University College of Southeast Norway
Faculty of Technology and Maritime Sciences
-
Master Thesis in System Engineering With Embedded Systems
Kongsberg Department of Engineering
May 27, 2020

Eivind Tangen Haldorsen

Path planning for multiple collaborative UAVs

Abstract

Unmanned aerial vehicles and autonomous robots have gained popularity in recent years, finding use in military applications, surveillance of urban areas, and hobbies. Autonomous drones require programming to follow distinct paths for their motion, and ultimately decide the shortest and most effective path in a vast network of nodes. This can range from maneuvering around in a city for surveillance and monitoring, to reach a certain destination during natural disasters as quickly as possible to gather life-saving environmental data.

This research is concerned with the matters of creating graph networks and ultimately calculate the shortest path from a given position to a given destination. Further motion planning for UAVs and their maneuverability based on the planned paths are also of great concern, whether it is based on following the path itself or maintaining stability of the system. Algorithms for determining the shortest path are investigated, regardless of the complexity of the graph network. Simulations are carried out and discussed, in order to illustrate and reflect various problem settings.

Acknowledgements

First and foremost, I would like to convey my sincere and deepest gratitude to my supervisor Dr. Antonio L. L. Ramos for his guidance and inspiration during the thesis. I am thankful for his feedback during my research, which has made the process much easier. I am also thankful for his suggestion of this topic, which I have enjoyed working on and learned a lot from.

I would like to thank my friends and colleagues at University of South-eastern Norway (USN), specifically Chrisander Brønstad for his helpful advice and encouragement, and Ole Marius Norderud, Bilgehan Günaydin and Huseyin Derbent as good colleagues during the Master's programme. I am also very thankful to the great lecturers at USN such as Dr. Dag A. H. Samuelsen and Dr. Antonio L. L. Ramos for introducing me to Objected Oriented and Embedded Systems programming, as well as Dr. Jose Ferreira for further embedded programming courses. Lastly, I would like to thank my closest family for being supportive and encouraging during the writing of this thesis.

Eivind Tangen Haldorsen
Kongsberg, Norway, May 27, 2020

Contents

List of Figures

List of Tables

List of Abbreviations

APSP	All Pair Shortest Path
CCW	Counter Clockwise
CPU	Central Processing Unit
CW	Clockwise
CIA	Central Intelligence Agency
FPS	Frames Per Second
GPU	Graphics Processing Unit
HTOL	Horizontal Take-Off and Landing
IDE	Integrated Development Environment
MV	Manipulated Variable
PV	Process Variable
PID	Proportional Integral Derivative
RAM	Random Access Memory
SAR	Search And Rescue
SLAM	Simultaneous Localization And Mapping
SP	Set Point
SSSP	Single Source Shortest Path
UAV	Unmanned Aerial Vehicle
USAF	United Sates Air Force
VTOL	Vertical Take-Off and Landing

Chapter 1

Introduction

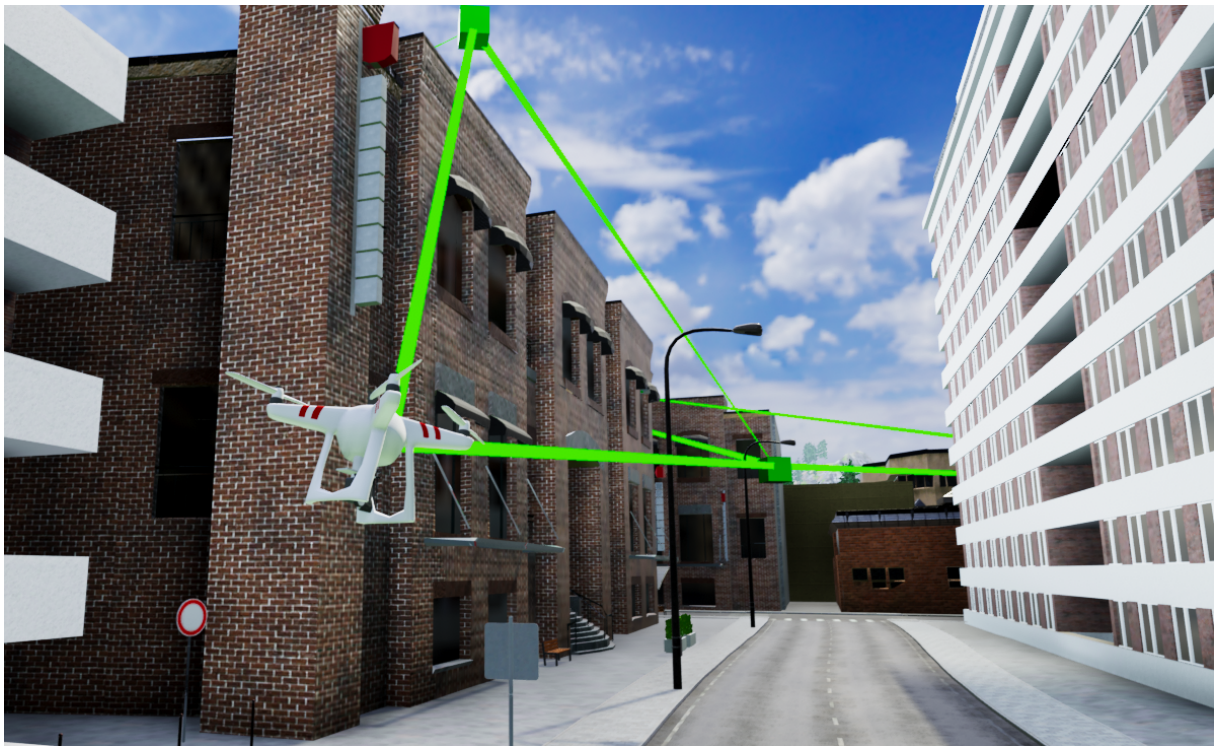


FIGURE 1.1: Simulated drone in Unity 3D game engine with representative paths.

The interest in unmanned aerial vehicles has grown over the last years with application examples such as surveillance of urban areas, hobbies, or military tools. Exploiting the versatility of sensors that can be mounted on the drones as well as the technology improvements for component's size and weight, have made it possible for unmanned aerial vehicles to operate in a wide range of applications. They have been around for over a century, especially in terms of military usage for monitoring or radio communication, which can be traced all the way back to 1900's. The United States of America started early with their development research on unmanned aerial vehicles, and tested various flying drones during the period of 1910 to World War II. The British military in 1915 initiated the groundwork for unmanned aerial vehicles, which utilized drones during the Battle of Neuve Chapelle to capture more than 1,500 sky view maps of the German fortifications [24].

The primary objective was to develop technology that could be utilized to monitor enemy defences, or strategies. However, the usage of drones in modern times have

increased towards urban areas in a civil manner such as camera footage of environment sales, or taking advantage of available sensors in search and rescue missions. This work is concerned with the autonomy of unmanned aerial vehicles and how they plan their path to a given destination, by avoiding obstacles such as city objects, or other drones on similar path to the same destination. Various algorithms for finding the shortest path have been investigated, as well as tools and techniques for maneuvering along the planned path. Issues related to environmental mapping, motion planning and localization of the unmanned aerial vehicle has also been addressed in this work.

1.1 Unmanned aerial vehicle maneuverability

Unmanned aerial vehicles (UAVs), also known as drones, are currently manufactured for a wide range of applications such as construction, journalism, security, monitoring, or even personal deliveries [13]. This further implies to being able to adapt to environmental changes and avoiding obstacles which may seem as a simple task for humans, but not that simple for robots. Not only are the UAVs supposed to maneuver in a collision-free path, but also satisfy other performance requirements such as maintaining stability and effectiveness of their flying. This is based around external surroundings or individual component weights and sizes [33]. Additionally, calculating the appropriate path may take an extensively long time, based on the complexity of available paths and microprocessor performance. This may cause latency issues, which prevents the UAVs from operating at a desired level due to inaccuracies and slow decisions. Environmental data such as mapping, UAV movement and localization and path finding algorithms are just a brief selection of areas that need to be taken into consideration when building autonomous UAVs.

1.1.1 Autonomous unmanned aerial vehicles

UAVs are usually operated by an external device controlled by a human. However autonomous unmanned aerial vehicles are becoming more popular for applications such as search and rescue (SAR) missions, and locating humans or objects with the use of thermal vision, cameras or other external sensors in difficult environment conditions [13]. The maneuverability and scalability of the UAVs makes them able to reach areas that we are not able to, especially when applying autonomous flying. In natural disasters, such as hurricanes or earthquakes, time is of the essence and making use of autonomous UAVs that calculate the shortest path to a certain destination may save lives by taking advantage of the monitoring possibilities and environmental data collection from a target site.

1.1.2 State of the Art

There are several types of unmanned aerial vehicles, whether it is based on military uses for remote bomb detonation, monitoring or communications, or civil uses in urban areas with drones as a hobby for enjoyment. The first notable use of robots in a SAR mission, was land based robots for locating victims during the terrorist attack on September 11th in New York [13]. The most frequently used UAVs, are quadcopters,

which are types of vertical take-off and landing (VTOL) drones that consists of four propellers arranged at each corner, as seen in Figure 1.2.

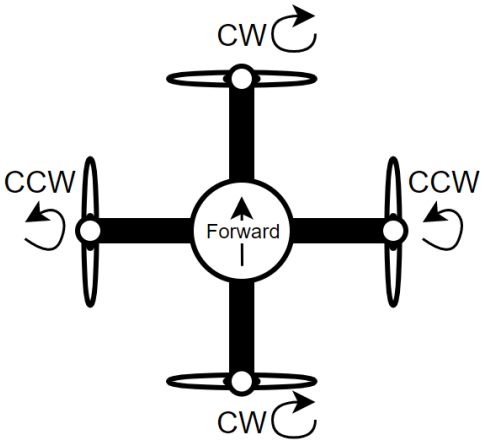


FIGURE 1.2: Quadcopter with indicated rotor rotation directions.

These machines differ from other VTOL drones, due to the fact that their rotors have to vary in thrust, in order to control pitch, roll and yaw angles respectively [4]. Initially, two of the rotors rotate clockwise (CW) and two anti-clockwise (CCW), and the quadcopter can further maneuver in three dimensions based on the power applied to each individual rotor.

In this work, the words drone and UAV are used interchangeably. In general, any unmanned aerial vehicle may be defined as *"any fixed rotary winged aircraft or lighter than air vehicle capable of flight without an on-board pilot or crew"* [9]. Initially, the DJI phantom [11], is the most common quadcopter sold for civil uses which typically comes with a camera mounted beneath as a surveillance tool for capturing videos and images for personal use. On the contrary, the Predator series are military drones being used by the United States Air Force (USAF) and Central Intelligence Agency (CIA). However, these are considerably different than the quadcopters discussed earlier, as it is an aircraft with propeller mounted pointing forward in the rear to improve visibility for sensors [14], which in turn creates a completely different flying mechanism. These drones are used for offensive operations by USAF and CIA, mostly in the reconnaissance role in certain areas. Additionally, there are two missiles mounted on the wings as well as sensors, a ground control station, and a primary satellite link for communication [5]. Nonetheless, the main focus in this work will be on quadcopters and their structure, as well as their maneuver possibilities based on rotor placements and thrust for yaw, pitch and roll.

1.2 Path planning

The benefits of UAVs is that they can be programmed accurately and are able to perform tasks precisely based on given commands, whether it is monitoring and analysing environments, or prominently reaching certain destinations. Path planning, which relates to travel from A to B as quick as possible, or finding the most effective route through a network of paths, is a very important term that is used in autonomy. It may consist on computing a collision-free path between two points in a three-dimensional

space. Multiple UAVs may also be used in some cases, to increase the likelihood of completing a given task successfully.

In short, path planning is based on how the robot will maneuver in a given environment in order to perform a given task. There are several algorithms for this, whether it is based on calculating the shortest path between two points in a known environment, global path planning, (Dijkstra's, A* or Floyd-Warshall[22]) or mapping the environment real-time and calculate optimal paths based on sensor data. The latter, also known as local path planning, can be achieved using an algorithm called Simultaneous Localization And Mapping (SLAM) [3], simultaneously, keep track of the robot's position within the environment. This is useful in scenarios for environments which are unstructured and unknown, and the robot needs to adapt its movement based on sensor readings.

On the other hand, using multiple UAVs may provide benefits in certain missions, although it is a difficult task to ultimately create coordinated robots. This is due to the UAVs need to continuously know other UAVs' current position. Swarm intelligence is a term that relates to the usage of multiple UAVs, in which they can analyze environments by maximizing the minimum distance between them [34]. However in unknown environments, it is difficult to know others three dimensional positions accurately, without appropriate communication capability implemented.

1.2.1 Multiple UAVs and their path

Assuming a set of UAVs A_i that have to reach a position G_i following a certain path P_i for each drone. Calculating this target path from start to goal before take-off is beneficial in cases where the environment is not changing, as it makes it possible to check if two paths intersect with each other, and re-evaluate before take-off [27]. However, ideally P_i should be calculated in real-time as each drone A_i presumably experiences environmental changes since the original planned path [17].

1.3 Objective

The objective of this work is to research and develop an autonomous flying system for UAVs that include path planning for finding the shortest path in a graph network. Further, the shortest path will be calculated using algorithms and optimizing parameters such as heuristic estimates, distance measurements and graph network complexity. Additionally, UAV movement in 3D space with rotation and vectors relative to the calculated path will also be addressed. The work method consists of using an agile methodology for carrying out tasks and reviewing previous work. This thesis can be derived into different sub-tasks:

- Analyse current solutions and applications of using autonomous UAVs.
- Literature review on UAVs movement and rotation with quaternions and movement with 3D-vectors.
- Implement a flying drone in a simulated environment, for further application in autonomous flight.
- Research basic graph theory and understanding graph networks.

- Investigate graph networks and shortest-path algorithms and implement graph networks that can be used for UAVs.
- Testing UAV simulation and graph network separately for faults and then integrate the two.
- System testing while optimizing algorithms and UAV movement to maximize performance.
- Final evaluation and assessment for possible future applications.

1.4 Main contributions

The main goal of this thesis was to simulate UAVs and their motion based on a planned path. This path is derived from a given start position to a given destination in a graph network. In order to illustrate and experiment with various path planning algorithms, some of the end results is summarized below.

UAV in simulated environment

The simulation environment in this work was created in a game engine that represents a city with environmental objects such as roads, buildings, signs or trees. The integration of a quadcopter within the environment has made it possible to illustrate its maneuverability, with a reference to the real-world. Functions for maintaining stability of the drone, as well as movement control using roll, pitch, and yaw has been implemented in order to carry out the available performance of a quadcopter, regarding its speed and angular velocities.

Graph network tool

A tool for creating vertices and edge connections between them, has been made for supporting and experimenting on path planning algorithms. With the help of this tool, vertices can be added and removed as needed, which increases the versatility, and ease of use regarding path planning testing. This means that execution times can be calculated during path planning, where a carefully placed vertex or edge connection can impact the algorithm performance.

Path planning algorithm

By using the graph network tool for creating custom graph networks, path planning algorithms can be applied based on given start- and goal-vertex. The implemented algorithms are compatible with the graph network tool, with a user interface for choosing certain graph networks or running tests. Specifically, the A* algorithm has been implemented where experiments can be performed in regards to parameters within the algorithm.

1.5 Outline

The remainder of this thesis structures as follows. Chapter 2 briefly introduce basic terms in regards to graph theory and unmanned aerial vehicle movement, as well as state-of-the-art path planning algorithms. Chapter 3 consists of the working methodology and basic graph networks for path planning algorithms and elementary UAV movement in 3D space. This also discusses programming tools and listings that can assist in simulating and achieving autonomous UAVs, that is further discussed in Chapter 4. Moreover, Chapter 4 elaborates implementations of flying drones with quaternions and 3D vectors, and graph networks with vertices and edges using simulations with game engine. Experimental test results and observations of the performance of the proposed system architecture will also be discussed. Finally, chapter 5 presents conclusions and summarizes achieved results of the proposed system solution. It elaborates various directions of possible future work in regards to new areas of application for autonomous UAVs, as well other possibilities for further optimization.

Chapter 2

Background

There are several types of unmanned aerial vehicles, and this chapter will cover some of the basic aspects regarding existing solutions and technologies. An overview of basic graph terms and terminology such as vertices and edges in a graph network, and various types of graph networks is also provided.

2.1 Unmanned aerial vehicles

Unmanned aerial vehicles can be represented as flying machines, which can be programmed to perform certain tasks whether it is based on a search and rescue mission with infra-red cameras or recording an environment with camera stabilization techniques. The structure of the different drones that currently exists, ranges from multi-rotor- to fixed wing UAVs. Below is a representation on some of the various types of drone structures, mostly based on their flying architecture such as number of rotors and their positions on the drone. Additionally, it is important to distinguish between propellers and rotors, in which rotors are used with lift, in order to provide vertical ascend and descent whereas propellers are usually referenced to when providing horizontal thrust. It is important to note that it is easier to maintain forward thrust than upward lift due to gravity, in which case main rotors are substantially larger than propellers on an UAV with same mass. Table 2.1 shows an overview of drone types based on their initial weight categorized by Brooke-Holland [6, 15].

TABLE 2.1: Drone categorization based on weight.

Type	**Weight range**
Nano Drones	< 200 g
Micro Drones	200 g - 2 kg
Mini Drones	2 kg - 20 kg
Small Drones	20 kg - 150 kg
Tactical Drones	150 kg - 600 kg
Strike Drones	> 600 kg

Single-rotor

Given by its name, these types of drones only have one rotor attached at the top and is typically structured as a helicopter, which provides vertical lift and hover flight possibilities [7]. Additionally, they might be powered by gas power which provides increased endurance and longer flight times. However, they are harder to fly and more training is needed in order to operate these drones correctly. These drones vary in size, and can generally be categorized as any drone type in Table 2.1 as both nano drones and tactical helicopter drones are used [15].

FIGURE 2.1: Single-rotor drone.

Multi-rotor

Multi-rotor drones can be categorized based on their weight typically from nano drones to mini drones, rarely larger. These types of drones are circular shaped, with 2 or more rotors attached perpendicular to each other in order to provide vertical lift and stability. Some of these drones are more commonly known as hexacopters, octocopters or the more famous quadcopter [7]. Their names represents the amount of rotor-positions that can provide vertical lift. They are the typical hobbyists choice, especially when it comes to aerial photography, and video inspections in certain areas [21]. This is due to their user friendly design and stability control which increases the ease of use when controlling and steering the drone. However, due to an increased amount of rotors, weight is increased requiring more power to provide enough lift at each rotor which in turn lowers the flight times. Just like their counterpart single-rotor drones, they are VTOL drones with hover flight possibilities [15].

FIGURE 2.2: Multi-rotor drone.

Fixed-wing

Fixed wing drones are inspired by the typical aeroplane with fixed wings to provide lift, rather than vertical rotors. This means that these drones need to have propellers attached in order to provide forward thrust that can create vertical lift as a whole on

the drone. They are not able to take-off vertically and is of the type Horizontal take-off and landing (HTOL), which restricts their usage as its launch requires a large space. Fixed-wing UAVs have a long endurance and greatly increased flight speed compared to the other rotor types which makes them perfect for aerial mapping and coverage, and power line inspection [7]. These drones are heavier due to the wings, and can usually be categorized as the upper range in Table 2.1, ranging from mini- to tactical drones.

FIGURE 2.3: Fixed-wing drone.

The DJI phantom 3 is an example of a multi-rotor VTOL quadcopter, and has a weight of 1280 grams with batteries and rotors included [33]. By Table 2.1, it can be observed that the DJI phantom 3 hobby drone can be categorized as a micro drone based on the weight from its specifications.

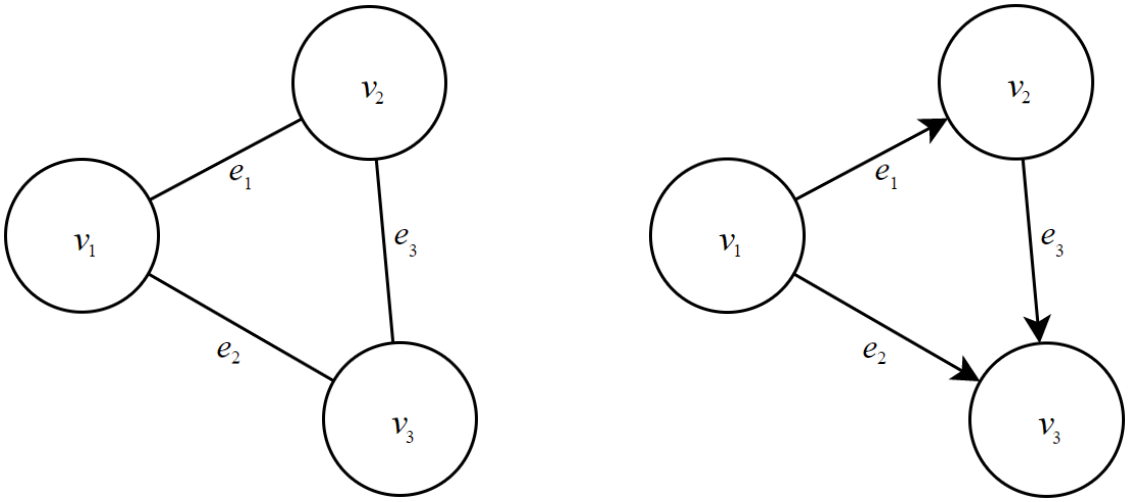
2.2 Graph theory and path planning algorithms

Graph theory is a study of graphs which describes a specific relationship between a set of objects or points. This relationship vary, based on the specific task at hand, however they all share the same idea, which is illustrating a hierarchical view of objects and their connections. A graph G can be defined as a set of two elements, namely vertices V, and edges E, as $G = (V, E)$. An edge element connects two vertices which may then be called an edge's end-vertices. Initially, two vertices can be divided as a subset of a given edge E. Thereby, a simple graph consists of no more than one edge between two vertices as well as no abnormal connections such as curves or loops, which are edges connecting a vertex to itself [20]. However these terms are commonly used in multi-graphs, where multiple edges and loops are used.

2.2.1 Vertices and edges

Assuming two connected vertices u and v, it is said that these are adjacent or neighbours of one another, specifically written as $u \sim v$. A vertex v can be represented as a tuple in world space such that $v_i = (x_i, y_i, z_i)$, for $i = 0, ..., n$ where n is the amount of vertices in a graph. Furthermore, in a simple graph network, each edge is associated with only two vertices, which can be represented as components such that $(u, v) \in E$ is the edge from u to v with a given length which is typically represented as a weight $w(u, v)$. Generally, in simple graph networks, the direction of the edge is irrelevant, such that $(u, v) \in E = (v, u) \in E$ is true. This means that the only meaningful information within a simple graph is which vertices are connected and the distance between them, this is referenced as undirected graph type. Figure 2.4a represents an undirected

graph which ignores the orientation, or direction of the edge connections. On the contrary, Figure 2.4b illustrates a set of edges with orientations [19].



(A) Undirected graph no orientation, $G = (3, 3)$. (B) Directed graph with orientation, $G = (3, 3)$.

FIGURE 2.4: Directed vs undirected graphs represented with 3 vertices.

Lastly, a weighted graph is a graph network where the edges are given a value that can, for example, represent the distance between its two vertex components. Figure 2.5 illustrates a weighted graph with numbers, or weights represented at each edge inclusively. Negative weights are also possible, although in this example all the numbers are positive, representing the distance between each vertex [19].

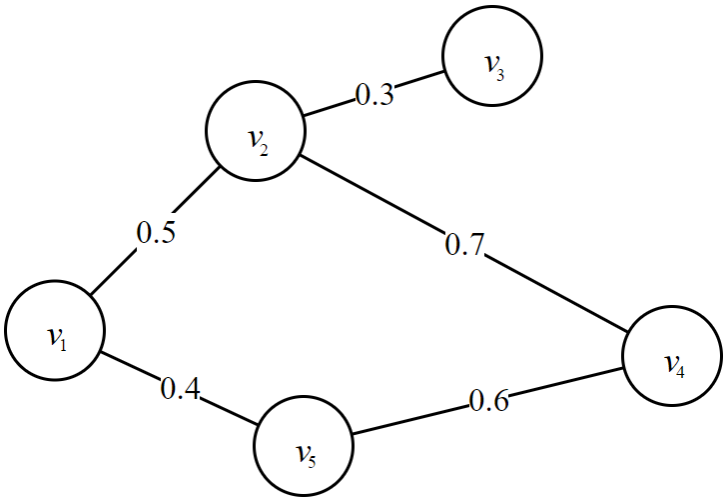


FIGURE 2.5: Weighted graph network with edges with weight cost $w(u, v)$, $G = (5, 5)$.

Identifying a vertex' neighbours is vital in graph theory due to solving certain problems and understanding algorithms, such as finding the shortest path or solving the traveling salesman problem. The latter, is a problem that was mathematically formulated in the 1800's, which asks a specific question: Consider a graph consisting of cities

$G = (C, E)$, what is the shortest path possible to visit all cities? Now, increasing the number of cities C further scales the complexity of the solution, due to the amount of possibilities. Assuming there are a total of three cities A, B and C, there would be a total of $3! = 6$ different edge connections E between them. Furthermore, solving the traveling salesman person may be done with brute force by trying each combination, however this is not effective as the number of cities increases.

2.2.2 Path planning algorithms

Based on any existing graph network, finding the shortest path between A and B can be performed in several ways. By this, it is necessary to have information of the whole network such as distances between the vertices, and the connections between them based on the neighbours. The most important aspect when choosing which algorithm to apply to a given graph network, is the calculation, or time complexity of the algorithm. A common term being used is the Big O notation, which indicates that worst-case calculated time. It is referenced as time complexity $T(n)$ with a calculation time of $O(n)$, where n is the size, or number of elements in a given problem. When $T(n) \in O(n)$, the time complexity is linear, meaning that the given algorithm goes through each element n once, until it is completed. Note that the Big O notation always represents the worst-case scenario. For instance, finding an item in an unsorted list with 10 elements has the worst-case time complexity of $O(10) = 10$ steps if the item we want to find is the last element. By this, we can understand the time complexity of different path planning algorithms.

Furthermore, assuming we have a set of edges and vertices in a graph network, each edge has a weight $w(u, v)$ associated with it. Calculating the length of this network $p = (v_0, v_1, ..., v_k)$ can be performed by summarizing each weight such that

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i), \tag{2.1}$$

where k is the number of vertices in the path. By this information, we can derive two main path planning algorithms, strictly known as single source shortest-path (SSSP) and All-pairs shortest path (APSP) [22]. We can use the weight information from Equation (2.1), and define the shortest path

$$\delta(u, v) = \begin{cases} w(p) : u \to v & \text{if there is a path from u to v,} \\ \infty & \text{otherwise,} \end{cases} \tag{2.2}$$

for any given vertices u and v in G. There are mainly four types of shortest path algorithms that are based on these equations [22].

Single pair

Single pair shortest path is based on finding the optimal path from a given vertex u to a given vertex v in a graph network G based on the initial weights for each pair. Typically referenced as finding the shortest path from vertex A to B.

Single source

Single source relates to finding the shortest path from a source vertex s to any given vertex v in G.

Single target

Typically, this is referenced as the shortest path from a target vertex t from every other vertex in G. These types of shortest path algorithms are beneficial when there is a goal vertex that needs to be reached from all available vertices.

All pairs

All pair type is a slightly different algorithm type, where the main goal is to find all the shortest paths between every vertex in G.

Chapter 3

Methodology

This chapter covers more details on maneuverability for UAVs, as well as localization problems based on their positions. It also presents a discussion on path planning algorithms, specifically single source shortest path or all pair shortest path algorithms. These are usually utilized when planning routes for UAVs.

3.1 UAV maneuverability

Maneuvering an UAV is not a simple task, and it becomes even harder when designing autonomous unmanned aerial vehicles that needs to plan their motion respectively. Motion planning can be described as how a system supposedly maneuvers in a certain sequence to reach a goal from a given start point. When maneuvering a robot, we have to take its position and rotation relative to the global space into consideration. This is due to the fact that when the drone has to navigate to a new point, it has to update its position in the system. By this, it is given that the robot knows its own local position based on previous events, however it does not know its global position. Usually, global position is referenced as the earth fixed frame F^E such as the UAV physical properties of roll, pitch, yaw, and angular velocities. On the contrary, local position is typically referenced as the body fixed frame F^B, in which some properties are measured such as linear acceleration of the UAV [4]. Assuming a 2D coordinate system, we can use homogeneous coordinates to transform a local point from a robot's perspective into global coordinates [8]. This means that a robot has its own frame that it operates in, independently from the global system, as seen in Figure 3.1.

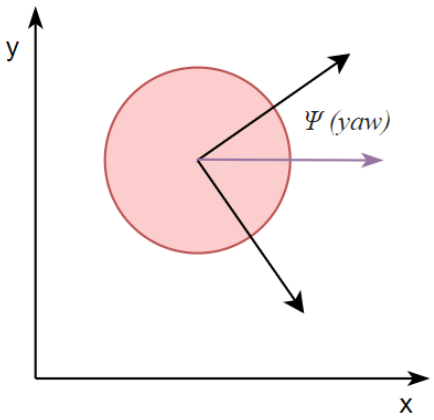


FIGURE 3.1: Robot position in global space.

By using homogeneous coordinates, we can represent a top-down illustration of the UAV in 2D, where the z-axis is set as a constant number [8]. Further, if the robot moves, a new local position X is found, which can be transformed to the global frame by multiplying the point with the robots local coordinate system, $\widetilde{\mathbf{v}}_{local}$, expressed as

$$\widetilde{\mathbf{v}}_{global} = X\widetilde{\mathbf{v}}_{local}. \tag{3.1}$$

To calculate its local position $\widetilde{\mathbf{v}}_{local}$, we have to multiply its rotation R with its position. Firstly, we can use trigonometric functions based on a direction ψ to define the rotation such that

$$R = \begin{pmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{pmatrix}, \tag{3.2}$$

which ultimately can calculate its local position $\widetilde{\mathbf{v}}_{local}$ by further adding the current position t of the UAV

$$\widetilde{\mathbf{v}}_{local} = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\psi & -\sin\psi & x \\ \sin\psi & \cos\psi & y \\ 0 & 0 & 1 \end{pmatrix}. \tag{3.3}$$

Here ψ is the direction, or yaw angle, while x and y represent position in local space [8]. For instance, assuming the robot moves 1m in x-axis which is the forward direction, a yaw angle rotation of $45°$, and position of $x = 0.7, y = 0.5$, we calculate the new global position to be

$$\widetilde{\mathbf{v}}_{global} = \begin{pmatrix} \cos 45° & -\sin 45° & 0.7 \\ \sin 45° & \cos 45° & 0.5 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1.41 \\ 1.21 \\ 1 \end{pmatrix}. \tag{3.4}$$

Additionally, it may be useful to do it the other way around, by converting a global position into local coordinates [4]. In such cases, this can be achieved by multiplying the inverse of the robot's pose, i.e.,

$$\widetilde{\mathbf{v}}_{local} = X^{-1}\widetilde{\mathbf{v}}_{global}. \tag{3.5}$$

3.1.1 Localization for UAVs

Based on Equations (3.1), (3.3) and (3.5) we can calculate a robot's position depending on how it maneuvers. However, there are other factors that affect a robots position such as minor inaccuracies in motors, or friction caused by wind and weather conditions [2]. This will ultimately increase the difficulty of maintaining an accurate position of an UAV. A term referenced as localization, can be used for UAVs to ensure that its position is correct, based on additional external data to calculate position in global space. Simultaneous localization and mapping (SLAM) is a navigation algorithm that uses localization techniques when maneuvering to ensure that a robots position is correct in its system. A valid technique is to observe landmark objects such as trees, buildings or generally any static object during navigation, and storing these objects in the system. Assuming a set of observation points p_n in global space with given landmark objects o_n. For each new observation point p, its known position fluctuates exponentially due

to the fact that earlier estimations of its position might be incorrect [1, 2]. This is illustrated in Figure 3.2, where each circle in p_i represents approximations of where the UAV might be positioned after navigating from one point to the next.

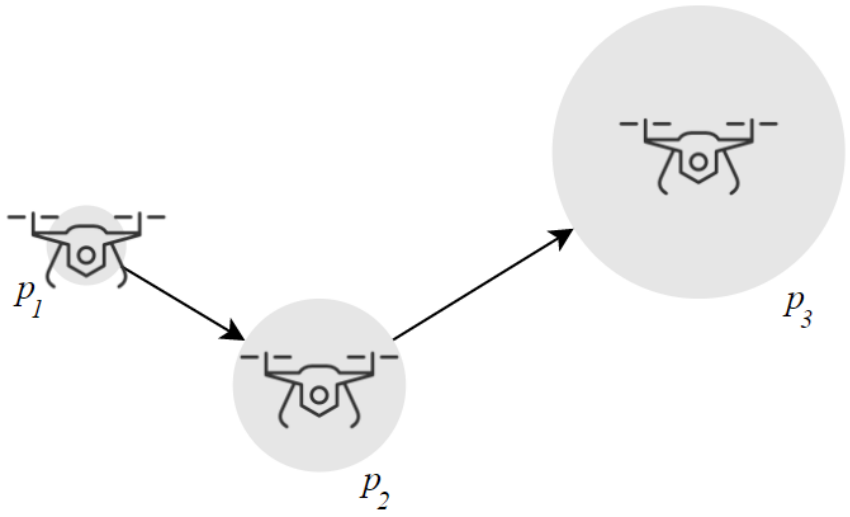


FIGURE 3.2: Robot localization without observing surroundings.

Furthermore, by having access to environment information such as trees, buildings and other landmark objects, we can estimate a drone's position in global space more accurately. From Figure 3.3, we can see that the UAV observes landmark objects o_1 and o_2 in position p_1, and an estimate of its position is calculated. After the UAV has navigated to p_2, it can perform a set of observations and store discovered landmark objects and their positions into its system.

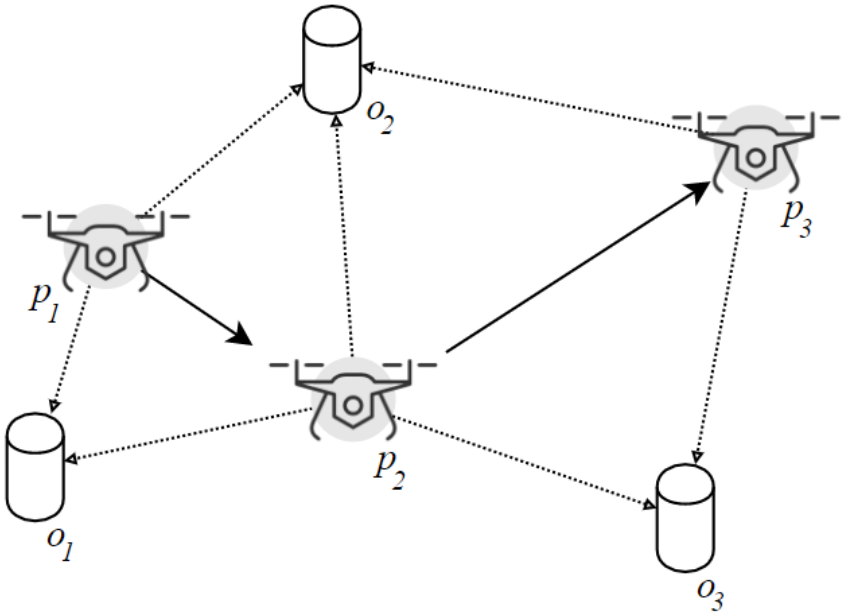


FIGURE 3.3: Robot localization by observing landmark objects to estimate position.

From this information, the UAV can calculate its position based on how it navigated using Equation (3.3), and refine the estimate using additional information from what it has observed in p_1 and p_2. In this regard, landmark object o_1 was discovered multiple times, which is utilized to retrieve an improved estimate of its position. However, this raises another aspect in regards to localization, which is how to distinguish the different landmark objects, and how to detect and recognize which ones have already been observed. The impact of having wrong associations with objects, greatly impacts a robots pose and, in worst case, will result in collisions.

3.1.2 Quadcopter maneuverability

Quadcopters, as discussed in Chapter 2, are of the type multi-rotor VTOL aircraft. Figure 3.4 illustrates how a quadcopter can utilize and maneuver in three dimensions, x, y and z respectively by applying power to the four different rotors. Yaw rotation rotates the quadcopter around the vertical axis (z) while maintaining horizontal plane stability. Pitch is referred to when the UAV tilts forward, around the side-to-side axis (y), typically for achieving forward momentum. Finally, roll can be achieved when the UAV rotates around the front-to-back axis (x) for sideways motion. Two of the rotors rotates counter-clockwise (CCW), and the other two in the opposite directions rotates clockwise (CW).

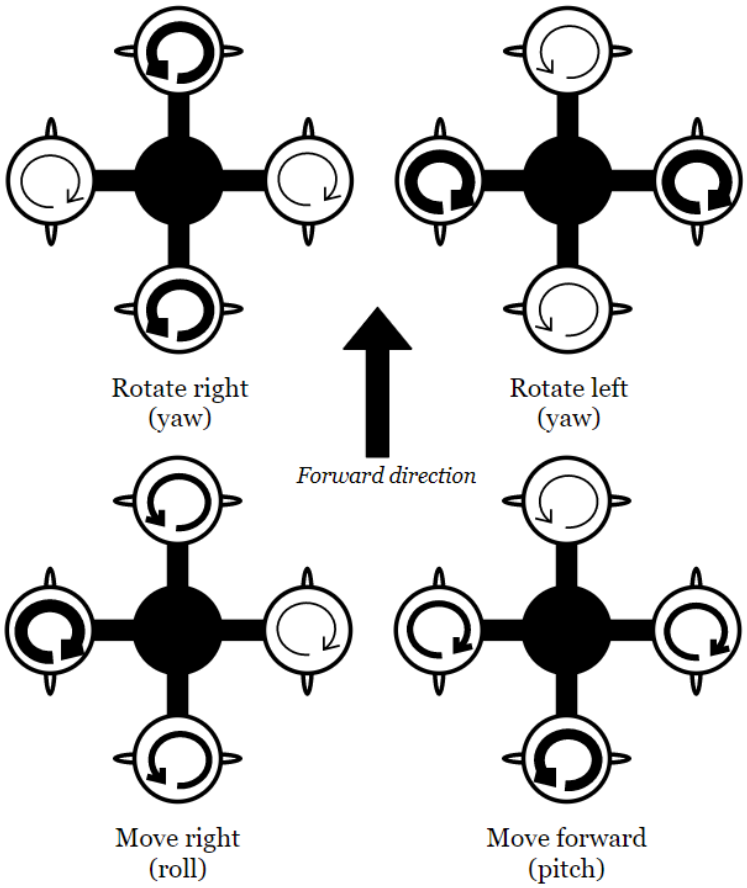


FIGURE 3.4: Quadcopter movement based on rotor thrust [4].

Understanding the basics of quadcopter movement can be achieved by observing the various forces that affects the UAV. Figure 3.5 illustrates four main forces and their directional pull on the quadcopter, and any typical aircraft.

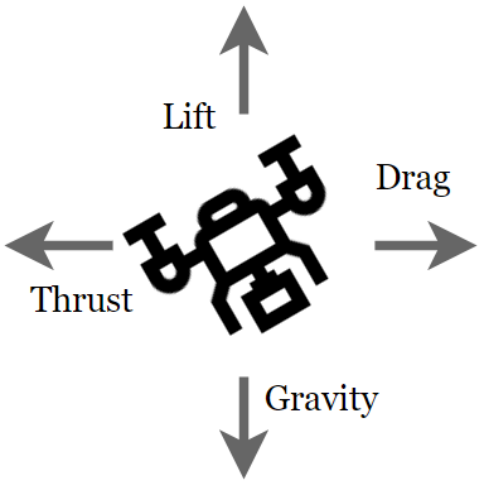


FIGURE 3.5: Forces acting on an unmanned aerial vehicles during flight.

From Figure 3.5, it is easily observed that to maintain height and stability, the amount of lift L must be greater than or equal to the gravitational force, such that $L \geq mg$. Not only for quadcopters but for any multi-rotor UAV, the sum of each force generated at each propulsor must be greater than mg in order to maintain height [4]. This means that the overall vertical ascend or descent F_z on the UAV can be represented as

$$F_z = \sum_{n=1}^{k} (L_n) - mg, \tag{3.6}$$

where k is the number of rotors on the UAV and m is the mass of the quadcopter respectively [4]. This equation is only true if each propulsor L_n generates the same amount of lift, otherwise the UAV will start to rotate in pitch, yaw, or roll directions, causing instability if not regulated correctly. If $F_z > 0$ the UAV will ascend, or the contrary if gravity is stronger than the lift generated by the rotors, i.e., $F_z < 0$. We achieve a hovering aircraft when $F_z = 0$, meaning the rotors generate just enough upwards lift to counteract the gravitational pull, maintaining the same altitude. Calculating lift L_n for each rotor depends on the air density and friction around the UAV, and the rotation speed of the blades in order to propel air downwards to create upward lift [25].

As seen from Figure 3.5, drag is a counteracting force which slows the UAV down due to friction and air in front of the UAV. The drag force F_d will be subtracted from the optimal thrust in vacuum in forward F_x or side direction F_y in order to create a realistic picture for UAV maneuverability. We can assume a quadcopter, and determine how to gain forward thrust based on the rotor individual lift. From Figure 3.4, we can observe that to achieve pitch and forward motion, the northern rotor lift L_N must be less than the southern rotor lift L_S. However, if the difference of rotor power between L_S and L_N is too large, the UAV's pitch angle will exceed its limits, which will cause loss in altitude and instability. The specification of the DJI phantom 3 quadcopter has a max tilt angle of $35°$.

3.2 Multiple UAVs and path cost

Using multiple unmanned aerial vehicles to communicate with each other and fly in formations, is a difficult task. In order to maintain control of the UAVs and prevent collisions, each UAV must know others exact position, as well as the velocities to approximate the positions in the next iteration. Based on the discussion earlier on graph theory, we can assume a set of multiple UAVs represented as A_i and starting positions in $S_i = \{x_i^s, y_i^s, z_i^s\}$ in global space. The total amount of way-points in the graph network that the UAVs can follow, consists of $Q - 1$ total points based on a path P_i of agent A_i from S_i to goal point $G_i = \{x_i^f, y_i^f, z_i^f\}$ [27]. Hence, the available path for each individual UAV is given as $P_i = \{W_0, ..., W_Q\}$, where each vertex is represented as a tuple $W_k = \{x_k, y_k, z_k\}$, for $k = 0, ..., Q$. This further gives the starting point when $k = 0$ and goal point when $k = Q$. Each UAV would then have a cost C_i to reach the goal which should be minimized. Vertical climb and decent costs should be taken into consideration between each vertex as the power usage increases during climbs and reduces during descents respectively. The cost can be calculated by finding the distance between vertices, and additionally use vertical climb and decent such that

$$C_i = \sum_{k=0}^{Q-1} (\|W_k - W_{k+1}\| + (V_c^k - V_d^k)), \tag{3.7}$$

which gives a cost based on path length between two vertices W_k and W_{k+1}, and vertical climb and descent gains V_c^k and V_d^k between each point, respectively [27]. By using this formula, we can derive different path planning algorithms from the calculated path cost. Paths can be calculated pre-takeoff, and later, determine if any planned paths intersect with other UAVs and recalculate respectively.

3.3 Proposed path planning algorithms

Based on the path planning algorithm types discussed in Chapter 2, several algorithms can be used for finding the shortest path whether it is a single source-, or an all-pair problem. Assuming a 2D environment with a tile-type grid, finding a path from source vertex s to v can be performed with simple greedy algorithms that chooses a vertex and calculates the costs for each neighbour. This follows the guidelines from shortest path given by Equation (2.2) in Chapter 2. Figure 3.6 illustrates an example environment in 2D, where the objective is to find the shortest path from source s to a given vertex v. This shows an example path that can be calculated using path planning algorithms for reaching the goal with the shortest path, while avoiding the obstacles at the same time.

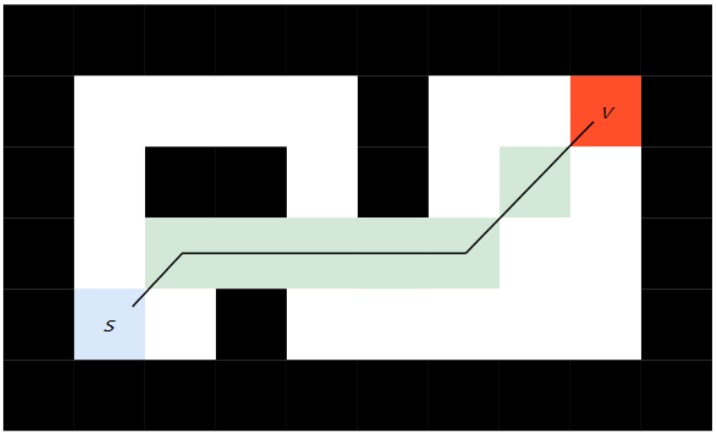


FIGURE 3.6: 2D tile-map of an environment illustrating the shortest path from s to v.

We can calculate the distances in a tile grid, since they always have a constant value from each other. The horizontal or vertical distance will always be equal to constant 1, which then ultimately can be put into distance Equation (3.9) to calculate the diagonal distance. This will get the following equation for distance between source vertex s to neighbour v in any 2D tile-grid:

$$\delta(s,v) = \begin{cases} 1 & \text{if } v \text{ is horizontal or vertical from } s, \\ \sqrt{2} & \text{if } v \text{ is diagonal from } s. \end{cases} \tag{3.8}$$

If there is no grid, the distance formula can be used to calculate the raw distance between two points s and v respectively as such

$$\delta(s,v) = \sqrt{(v_x - s_x)^2 + (v_y - s_y)^2}. \tag{3.9}$$

These equations can be used to solve for any 2D environment, where we can find the shortest path by applying any path planning algorithm, since they rely on the distance function [29]. However, when applying a new dimension to the problem so that it becomes more realistic for real life integration, height z also needs to be considered in $\delta(u,v)$, but the same distance formula still applies.

3.3.1 Dijkstra's algorithm

In regards to single source shortest path, an algorithm referenced as Dijkstra's algorithm can be used. This algorithm strictly starts by creating an empty list that will keep track of vertices that has been evaluated with a distance cost from source vertex s [10]. Further, the algorithm works by assigning a distance value of infinity to all vertices in order to compare the weights $w(u,v)$ with new distance values that has been found subsequently [16]. The time complexity of this algorithm is based on its need of iterating through each vertex v in a graph G exponentially. This yields a time complexity using the big O notation of $T(V,E) = O(|V|^2 + |E|)$.

Dijkstra's algorithm pseudocode

Algorithm 1 demonstrates the basics of Djikstra's algorithm, where it starts by initializing each vertex in a graph G with a value of infinity. Further, a list Q will be filled with all vertex elements in the graph, such that the loop in line 4 will evaluate all possible vertices, hence $O(|V|^2 + |E|)$. The goal is to start from the source vertex s, and arbitrarily visit its neighbours and giving them a respective cost based on the distance from s. In the next iteration, s' neighbours has an updated cost which is lower than the other vertices' cost of ∞, where the vertex with the lowest cost is chosen for evaluation. Assuming vertex u is the target for evaluation, its neighbour v will have a new cost of $C_u = w(s, v) + w(u, v)$ [10], as seen on line 8 in Algorithm 1. This is iteratively performed until queue Q is empty, and no more vertices are left to be evaluated. However, if vertex u for evaluation is equal to the target goal vertex, the algorithm has successfully found the shortest path, and a trace-back of the discovered path can be performed based on which vertices each vertex came from.

Algorithm 1 Dijkstra's algorithm in pseudocode [10].

```
1:  function DIJKSTRA(graph, source)
2:      set each vertex v cost to ∞
3:      add all vertices to queue Q
4:      while Q is not empty do
5:          u := minDist(Q, dist)
6:          remove u from Q
7:          for each neighbour v adjacent to u do
8:              v.cost = min(v.oldCost, u.cost + w(u, v))
9:          end for
10:     end while
11: end function
```

3.3.2 A* search algorithm

An improvement to the Dijkstra's algorithm by taking advantage of calculating a heuristic estimate to the goal vertex. This algorithm is referenced as the A* algorithm, where the idea is to find the lowest value of a given vertex v from source s as $g(n) = s \rightarrow v$, plus the heuristic estimate $h(n)$ from v to the goal in a graph G [32]. This can be calculated as a cost function $f(n)$ with a target node n such that

$$f(n) = g(n) + h(n). \tag{3.10}$$

Heuristic estimate

Finding the heuristic value to the end goal from a given vertex greatly determines the efficiency of the A* algorithm. It is important that the heuristic is admissible, which means that it never overestimates, but rather underestimates and stays lower than or equal to the actual cost [32]. Assuming a node n and a heuristic h, we can derive a heuristic estimate $h(n)$ with an optimal cost of $h^*(n)$ to meet the requirement of an admissible heuristic if

$$h(n) \leq h^*(n). \tag{3.11}$$

There are several way to estimate the distance to the goal vertex. Two of the most commonly used are the Euclidean and Manhattan distance estimations.

Euclidean distance

Calculating the raw distance from a target vertex to the goal using a distance formula can be used as a simple heuristic estimate, but it will always yield an estimate that is lower than the actual distance. The issue with that however, is that it does not consider any potential obstacles that interferes with the calculated line, thus may result in inaccurate paths. For two given points in three-dimensional space, u and v, the distance δ between the points is simply given by

$$\delta(u, v) = \sqrt{(v_x - u_x)^2 + (v_y - u_y)^2 + (v_z - u_z)^2}. \tag{3.12}$$

Manhattan

Manhattan distance, also referred to as Taxicab geometry or rectilinear distance L_n, simply calculates the distance between two points by adding the absolute differences of their Cartesian coordinates. In three dimensions, the following taxicab distance between point u and v is given by

$$\delta(u, v) = |u_x - v_x| + |u_y - v_y| + |u_z - v_z|. \tag{3.13}$$

Figure 3.7 represents the difference between the two mentioned heuristic estimates in two dimensions by subsequently using Equations (3.12) and (3.13) to calculate two different distance metrics.

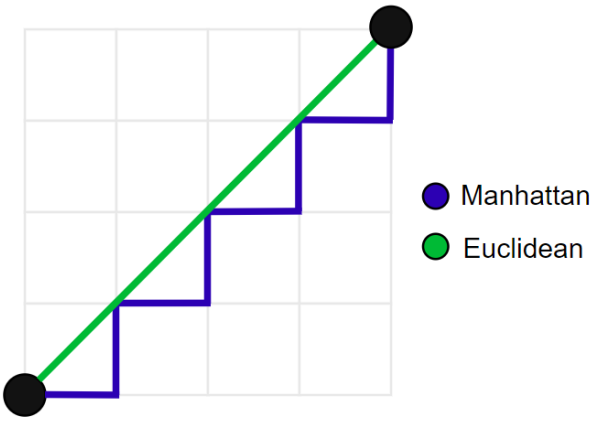


FIGURE 3.7: Manhattan- vs euclidean distance in a 2D tile grid environment.

A* algorithm pseudocode

Algorithm 2, illustrates a step by step implementation of the functionality of how the A star algorithm works. Allegedly, the first steps of this algorithm is to determine the start vertex and the goal vertex, given by the parameters in function call. Further, two arrays named openSet and closedSet contains all the vertices that need to be considered, and those that has already been checked respectively [31]. The vertex that is being evaluated found within openSet list, is referenced as $current$ in the given pseudocode with the lowest f score. The f score as discussed earlier, is a combined score of the calculated distance it has already travelled from the start vertex $g(n)$ and an heuristic estimate to the finish $h(n)$.

The algorithm further finds every neighbour of $current$ and iterates through if the target neighbour does not exist in the closed set. If it already exists in the closed set, then it has already been evaluated and the iteration continues to the next neighbour. If a neighbour has not yet been evaluated, the tentative g score must be calculated, which is the current vertex g score plus the distance between current and the target neighbour. Based on whether the neighbour does not exist in openSet, or has a lower tentative g score than its current g score, a new path has been found, and this neighbour vertex is part of the shortest path from start- to goal-vertex [32].

If $current$ is equal to goal vertex, then the algorithm has successfully found a path from start- to goal-vertex. This means a trace-back of the found path must be performed, hence a $cameFrom$ variable is being used. Since each vertex has the variable as seen in line 37 in Algorithm 2, we can use this to trace which vertex it originally came from, with a loop that iterates through these vertices. This will continue until we reach the start vertex, which supposedly did not come from any vertex as this was the starting point. This iteration will yield an array of all the vertices in the shortest path in descending order, starting from goal- to the start vertex, and we have successfully found the shortest path. Ultimately, if openSet list is empty, and if the goal vertex has not been found after we having iterated through every possible vertex from the start point, the function returns a failure as no path is available in given graph network.

Algorithm 2 Pseudocode for the A$*$ algorithm [31, 32].

```
1: function ASTARALGORITHM(startVertex, goalVertex)
2:     openSet := {startVertex}
3:
4:     while openSet is not empty do
5:         current := the node in openSet with lowest fScore
6:
7:         if current = goalVertex then
8:             tempVertex := current
9:             while tempVertex has no  cameFrom do
10:                add tempVertex to resultPath[]
11:                set tempVertex to vertex it came from
12:            end while
13:            return resultPath[]
14:        end if
15:
16:        add current to closedSet
17:        remove current from openSet
18:
19:        for each neighbour of current do
20:            if closedSet contains neighbour then
21:                continue                    ▷ //Already checked, skip iteration
22:            end if
23:            tentative_g := g[current] + d(current, neighbour)
24:            newPath = false
25:            if openSet contains neighbour then
26:                if tentative_g < g[neighbour] then
27:                    g[neighbour] = tentative_g
28:                    newPath = true
29:                end if
30:            else
31:                g[neighbour] = tentative_g
32:                add neighbour to openSet
33:                newPath = true
34:            end if
35:            if newPath is true then    ▷ //If better g_score, or does not exists in openSet
36:                f[neighbour] = g[neighbour] + h[neighbour]
37:                set neighbour's cameFrom to current
38:            end if
39:        end for
40:    end while
41:    return failure                          ▷ //openSet is empty, but goal not found
42: end function
```

3.4 PID Controller

Proportional-integral-derivative controller (PID, or three-term controller) is an iteration process for calculating error signals and responding with feedback for control systems [18]. In this regard, we initially have a path for the UAV to follow based on earlier path planning algorithms, in which its position relative to this path can be used. PID controller can be derived into four main aspects, typically

- The **sensed position** of target unit represented as a process-variable PV;
- The **desired position** referenced as setpoint SP;
- An **error** e calculated by differentiating $SP - PV$; and
- The **input** of the whole process, such as applied power to an UAV's rotors (propulsors) so that its position changes accordingly. Represented as the **manipulated value** MV or **control variable** CV.

Based on this information we can use the PID controller algorithm as seen from Equation (3.14) to control the motion of the UAV so that it follows the shortest path [18]. The manipulated value $MV(t)$ is then how much we want to adjust the UAV position so

$$MV(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}, \tag{3.14}$$

where t is the instantaneous time and τ is the current variable of integration that takes values from the start to present t. Moreover, K_p is the proportional tuning parameter, K_i is the integral tuning parameter, and K_d is the derivative tuning parameter. Choosing the correct values for these parameters substantially impacts the performance of the PID controller [18].

3.4.1 The proportional term

The proportional gain constant K_p is multiplied with the current error value in order to create a number proportional to the error. This is the first part of the PID controller algorithm, where the proportional output is

$$P_{Out} = K_p e(t). \tag{3.15}$$

This means that a high K_p makes a high output value P_{out} and in turn makes $MV(t)$ change drastically based on a given error. This will make the UAV become unstable due to the high output values for propulsors. On the contrary, a small number impacts the system in a way that the adjustments in the control system are too minor, resulting in a less sensitive control system [18].

3.4.2 The integral term

It is important to measure the error signal over time, meaning that the integral term refers to both the magnitude and duration of the given error e over time t. It is generally

the sum of the instantaneous errors over a time-span, represented as

$$I_{Out} = K_i \int_0^t e(\tau)d\tau, \tag{3.16}$$

accumulates previous errors that should have been corrected in the system. The benefits of using an integral, is that it prevents the residual steady-state error from occurring [12]. The steady-state is simply the difference between the actual output value of $MV(t)$ and the desired output value. Furthermore, by using the integral in the algorithm, the movement towards SP is accelerated based on previous errors, however might overestimate, or overshoot, the output value to SP due to previous errors. Finally, K_i is multiplied with this integral, which in turn can be used to fine-tune the performance of the algorithm [18].

3.4.3 The derivative term

The derivative of the algorithm is optimally not causal, meaning it only calculates future behaviour of the UAV. This can be done by determining the slope of the error over time to ultimately create a stabilized system and multiplying this rate with the derivative gain K_d. The derivative output D_{Out} of the equation is

$$D_{Out} = K_d \frac{de(t)}{dt}. \tag{3.17}$$

However, since the derivative part of the algorithm is not causal, it is rarely used in real-world systems, where predictions are never real values and only estimates which may lead to unreliable, unstable systems [12, 18].

3.4.4 Paths and positions for UAV

Applying PID controller to UAV motion and following a path can be done by firstly determining what is the set-point, process-variable, and the manipulated value. Ultimately, Equation (3.14) can be used to calculate positional errors of the UAV, based on the path it is supposed to follow. From Figure 3.8, it can be observed that we can assign the process-variable PV as the global position of the quadcopter, and a desired position towards current edge to be followed in the planned shortest path as set-point SP.

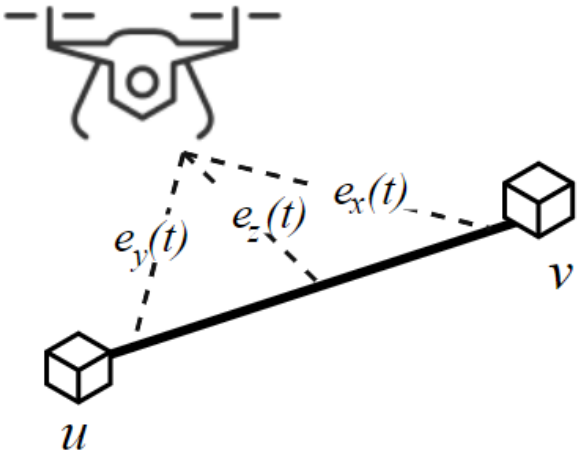


FIGURE 3.8: UAV error relative the target path from edge $e(u, v)$.

The error can be calculated by determining the positional errors in each direction, such that each set-point and process-variable in each dimension is

$$\begin{cases} MV_x(t) = e_x(t) = SP_x(t) - PV_x(t), \\ MV_y(t) = e_y(t) = SP_y(t) - PV_y(t), \\ MV_z(t) = e_z(t) = SP_z(t) - PV_z(t). \end{cases} \tag{3.18}$$

For instance, observing an error value in the roll-axis $e_x(t)$, the control system knows that it can apply power accordingly to certain rotors to roll sideways towards the path. To achieve this, it is necessary that the UAV knows the position of the edge between vertex u and v. However, this edge does not strictly have a position, but rather a direction and a start and end-point. In this case, finding the set-point is troublesome, due to the uncertainty of the exact location between two vertices [26].

Figure 3.9 illustrates further how to calculate the error from the UAV to the path, by creating more vertices between main points, in order to assign more optimal set-points SP.

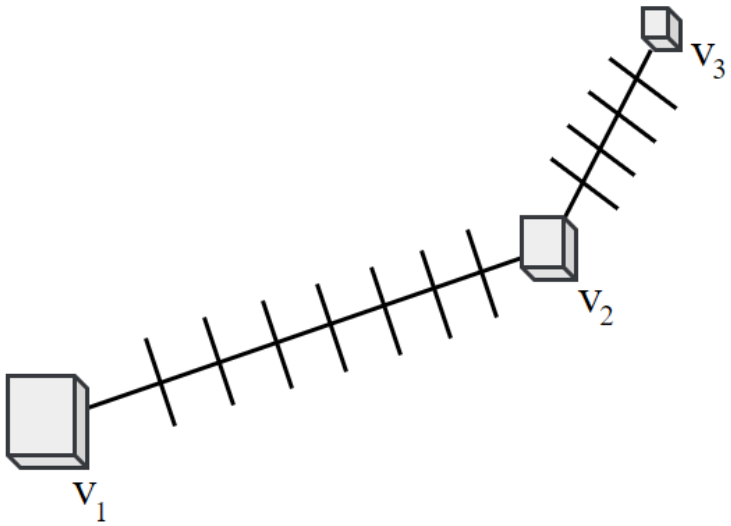


FIGURE 3.9: Graph network $G = (3, 3)$ where more vertices can be added represented as lines for increasing network accuracy.

Adding more vertices between two points is beneficial so there are more available positions to be assigned as SP. Optimally, SP is set as the closest vertex to the UAV in the graph network. By this, the PID-controller maneuvers the UAV accordingly to stay on the path. Once the UAV is on the path, it ultimately knows the next target vertex and can maneuver towards this point accordingly.

Chapter 4

Simulations and implementation

This chapter discusses simulations and implementation aspects of various UAV behaviors and paths, to demonstrate the feasibility of the proposed path planning system.

4.1 Unity 3D simulation engine

In this work, two simulation engines have been researched, referenced as game engines Unity 3D and Unreal Engine respectively. The benefits of using game engines, is that they provide a good user interface and built-in modules ranging from movement physics, gravity, rigidbodies and quaternions for rotation physics [23]. Further, both Unity 3D and Unreal engine provide modules for simulations and game creation. However, Unity has more built-in modules for managing textures and shaders, which results in simplicity for development that in turn provides less versatility for graphics. Nonetheless, Unreal engine is more focused towards graphics and provides a generally more appealing look to the game simulation [23, 28]. Ultimately, graphics and general view of the simulation is not as important as the overall physics and object behaviors. Additionally, based on previous experiences with simulations and programming Unity 3D has been used in this work.

The **software requirements** for simulation can be seen in the list below, based on tools that are needed to provide simulated results.

Unity 3D student package

In this work, the student plan for Unity 3D has been used, in which all files and materials used are open-source and free to use [23]. This is important in order to provide availability of source files and readability, especially in regards to re-creating simulations.

Visual studio 2017

Visual studio has been used for code editing and as an integrated development editor (IDE) in regards to the code scripts used by Unity. Unity currently uses the C# programming language, which provides a user friendly way of changing attributes and behaviors of objects found within the Unity simulation.

Unity asset - Windridge City Demo

Furthermore, an asset from the unity store to represent a city for drone simulations has been used [35]. This asset is completely free and consists of materials and textures of

city objects. Additionally, the Post Processing Stack v2 included in the Unity installation can be used for solely improving the graphics.

AirSim - Quadcopter model

The AirSim is an open-source library for both Unity and Unreal engine [30]. However, only the quadcopter model has been used in this work. No drone physics or maneuverability has been used from this library, as everything has been implemented independently.

As a reference during the experiments regarding calculation times and performance, the used **hardware** from a desktop computer can be seen in Table 4.1.

TABLE 4.1: Simulation computer hardware specifications.

Desktop computer specifications	
CPU	Intel i7 4790 @ 3.60 GHz
GPU	Nvidia GeForce GTX 1070
RAM	4x KHX1600C9D3 Kingston 4GB DDR3
Motherboard	MSI Z97 GAMING 7

4.1.1 Development in Unity 3D

Creating simulations and virtual environments in Unity 3D can be done by adding wanted objects directly into any given scene. A scene in Unity is a representation of a created virtual environment, with a hierarchical view of all objects in the simulation, and their given rotation, global position, and size as attributes. Figure 4.1 illustrates all simulated game objects within the simulation environment, ranging from landscape objects, controllers, lightning as well as the drone itself.

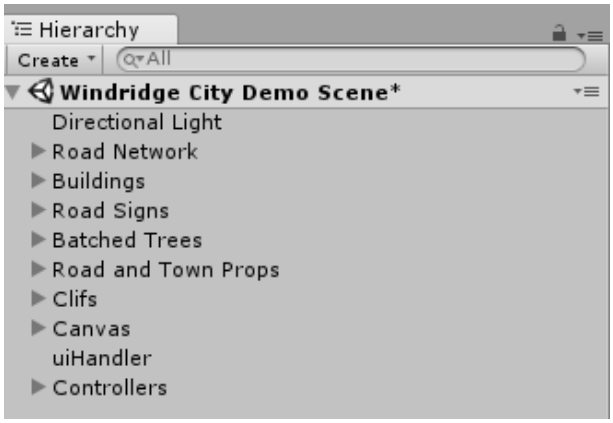


FIGURE 4.1: Unity scene hierarchy for Windridge, containing all objects in simulation.

One can simply right click in the scene viewer to create new objects, whether they are empty game objects, or lightning objects such as city lights and lamps which supports lightning from the built-in physics engine. Figure 4.2 shows an overview of the Windridge city environment from Unity assets, which is the environment being used in this work.

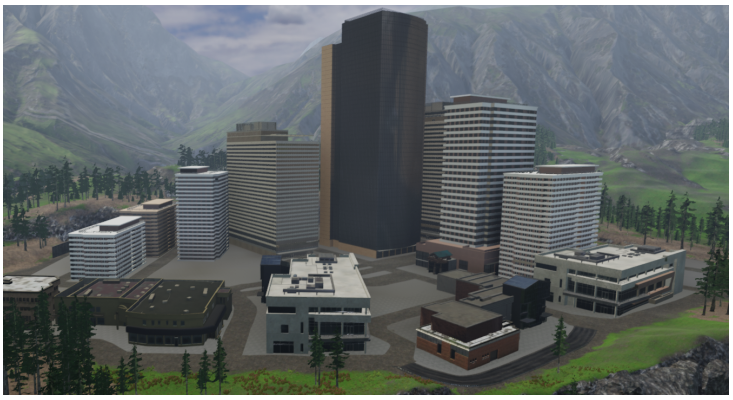


FIGURE 4.2: Windridge city in Unity, used for simulation as environment for drone.

Gameobjects within scene

Furthermore, each of these objects within the hierarchy window can be inspected such that attributes of the object is shown. This can be seen from Figure 4.3, where the global earth fixed frame F^E position, can be changed into any x, y and z coordinates.

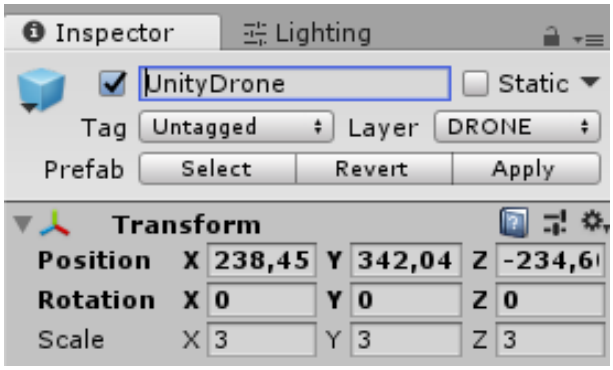


FIGURE 4.3: Inspector window of a given game object, in this case the drone itself.

Additionally, the rotation of the object is represented as Euler angles for ease of use by the user. However, Unity internally uses quaternions to prevent the case of *Gimbal Lock*. The rotation represented in this window is relative to the world axis.

When creating objects that are to be simulated, one can use the module known as a rigidbody. This module can be observed in Figure 4.4 that contains built-in attributes such as gravity, which is 9.81 ms^2 by default. Additionally, drag for air resistance and friction can be changed, as well as angular drag that slows rotation due to internal frictions within the game object.

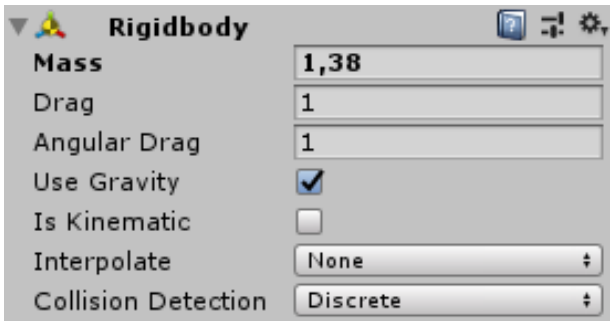


FIGURE 4.4: Unity rigidbody type with mass, and both drag and angular drag for air friction.

Rigidbody is a module with built-in physics, where the most common used is *AddForce(Vector3 dir)* function. The passed *dir* parameter indicates the amount of force applied at the origo of the game objects position, i.e., the center of the UAV. This means to initially get a better representation of a real drone, a function that is widely used is *AddForceAtPosition(Vector3* dir, Vector3 pos) function. This function takes two parameters that firstly is a force as a 3D direction vector with magnitude, as well as the position on the object the force is applied. By this, we can apply four forces in the UAVs current upwards direction, initially where the four rotors are to be positioned. Utilizing these functions is performed by attaching a script module to the game object, where an example script of the drone can be seen in Figure 4.5.

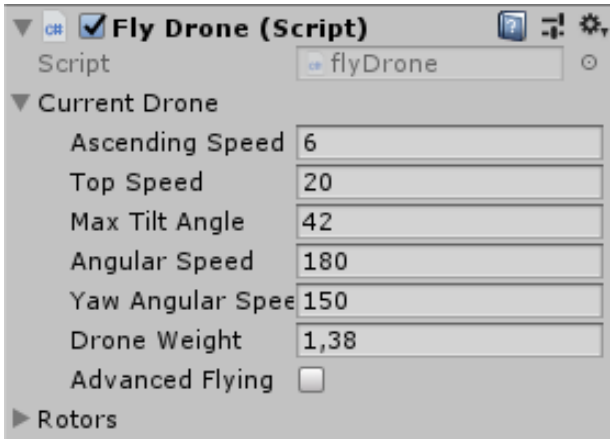


FIGURE 4.5: Script attached to drone object for configurations.

This script is a C# file (.cs) which can be opened in a IDE to be edited. Algorithm 3 shows the different variables that can be declared in the IDE, which will automatically update in the inspector window.

Algorithm 3 : Drone class specification for updating in the inspector window.

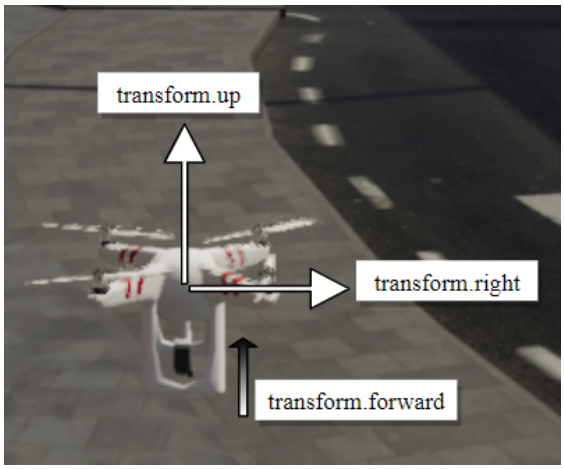
```
1: function DRONECONFIGURATIONS
2:     public float ascendingSpeed = 6.0f; // m/s
3:     public float topSpeed = 20f; // m/s
4:     public float maxTiltAngle = 42; // degrees
5:     public float angularSpeed = 150; // degrees/s
6:     public float yawAngularSpeed = 150; // degrees/s
7:     public float droneWeight = 1.380f; // grams
8: end function
```

This gives an easy user friendly way of changing simulation values, for testing various simulation results. For instance, the weight of the drone can be changed in order to simulate different types of quadcopters based on their specifications found in the spreadsheets.

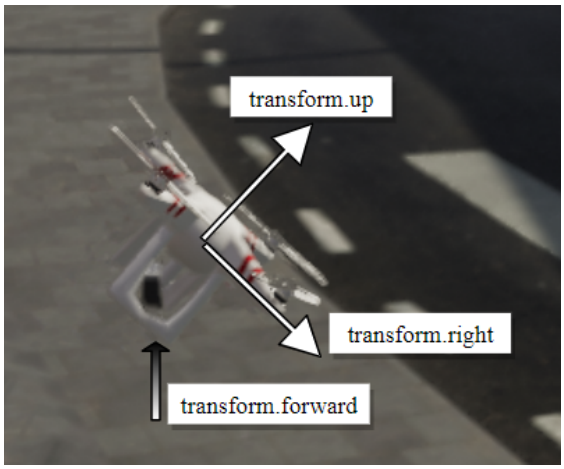
4.2 Simulating an unmanned aerial vehicle

Using Unity 3D, simulating unmanned aerial vehicle can be achieved with the built-in modules which handles collisions based on boundary boxes around each object as well as movement and physics. In this work, a quadcopter has been created from an empty object which ultimately consists of 4 points at each corner that represents the rotors. Based on the drone variables derived from Algorithm 3, it is possible to simulate the drone physics. Firstly, a hover force H is calculated based on the given mass and physics with Equation (3.6). This means that the sum of all rotors applied is equal to the gravitational pull on the aircraft. However, another solution is to have a main force F_z in the quadcopters up direction, and use the rotation functions to roll, pitch, and yaw the UAV.

Figure 4.6a illustrates the flying drone in the simulation environment in a stable state, with its respective body fixed frame F^B axis, here transform.up is parallel to global frame Vector3.up.



(A) Quadcopter in a given simulation environment where transform.up ∥ Vector3.up.



(B) Quadcopter is rolling around the forward axis where transform.up ∦ Vector3.up.

FIGURE 4.6: Indicating drone movement in global space, hovering vs roll.

Table 4.2 shows the axes in the simulation environment where F^B is represented with the transform keyword in Unity, whereas the earth-fixed frame F^E is represented with Vector3 keyword.

TABLE 4.2: Axes in simulation environment.

Axis	**Global frame F^E**	**Body frame F^B**
x	Vector3.right	transform.right
y	Vector3.up	transform.up
z	Vector3.forward	transform.forward

Based on that, it is possible to simulate drone movement with rotation around its corresponding body frame F^B axes, as roll can be achieved by rotating around transform.forward axis, pitch around transform.right and yaw around transform.up axis respectively.

To achieve rotation on the UAV, one can either apply varying power to each rotor in order to achieve roll, pitch and yaw as discussed from Figure 3.4. However, a simpler method is by using the Rotate function within Unity. The function is namely Rotate(*Vector3 eulers*, *Space relativeTo*), where the *eulers* parameter takes the three coordinates x, y and z and rotates based on this. A Vector3 eulers parameter value of $v = (0, 5, 0)$ will rotate the object in 5 degrees clockwise around the y-axis, or the up direction. This will in turn create a yaw rotation of the UAV, where the same can be applied to x and z axes to achieve pitch and roll respectively. The *relativeTo* parameter decides whether the object is rotating in regards to world space from earth-fixed frame F^E or locally with the body-fixed frame F^B.

On the other hand, a camera has been attached beneath the drone, for surveillance and monitoring the environment for landmark objects and other external information. This camera can be tilted up and down around local x-axis, and can be seen in Figure 4.7.

FIGURE 4.7: Drone camera for monitoring and usage for environmental data.

Like other cameras found on modern quadcopters, a stabilizing technique is performed, using gyro. This means that the only direction the camera is supposed to change if the UAV rotates, is the yaw direction, whereas the UAV can roll and pitch regardless without affecting the camera rotation. This is beneficial for gather environmental data to minimize the noise from vibrations and UAV movement.

4.2.1 Overview of drone simulation

Figure 4.8 illustrates the drone simulation as discussed in this work. Information about the drone is provided, ranging from speed to rotation in angles in each direction.

FIGURE 4.8: Drone simulation with additional information such as drone status, and camera feed.

Firstly, the green panel contains information regarding the current performance of the simulation, referenced as frames per second (FPS) as well as the vertical and horizontal speed of the UAV. Additionally, the blue panel gives information about its rotation, as well as the hover force, which is the amount of force needed to achieve a hovering state of the UAV based on its weight. Altitude from sea level as well as the drone camera are also shown.

4.3 Path planning implementation

As the discussion continues, the UAV now has to follow, and plan out a path. A graph network tool has been created in the same simulation environment which represents various nodes, or vertices with corresponding edges. This graph network tool has been made so that optimially the UAV can maneuver and map out various vertices and store their representative positions into its system. Furthmore, algorithms for planning the shortest path has been implemented, such as the A^* algorithm, as well as the Dijkstra's algorithm to some extent.

4.3.1 Graph network

The implementation of the graph network in this thesis is created as a C# script attached to an empty game object, which allegedly consists of three, objects, vertex, edge, and the network as a whole. The graph network object G further contains two lists of vertex V- and edge E objects, giving that $G = (V, E)$. Additionally, important functions such as adding vertices, adding edge connections and getting total number of

edges and edges is found within the graph network object. Each vertex has a position represented as a Vector3 in C# format, which purportedly contains information of three-dimensional space relative to an origin. Adding a vertex to the graph network can be seen in Algorithm 4, which takes two parameters such as the position in global space of new vertex, and its neighbour vertices.

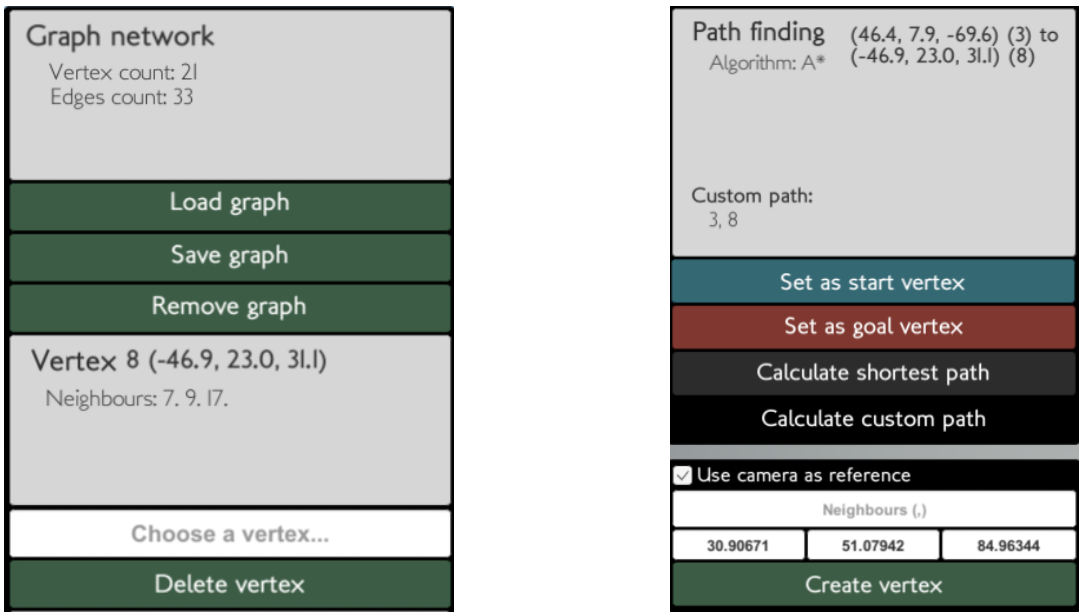
Algorithm 4 Add vertex to graph network.

```
1: function ADDVERTEX(position, neighbours[])
2:     add vertex index to vertexList[] with position in graphNetwork
3:     for each neighbour in neighbours[] do
4:         create edge between neighbour and new vertex
5:         update neighbour's neighbourList[] with new vertex
6:     end for
7: end function
```

Implementing the graph network this way, means that we get a list of every vertex that exists in the network represented as a $N-$tuple where N is the number of vertices in the network. If there is the necessity of accessing a vertex' neighbour, we can simply look into this tuple, to access the vertex objects. This is required to successfully calculate the shortest path and use algorithms, such as the A* algorithm.

4.3.2 Graph network user interface

Figure 4.9a represents the general user interface of the graph network tool, which ultimately can load, save, and remove a graph respectively.



(A) Graph management and deleting vertices.

(B) Path finding algorithm specification as well as creating new vertices.

FIGURE 4.9: Graph network tool user interface.

Additionally, Figure 4.9b represents the path finding algorithm specifications, such as the source vertex s to be used as a starting point to a target goal vertex v. Further,

Figure 4.9b also has the ability to create new vertices in the network as follows. Firstly, the new vertex neighbours can be selected based on the index of target neighbours. For instance, an input in the text field of $0, 3, 5$ indicates that the new vertex to be created v_{n+1} has neighbours 0, 3, and 5 from the graph network. The three boxes above the create vertex button can be used to determine the position of the vertex in global space in x, y and z respectively. However, when integration with the drone, an option that is seen in Figure 4.9b as *"Use camera as reference"*, can be used to create vertices based on the position of the camera, or in this case, the UAV global position. By this, it is possible for the UAV to fly around in an area to map a graph network, in which all the vertices can be accessed regardless of the UAV position. When a vertex is being created, it is drawn in the simulation scene window, so that it is available to observe the position of the different vertices in the network. Lastly, based on the neighbour inputs, edges are being created which are strictly drawn lines in the simulation. Note that for simulation purposes, there exists two graph networks, one that is visually appearing for the user such as drawn squares and lines, and another that is the graph network the drone has stored into its system.

4.3.3 Path planning algorithms

Since the foundation of the graph network has been developed, we can simply integrate the path finding algorithms discussed earlier, based on their pseudo-code algorithms, as seen from Algorithms 1 and 2. However, it is also necessary to communicate information between the game objects and controllers, such as the number of neighbours to a vertex or, more specifically, the complexity of the graph network. Based on the graph network discussed from Algorithm 4, we additionally need **get** and **set** functions in the object. This means that we can simply call a function such as getNumNeighbours(*int vertexIndex*) or getNumVerticies() which both return number of neighbours to passed parameter vertex, and number of vertices in a given graph network respectively.

4.4 Simulation results and discussion

From the implementation discussed earlier, we are able to derive test results from observing the performance of the discovered path planning algorithms and additionally the flying capabilities of the UAV simulation.

4.4.1 Path planning

Figure 4.10 shows a simulated environment with the graph network tool. This tool can be utilized to simulate different paths calculated with algorithms. In this specific case in Figure 4.10, the standard A* algorithm was used with the implementation discussed in Section 3.3.2, to identify the shortest path from vertices source vertex $s = 0$ and target vertex $v = 17$ using the Manhattan heuristic.

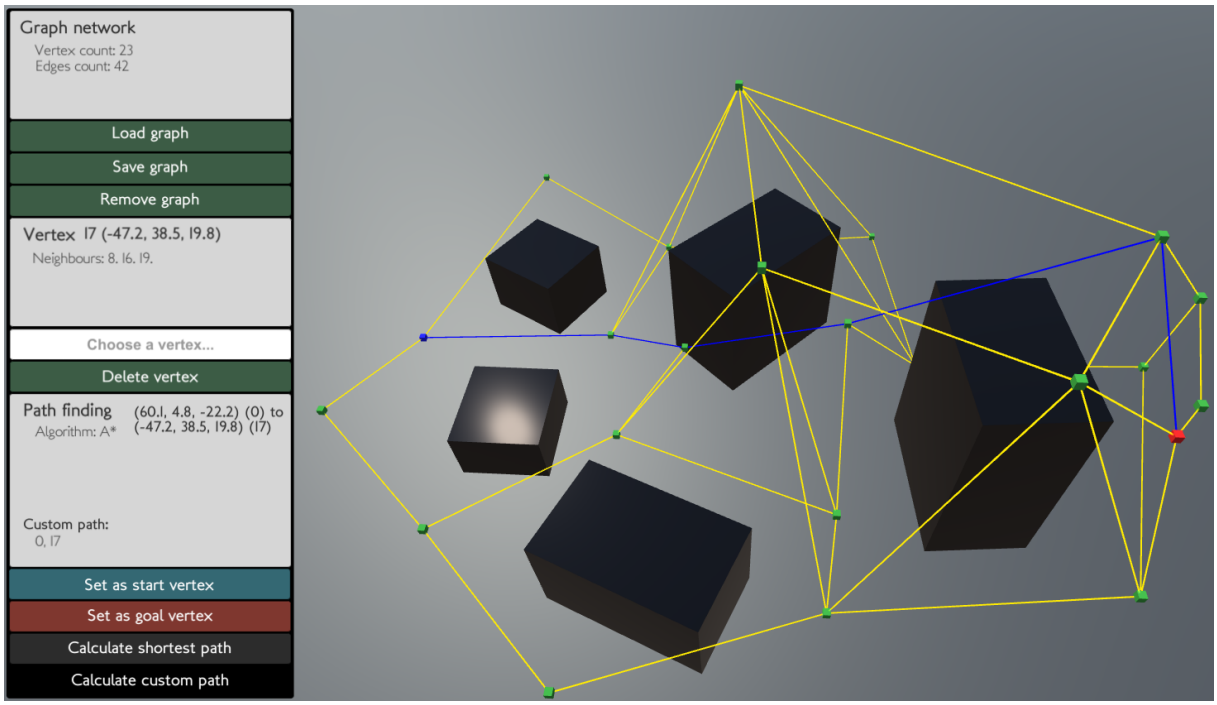


FIGURE 4.10: Graph network with shortest path from a source vertex s and goal vertex v.

From earlier simulations, we can also get numerical values such as execution time for the various path planning algorithms based on the chosen parameters. This includes heuristic estimates for the A* algorithm, the implemented algorithm type, as well as the complexity of the graph network. From Figure 4.11 we get information regarding the path planning algorithm, including the time spent calculating the path.

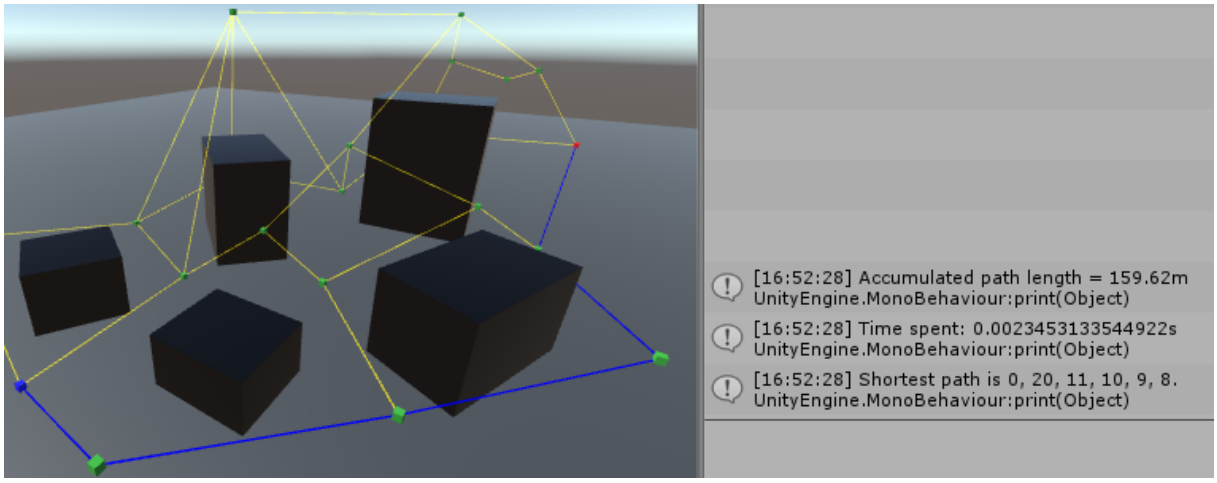


FIGURE 4.11: Calculated shortest path from vertex $v_0 \rightarrow v_8$.

However, since this is a simulation, we have to account for the fact that there are other simulation objects, such as rendering physical objects, and drawing the user interface. By this, we can initially create a graph network consisting of only one node, and thereby calculating the time spent finding shortest path. Based on the implementation from Algorithm 2, we see that once the current vertex being tested is equal to the goal vertex, the algorithm finishes. This means that calculating the shortest path

from the same target vertex to itself, only takes one cycle. Furthermore, by utilizing the $Time$ library within the Unity C# scripts, we can set a variable $startTime$ to the internal system clock on the simulation device from Table 4.1 as the initial time before calculating the path. After the shortest path has been calculated, we can compare the new system clock with the $startTime$ and get the final computational time for the algorithm. Firstly, by having a graph network $G = (1, 0)$, we get an approximate calculation time of $1.910ms$, which is managing the game objects behavior and external simulation information that is inconsequential for the algorithm performance. Table 4.3 illustrates 10 runs with $G = (1, 0)$ to get a mean base value for the average calculation times based on rendering and game object physics.

TABLE 4.3: Mean value of calculation times **without** path planning algorithm.

Iteration	**Execution time**
1	1.892ms
2	1.892ms
3	1.892ms
4	1.953ms
5	1.892ms
6	2.014ms
7	1.892ms
8	1.892ms
9	2.014ms
10	1.770ms
Mean	1.910ms

We can set this as the base time $b = 1.910$ms and can use this to subtract from the time we get during testing of the various algorithms and parameters. For instance, as shown in Figure 4.11 we can see that the time spent calculating the shortest path in a graph network $G = (21, 33)$ took $E_t = 2.345$ms with the A* algorithm with Manhattan heuristic from the console window to the right. Note that this is programmed to iterate $n = 10$ times to get a more accurate numerical representation. To eliminate rendering times and other external factors, we get a calculation time of

$$T(G) = E_t - b, \tag{4.1}$$

and further

$$T(G = (21, 33)) = 2.345\text{ms} - 1.910\text{ms} = 0.435\text{ms}. \tag{4.2}$$

Based on this given information, we can derive tests for various parameters to calculate the execution times for the algorithms in a given graph network. Table 4.4 illustrates the calculation times $T(G)$ for the different parameter values in three different graph networks. Which in this case, is by using the Euclidean heuristic estimate.

TABLE 4.4: Calculation times using the A* algorithm with Euclidean heuristic estimate.

Graph Network G = (V, E)	**Heuristic estimate**	**Calculation time** $T(G) = E_t - b$
G_1 = (21, 33)	Euclidean	0.484ms
G_2 = (21, 25)	Euclidean	0.478ms
G_3 = (50, 80)	Euclidean	1.159ms

Further, we have the Manhattan heuristic estimate as seen from Table 4.5 implemented using Equation (3.13).

TABLE 4.5: Calculation times using the A* algorithm with Manhattan heuristic estimate.

Graph Network G = (V, E)	**Heuristic estimate**	**Calculation time** $T(G) = E_t - b$
G_1 = (21, 33)	Manhattan	0.395ms
G_2 = (21, 25)	Manhattan	0.388ms
G_3 = (50, 80)	Manhattan	0.934ms

From Equations (3.12) and (3.13), and Tables 4.4 and 4.5, it is seen that using a quadratic formula for calculating the distances significantly slows down the execution times, while on the contrary the Manhattan heuristic is much faster due to calculating only the absolute value differences in each axis.

Additionally, during the test of the A* algorithm, the source vertex s and target goal vertex v were equal for both the Euclidean and Manhattan test in G_1, G_2, G_3 respectively.

4.4.2 UAV - Quadcopter simulation results

Finally, we can utilize the simulated drone in the Windridge city asset, and maneuver around the city based on a given implemented graph network. Figure 4.12 illustrates an example path around a city block in Windridge city, for the UAV to follow.

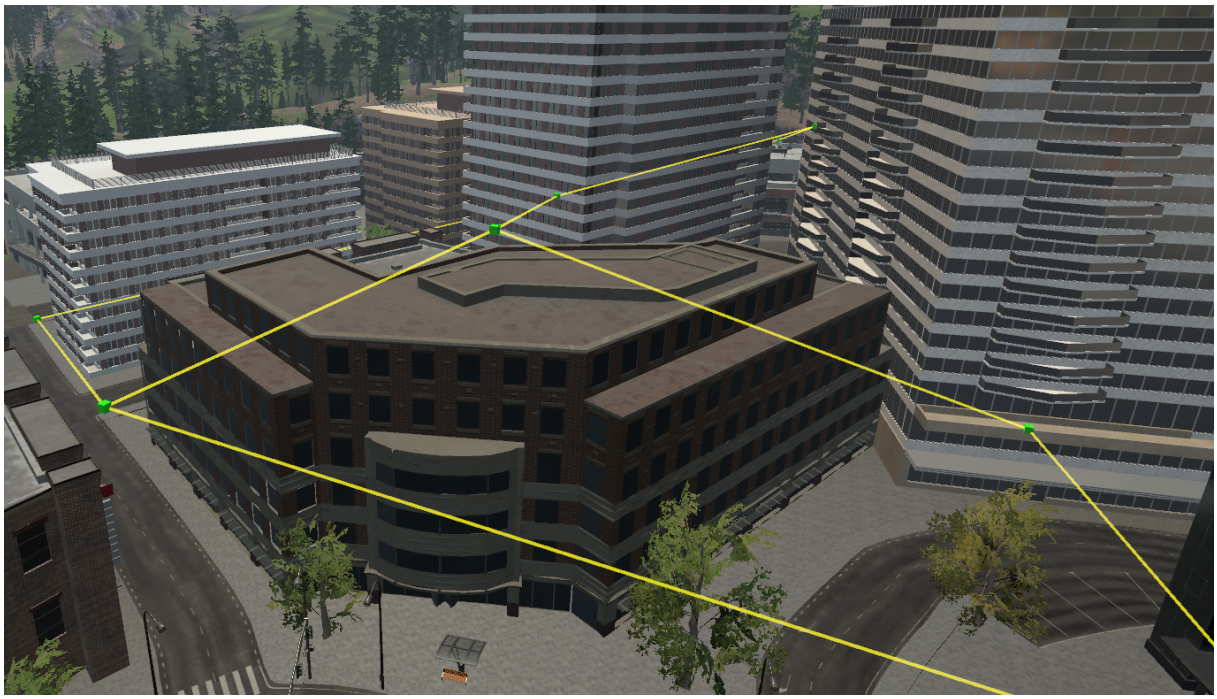


FIGURE 4.12: Example graph network in the city, increasing the number of vertices may be beneficial for accuracy, $G = (10, 12)$.

Furthermore, Figure 4.13 represents the UAV alongside the path to follow, which can be applied anywhere in the city.

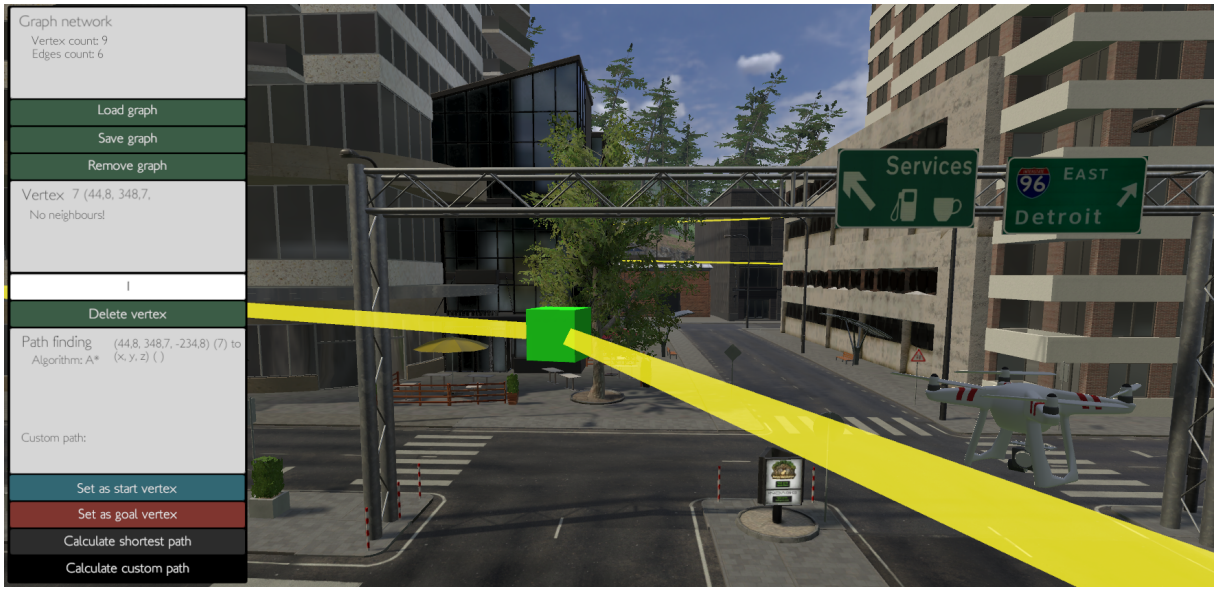


FIGURE 4.13: Drone following path in the city environment from graph network.

By this, it can be seen that it is possible to map out an entire city by flying an UAV and mapping the vertices and storing their position and connections internally within the UAV, and then further calculate the shortest path to any target position in the city. This will ultimately generate a path for the UAV to follow, in order to reach the destination as effectively as possible.

Lastly, Figure 4.14 illustrates a small graph network in the city, and additionally the shortest path between source vertex $s = 4$ and target $v = 15$.

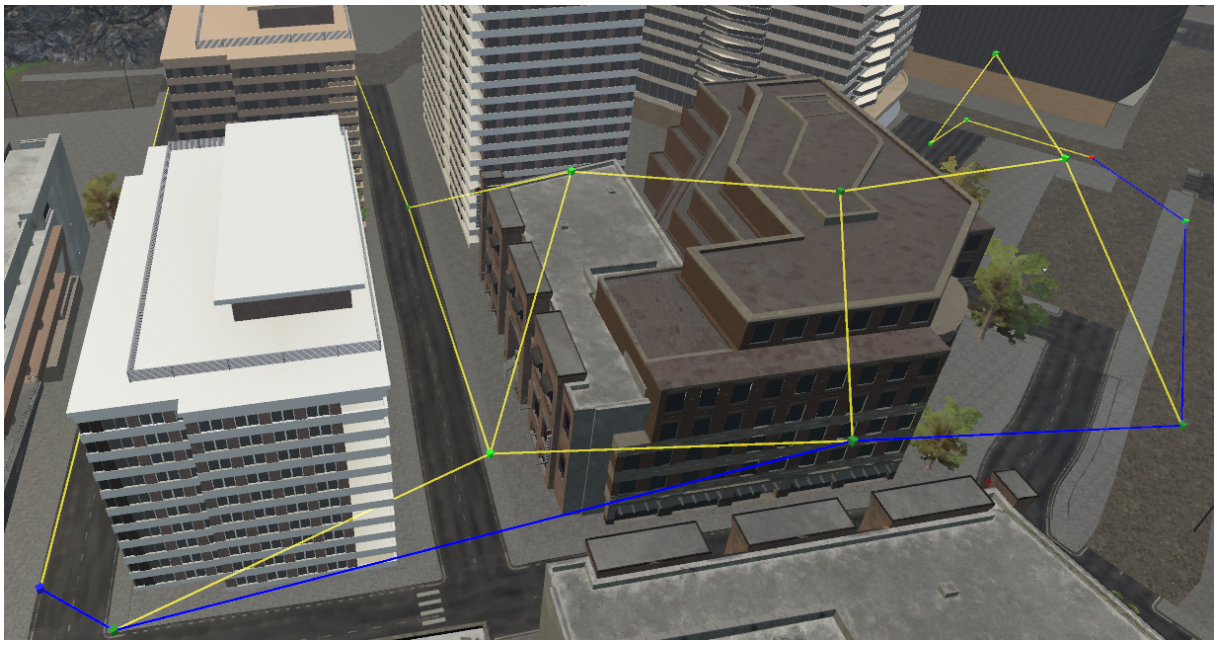
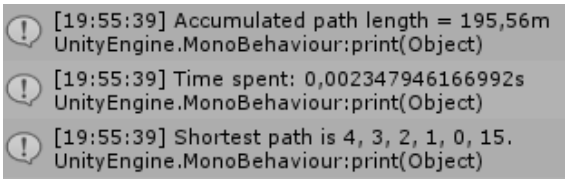
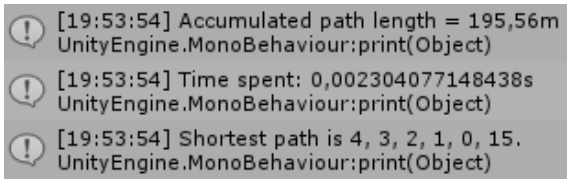


FIGURE 4.14: Small graph network in city, representing shortest path with the blue line indication from s to v, $G = (16, 21)$.

We can calculate the execution time for finding the shortest path by subtracting the base execution time b from the actual execution time gathered from the output in the console as seen in Figures 4.15a and 4.15b with Euclidean and Manhattan heuristics respectively.



(A) Euclidean heuristic used in Figure 4.14.



(B) Manhattan heuristic used in Figure 4.14.

FIGURE 4.15: Euclidean vs Manhattan heuristic in A* algorithm.

We can see the calculated time for generating the shortest path for both Euclidean and Manhattan heuristics, and by using Equation (4.1), we get

$$T(G = (16, 21))_{Euclidean} = 2.347\text{ms} - 1.910\text{ms} = 0.437\text{ms}, \text{ and} \tag{4.3}$$

$$T(G = (16, 21))_{Manhattan} = 2.304\text{ms} - 1.910\text{ms} = 0.394\text{ms}. \tag{4.4}$$

4.4.3 Heuristic estimates and performance

Both the Manhattan and Euclidean heuristics calculated the same exact path, which is the optimal route from s to v in the graph network from Figure 4.14. This means that in

simple weighted graph networks, the Manhattan heuristic might be preferred due to the fact that it executes an average of 18.38% faster than Euclidean based from Table 4.2. Additionally, it is illustrated from this table that a calculation time in $G = (V, E)$ can be created as a function based on number of vertices and edges in a network as

$$f(G = (V, E)) = K_V V + K_E E, \tag{4.5}$$

where K_E and K_V are constant numbers from execution times of edges and vertices respectively. K_E can be calculated based on numbers from Tables 4.4 and 4.5. The idea is that G_1 and G_2 contains the same amount of vertices, however the only difference is the amount of edges found in each network. We can then calculate how long it takes to calculate one edge with

$$K_E = |\frac{T(G_1)}{E_{G_1}} - \frac{T(G_2)}{E_{G_2}}|, \tag{4.6}$$

and further

$$K_E = |\frac{0.484\text{ms}}{33} - \frac{0.478\text{ms}}{25}| = 0.00445\text{ms}. \tag{4.7}$$

Similarly, K_V can be calculated by finding the fraction between execution time $T(G)$ and number of vertices V. It is also necessary to subtract the total execution time by execution time for each edge, i.e.,

$$K_V = \frac{T(G) - K_E E}{V}, \tag{4.8}$$

and further from G_1 with Euclidean,

$$K_V = \frac{0.484\text{ms} - (0.00445\text{ms} \cdot 33)}{21} = 0.01605\text{ms}. \tag{4.9}$$

This can be validated by using Equation (4.5) and comparing a tested graph in Table 4.4. For instance, $G_3 = (50, 80)$ giving

$$T(G) = f(G = (50, 80)) = 0.01605\text{ms} \cdot 50 + 0.00445\text{ms} \cdot 80 = 1.1585\text{ms}, \tag{4.10}$$

which is fairly close to the actual result. From these equations, we can derive a graph where we can plot the x-axis as number of vertices, y-axis as number of edges and z as the execution time. In Figure 4.16, the graph is illustrated, with an example calculation time of network $T(G = (25, 52)) = 0.6326$ms.

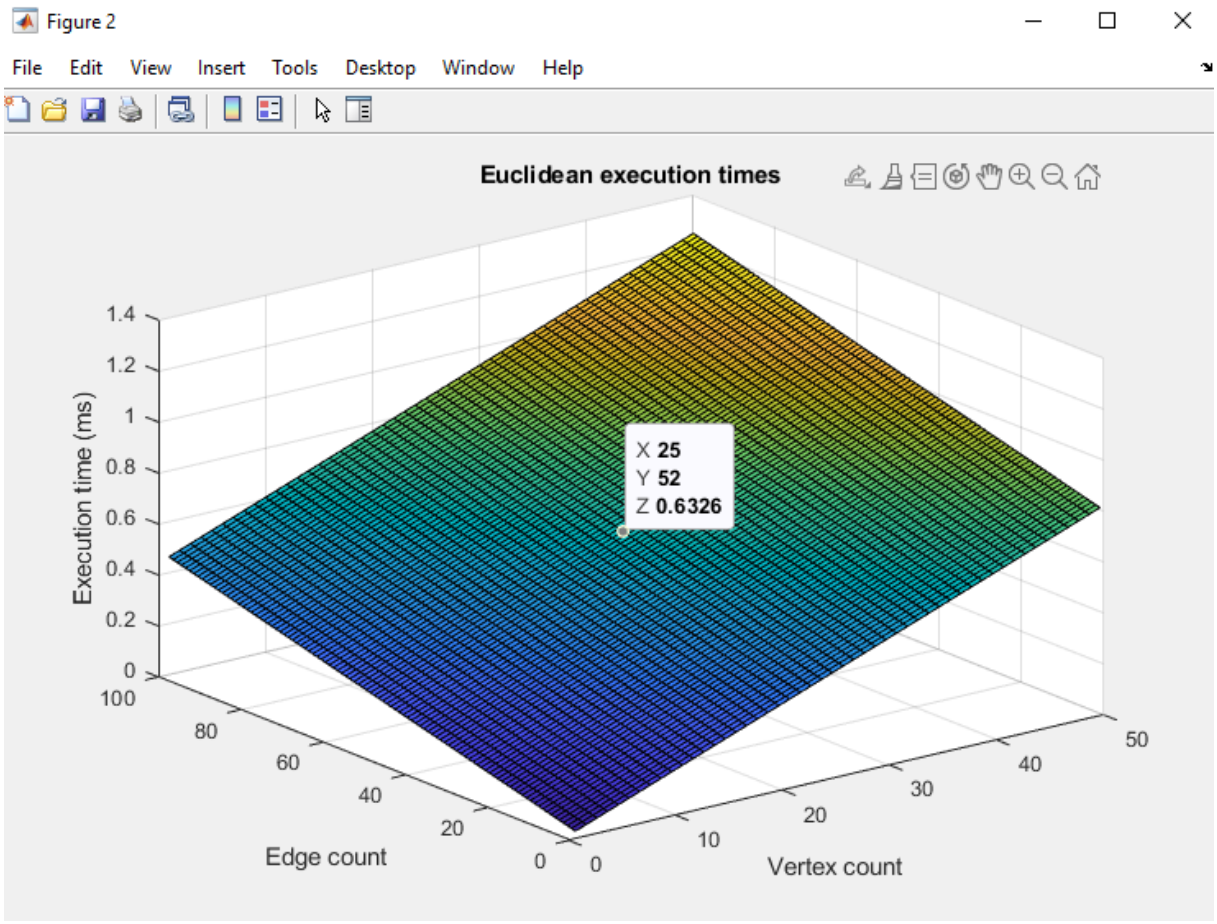


FIGURE 4.16: Function for execution times based on variables V and E from Equation (4.5).

The same thing can be applied with the Manhattan heuristic and re-calculate K_V and K_E from Table 4.5. By using same equations we get $K_V = 0.00355$ms and $K_E = 0.01323$ms. This generates a new graph, as seen in Table 4.17 illustrating the same graph network with a better execution time of $T(G = (25, 52)) = 0.5153$ms.

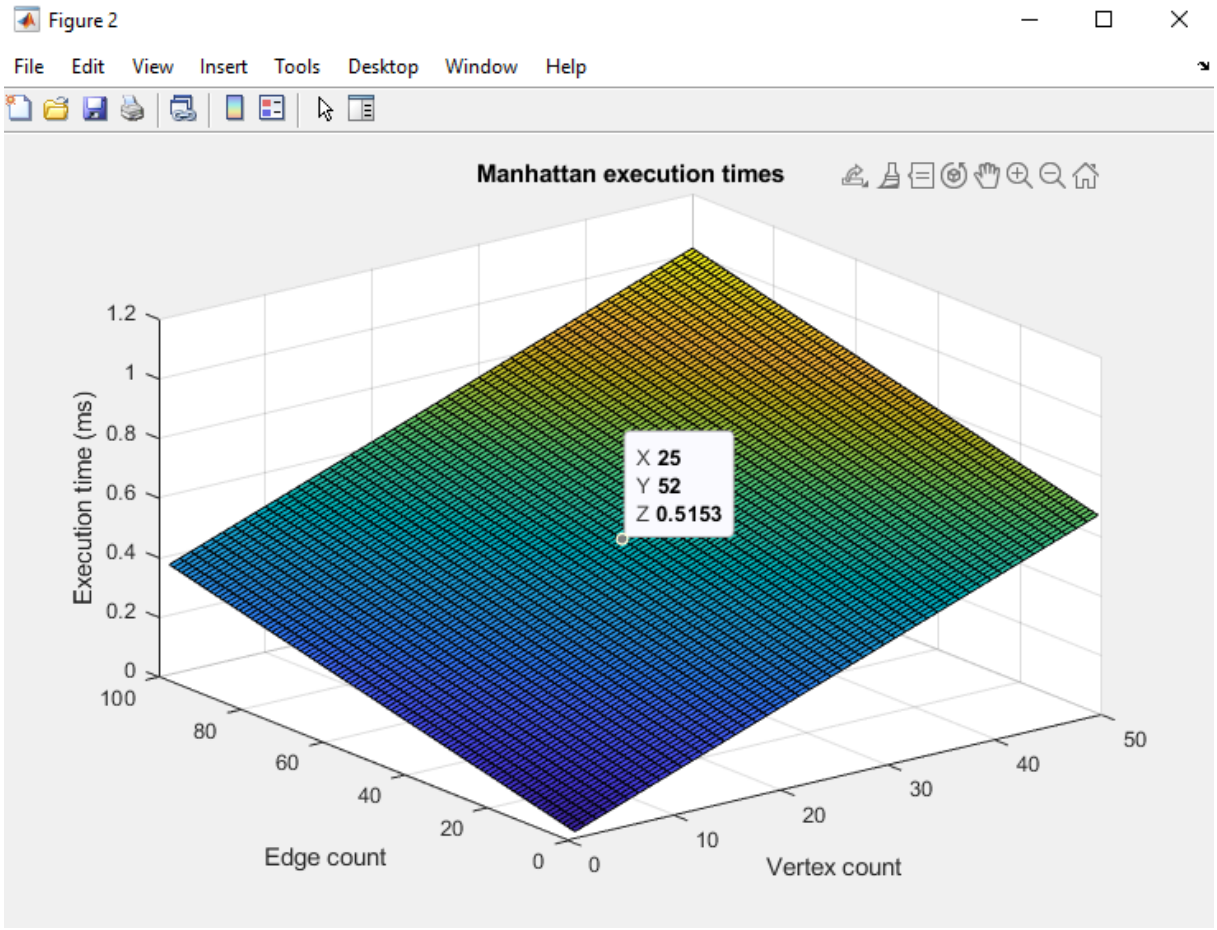


FIGURE 4.17: Function for execution times based on variables V and E from Equation (4.5).

The graphs represented above are only meant to demonstrate an approximation of the execution times based on number of edges and vertices found within a graph. Firstly, it is important to note that in these tests, the same source vertex s and target v was applied for each network. This is due to ambiguity with the placement of the vertices, because the algorithm finishes once it finds the target vertex. If s and v were neighbours, the algorithm would be finished in one cycle, meaning that the other vertices in the network does not affect the execution time. By this, the selection of s and v in these tests have been strategically chosen so that each vertex has been taken into consideration. Finally, there is a constraint that is not illustrated in the graphs, where there is no possibility to have a graph network of specification $G = (2, 4)$ for instance. This is due to the fact that it is only valid to have one edge in a graph with two vertices. To fulfil this constraint, the graph network can maximally have

$$E_{Max} = \frac{V(V-1)}{2}, \tag{4.11}$$

number of edges based on vertices V in given graph network [20].

Chapter 5

Conclusion and Future Work

This work covers the state-of-the-art UAVs and path planning algorithms for the design of autonomous unmanned aerial vehicles that can be utilized in a wide range of applications. Following are some conclusions and potential future developments of this work.

5.1 Conclusion

The development of autonomous vehicles is one of the hottest topics of modern engineering. It is also a multidisciplinary discipline and indeed a very challenging endeavor that has captured the attention and imagination of many researchers around the world. This work is concerned with autonomous aerial vehicles in particular, and I had to dive among fields such as graph theory, for mapping out city areas and perform path planning, control engineering, quaternions, that can be regarded as an extension of complex numbers to higher dimensions, and, of course, programming.

Various graph networks and unmanned aerial vehicle physics have been implemented and demonstrated through simulations using Unity 3D engine. Moreover, integrating simulations based on research and literature review, has made it possible to demonstrate proposed path planning algorithms and their performances. Choosing the optimal path planning algorithm is the most crucial step, where it has been demonstrated in this work that Dijkstra's algorithm is slower than the A* algorithm due to lack of heuristic estimates. Using the latter, finding the shortest path is performed by calculating a heuristic distance to a given destination from each vertex plus the distance travelled. This means there are fewer vertices that needs to be explored, thus improving execution times. A* algorithm may use more resources in terms of memory, although it will always yield the optimal path in a graph network more effectively than Dijkstra's algorithm, if one exists. The heuristic estimate being used also determines the efficiency of the algorithm. From the results carried out in this research, it was discovered that the Manhattan heuristic executed almost 20% faster than the Euclidean heuristic, where both calculated the same optimal path in a representative selection of graph networks. Calculation time depends on structure, size, and complexity of a graph network as well as source vertex s and target v position in the graph network.

Using the graph tool that was created for simulation purposes, made it easy to experiment on various graph networks. The path planning algorithms was conveniently applied to any given graph network. Vertices and edges could be removed in order to experiment how the different relationships between vertices affect the performance of the planned path. Similarly, using read and write file functions, it was possible to save and load graphs in a given environment, from a position in global space. This

was achieved by saving the graph network architecture such as each individual vertex' position and its neighbours into a file. The graph tool served a great purpose for accumulating the shortest path, which could be tested using the custom path calculation function. Vertices could be selected, and the distance from the first selected vertex to the last was given in the console window. By this, it was possible to compare the actual shortest path, with a calculated path from the algorithms and check if the optimal path has been found.

Furthermore, simulating UAVs in Unity has shown promising results. Quaternions and Euler angles has been useful for integrating pitch, roll and yaw attributes to the UAV, and similarly using gravity physics within Unity to calculate the necessary force for upwards lift. This was done in order to achieve a hovering state for the drone, where force can be removed or applied to ascend or descend, respectively. A functional and flexible simulation environment has been developed, taking into account the current state-of-the-art on UAVs. Issues such as maneuverability and enhanced localization of the quadcopter, utilizing environmental data such as landmarks to recognize an UAV's body-fixed frame position in global space, have also been given due attention.

Simulating an unmanned aerial vehicle in a representative city within a graph network has been demonstrated through the results presented in this work. Graphs can be used for mapping out paths in a city environment for the UAV to follow, where the vertices' position can be stored in the UAV. Thereby, the UAV can be given a command to maneuver to a specific destination, and would opt to follow the shortest path based on stored vertex positions. This path can be calculated preemptively based on the network, using the A* algorithm with an admissible heuristic.

5.2 Future work

This research has created a groundwork for autonomous UAVs, with the use of graph networks and state-of-the-art path planning algorithms. Despite the promising results, the experiments still need optimization regarding UAV motion planning. The following issues would be worth investigating further:

- Some aspects regarding algorithms for multiple UAVs has not been simulated, e.g., the particle swarm optimization algorithm. This could be investigated further, by using this work as a foundation, where a path cost function for multiple UAVs is carried out in Equation (3.7) in Chapter 3.

- Optimizing the vertex placement in a graph network as well as integrating curved edges for smoother and more efficient paths for the UAVs to follow, as they can maintain their speed more effectively.

- A modified version of the A* algorithms for shorter calculation times.

- The most interesting direction would be towards motion planning for UAVs, as the planned path is already determined regardless of algorithm efficiency. In other words, utilizing the PID controller by fine-tuning the hyper-parameters such as the proportional K_p, integral K_i and derivative K_d gains, to increase the stability of the system.

Appendix A

Code listings for Unity and Matlab

Most important code sections for implementation in Unity can be seen here, both in regards to path planning algorithms and UAV movement. This is included in order to support the discussed topics, for instance the parameters and function for the given Matlab graph representation of execution times from results.

A.1 Graph network implementation

```
// Vertices
public class vertex {
    public Vector3 p;
    public List<int> neighbours;

    public vertex(Vector3 pos, int[] n) { // Constructor.
        p = pos;
        neighbours = new List<int>();
        neighbours.AddRange(n);
    }

    public void addNeighbour(int newNeighbour) {
        neighbours.Add(newNeighbour);
        neighbours.Sort();
    }

    public void removeNeighbour(int ind) {
        for(int i = 0; i < neighbours.Count; i++) {
            if (neighbours[i] == ind)
            {
                neighbours.RemoveAt(i);
            }
        }
        neighbours.Sort();
    }

    // used to lower all neighbours indexes above target vertex
    public void shiftNeighbours(int ind) {
        for(int i = 0; i < neighbours.Count; i++) {
            if (neighbours[i] > ind) {
                neighbours[i] = neighbours[i] - 1;
            }
        }
    }
}
```

```
// Edges
public class edge {
    public vertex vertexA;
    public vertex vertexB;

    public float len;
    public edge(vertex a, vertex b) {
        vertexA = a;
        vertexB = b;
        len = Vector3.Distance(vertexA.getPos(), vertexB.getPos());
    }

    public float getLength() {
        return len;
    }

    public vertex getVertexA() {
        return vertexA;
    }

    public vertex getVertexB() {
        return vertexB;
    }
}
```

A.2 Path planning algorithm

```
// A star heuristics and execution time variables
public class PathFindingController : MonoBehaviour {

  // Used for calculating execution speed
    public double startTime = 0.0f;

  // Source and target vertex
    public int startVertex = -1;
    public int goalVertex = -1;

  // Accumulated path length
    float pathLength = 0.0f;

  // Euclidean function
  public void calculateEuclideanHeuristic(Vector3 goalPos) {
    h = Vector3.Distance(pos, goalPos);
  }

  // Manhattan function
  public void calculateManhattanHeuristic(Vector3 goalPos) {
    h = Mathf.Abs(pos.x - goalPos.x) +
    Mathf.Abs(pos.y - goalPos.y) +
    Mathf.Abs(pos.z - goalPos.z);
  }
  public void updateFScore() {
    f = g + h;
  }
}
```

```
// A star algorithm with chosen heuristic
// Returns path with integer array with vertex indexes
    int[] a_star() {
    List<int> result = new List<int>();
    openSet.Add(vertices[startVertex]);
    while (openSet.Count > 0) {
        int current = 0;
        for(int i = 0; i < openSet.Count; i++) {
            if (openSet[i].f < openSet[current].f) {
                current = i;
            }
        }
        VertexValuesAStar currentVertex = openSet[current];
        if (currentVertex.ID == goalVertex) {
            VertexValuesAStar tmp = currentVertex;
            while (tmp.cameFrom != -1) {
                pathLength += Vector3.Distance(
                gc.getVertexPosition(tmp.ID),
        gc.getVertexPosition(tmp.cameFrom));
                result.Add(tmp.ID);
                tmp = vertices[tmp.cameFrom];
            }
            result.Add(startVertex);
      return result.ToArray();
        }
        closedSet.Add(currentVertex);
        openSet.Remove(currentVertex);
        int[] currentNeighbours =
    gc.getVertexNeighbours(currentVertex.ID);
        for (int n = 0; n < currentNeighbours.Length; n++) {
            int neigh = currentNeighbours[n];
            if (closedSet.Contains(vertices[neigh])) {
                continue;
            }
            float tempG = currentVertex.g +
      Vector3.Distance(gc.getVertexPosition(currentVertex.ID),
      gc.getVertexPosition(neigh));
            bool newPath = false;
            if (openSet.Contains(vertices[neigh])) {
                if (tempG < vertices[neigh].g)
                {
                    vertices[neigh].g = tempG;
                }
            } else {
                vertices[neigh].g = tempG;
                openSet.Add(vertices[neigh]);
            }
      vertices[neigh].updateFScore();
      vertices[neigh].cameFrom = currentVertex.ID;
        }
    }
    return null;
}
```

```
// Matlab code with linear execution times based on edges/vertices
ke = 0.00355; %Manhattan. Euclidean = 0.00445 ms/E
kv = 0.01323; %Manhattan. Euclidean = 0.01605 ms/V

x = 1:50;
y = 1:100;
[X,Y] = meshgrid(x,y);
z = @(x,y) (kv*x) + (ke*y);  % Function
figure
surf(x, y, z(X,Y))
title(['Manhattan execution times'])
xlabel('Vertex count');
ylabel('Edge count');
zlabel('Execution time (ms)');
grid on
figure
grid
```

A.3 UAV

```
public class flyDrone : MonoBehaviour {
    public class DroneConfigurations {
        public float ascendingSpeed = 6.0f;      // m/s
        public float topSpeed = 20f;            // m/s
        public float maxTiltAngle = 42;          // degrees
        public float angularSpeed = 200;         // degrees/s
        public float yawAngularSpeed = 150;      // degrees/s
        public float droneWeight = 1.380f;       // kilograms
    }
    public DroneConfigurations currentDrone;
    float thrust = 0.0f;
    Rigidbody rb;
    void Start() {
        rb = GetComponent<Rigidbody>();
        rb.mass = currentDrone.droneWeight;
    }
    // Update is called once per frame
    void FixedUpdate() {
            stabilize();
            updateThrust();
            rb.AddForce(thrust * transform.up);
        }
    }
    float getDroneLift() {
        // returns newton value to keep drone hovering
        float newtons = rb.mass *
    Mathf.Abs(Physics.gravity.y);
        return newtons;
    }
    void updateThrust() {
        if (!hasLanded()) {
            thrust = getDroneLift() +
      (getThrustInput() *
      currentDrone.ascendingSpeed *
      currentDrone.droneWeight);
        }
    }
    void stabilize() {
        int yawDir = 0;
        if (Input.GetButton("LB_YAW")) {
            yawDir = -1;
        }
        if (Input.GetButton("RB_YAW")) {
            yawDir = 1;
        }
        Vector3 r = transform.eulerAngles;
        r.x = Input.GetAxis("L_PITCH") * currentDrone.maxTiltAngle;
        r.z = Input.GetAxis("L_ROLL")  * currentDrone.maxTiltAngle;
        transform.rotation = Quaternion.RotateTowards(
    Quaternion.Euler(transform.eulerAngles),
    Quaternion.Euler(r),
    Time.deltaTime * currentDrone.angularSpeed);
```

```
53          float yawRot = currentDrone.yawAngularSpeed
54      * yawDir * Time.deltaTime;
55          transform.Rotate(0, yawRot, 0);
56      }
57      public Vector3 getVelocity() {
58          return rb.velocity;
59      }
60      public float getWeight() {
61          return rb.mass;
62      }
63  }
```

Bibliography

[1] Hanafi Anis et al. "Automatic Quadcopter Control Avoiding Obstacle Using Camera with Integrated Ultrasonic Sensor". In: *Journal of Physics: Conference Series* 1011 (Apr. 2018), p. 012046. DOI: 10.1088/1742-6596/1011/1/012046.

[2] Jorge Artieda et al. "Visual 3-D SLAM from UAVs". In: *J. Intell. Robotics Syst.* 55 (Jan. 2009), pp. 299–.

[3] T. Bailey and H. Durrant-Whyte. "Simultaneous localization and mapping (SLAM): part II". In: *IEEE Robotics Automation Magazine* 13.3 (Sept. 2006), pp. 108–117. ISSN: 1558-223X. DOI: 10.1109/MRA.2006.1678144.

[4] Zoran Benić, Petar Piljek, and Denis Kotarski. "Mathematical Modelling of Unmanned Aerial Vehicles with Four Rotors". In: *Interdisciplinary Description of Complex Systems* 14 (Jan. 2016), pp. 88–100. DOI: 10.7906/indecs.14.1.9.

[5] William W. Bierbaum. *UAV Predator specifications and specifics*. 2013. URL: https://web.archive.org/web/20130601053222/http://www.airpower.maxwell.af.mil/airchronicles/cc/uav.html.

[6] Louisa Brooke-Holland. *Unmanned Aerial Vehicles (drones): an introduction*. Dec. 2012.

[7] Andrew Chapman. "Types of Drones: Multi-Rotor vs Fixed-Wing vs Single Rotor vs Hybrid VTOL". In: Dec. 2019.

[8] Daniel Cremers. *Lecture 2.1: Recap on Linear Algebra*. 2014.

[9] Martin E Dempsey. *'Eyes of the Army': U.S. Army Roadmap for Unmanned Systems, 2010-2035*. Apr. 2010. URL: https://www.hsdl.org/?abstract&did=705357.

[10] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numer. Math.* 1.1 (Dec. 1959), 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390.

[11] *DJI Phantom 4 Pro – Specs, Tutorials & Guides – DJI*. URL: https://www.dji.com/no/phantom-4-pro/info.

[12] Liptak Bela G. *Instrument engineers handbook Process control and optimization (4th ed.)* CRC Press, 2003, pp. 100–110.

[13] Sean Grogan, Robert Pellerin, and Michel Gamache. "The use of unmanned aerial vehicles and drones in search and rescue operations – a survey". In: Sept. 2018.

[14] Victor Hansen. *Predator Drone Attacks*. 2012. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2009313.

[15] Mostafa Hassanalian and Abdessattar Abdelkefi. "Classifications, applications, and design challenges of drones: A review". In: *Progress in Aerospace Sciences* (May 2017).

[16] Adeel Javaid. "Understanding Dijkstra Algorithm". In: *SSRN Electronic Journal* (Jan. 2013). DOI: 10.2139/ssrn.2340905.

[17] J. Kennedy and R. Eberhart. "Particle swarm optimization". In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. Vol. 4. 1995, 1942–1948 vol.4.

[18] Kiam Heong Ang, G. Chong, and Yun Li. "PID control system analysis, design, and technology". In: *IEEE Transactions on Control Systems Technology* 13.4 (2005), pp. 559–576.

[19] Dixit Prasanna Kumar, Sahoo Archana, and Badajena Tushar Kumar. "Graph Theory in an Object Oriented Approach". In: *Journal of Computer Sciences and Applications* 3.6 (2015), pp. 123–126. ISSN: 2328-725X. DOI: 10.12691/jcsa-3-6-2. URL: http://pubs.sciepub.com/jcsa/3/6/2.

[20] Klavdija Kutnar and Dragan Marušič. "Some Topics in Graph Theory". In: vol. 613. Jan. 2009, pp. 3–22.

[21] Hua-Ying Liu et al. *Drone-based all-weather entanglement distribution*. 2019. arXiv: 1905.09527 [quant-ph].

[22] Amgad Madkour et al. "A Survey of Shortest-Path Algorithms". In: (May 2017).

[23] Wei Meng et al. "ROS+unity: An efficient high-fidelity 3D multi-UAV navigation and control simulator in GPS-denied environments". In: vol. 92. Nov. 2015, pp. 931–944.

[24] Shea O'Donnell. *A Short History of Unmanned Aerial Vehicles*. June 2019. URL: https://consortiq.com/en-gb/media-centre/blog/short-history-unmanned-aerial-vehicles-uavs.

[25] Jamal Osama et al. "Design and Manufacturing of Quadcopter". In: Aug. 2019.

[26] Robert Paz. "The Design of the PID Controller". In: (Jan. 2001).

[27] Sujit P.B and Randal Beard. "Multiple UAV Path Planning using Anytime Algorithms". In: July 2009, pp. 2978 –2983.

[28] Sirawat Pitaksarit. "Objectively comparing Unity and Unreal Engine". In: Jan. 2019.

[29] N. Sariff and Norlida Buniyamin. "An Overview of Autonomous Mobile Robot Path Planning Algorithms". In: July 2006, pp. 183 –188. ISBN: 978-1-4244-0526-8.

[30] Shital Shah et al. "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles". In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065.

[31] Harshita Sharma et al. "Determining similarity in histological images using graph-theoretic description and matching methods for content-based image retrieval in medical diagnostics". In: *Diagnostic pathology* 7 (Oct. 2012), p. 134.

[32] Daniel Shiffman The Coding Train. "Coding Challenge 51.1: A* Pathfinding Algorithm - Part 1". In: (Jan. 2017). URL: https://youtu.be/aKYlikFAV4k?t=85.

[33] Esteban Valencia Torres, Victor HIDALGO DIAZ, and Orlando Calle. "Methodology for Weight and Performance Assessment of an UAV for Precision Agriculture at Cruise Condition". In: *53rd AIAA/SAE/ASEE Joint Propulsion Conference* (July 2017). DOI: 10.2514/6.2017-4868.

[34] G. Varela et al. "Swarm intelligence based approach for real time UAV team coordination in search operations". In: *2011 Third World Congress on Nature and Biologically Inspired Computing*. Oct. 2011, pp. 365–370. DOI: 10.1109/NaBIC.2011.6089619.

[35] *Windridge City: 3D Roadways: Unity Asset Store*. URL: https://assetstore.unity.com/packages/3d/environments/roadways/windridge-city-132222.