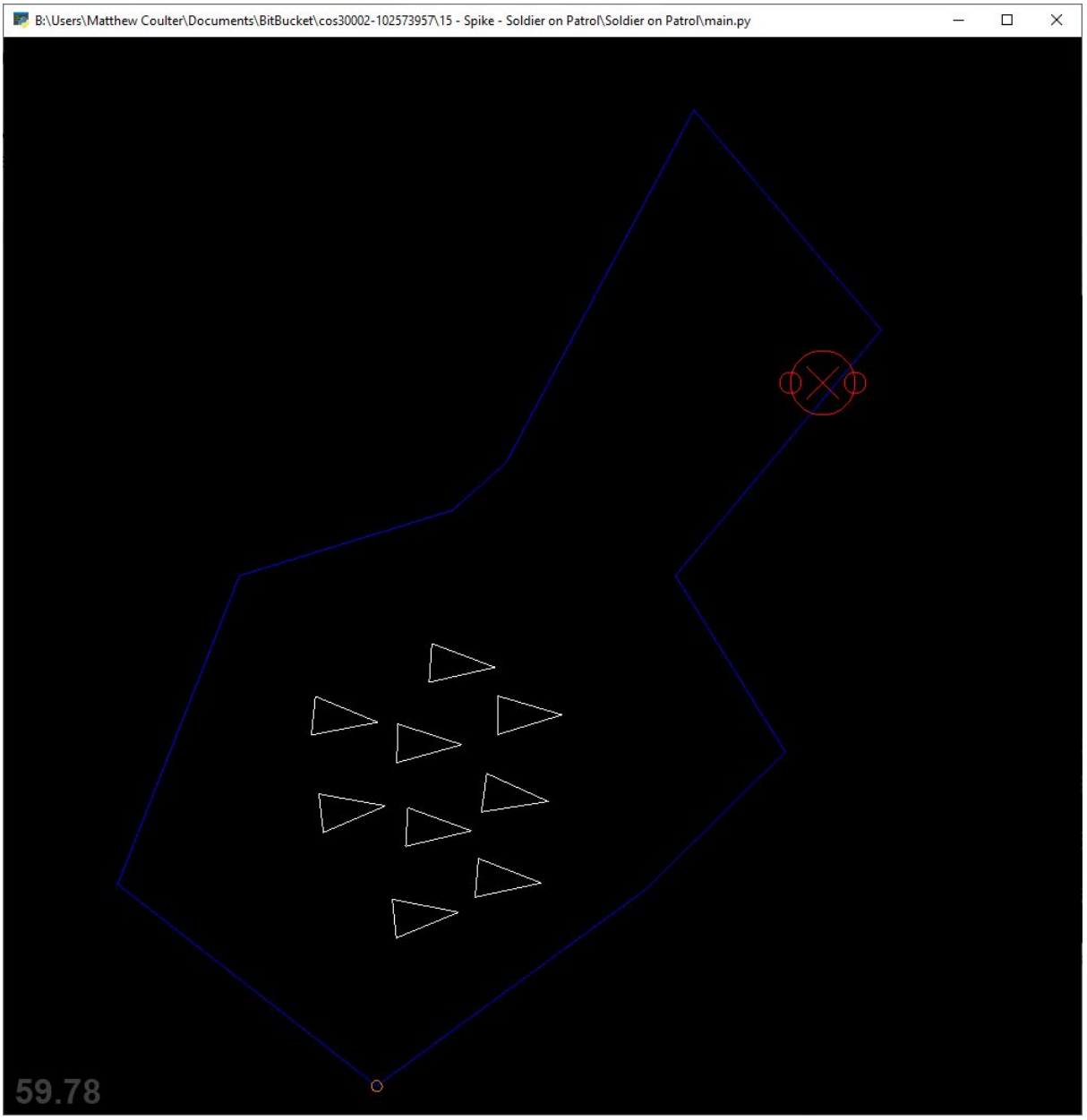
Multiple agents in patrol mode fly around the saem path with distance:



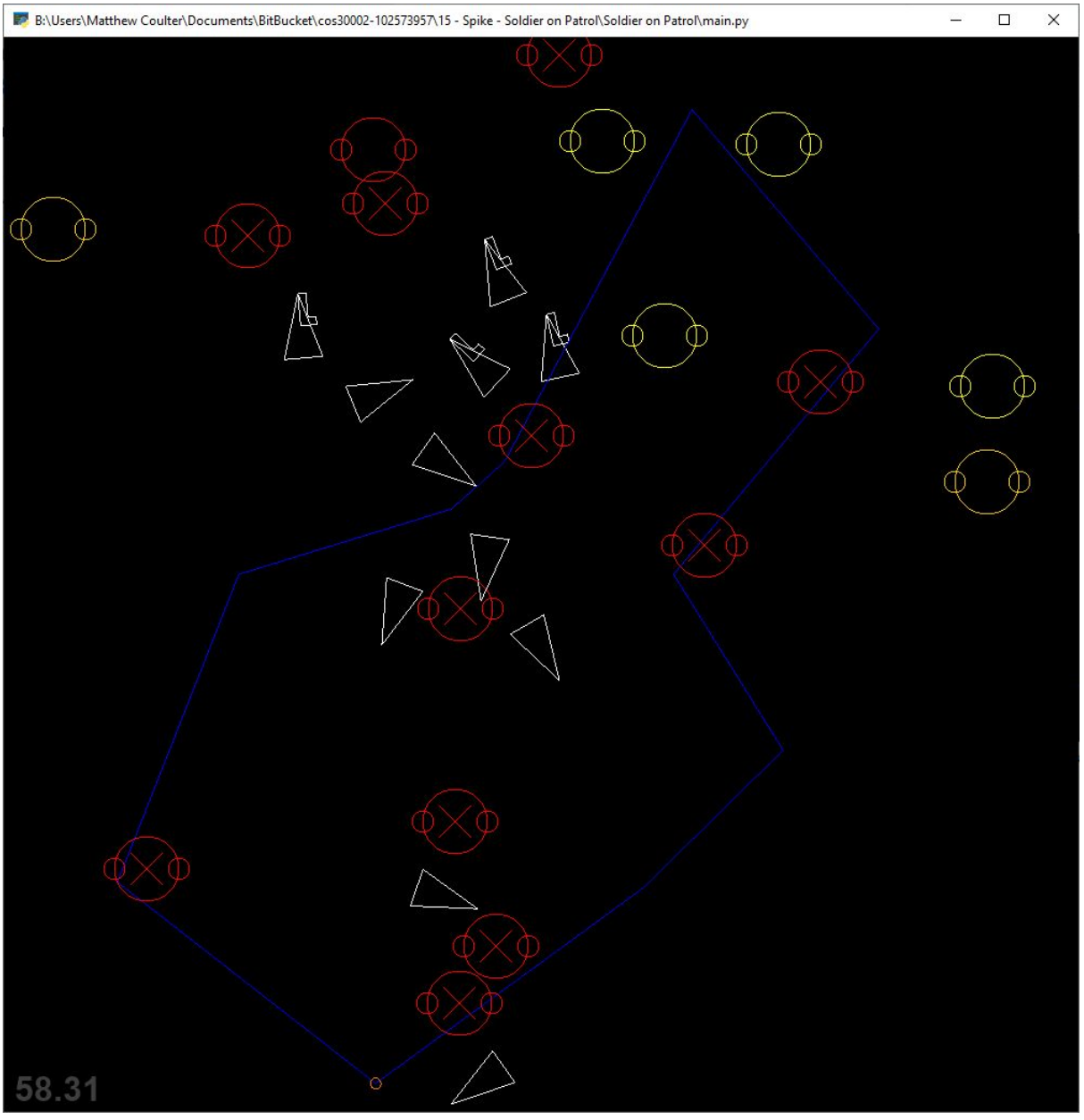
When we spawn an enemy in, their mode updates to attack:



The enemy dies when it has no health and the agent begin patrolling again:



When an agent has no ammo left, it will begin to hide behind other dead bodies:



The ones with guns in their hand are still attacking, the agents near the bottom of the screen don't have guns and are reloading and are in the hiding spot behind the dead body. The agents in the middle of the screen are also reloading but are approaching a hiding spot.

(Problem) Agent may get stuck shooting a dead body as it is inline with an alive enemy