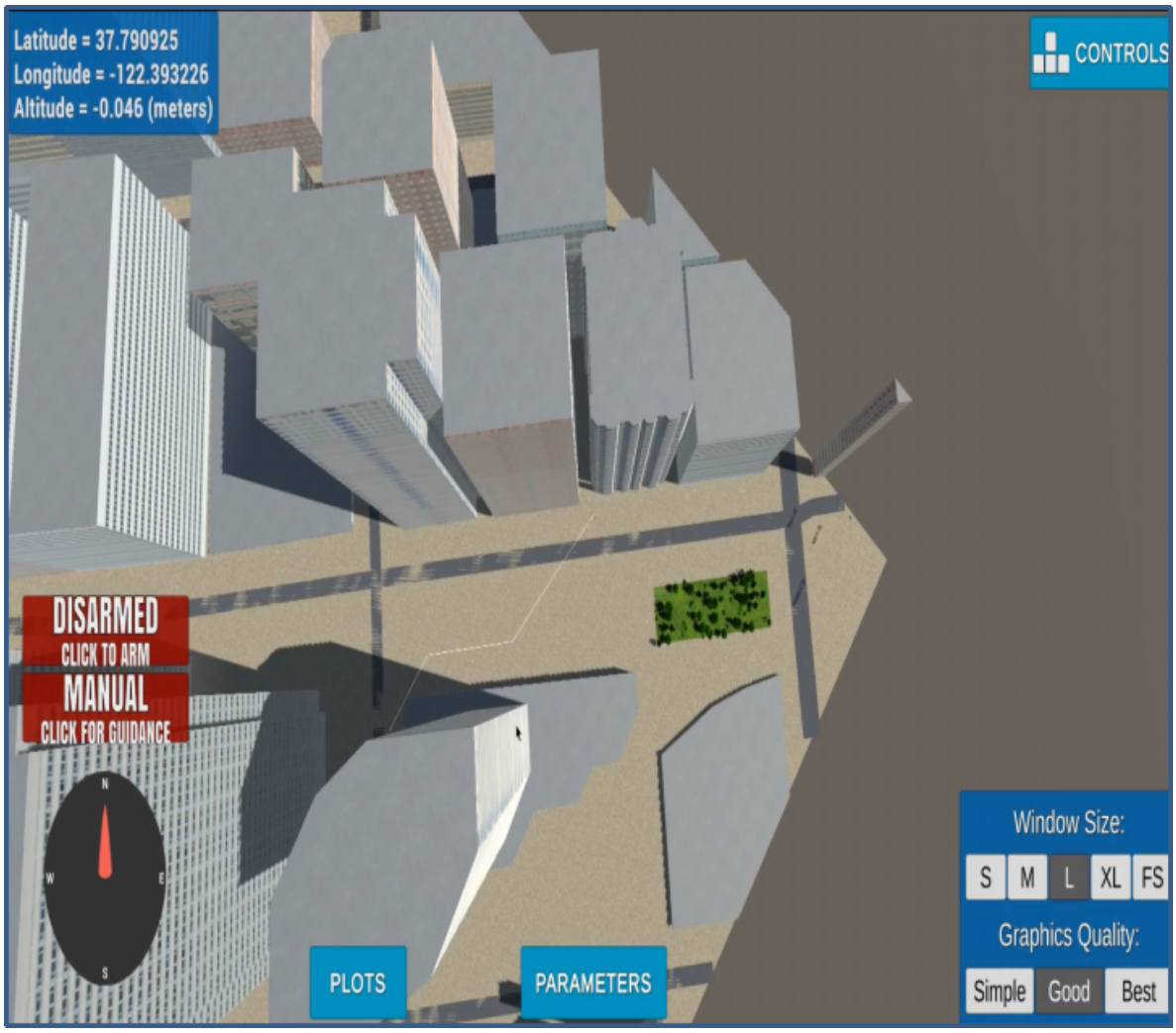
Project: 3D Motion Planning by Yassine Elhallaoui.

Required Steps for a Passing Submission:

Rubric Points

1. Load the 2.5D map in the colliders.csv file describing the environment.
2. Discretize the environment into a grid or graph representation.
3. Define the start and goal locations.
4. Perform a search using A* or other search algorithm.
5. Use a collinearity test or ray tracing method (like Bresenham) to remove unnecessary waypoints.
6. Return waypoints in local ECEF coordinates (format for `self.all_waypoints` is [N, E, altitude, heading], where the drone's start location corresponds to [0, 0, 0, 0].
7. Write it up.
8. Done!



Starter Code :

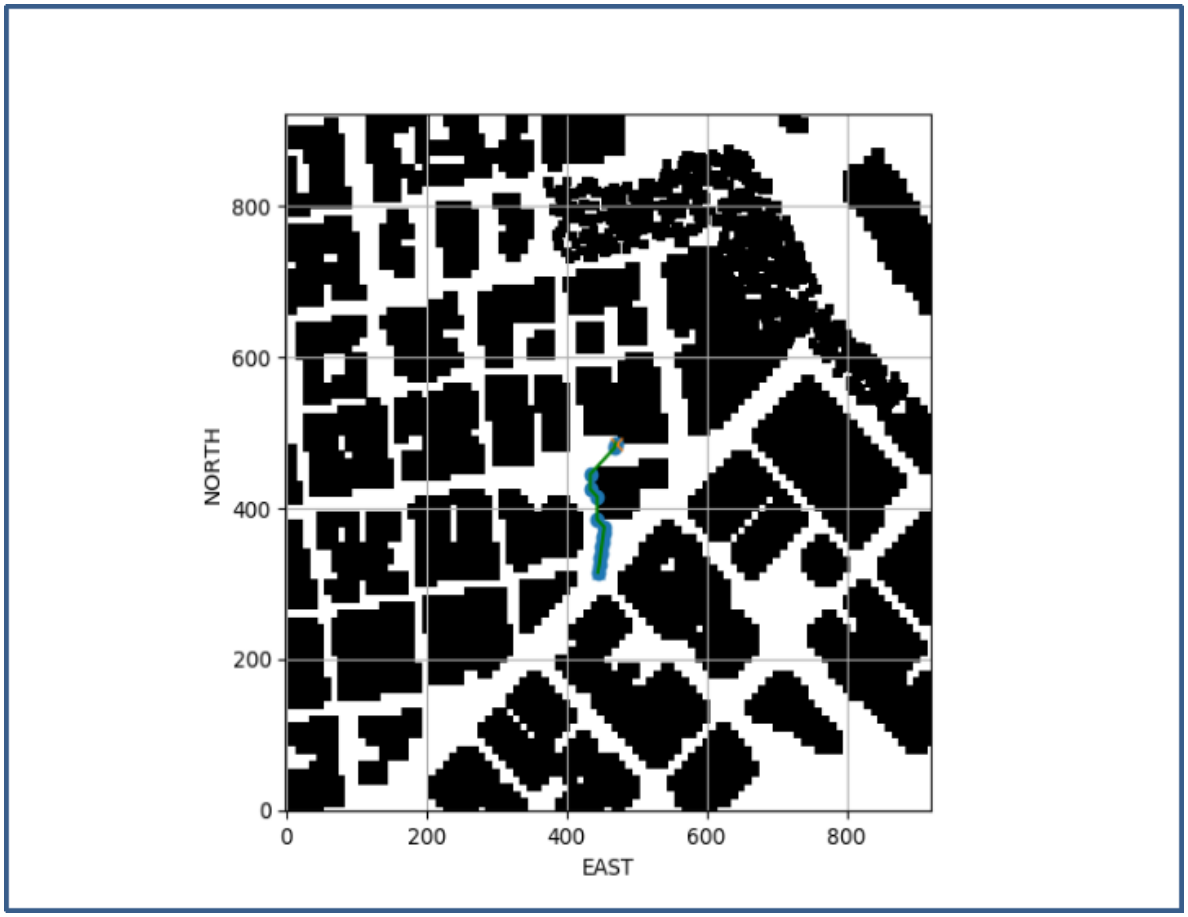
1. motion_planing.py Functionality compared with backyard_flyer_solution.py:

- Planning state added.
- Replacing the function ***calculate_box()*** with ***plan_path()*** to calculate the new path in-between the the obstacles [buildings] and also configuring the Altitude of the Drone and the maximum distance to the obstacles.
- Send_waypoint() function to draw the way-points on the simulator 3D environment..

2. planning_utils.py file review :

- create_grid() : creating a 2D grid and declare numerically to clear zones as 0 and zone with obstacles as 1.
- Action() : returning the valid actions to the Drone to reach the Goal via way-points.
- a_star() : used to generate the path from grid_start to grid_goal, given an heuristic
- heuristic() : given a distance estimates from the current location to the Goal.

Implement the Path Planning Algorithm :



- **global home position** : by reading the first line of the csv file, extract lat0 and lon0 as floating point values and use the self.set_home_position() method to set global home.
- Drone Position Update : retrieving the drone's location in global coordinates and then this location is converted into local position by using the global_to_local() function.
- **Grid for start position from local position** : The self.local_position output used to get the north and east offsets converted to integers numbers in order to use it to calculate the path using the grid.

- **Grid for goal position from geodetic coordinates** : In order to set the goal position, I use the grid_start position and add random values within the limits of the north and east offsets. Moreover, we make sure that the chosen point is not within an obstacle.
- **Diagonal motion** : planning_utils() has been modified in order to include diagonal motions on the grid that will control the physical flight behavior of the drone.
- **Cull waypoints** : calculating the Full path to be used into the shorten_path() function in planning_utils(), which returns a shorter list of waypoints after the collinearity check between each 3 successive points of the full path.

Executing the flight Plan :

1. **Does it work?**

Definitely Yes. :D

The flight plan was successful with no complication in all the way-points will till the landing phase as on the video on the github 'Motion Planning.mkv for the all flight, the only problem encounter was it the distance between the start point and the Goal is too far the simulator hanged for a while and took many time to calculate the path, also the Drone initial position on the 'MOTION PLANNING' is under the cemented platform i had to takeoff manually to put it on the cemented platform first '3D Environment glitch'

motion_planning.py

1. Importing all the necessary Python Libraries including the Udacity Drone Library.

```
import argparse
import time
import msgpack
from enum import Enum, auto

import numpy as np
import random

from planning_utils import *
from udacidrone import Drone
from udacidrone.connection import MavlinkConnection
from udacidrone.messaging import MsgID
from udacidrone.frame_utils import global_to_local

import matplotlib.pyplot as plt
```

2. Declaring the state of the Drone and assigning an automatic values.

```
class States(Enum):
    MANUAL = auto()
    ARMING = auto()
    TAKEOFF = auto()
    WAYPOINT = auto()
    LANDING = auto()
    DISARMING = auto()
    PLANNING = auto()
```

3. MotionPlanning class under the Udacity Drone main class to manage all the variables and callback functions for every state and parameters.

```
class MotionPlanning(Drone):

    def __init__(self, connection):
        super().__init__(connection)

        self.target_position = np.array([0.0, 0.0, 0.0])
        self.waypoints = []
        self.in_mission = True
        self.check_state = {}

        # initial state
        self.flight_state = States.MANUAL

        # register all your callbacks here
        self.register_callback(MsgID.LOCAL_POSITION, self.local_position_callback)
        self.register_callback(MsgID.LOCAL_VELOCITY, self.velocity_callback)
        self.register_callback(MsgID.STATE, self.state_callback)
```

the class contain many functions :

- local_position_callback: sending position of the drone in every state callback.
- velocity_callback: sending the velocity of the drone every-time it changes.
- state_callback: reporting the flight state.
- arming_transition: switch the drone state to Arming. [from Manual to Autonomous]
- takeoff_transition: Switch the Drone state to TAKEOFF [send the Altitude data]
- waypoint_transition: Switch the Drone state to WAYPOINT and set targets to next waypoints till the Goal.
- landing_transition: Switch the Drone to state LANDING to trigger the landing procedures.
- disarming_transition: Switch the Drone state to DISARMING to prepare fir MANUAL state.
- manual_transition: Switch the Drone state to MANUAL and stop drone and set the mission to False.
- send_waypoints: parse the waypoints data to the simulator for visualization.
- plan_path: Switch the Drone state to PLANNING to set the necessary parameters to the drone [safety distance, target altitude], also reading the geospacial data from 'colliders.csv' to declare the home & target positions and altitude.
- start: initializing the Log and start the connection with the UDACITY Drone simulator.

4. preparing the connection parameters to be parsed to the MAVLINK connection protocol, waiting for 1sc till the connection established and later run the start function from the MotionPlanning class.

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--port', type=int, default=5760, help='Port number')
    parser.add_argument('--host', type=str, default='127.0.0.1', help="host address, i.e. '127.0.0.1'")
    args = parser.parse_args()

    conn = MavlinkConnection('tcp:{0}:{1}'.format(args.host, args.port), timeout=60)
    drone = MotionPlanning(conn)
    time.sleep(1)

    drone.start()
```

planning_utils.py

1. Importing the necessary python Libraries :

```
from enum import Enum
from queue import PriorityQueue
import numpy as np
```

2. create_grid() : function to create numpy arrays from the 2.5D map data from the 'colliders.csv' file and convert the array to grids contain the data about the map obstacles [1 for cells with obstacles and 0 for free cells].

```python
def create_grid(data, drone_altitude, safety_distance):
    """
    Returns a grid representation of a 2D configuration space
    based on given obstacle data, drone altitude and safety distance
    arguments.
    """

    # minimum and maximum north coordinates
    north_min = np.floor(np.min(data[:, 0] - data[:, 3]))
    north_max = np.ceil(np.max(data[:, 0] + data[:, 3]))

    # minimum and maximum east coordinates
    east_min = np.floor(np.min(data[:, 1] - data[:, 4]))
    east_max = np.ceil(np.max(data[:, 1] + data[:, 4]))

    # given the minimum and maximum coordinates we can
    # calculate the size of the grid.
    north_size = int(np.ceil(north_max - north_min))
    east_size = int(np.ceil(east_max - east_min))

    # Initialize an empty grid
    grid = np.zeros((north_size, east_size))

    # Populate the grid with obstacles
    for i in range(data.shape[0]):
        north, east, alt, d_north, d_east, d_alt = data[i, :]
        if alt + d_alt + safety_distance > drone_altitude:
            obstacle = [
                int(np.clip(north - d_north - safety_distance - north_min, 0, north_size-1)),
                int(np.clip(north + d_north + safety_distance - north_min, 0, north_size-1)),
                int(np.clip(east - d_east - safety_distance - east_min, 0, east_size-1)),
                int(np.clip(east + d_east + safety_distance - east_min, 0, east_size-1)),
            ]
            grid[obstacle[0]:obstacle[1]+1, obstacle[2]:obstacle[3]+1] = 1

    return grid, int(north_min), int(east_min)
```

3. Action class that is the manager of all the action that the Drone will take on the grid and associate costs and deltas of every movement to every cell on the grid [WEST, EAST, NORTH, SOUTH] and also diagonal moves [NORTH EAST, NORTH WEST, SOUTH EAST, SOUTH WEST].

```python
# Assume all actions cost the same.
class Action(Enum):
    """
    An action is represented by a 3 element tuple.
    The first 2 values are the delta of the action relative
    to the current grid position. The third and final value
    is the cost of performing the action.
    """

    WEST = (0, -1, 1)
    EAST = (0, 1, 1)
    NORTH = (-1, 0, 1)
    SOUTH = (1, 0, 1)

    NORTH_EAST = (-1, 1, 1.14) #cost of sqrt(2)
    NORTH_WEST = (-1, -1, 1.14)
    SOUTH_EAST = (1, 1, 1.14)
    SOUTH_WEST = (1, -1, 1.14)

    @property
    def cost(self):
        return self.value[2]

    @property
    def delta(self):
        return (self.value[0], self.value[1])
```

the class contains :

- valid_actions : determine which valid action that need to be computed and decide the moves.
- a_star : store all valid actions and create a set of already visited cells that will help create a dictionary of the valid paths already scouted
- heuristic : reporting the estimated cost from the position of the Drone on the grid to the Goal in each step.
- collinearity_check: eliminating unnecessary waypoints in your path and evaluate it.
- shorten_path : remove all the intermediary points on the path returned from the a_start function in every 3 points will remove the middle point if the path does not collide cause of obstacles.

THANK YOU.

Yassine.H