OBSTACLE AVOIDANCE USING A-STAR AND DEEP Q-NETWORK REINFORCEMENT LEARNING

ABHIRAJ CHAUDHARY

16BCE1385

INTERNED AT : INSTITUTE OF SYSTEMS STUDIES AND ANALYSIS , DRDO, NEW DELHI

INTRODUCTION

Nowadays, use of drones, robots or aerial vehicles have increased by the armed forces or the intelligence services for gathering information, purpose of combat or monitoring various situations or keeping an eye along the line of control. There are times, when some actions need to be taken, without getting in surveillance of the enemy, or neighboring countries using UAVs or aerial vehicles. In the above-mentioned scenario escaping various obstacles put in the path needs to be avoided, so that the target is reached and desired action to be performed without raising any alarms, whether its avoiding enemies' radars, or various infrared equipped gadgets or any other type of obstacle. If a system is in place to find the best path or getting information about location of all obstacles beforehand, it will help the armed forces in executing their actions with more precision and higher efficiency.

The main aim of this project is obstacle avoidance, where obstacle can be in the form of radars or any other type of equipment's. This was achieved using A-Star algorithm, which draws the best possible path through static obstacles for most time saving and efficient actions to be executed and Deep Q-Network Reinforcement Learning, which constantly learns from dynamic obstacles, and after a certain period of time of training, it is able to avoid all dynamic obstacles. Further on visualization of both algorithms was done using Pygame for A- Star approach and Director for Deep Q-Network Reinforcement Learning.

The main challenge in implementing this approach was to the right type of algorithms, which was overcome by insights the project guide and research manuals. Further on choosing the algorithms, main obstacle was to develop the algorithm obstacle avoidance, keeping in mind various scenarios.

Other time consuming issue was, to choose the appropriate way for visualization of the approaches, because the user won't require any type of data, user requires visualization of their object (aircraft, UAV, etc.) avoiding obstacles, so that they are able to have the information before hand and have an insight of the situation. For this, this project uses Pygame which visualizes the path to be taken avoiding static obstacles and Director which helps the object in detecting obstacles and learning to avoid them.

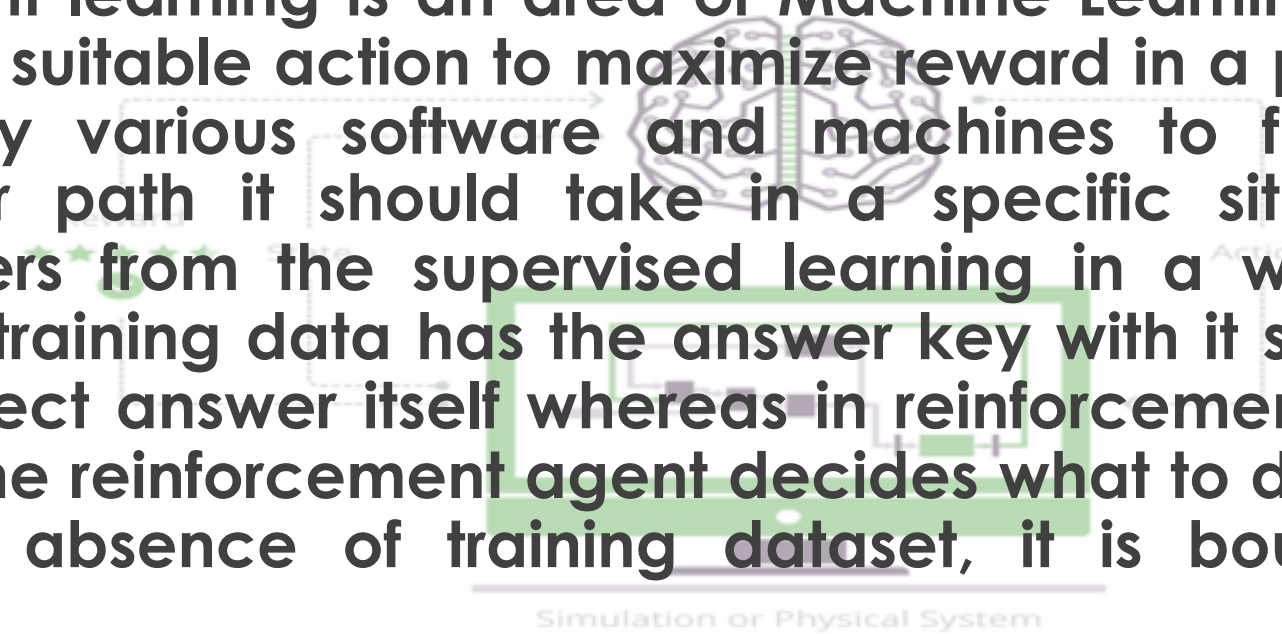
PRE-REQUISITE KNOWLEDGE

A-STAR

A-Star is one of the most popular methods for finding the shortest path between two locations in a mapped area. A* was developed in 1968 to combine heuristic approaches like Best-First-Search (BFS) and formal approaches like Dijsktra's algorithm. The defining characteristics of the A* algorithm are the building of a "closed list" to record areas already evaluated, a "fringe list" to record areas adjacent to those already evaluated, and the calculation of distances travelled from the "start point" with estimated distances to the "goal point". It is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

REINFORCEMENT LEARNING

Reinforcement learning is an area of Machine Learning. Reinforcement. It is about taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible behaviour or path it should take in a specific situation. Reinforcement learning differs from the supervised learning in a way that in supervised learning the training data has the answer key with it so the model is trained with the correct answer itself whereas in reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. In the absence of training dataset, it is bound to learn from its experience.



Q LEARNING

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations. For any finite Markov decision process (FMDP), Q-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" names the function that returns the reward used to provide the reinforcement and can be said to stand for the "quality" of an action taken in a given state. The ‘q’ in q-learning stands for quality. Quality in this case represents how useful a given action is in gaining some future reward.

DEEP Q LEARNING

- Q-learning is a simple yet quite powerful algorithm to know the expected reward for each state for our agent. This helps the agent figure out exactly which action to perform. In situation with a large environment and numerous actions to be taken from Q-Learning is not that feasible. In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output.
- Steps involved in reinforcement learning using deep Q-learning networks (DQNs) are:
- All the past experience is stored by the user in memory
- The next action is determined by the maximum output of the Q-network
- The loss function is mean squared error of the predicted Q-value and the target Q-value – Q*. Hence basically a regression problem. Actual or target values are not known as a reinforcement learning problem is dealt with. Going back to the Q-value update equation derived from the Bellman equation :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

A-STAR ALGORITHM FOR OBSTACLE AVOIDANCE

PART -1

ALGORITHM

PART 1

A-Star works by making a lowest-cost path tree from the start node to the target node. A-Star is better than other alternative algorithms because for each node, A-Star uses a function f(n) that gives an estimate of the total cost of a path using that node. Therefore, A-Star is a heuristic function, which differs from an algorithm in that a heuristic is more of an estimate and is not necessarily provably correct. A-Star expands paths that are already less expensive by using this function:

f(n) = g(n) + h(n),

f(n) = total estimated cost of path through node n

g(n) = cost so far to reach node n

h(n) = estimated cost from n to goal. This is the heuristic part of the cost function

Using a good heuristic is important in determining the performance of A-Star. The value of h(n) would ideally equal the exact cost of reaching the destination. This is, however, not possible because path is not known. Nonetheless, choose a method that will give us the exact value some of the time, such as when traveling in a straight line with no obstacles. This will result in a perfect performance of A-Star in such a case. Primary aim is to select a function h(n) that is less than the cost of reaching our goal. This will allow h to work accurately, if we select a value of h that is greater, it will lead to a faster but less accurate performance. Thus, it is usually the case that we choose an h(n) that is less than the real cost.

If the heuristic function *h* is admissible, meaning that it never overestimates the actual minimal cost of reaching the goal, then A-Star is itself admissible (or optimal) if a closed set is not used. If a closed set is used, then *h* must also be monotonic (or consistent) for A-Star to be optimal. This means that for any pair of adjacent nodes x and *y*, where *d(x,y)* denotes the length of the edge between them, :

$$h(x) \leq d(x, y) + h(y)$$

This ensures that for any path *X* from the initial node to x:

$$L(X) + h(x) \leq L(X) + d(x, y) + h(y) = L(Y) + h(y)$$

where *L* is a function that denotes the length of a path, and *Y* is the path *X* extended to include *y*. In other words, it is impossible to decrease (total distance so far + estimated remaining distance) by extending a path to include a neighbouring node. Monotonicity implies admissibility when the heuristic estimate at any goal node itself is zero, since (letting $P = (f, v_1, v_2, \ldots, v_n, g)$ be a shortest path from any node f to the nearest goal g):

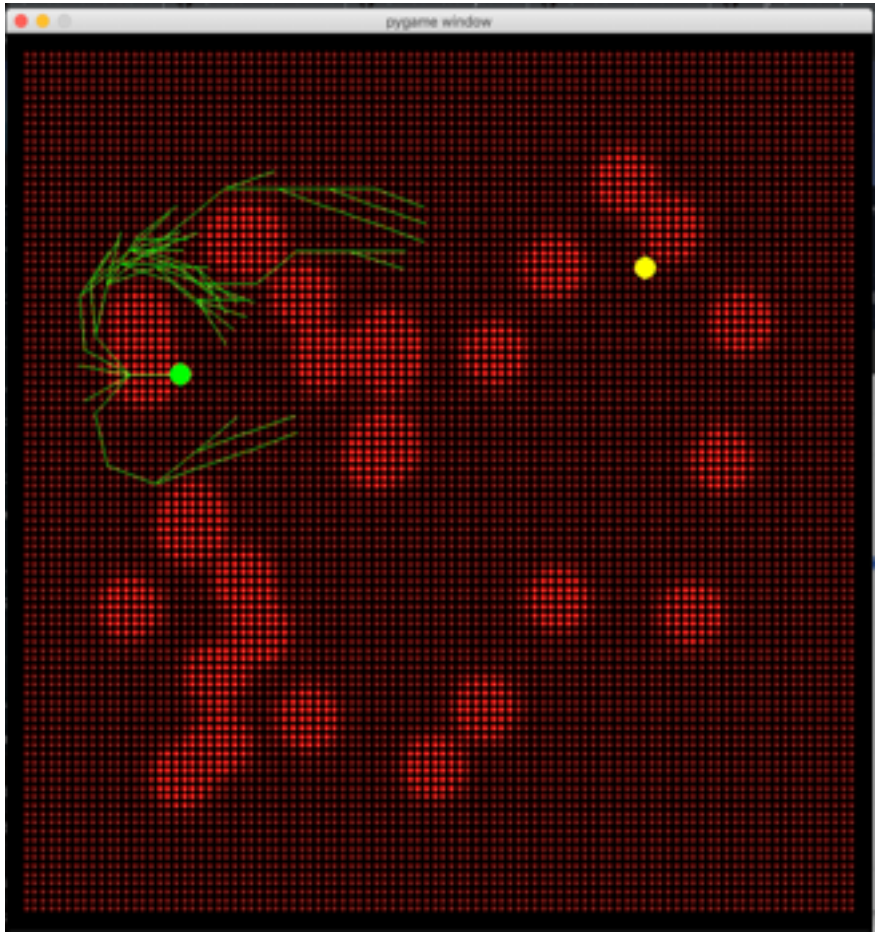
$$h(f) \leq d(f, v_1) + h(v_1) \leq d(f, v_1) + d(v_1, v_2) + h(v_2) \leq \ldots \leq L(P) + h(g) = L(P)$$

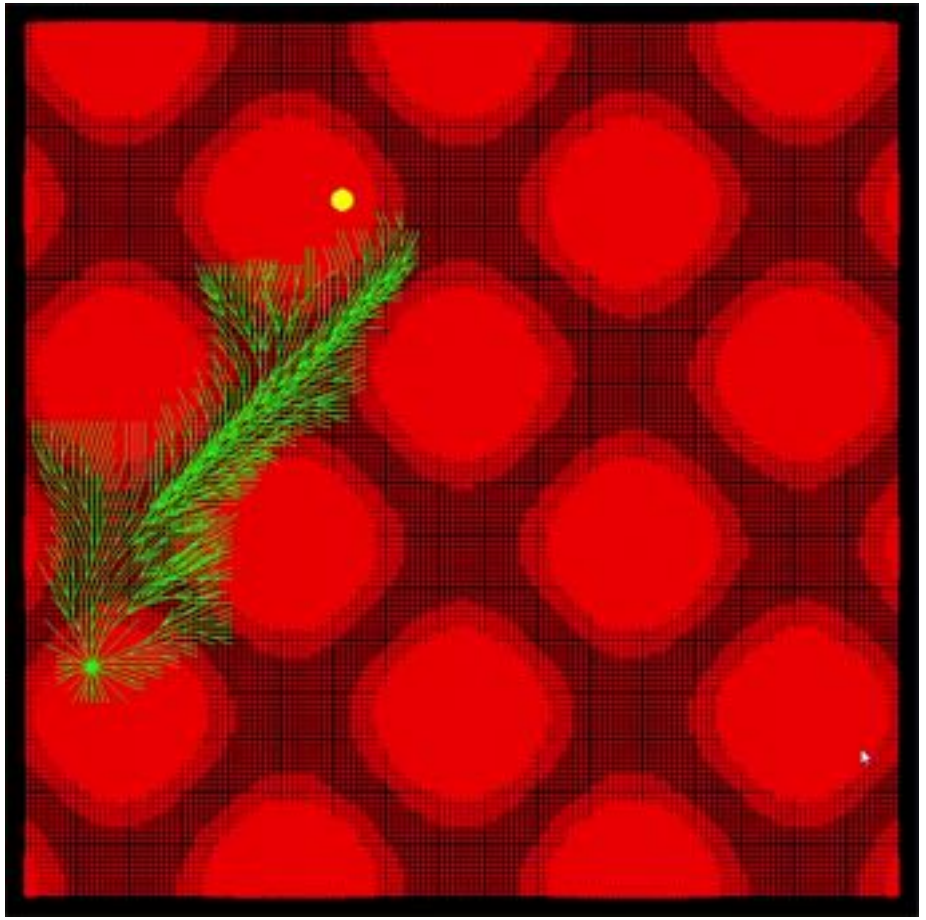
A-Star is also optimally efficient for any heuristic *h*, meaning that no optimal algorithm employing the same heuristic will expand fewer nodes than A-Star, except when there are multiple partial solutions where *h* exactly predicts the cost of the optimal path. A-Star is admissible and, in some circumstances, considers fewer nodes than any other admissible search algorithm with the same heuristic. This is because A-Star uses an "optimistic" estimate of the cost of a path through every node that it considers—optimistic in that the true cost of a path through that node to the goal will be at least as great as the estimate. But, critically, as far as A-Star "knows", that optimistic estimate might be achievable. To prove the admissibility of A-Star, the solution path returned by the algorithm is used as follows:
When A terminates its search, it has found a path whose actual cost is lower than the estimated cost of any path through any open node. But since those estimates are optimistic, A* can safely ignore those nodes. In other words, A* will never overlook the possibility of a lower-cost path and so is admissible.*

IMPLEMENTATION

Implementation starts with defining a map with obstacles, start position and the target. Locations of above are assigned to various nodes, like on a map, following which these nodes are mapped to coordinates on screen. Object starts from a starting position and starts exploring various nodes, using A-Star algorithm. Exploration is represented using green colour and optimal path is displayed using yellow colour between start and target, avoiding all the obstacles. For all visualization of the process, Pygame (library of Python) is used.

OUTPUT





RESULT

- Result from obstacle avoidance using A-Star algorithm is a starting position and a target for an object is set. Following which obstacles (which can be radars, high mountains and etc) are determined in path between target and starting position. It can be clearly seen that algorithm has tried all possible nodes, but has rejected those with obstacles or rather a high cost. Then algorithm starts working and finds the most optimal path to the target, escaping from all the obstacles one by one. In the end, a yellow line is shown to represent the final path i.e. without colliding into any obstacles and shortest possible distance from start to target, hence obstacle avoidance is achieved.

DEEP Q-NETWORK REINFORCEMENT LEARNING AGENT FOR OBSTACLE AVOIDANCE

PART 2

ALGORITHM

PART 2

- A Markov chain is a mathematical model that experiences transition of states with probabilistic rules. A Markov Decision Process (MDP) is an extension of the Markov chain and it is used to model more complex environments. In this extension, add the possibility to make a choice at every state which is called an action. Also add a reward which is a feedback from the environment for going from one state to another taking one action. This is called a stochastic environment (random), in the sense that for one same action taken in the same state, there may be different results.
- The reward is the feedback from the environment that tells us how well something is doing. Goal is to maximize the total reward.

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots$$

- This is the total reward starting at some point t in time. The total reward is the sum of all the immediate rewards we get for taking actions in the environment. Defining reward leads to a few problems that sum can potentially go to infinity, which doesn't make sense since aim is to maximize it or are accounting as much for future rewards as for immediate rewards, hence use a decreasing factor for future rewards.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$$

- Setting γ=1 takes us back to the first expression where every reward is equally important. Setting γ=0 results in only looking for the immediate reward (always acting for the optimal next step). Setting γ between 0 and 1 is a compromise to look more for immediate reward but still account for future rewards.

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \dots) \\ &= r_{t+1} + \gamma R_{t+1} \end{aligned}$$

- A policy is a function that tells what action to take when in a certain state. This function is usually denoted *π(s,a)* and yields the probability of taking action *a* in state *s*. Aim is to find the policy that maximizes the reward function.

$$\sum_a \pi(s,a) = 1$$

- Below is the expected immediate reward for going from state *s* to state *s'* through action *a*.

$$R^a_{ss'} = \mathbb{E}\left[r_{t+1} \quad | \quad s_t = s, a_t = a, s_{t+1} = s'\right]$$

- Below is the transition probability of going from state *s* to state *s'* through action *a*.

$$P^a_{ss'} = \mathbb{P}\left[s_{t+1} = s' \quad | \quad s_t = s, a_t = a\right]$$

- The state value function, and the action value function. These functions are a way to measure the "value", or how good some state is, or how good some action is, respectively

$$V^\pi(s) = \mathbb{E}\left[R_t \quad | \quad s_t = s\right]$$

- The value of a state is the expected total reward we can get starting from that state. It depends on the policy which tells how to take decisions.

$$Q^{\pi}(s,a) = \mathbb{E}\big[R_t \quad | \quad s_t = s, a_t = a\big]$$

- The value of an action taken in some state is the expected total reward we can get, starting from that state and taking that action. It also depends on the policy.
- The expected operator is linear.

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}\big[R_t \quad | \quad s_t = s\big] \\ &= \mathbb{E}\big[r_{t+1} + \gamma R_{t+1} \quad | \quad s_t = s\big] \\ &= \mathbb{E}\big[r_{t+1} \quad | \quad s_t = s\big] + \gamma \mathbb{E}\big[R_{t+1} \quad | \quad s_t = s\big] \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} R^a_{ss'} + \gamma \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} V^{\pi}(s') \end{aligned}$$

- Below, is expansion of the action value function.

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}\big[R_t \quad | \quad s_t = s, a_t = a\big] \\ &= \mathbb{E}\big[r_{t+1} + \gamma R_{t+1} \quad | \quad s_t = s, a_t = a\big] \\ &= \mathbb{E}\big[r_{t+1} \quad | \quad s_t = s, a_t = a\big] + \gamma \mathbb{E}\big[R_{t+1} \quad | \quad s_t = s, a_t = a\big] \\ &= \sum_{s'} P^a_{ss'} R^a_{ss'} + \gamma \sum_{s'} P^a_{ss'} V^{\pi}(s') \end{aligned}$$

- This form of the Q-Value is very generic. It handles stochastic environments

$$Q^{\pi}(s_t, a_t) = r(s_t, a_t) + \gamma V^{\pi}(s_{t+1})$$

- In a greedy policy context, below is relation between the state value and the action value functions.

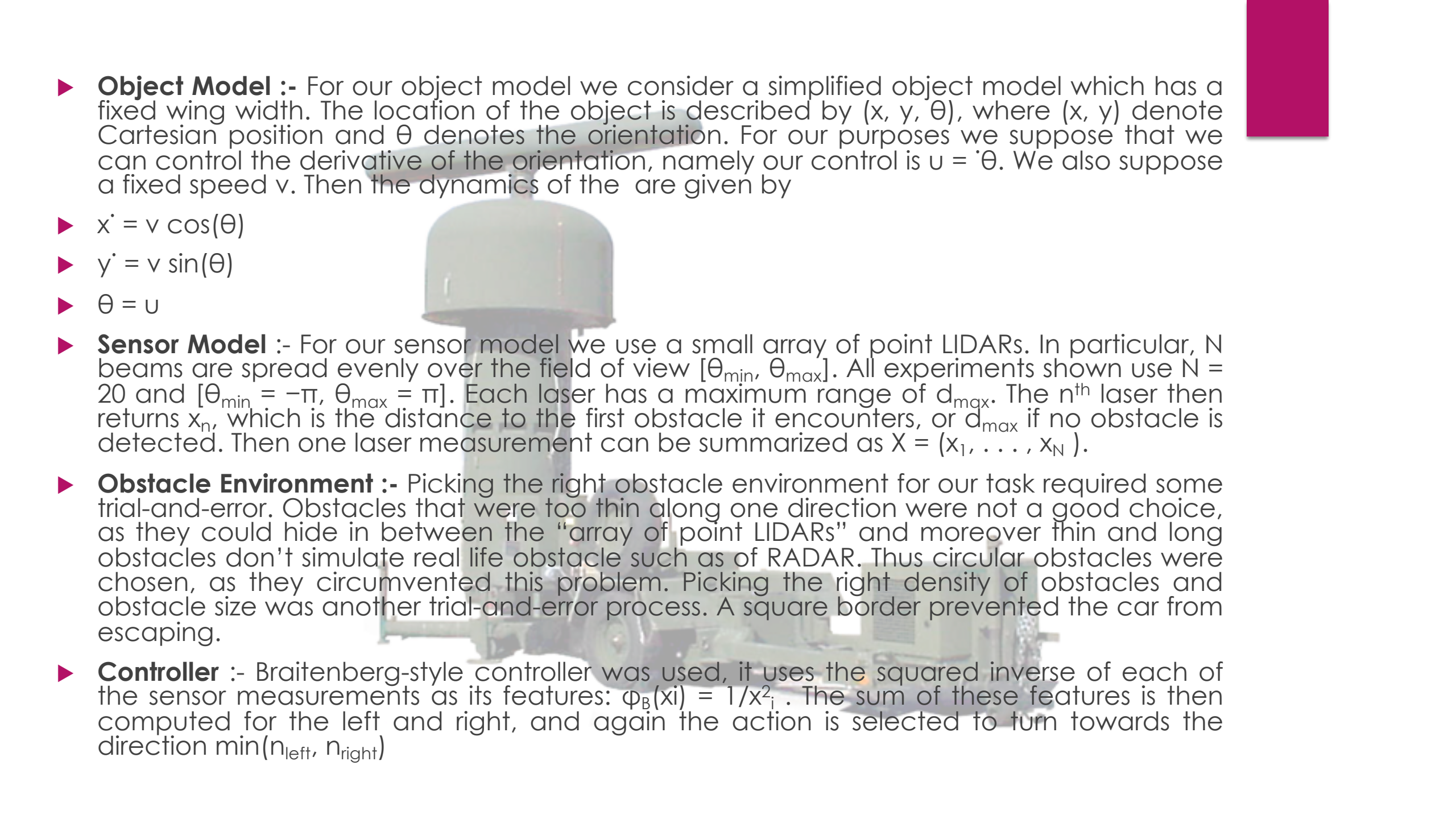
$$V(s_t) = \max_a Q(s_t, a)$$

- Q-Value of a (state, action) pair in a deterministic environment, following a greedy policy.

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

- And this is the Bellman equation in the Q-Learning. It tells that the value of an action *a* in some state *s* is the immediate reward you get for taking that action, to which add the maximum expected reward which can be received in the next state.

ENVIRONMENT

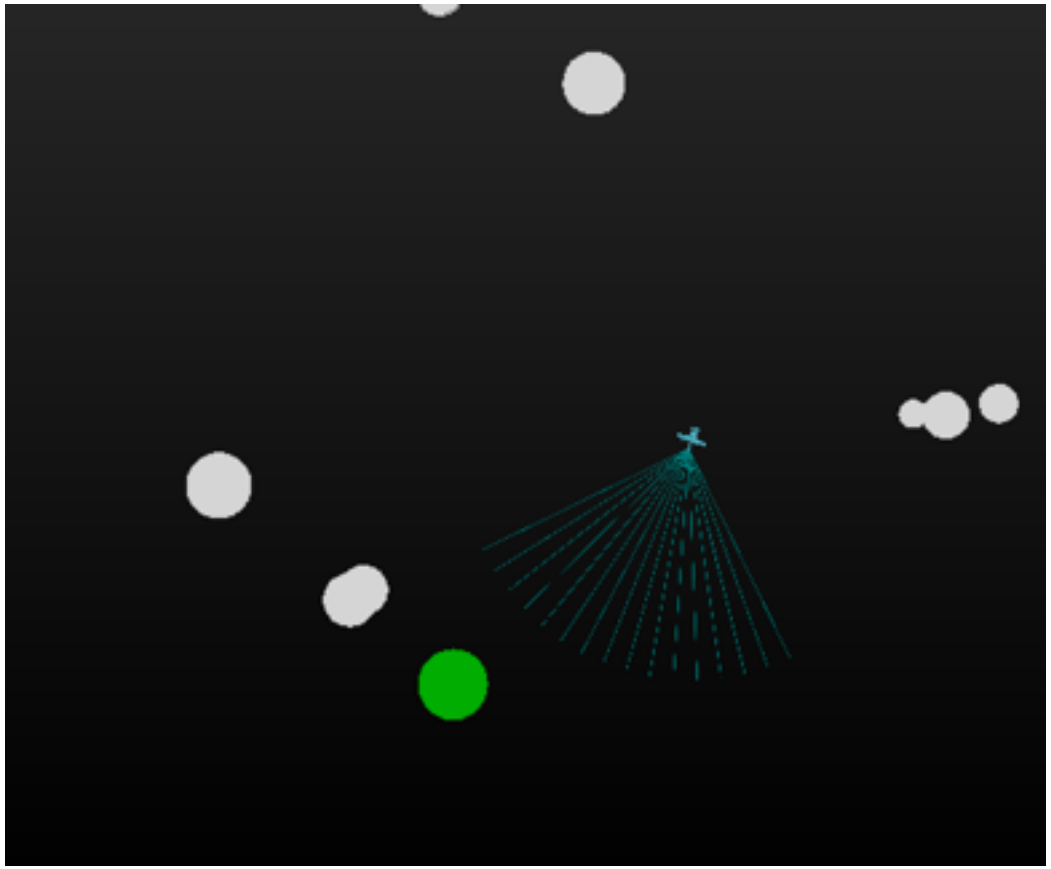


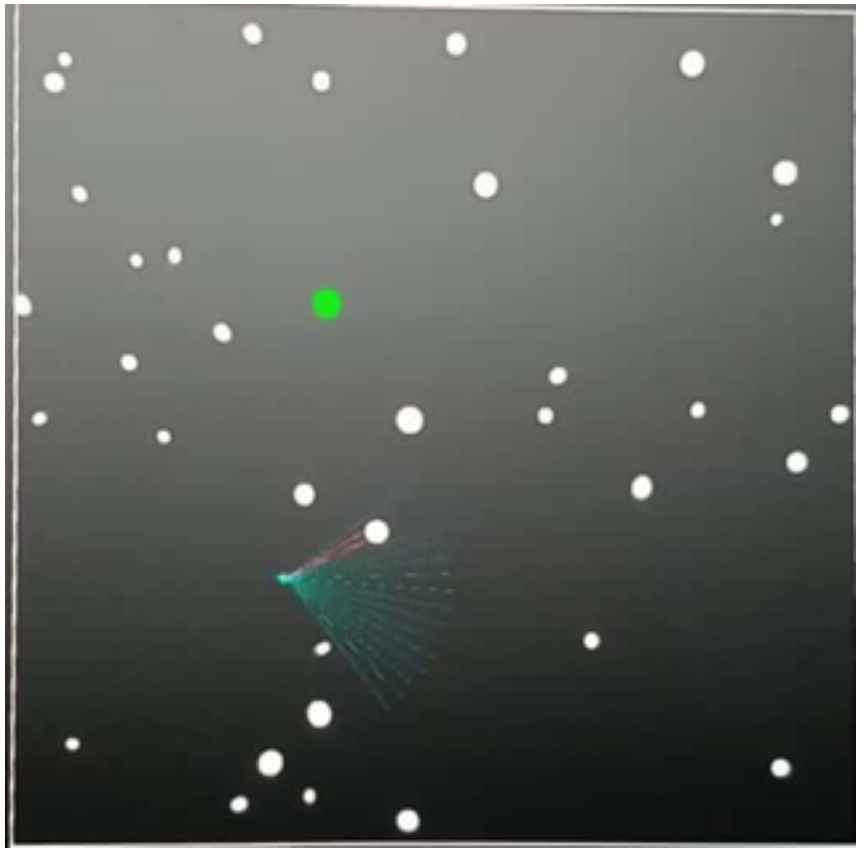
- **Object Model :-** For our object model we consider a simplified object model which has a fixed wing width. The location of the object is described by (x, y, θ), where (x, y) denote Cartesian position and θ denotes the orientation. For our purposes we suppose that we can control the derivative of the orientation, namely our control is $u = \dot{\theta}$. We also suppose a fixed speed v. Then the dynamics of the are given by
- $\dot{x} = v \cos(\theta)$
- $\dot{y} = v \sin(\theta)$
- $\theta = u$
- **Sensor Model** :- For our sensor model we use a small array of point LIDARs. In particular, N beams are spread evenly over the field of view $[\theta_{min}, \theta_{max}]$. All experiments shown use N = 20 and $[\theta_{min} = -\pi, \theta_{max} = \pi]$. Each laser has a maximum range of d_{max}. The n^{th} laser then returns x_n, which is the distance to the first obstacle it encounters, or d_{max} if no obstacle is detected. Then one laser measurement can be summarized as $X = (x_1, \ldots, x_N)$.
- **Obstacle Environment :-** Picking the right obstacle environment for our task required some trial-and-error. Obstacles that were too thin along one direction were not a good choice, as they could hide in between the "array of point LIDARs" and moreover thin and long obstacles don't simulate real life obstacle such as of RADAR. Thus circular obstacles were chosen, as they circumvented this problem. Picking the right density of obstacles and obstacle size was another trial-and-error process. A square border prevented the car from escaping.
- **Controller** :- Braitenberg-style controller was used, it uses the squared inverse of each of the sensor measurements as its features: $\varphi_B(xi) = 1/x^2_i$. The sum of these features is then computed for the left and right, and again the action is selected to turn towards the direction $\min(n_{left}, n_{right})$

IMPLEMENTATION

- Deep Q-Network (DQN) reinforcement learning agent is set which navigates an object in a simulator to a target waypoint while avoiding obstacles. For this project assumption a map of the environment a priori is not taken but rather rely on sensor (in this case a LIDAR) to inform control decisions. Software stack is integrated with a visualization tool called Director. Director code is used for visualization and raycasting. The simulation environment is written in Python. This implementation has a main class called Simulator which combines together several other modules, e.g. moving_object, Sensor, World, Controller, net and etc, to construct a working simulation class. One nice feature of the Director application is that it uses Visualization Toolkit as the underlying graphics engine, and thus raycast calls are actually in C++ and thus very efficient. A timestep of dt = 0.05 is used, and scipy.integrate is used to for integrating forward the dynamics.

OUTPUT



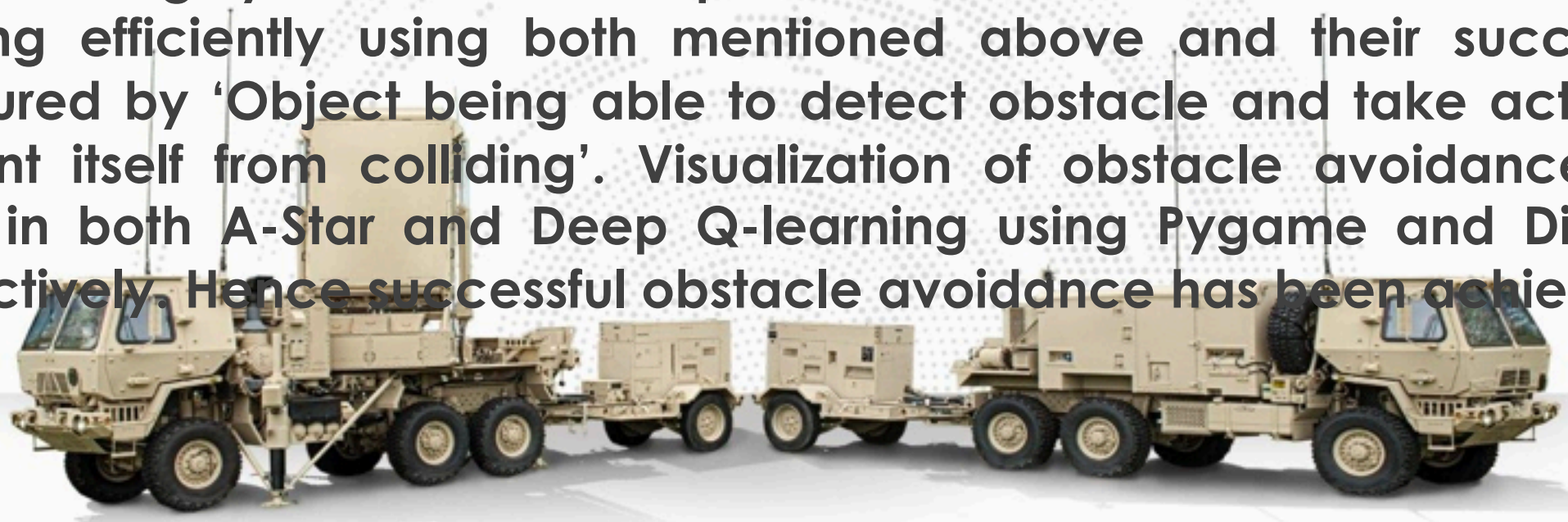


RESULT

- Result from using Deep Q-Network (DQN) Reinforcement learning agent for obstacle avoidance is a moving object is seen with rays protruding from the object, which are actually to detect obstacles and learn from. Then a starting position and target is defined, following which object starts moving through clutter of obstacles. Then training and learning of object starts, when it collides with an obstacle it learns from it , so that next time it doesn't collide. After a few iterations of learning, object starts detecting obstacle (rays turn red when an obstacle is in course) and it diverts from the course and keeps looking forward for the target

CONCLUSION

A-Star and Deep Q-Learning algorithms were successfully studied and implemented for obstacle avoidance. Both the mentioned techniques were thoroughly studied and implementation of obstacle avoidance is working efficiently using both mentioned above and their success is measured by 'Object being able to detect obstacle and take action to prevent itself from colliding'. Visualization of obstacle avoidance was done in both A-Star and Deep Q-learning using Pygame and Director respectively. Hence successful obstacle avoiddnce has been achieved.



CODE FOR A-STAR ALGORITHM FOR OBSTACLE AVOIDANCE

```
import numpy as np
import math
import random
import pygame
import time
import collections
# Convert coordinates form cartesian to screen coordinates (used to draw in pygame screen)
def cartesian_to_screen(car_pos):
    factor = 0.021
    screen_pos = np.array([center[0]*factor+car_pos[0],center[1]*factor-car_pos[1]])/factor
    screen_pos = screen_pos.astype(int)
    return screen_pos
# Convert coordinates form pygame coordinates to screen coordinates
def screen_to_cartesian(screen_pos):
    car_pos = np.array([screen_pos[0]-center[0],center[1]-screen_pos[1]])
    car_pos = car_pos.astype(float)
    return car_pos
def heuristic_cost_estimate(st, en):
    return  np.linalg.norm(st.pos - en.pos)
            #+ abs(math.atan2(math.sin(st.ang-en.ang), math.cos(st.ang-en.ang)))/5
def heuristic_cost_estimate_neighbour(st, en):
    w = (weights[st.coor[0],st.coor[1]]+weights[en.coor[0],en.coor[1]])/2
    return  np.linalg.norm(st.pos - en.pos)*w
def get_pos_by_coor(coor):
    return np.array([xValues[coor[0]], yValues[coor[1]]])
def get_coor_by_pos(pos):
    x =int(round((pos[0]-(-8))/(8-(-8))*((n-1))))
    y =int(round((pos[1]-(-8))/(8-(-8))*((n-1))))
    k =int(round((pos[2]-(0))/(2*math.pi-(0))*((m-1))))
    coor = np.array([x,y,k], dtype=int)
    return coor
class Node:
    def __init__(self, state, parent):
        self.gScore = math.inf
        self.fScore = math.inf
        self.parent = parent
        self.state = state
        self.pos = state['pos']
        self.ang = state['b']
        self.coor = get_coor_by_pos(np.array([self.pos[0],self.pos[1],self.ang]))
        graph.availableCoors[self.coor[0]][self.coor[1]][self.coor[2]] = self
    def get_neighbors(self):
        neightbours = []
        v3=[1,0]
        R = np.array([[np.cos(self.ang), -np.sin(self.ang)], [np.sin(self.ang), np.cos(self.ang)]])
        v3rot = np.matmul(R, v3)
        neighbourPos = self.pos + v3rot
        for angle in kValues:
            if abs(math.atan2(math.sin(self.state['b']-angle), math.cos(self.state['b']-angle))) < 1 and -8 <neighbourPos[0] < 8 and -8 <neighbourPos[1] <8:
                neighbourCoor =  get_coor_by_pos(np.array([neighbourPos[0],neighbourPos[1], angle]))
                # neighbourPos = get_pos_by_coor(np.array([neighbourCoor[0],neighbourCoor[1]]))
                if (graph.availableCoors[neighbourCoor[0]][neighbourCoor[1]][neighbourCoor[2]] == 0):
                    node = Node({'pos':neighbourPos, 'b':angle}, self)
                else:
node = graph.availableCoors[neighbourCoor[0]][neighbourCoor[1]][neighbourCoor[2]]
                neightbours.append(node)
        return  neightbours
class Graph:
    def __init__(self):
        self.all = []
        self.availableCoors = [[[0 for j in range(m)] for i in range(n)] for k in range(n)]
    def prepare(self, start, end):
        print('new')
        self.start = start
        self.end = end
        # The set of nodes already evaluated
        self.closedSet = []
        # The set of currently discovered nodes that are not evaluated yet.
        #  Initially, only the start node is known.

        self.openSet = [start]

        #         # For each node, which node it can most efficiently be reached from.
        #         # If a node can be reached from many nodes, cameFrom will eventually contain the
        #         # most efficient previous step.

    def search(self):
        current = self.start
        while len(self.openSet) > 0:  # While not close enough to end
            minScore = math.inf
            for node in self.openSet:
                if node.fScore < minScore:
                    minScore = node.fScore
                    current = node
            # current = the node in openSet having the lowest fScore[] value
            self.reconstruct_path(self.start, current,green,1)
            #print(current.pos)
            if np.linalg.norm(current.pos - self.end.pos) <0.3:
                    #and current.ang == self.end.ang:
                print(current.parent)
                return self.reconstruct_path(self.start, current,yellow,3)
            self.openSet.remove(current)
            self.closedSet.append(current)
            neighbors = current.get_neighbors()
            for neighbor in neighbors:
                if neighbor in self.closedSet:
                    continue  # Ignore the neighbor which is already evaluated.
                # The distance from start to a neighbor
                tentative_gScore = current.gScore + heuristic_cost_estimate_neighbour(current, neighbor)
                if neighbor not in self.openSet:  # Discover a new node
                    self.openSet.append(neighbor)
                elif tentative_gScore >= neighbor.gScore:
                    continue
                # This path is the best until now. Record it!
                neighbor.parent = current
                neighbor.gScore = tentative_gScore
                neighbor.fScore = neighbor.gScore  + heuristic_cost_estimate(neighbor, self.end)
    def add_node(self, state):
        node = Node(state)
        self.availableCoors[coor[0]][coor[1]] = node
        self.all.append(node)
        return node
    def reconstruct_path(self, start, end, color, w):
        pygame.event.get()
        path = []
        current = end
        while current.parent != None:
            current = current.parent
            path.append(current)
        for p in range(len(path)-1):
            pygame.draw.line(screen, color, cartesian_to_screen(path[p].pos),cartesian_to_screen(path[p+1].pos), w)
```

```
        pygame.display.flip()
        return path
# Screen parameters
width = 800
height = 800
center = np.array([width/2, height/2])
screen = pygame.display.set_mode((width, height))
# Colors
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)
white = (255, 255, 255)
yellow = (255,255, 0)
fpsClock = pygame.time.Clock()
fps = 40
n = 100
m =20
# Construct grid
weights = np.ones((n,n))*1
# weights += np.random.rand(n,n)*2

mean_weights =  np.mean(weights)
print(np.mean(weights))
# weights = np.random.rand(100,100)*5
xValues = np.linspace(-8, 8, n)
yValues = np.linspace(-8, 8, n)
kValues = np.linspace(0, 2*math.pi, m)
np.random.seed(1)
bumps = np.random.uniform(-6, 6, (30, 2))
def get_w(p):
    sigma = 0.3
    w=0
    for bump in bumps:
        w += 1/(sigma * np.sqrt(2 * np.pi)) *np.exp( - (np.linalg.norm(bump-p))**2 / (2 * sigma**2))*5
    w +=1
    return w
for x in range(n):
    for y in range(n):

            weights[x,y] =get_w(get_pos_by_coor([x,y]))
            # weights[x, y, k] =1
# Game loop
while True:
    screen.fill((0, 0, 0))
```

CODE FOR DEEP Q-NETWORK REINFORCEMENT LEARNING AGENT FOR OBSTACLE AVOIDANCE

The code includes various python files, which have specific functionality :

- **Moving_Object.py** – This python file, makes a moving object, which gets a random start and target location. In this the object is made to move randomly and learn when collide and start again, iteratively
- **Net.py** – This python file is responsible for setting up of the deep Q-Network and implement the learning feature for the object
- **Sensor.py** – This executes the sensor attached to object to detect obstacles and change movement / path accordingly
- **Simulator.py** – This python file, simulates all the visualization into one screen
- **World.py** – This python file, sets up the environment which has number of obstacles and their dynamic movement

Moving_Object.py

```
import numpy as np
from net import Controller
from director import vtkAll as vtk
from director.debugVis import DebugData
from director import ioUtils, filterUtils


class MovingObject(object):

    """Moving object."""

    def __init__(self, velocity, polydata):
        """Constructs a MovingObject.
        Args:
            velocity: Velocity.
            polydata: Polydata.
        """
        self._state = np.array([0., 0., 0.])
        self._velocity = float(velocity)
        self._raw_polydata = polydata
        self._polydata = polydata
        self._sensors = []
```

```
    @property
    def x(self):
        """X coordinate."""
        return self._state[0]

    @x.setter
    def x(self, value):
        """X coordinate."""
        next_state = self._state.copy()
        next_state[0] = float(value)
        self._update_state(next_state)

    @property
    def y(self):
        """Y coordinate."""
        return self._state[1]

    @y.setter
    def y(self, value):
        next_state = self._state.copy()
        next_state[1] = float(value)
        self._update_state(next_state)

    @property
    def theta(self):
        """Yaw in radians."""
        return self._state[2]

    @theta.setter
    def theta(self, value):
        """Yaw in radians."""
        next_state = self._state.copy()
        next_state[2] = float(value) % (2 * np.pi)
        self._update_state(next_state)

    @property
    def velocity(self):
        """Velocity."""
        return self._velocity

    @velocity.setter
    def velocity(self, value):
        """Velocity."""
        self._velocity = float(value)

    @property
    def sensors(self):
        """List of attached sensors."""
        return self._sensors

    def attach_sensor(self, sensor):
        """Attaches a sensor.
        Args:
            sensor: Sensor.
        """
        self._sensors.append(sensor)

    def _dynamics(self, state, t, controller=None):
        """Dynamics of the object.
        Args:
            state: Initial condition.
            t: Time.
        Returns:
            Derivative of state at t.
        """
        dqdt = np.zeros_like(state)
        dqdt[0] = self._velocity * np.cos(state[2])
        dqdt[1] = self._velocity * np.sin(state[2])
        dqdt[2] = self._control(state, t)
        return dqdt * t

    def _control(self, state, t):
        """Returns the yaw given state.
        Args:
            state: State.
            t: Time.
        Returns:
            Yaw.
        """
        raise NotImplementedError

    def _simulate(self, dt):
        """Simulates the object moving.
        Args:
            dt: Time length of step.
        Returns:
            New state.
        """
        return self._state + self._dynamics(self._state, dt)

    def move(self, dt=1.0/30.0):
        """Moves the object by a given time step.
        Args:
            dt: Length of time step.
        """
        state = self._simulate(dt)
        self._update_state(state)

    def _update_state(self, next_state):
        """Updates the moving object's state.
        Args:
            next_state: New state.
        """
        t = vtk.vtkTransform()
        t.Translate([next_state[0], next_state[1], 0.])
        t.RotateZ(np.degrees(next_state[2]))
        self._polydata = filterUtils.transformPolyData(self._raw_polydata, t)
        self._state = next_state
        list(map(lambda s: s.update(*self._state), self._sensors))

    def to_positioned_polydata(self):
        """Converts object to visualizable poly data.
        Note: Transformations have been already applied to this.
        """
        return self._polydata

    def to_polydata(self):
        """Converts object to visualizable poly data.
        Note: This is centered at (0, 0, 0) and is not rotated.
        """
        return self._raw_polydata


class Robot(MovingObject):

    """Robot."""
```

```
    def __init__(self, velocity=25.0, scale=0.15, exploration=0.5,
                 model="A10.obj"):
        """Constructs a Robot.
Args:

            velocity: Velocity of the robot in the forward direction.
            scale: Scale of the model.
            exploration: Exploration rate.
            model: Object model to use.
        """
        self._target = (0, 0)
        self._exploration = exploration
        t = vtk.vtkTransform()
        t.Scale(scale, scale, scale)
        polydata = ioUtils.readPolyData(model)
        polydata = filterUtils.transformPolyData(polydata, t)
        super(Robot, self).__init__(velocity, polydata)
        self._ctrl = Controller()

    def move(self, dt=1.0/30.0):
        """Moves the object by a given time step.
        Args:
            dt: Length of time step.
        """
        gamma = 0.9
        prev_xy = self._state[0], self._state[1]
        prev_state = self._get_state()
        prev_utilities = self._ctrl.evaluate(prev_state)
        super(Robot, self).move(dt)
        next_state = self._get_state()
        next_utilities = self._ctrl.evaluate(next_state)
        print("action: {}, utility: {}".format(
            self._selected_i, prev_utilities[self._selected_i]))

        terminal = self._sensors[0].has_collided()
        curr_reward = self._get_reward(prev_xy)
        total_reward =\
            curr_reward if terminal else \
            curr_reward + gamma * next_utilities[self._selected_i]
        rewards = [total_reward if i == self._selected_i else prev_utilities[i]
                   for i in range(len(next_utilities))]
        self._ctrl.train(prev_state, rewards)

    def set_target(self, target):
        self._target = target

    def set_controller(self, ctrl):
        self._ctrl = ctrl

    def at_target(self, threshold=3):
        """Return whether the robot has reached its target.
        Args:
            threshold: Target distance threshold.
        Returns:
            True if target is reached.
        """
        return (abs(self._state[0] - self._target[0]) <= threshold and
                abs(self._state[1] - self._target[1]) <= threshold)

    def _get_reward(self, prev_state):
        prev_dx = self._target[0] - prev_state[0]
        prev_dy = self._target[1] - prev_state[1]
        prev_distance = np.sqrt(prev_dx ** 2 + prev_dy ** 2)
        new_dx = self._target[0] - self._state[0]
        new_dy = self._target[1] - self._state[1]
        new_distance = np.sqrt(new_dx ** 2 + new_dy ** 2)
        if self._sensors[0].has_collided():
            return -20
        elif self.at_target():
            return 15
        else:
            delta_distance = prev_distance - new_distance
            angle_distance = -abs(self._angle_to_destination()) / 4
            obstacle_ahead = self._sensors[0].distances[8] - 1
            return delta_distance + angle_distance + obstacle_ahead

    def _angle_to_destination(self):
        x, y = self._target[0] - self.x, self._target[1] - self.y
        return self._wrap_angles(np.arctan2(y, x) - self.theta)

    def _wrap_angles(self, a):
        return (a + np.pi) % (2 * np.pi) - np.pi

    def _get_state(self):
        dx, dy = self._target[0] - self.x, self._target[1] - self.y
        curr_state = [dx / 1000, dy / 1000, self._angle_to_destination()]
        return np.hstack([curr_state, self._sensors[0].distances])

    def _control(self, state, t):
        """Returns the yaw given state.
        Args:
            state: State.
            t: Time.
        Returns:
            Yaw.
        """
        actions = [-np.pi/2, 0., np.pi/2]

        utilities = self._ctrl.evaluate(self._get_state())
        optimal_i = np.argmax(utilities)
        if np.random.random() <= self._exploration:
            optimal_i = np.random.choice([0, 1, 2])

        optimal_a = actions[optimal_i]
        self._selected_i = optimal_i
        return optimal_a


class Obstacle(MovingObject):

    """Obstacle."""

    def __init__(self, velocity, radius, bounds, height=1.0):
        """Constructs a Robot.
        Args:
import os
import tensorflow as tf


class NeuralNetwork(object):

    def __init__(self, input_size, output_size, hidden_layer_sizes,
                 learning_rate, dtype=tf.float32):
        self._x = tf.placeholder(dtype, [None, input_size])
        self._weights = []
```

```
        self._biases = []
        self._layers = []

        prev_layer = self._x
        for layer_size in hidden_layer_sizes:
            # Add hidden layer with RELU activation.
            prev_layer = self._add_layer(prev_layer, input_size, layer_size,
                                         tf.nn.relu)
            input_size = layer_size

        # Add output layer with linear activation.
        self._y = self._add_layer(prev_layer, input_size, output_size,
                                  lambda x: x)

        # Set up trainer.
        self._y_truth = tf.placeholder(dtype, [None, output_size])
        loss = tf.reduce_mean(tf.square(self._y_truth - self._y)) / 2
        optimizer = tf.train.GradientDescentOptimizer(learning_rate)
        self._trainer = optimizer.minimize(loss)

        # Set up session.
        self._session = tf.Session()
        self._session.run(tf.initialize_all_variables())

        self._saver = tf.train.Saver()

    def _add_layer(self, prev_layer, input_size, output_size, activation):
        # Build layer.
        W = tf.Variable(tf.truncated_normal([input_size, output_size]))
        b = tf.Variable(tf.zeros([output_size]))
        out = activation(tf.matmul(prev_layer, W) + b)

        # Maintain references.
        self._weights.append(W)
        self._biases.append(b)
        self._layers.append(out)

        return out

    @property
    def weights(self):
        return self._session.run(self._weights)

    @property
    def biases(self):
        return self._session.run(self._biases)

    def evaluate(self, x):
        return self.evaluate_many([x])[0]

    def evaluate_many(self, xs):
        return self._session.run(self._y, feed_dict={self._x: xs})

    def train(self, x, y):
        self.train_many([x], [y])

    def train_many(self, xs, ys):
        self._session.run(self._trainer,
                          feed_dict={self._x: xs, self._y_truth: ys})

    def save(self, save_path="model.ckpt"):
        save_path = self._saver.save(self._session, save_path)
        print("model saved to file: {}".format(save_path))
    def load(self, load_path="model.ckpt"):
        if os.path.exists(load_path):
            self._session = tf.Session()
            self._saver.restore(self._session, load_path)
            print("model restored from: {}".format(load_path))


class Controller(object):

    def __init__(self, learning_rate=0.01):
        self._nn = NeuralNetwork(19, 3, [11, 7], learning_rate)

    def evaluate(self, x):
        return self._nn.evaluate(x)

    def train(self, x, actual):
        self._nn.train(x, actual)

    def load(self):
        self._nn.load()

    def save(self):
        self._nn.save()
```

Sensor.py

```
import numpy as np
import math
import director.vtkAll as vtk
from director.debugVis import DebugData


class RaySensor(object):

    """Ray sensor."""

    def __init__(self, num_rays=16, radius=40, min_angle=-45, max_angle=45):
        """Constructs a RaySensor.
        Args:
            num_rays: Number of rays.
            radius: Max distance of the rays.
            min_angle: Minimum angle of the rays in degrees.
            max_angle: Maximum angle of the rays in degrees.
        """
        self._num_rays = num_rays
        self._radius = radius
        self._min_angle = math.radians(min_angle)
        self._max_angle = math.radians(max_angle)

        self._locator = None
        self._state = [0., 0., 0.]  # x, y, theta

        self._hit = np.zeros(num_rays)
        self._distances = np.zeros(num_rays)
        self._intersections = [[0, 0, 0] for i in range(num_rays)]

        self._update_rays(self._state[2])

    @property
    def distances(self):
```

```
        """Array of distances measured by each ray."""
        normalized_distances = [
            self._distances[i] / self._radius if self._hit[i] else 1.0
            for i in range(self._num_rays)
        ]
        return normalized_distances

    def has_collided(self, max_distance=0.05):
        """Returns whether a collision has occured or not.
        Args:
            max_distance: Threshold for collision distance.
        """
        for hit, distance in zip(self._hit, self._distances):
            if hit and distance <= max_distance:
                return True

        return False

    def set_locator(self, locator):
        """Sets the vtk cell locator.
        Args:
            locator: Cell locator.
        """
        self._locator = locator

    def update(self, x, y, theta):
        """Updates the sensor's readings.
        Args:
            x: X coordinate.
            y: Y coordinate.
            theta: Yaw.
        """
        self._update_rays(theta)
        origin = np.array([x, y, 0])
        self._state = [x, y, theta]

        if self._locator is None:
            return

        for i in range(self._num_rays):
            hit, dist, inter = self._cast_ray(origin, origin + self._rays[i])
            self._hit[i] = hit
            self._distances[i] = dist
            self._intersections[i] = inter

    def _update_rays(self, theta):
        """Updates the rays' readings.
        Args:
            theta: Yaw.
        r = self._radius
        angle_step = (self._max_angle - self._min_angle) / (self._num_rays - 1)
        self._rays = [
            np.array([
                r * math.cos(theta + self._min_angle + i * angle_step),
                r * math.sin(theta + self._min_angle + i * angle_step),
                0
            ])
            for i in range(self._num_rays)
        ]

    def _cast_ray(self, start, end):
        """Casts a ray and determines intersections and distances.
        Args:
            start: Origin of the ray.
            end: End point of the ray.
        Returns:
            Tuple of (whether it intersected, distance, intersection).
        """
        tolerance = 0.0                 # intersection tolerance
        pt = [0.0, 0.0, 0.0]            # coordinate of intersection
        distance = vtk.mutable(0.0)     # distance of intersection
        pcoords = [0.0, 0.0, 0.0]       # location within intersected cell
        subID = vtk.mutable(0)          # subID of intersected cell

        hit = self._locator.IntersectWithLine(start, end, tolerance,
                                              distance, pt, pcoords, subID)

        return hit, distance, pt

    def to_polydata(self):
        """Converts the sensor to polydata."""
        d = DebugData()
        origin = np.array([self._state[0], self._state[1], 0])
        for hit, intersection, ray in zip(self._hit,
                                          self._intersections,
                                          self._rays):
            if hit:
                color = [1., 0.45882353, 0.51372549]
                endpoint = intersection
            else:
                color = [0., 0.6, 0.58823529]
                endpoint = origin + ray

            d.addLine(origin, endpoint, color=color, radius=0.05)

        return d.getPolyData()
```

Simulator.py

```
import argparse
import numpy as np
from world import World
from PythonQt import QtGui
from net import Controller
from sensor import RaySensor
from director import applogic
from moving_object import Robot
from director import vtkAll as vtk
from director import objectmodel as om
from director.debugVis import DebugData
from director import visualization as vis
from director.consoleapp import ConsoleApp
from director.timercallback import TimerCallback


class Simulator(object):

    """Simulator."""

    def __init__(self, world):
        """Constructs the simulator.
        Args:
            world: World.
```

```
    """
    self._robots = []
    self._obstacles = []
    self._world = world
    self._app = ConsoleApp()
    self._view = self._app.createView(useGrid=False)

    # performance tracker
    self._num_targets = 0
    self._num_crashes = 0
    self._run_ticks = 0

    self._initialize()

def _initialize(self):
    """Initializes the world."""
    # Add world to view.
    om.removeFromObjectModel(om.findObjectByName("world"))
    vis.showPolyData(self._world.to_polydata(), "world")

def _add_polydata(self, polydata, frame_name, color):
    """Adds polydata to the simulation.
    Args:
        polydata: Polydata.
        frame_name: Frame name.
        color: Color of object.
    Returns:
        Frame.
    """
    om.removeFromObjectModel(om.findObjectByName(frame_name))
    frame = vis.showPolyData(polydata, frame_name, color=color)

    vis.addChildFrame(frame)
def add_target(self, target):
    data = DebugData()

    center = [target[0], target[1], 1]
    axis = [0, 0, 1]  # Upright cylinder.
    data.addCylinder(center, axis, 2, 3)
    om.removeFromObjectModel(om.findObjectByName("target"))
    self._add_polydata(data.getPolyData(), "target", [0, 0.8, 0])

def add_robot(self, robot):
    """Adds a robot to the simulation.
    Args:
        robot: Robot.
    """
    color = [0.4, 0.85098039, 0.9372549]
    frame_name = "robot{}".format(len(self._robots))
    frame = self._add_polydata(robot.to_polydata(), frame_name, color)
    self._robots.append((robot, frame))
    self._update_moving_object(robot, frame)

def add_obstacle(self, obstacle):
    """Adds an obstacle to the simulation.
    Args:
        obstacle: Obstacle.
    """
    color = [1.0, 1.0, 1.0]
    frame_name = "obstacle{}".format(len(self._obstacles))
    frame = self._add_polydata(obstacle.to_polydata(), frame_name, color)
    self._obstacles.append((obstacle, frame))
    self._update_moving_object(obstacle, frame)

def _update_moving_object(self, moving_object, frame):
    """Updates moving object's state.
    Args:
        moving_object: Moving object.
        frame: Corresponding frame.
    """
    t = vtk.vtkTransform()
    t.Translate(moving_object.x, moving_object.y, 0.0)
    t.RotateZ(np.degrees(moving_object.theta))
    frame.getChildFrame().copyFrame(t)

def _update_sensor(self, sensor, frame_name):
    """Updates sensor's rays.
    Args:
        sensor: Sensor.
        frame_name: Frame name.
    """
    vis.updatePolyData(sensor.to_polydata(), frame_name,
                       colorByName="RGB255")

def update_locator(self):
    """Updates cell locator."""
    d = DebugData()

    d.addPolyData(self._world.to_polydata())
    for obstacle, frame in self._obstacles:
        d.addPolyData(obstacle.to_positioned_polydata())

    self.locator = vtk.vtkCellLocator()
    self.locator.SetDataSet(d.getPolyData())
    self.locator.BuildLocator()

def run(self, display):
    """Launches and displays the simulator.
    Args:
        display: Displays the simulator or not.
    """
    if display:
        widget = QtGui.QWidget()
        layout = QtGui.QVBoxLayout(widget)
        layout.addWidget(self._view)
        widget.showMaximized()

        # Set camera.
        applogic.resetCamera(viewDirection=[0.2, 0, -1])

    # Set timer.
    self._tick_count = 0
    self._timer = TimerCallback(targetFps=120)
    self._timer.callback = self.tick
    self._timer.start()

    self._app.start()

def tick(self):
    """Update simulation clock."""
    self._tick_count += 1
    self._run_ticks += 1
    if self._tick_count >= 500:
        print("timeout")
        for robot, frame in self._robots:
            self.reset(robot, frame)
```

```
        need_update = False
        for obstacle, frame in self._obstacles:
            if obstacle.velocity != 0.:
                obstacle.move()
                self._update_moving_object(obstacle, frame)
                need_update = True

        if need_update:
            self.update_locator()

        for i, (robot, frame) in enumerate(self._robots):
            self._update_moving_object(robot, frame)
            for sensor in robot.sensors:
                sensor.set_locator(self.locator)
            robot.move()
            for sensor in robot.sensors:
                frame_name = "rays{}".format(i)
                self._update_sensor(sensor, frame_name)
                if sensor.has_collided():

                    self._num_crashes += 1
                    print("collided", min(d for d in sensor._distances if d > 0))
                    print("targets hit", self._num_targets)
                    print("ticks lived", self._run_ticks)
                    print("deaths", self._num_crashes)
                    self._run_ticks = 0
                    self._num_targets = 0
                    new_target = self.generate_position()
                    for robot, frame in self._robots:
                        robot.set_target(new_target)
                    self.add_target(new_target)
                    self.reset(robot, frame)

            if robot.at_target():
                self._num_targets += 1
                self._tick_count = 0
                new_target = self.generate_position()
                for robot, frame in self._robots:
                    robot.set_target(new_target)
                self.add_target(new_target)

    def generate_position(self):
        return tuple(np.random.uniform(-75, 75, 2))

    def set_safe_position(self, robot):
        while True:
            robot.x, robot.y = self.generate_position()
            robot.theta = np.random.uniform(0, 2 * np.pi)
            if min(robot.sensors[0].distances) >= 0.30:
                return

    def reset(self, robot, frame_name):
        self._tick_count = 0
        self.set_safe_position(robot)
        self._update_moving_object(robot, frame_name)
        robot._ctrl.save()


def get_args():
    """Gets parsed command-line arguments.
    Returns:
        Parsed command-line arguments.
    """
    parser = argparse.ArgumentParser(description="avoids obstacles")
    parser.add_argument("--obstacle-density", default=0.01, type=float,
                        help="area density of obstacles")
    parser.add_argument("--moving-obstacle-ratio", default=0.0, type=float,
                        help="percentage of moving obstacles")
    parser.add_argument("--exploration", default=0.5, type=float,
                        help="exploration rate")
    parser.add_argument("--learning-rate", default=0.01, type=float,
                        help="learning rate")
    parser.add_argument("--no-display", action="store_false", default=True,
                        help="whether to display the simulator or not",
                        dest="display")

    return parser.parse_args()


if __name__ == "__main__":
    args = get_args()

    world = World(200, 200)
    sim = Simulator(world)
    for obstacle in world.generate_obstacles(args.obstacle_density,
                                             args.moving_obstacle_ratio):
        sim.add_obstacle(obstacle)

    sim.update_locator()

    target = sim.generate_position()
    sim.add_target(target)

    controller = Controller(args.learning_rate)
    controller.load()

    robot = Robot(exploration=args.exploration)
    robot.set_target(target)
    robot.attach_sensor(RaySensor())
    robot.set_controller(controller)
    sim.set_safe_position(robot)
    sim.add_robot(robot)

    sim.run(args.display)
    controller.save()
```

World.py

```
import numpy as np
from moving_object import Obstacle
from director.debugVis import DebugData


class World(object):

    """Base world."""

    def __init__(self, width, height):
        """Construct an empty world.
        Args:
            width: Width of the field.
            height: Height of the field.
        """
        self._data = DebugData()
```

```
        self._width = width
        self._height = height
        self._add_boundaries()
    def _add_boundaries(self):
        """Adds boundaries to the world."""
        self._x_max, self._x_min = self._width / 2, -self._width / 2
        self._y_max, self._y_min = self._height / 2, -self._height / 2

        corners = [
            (self._x_max, self._y_max, 0),  # Top-right corner.
            (self._x_max, self._y_min, 0),  # Bottom-right corner.
            (self._x_min, self._y_min, 0),  # Bottom-left corner.
            (self._x_min, self._y_max, 0)   # Top-left corner.
        ]

        # Loopback to begining.
        corners.append(corners[0])

        for start, end in zip(corners, corners[1:]):
            self._data.addLine(start, end, radius=0.2)

    def generate_obstacles(self, density=0.05, moving_obstacle_ratio=0.20,
                           seed=None):
        """Generates randomly scattered obstacles to the world.
        Args:
            density: Obstacle to world area ratio, default: 0.1.
            moving_obstacle_ratio: Ratio of moving to stationary obstacles,
                default: 0.2.
            seed: Random seed, default: None.
        Yields:
            Obstacle.
        """
        if seed is not None:
            np.random.seed(seed)

        field_area = self._width * self._height
        obstacle_area = int(field_area * density)

        bounds = self._x_min, self._x_max, self._y_min, self._y_max
        while obstacle_area > 0:
            radius = np.random.uniform(1.0, 3.0)
            center_x_range = (self._x_min + radius, self._x_max - radius)
            center_y_range = (self._y_min + radius, self._y_max - radius)
            center_x = np.random.uniform(*center_x_range)
            center_y = np.random.uniform(*center_y_range)
            theta = np.random.uniform(0., 360.)
            obstacle_area -= np.pi * radius ** 2

            # Only some obstacles should be moving.
            if np.random.random_sample() >= moving_obstacle_ratio:
                velocity = 0.0
            else:
                velocity = np.random.uniform(-30.0, 30.0)

            obstacle = Obstacle(velocity, radius, bounds)
            obstacle.x = center_x
            obstacle.y = center_y
            obstacle.theta = np.radians(theta)
            yield obstacle

    def to_polydata(self):
        """Converts world to visualizable poly data."""
        return self._data.getPolyData()
```