# Drone Dash 2023

Summer of Innovation 2023

June, 2023

**Abstract**

The objective of the Drone Dash challenge is to develop an obstacle avoidance algorithm for a drone equipped with a color and depth camera. The goal is to navigate through an obstacle course (Fig1) while minimizing collisions and completing the course in the shortest time possible. The drone should utilize any preferred algorithm to detect and avoid obstacles using the camera data. The challenge concludes when the drone successfully detects an ArUco marker below it, indicating the end of the course, at which point the drone should land.

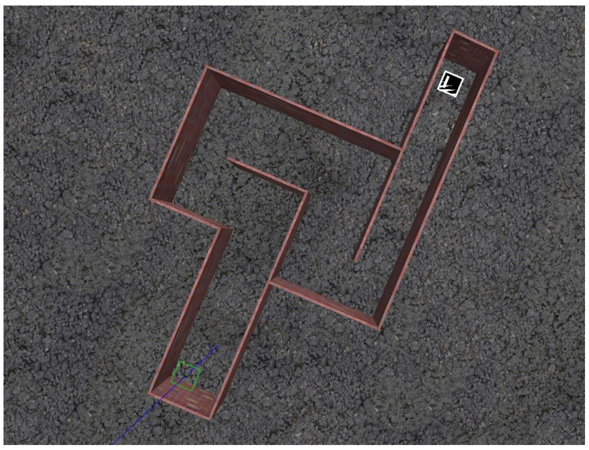

**Figure 1:** Image of Demo world

## 1. Introduction

The drone and the code setup were provided using the GitHub repository of the Catkin workspace. Along with this, we were provided with a **demo_node.py** code, which contained the basic movement of the drone. The initial setup was as follows:

*1.1. Description*

- The RGB (Fig2) images topic: */camera/rgb/image_raw*
- The depth (Fig3) images topic: */camera/depth/image_raw*
- The state of the drone is updated continuously, in my case every $50ms$

## 2. Assumptions

To proceed with the algorithms the following assumptions were made, some of them do require improvements.

- All walls are of similar types, i.e., the colour and structure of the wall
- All walls and obstacles are perpendicular to each other

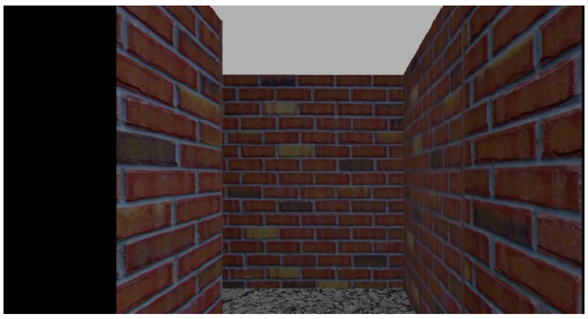

**Figure 2:** Sample RGB Image

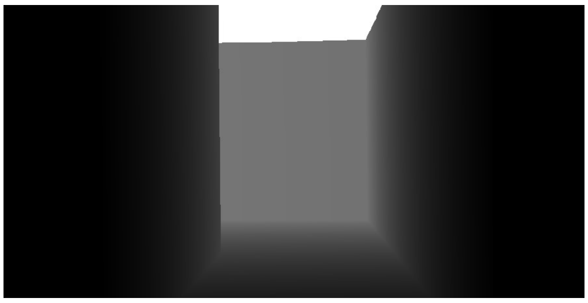

**Figure 3:** Sample Depth Image

## 3. Terminologies

To describe the algorithm the following terminologies are used:

- **Node:** Node is defined as any point from where multiple paths intersect. It has attributes coordinate and neighbours
- **Neighbour:** A node can have 4 neighbours (according to assumption). Each can be either **FREE** or **WALL** or **AnotherNode**
- **Movement:** It describes the movement of the drone such as **rotation** and **forward** movement
- **Controller:** This gives a control signal to the drone stating where to move, the path to follow, or detecting walls.
- **NodeList:** It maintains list of all nodes visited till now

Giving an analogy of networking, **Controller.py** basically is the IP layer (calculates the path), and **Movement.py** is Link Layer (actual movement)

## 4. Approach

The approach that I followed is a **State Machine** based algorithm. The entire algorithm is based on **5** states, namely:

1. **INIT**: Initialize the current node and add it to the list of nodes; if not exists
2. **ANALYZING**: Analyze 4 directions around current and update **Neighbours**
3. **THINKING**: Based on **NodeList** come up with a path to follow
4. **PATH_TRAVERSE**: Set a new destination based on the next node in the path
5. **FREE_ROAM**: Free roam to the next destination or until a wall is detected

The state transition diagram is shown here (Fig4) The details of each state are given below.

### *4.1. INIT*

The drone starts off at the init state. If the current node already exists, it updates the children, or else it adds a new node to **NodeList**.

### *4.2. ANLYZING*

The drone intermittently turns 90 degrees 4 times and uses the **Wall Detection Model** (later section) to update the neighbour attributes.

### *4.3. THINKING*

This uses the entire **NodeList** and uses **Modified Dijkstra Algorithm** (later section) to find the nearest node with a **FREE** direction. This simple heuristic makes sure that we explore newer nodes quickly.

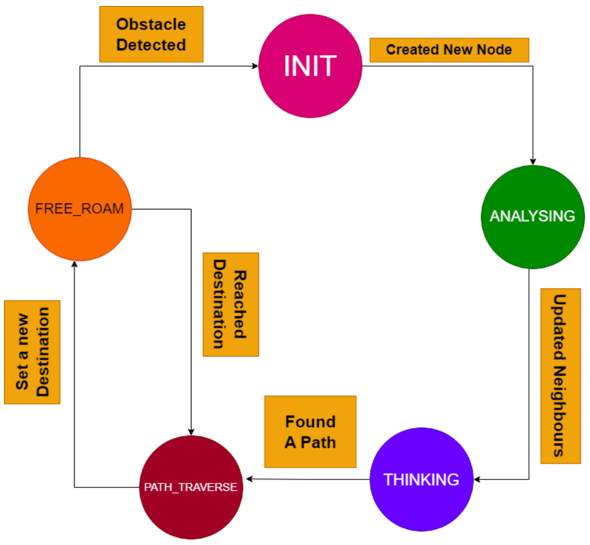

**Figure 4:** State Transition Diagram

### *4.4. PATH_TRAVERSE*

Once the path is ready, It sets the next destination. If the path has ended, the destination is set to **None**.

### *4.5. FREE_ROAM*

Here the drone moves from the current node to a destination node if the destination is **None**, the drone free roams until an obstacle is detected; in this case, the state is reset back to **INIT**.

## 5. Modified Dijkstra Algorithm

This algorithm finds the nearest node to the current node with a **FREE** neighbour. To do this, the original Dijkstra algorithm breaks the execution when a newly popped node from an open list satisfies at least one **FREE** direction condition. Then the path from the current node to that node is computed using *parent* pointers. The code is shown below.

Link To Algorithm

## 6. Wall Detection

To detect wall or an obstacle, a dataset was created by manually moving the drone to target regions and images were collected. Using around **2000** image datapoints, a **Conv. Neural Network** was trained to detect a wall using the input image stream. The dataset and lab notebook can be found here: Click here! Using this we can detect a wall with an accuracy of **95.09 %**.

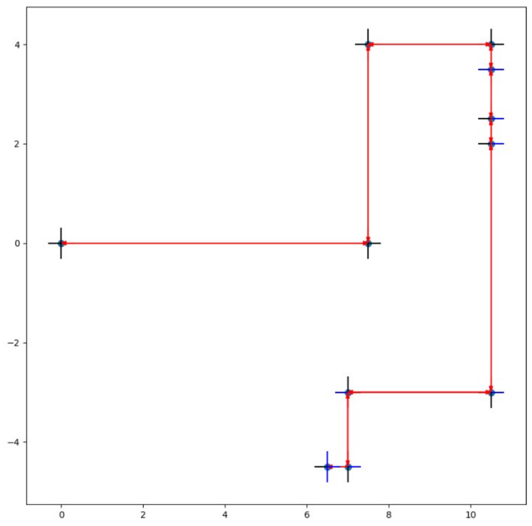

**Figure 5:** Nodes and Paths Learnt

## 7. Observations on Demo World

Since the algorithm is based on an ML model, and the detection is only for specific coloured, perpendicular walls, The drone on average could cover **80 %** of the **Demo World** map, before looping around the same place. But once the walls are detected and a path is formed, the drone moves correctly.

The following image shows paths, walls and nodes learnt by the algorithm during its run.

## 8. Conclusion

The following improvements can be made to the algorithms:

1. More dynamic way of obstacle detection instead of static
2. Learning of more varied and complex obstacles using a depth camera
3. More work on landing algorithm

In conclusion, this was a very challenging and fun event, with lots of new technologies used. It was a very nice learning experience.