Chiroptera: A Biomimicry Inspired Approach to Simultaneous Localization and Mapping for Autonomous Vehicles

Chiroptera: A Biomimicry Inspired Approach to Simultaneous Localization and Mapping for Autonomous Vehicles

Alexander Shopov

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

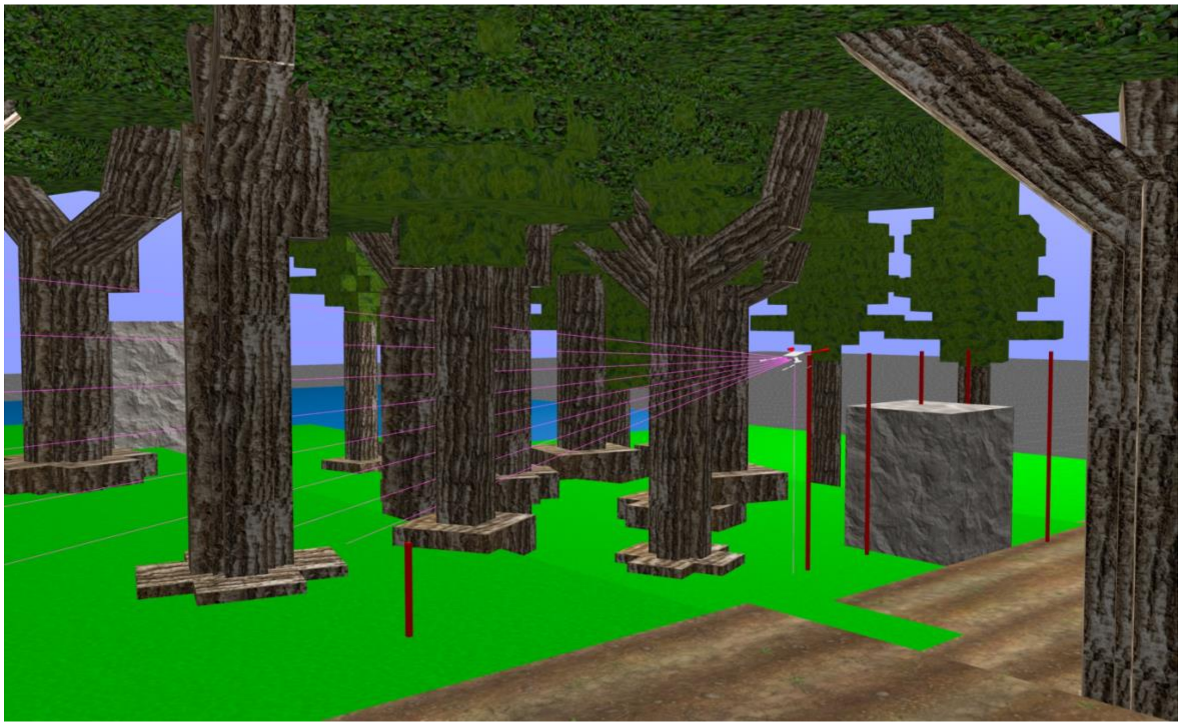
May 2024

Abstract

Biomimicry is a technique for solving complex technical challenges by applying biologically inspired processes. When considering flight through challenging conditions, bats (order *chiroptera*) have demonstrated a keen ability to use non-visual senses to navigate surroundings, avoiding obstacles, and create a mental map of the environment in which they live and operate.

Unmanned aerial drones are controlled in flight by an intelligent agent, be it a human pilot or artificial intelligence (AI) based autopilot. Simultaneous Localization and Mapping (SLAM) is a technique for building a map of an environment while keeping track of a vehicle's position within that environment. This thesis shows that is possible to use bats as the inspiration for the development of an AI autopilot, Chiroptera, that can conduct SLAM operations to navigate a drone through an unknown environment. A bespoke 3D drone flight simulator, Bat Cave, will be developed concurrently within which Chiroptera's SLAM capabilities can be tested and its behavior refined.

Frontispiece



A still frame taken from a test flight conducted in the Bat Cave flight simulator and described in Chapter VI.4 shows Chiroptera navigating a virtual drone through the wilderness, marking its path as it goes.

Acknowledgments

With deepest thanks and appreciation to my thesis director Professor James L. Frankel for his guidance and mentorship through a myriad of challenges as this thesis was brought to life. Likewise to my research advisor Professor Hongming Wang for her valuable feedback throughout every step of the process.

Thank you as well to my partner Melissa and to my family. This master’s degree journey would not have been possible without their unconditional support over the last several years.

Table of Contents

List of Tables

List of Figures

Chapter I.

Introduction

Intelligent agents are entities that can leverage sensor data to collect information about their environment and use that information to make decisions about how to interact with that environment. Biological organisms (human or otherwise) and artificial intelligence (AI) systems both fit this definition. Biological organisms collect data using sensors including, but not limited to, their eyes, ears, and noses, while AIs rely on artificial sensors to determine distance, forces, and positional measurements. Evolution has granted biological organisms the ability to synthesize that sensor data into an understanding of their environment such that they can effectively recognize and navigate a path between two points. Software based approaches to finding a route between two points are known as pathfinding and are categorized as a subset of graph theory. An outstanding problem in artificial intelligence involves the determination of how to find a path between two points in an unknown environment while avoiding obstacles along the way. One approach to this problem is Simultaneous Localization and Mapping (SLAM), an operation in which an autonomous agent creates a map of its surroundings while keeping track of its position within that map and using the map as an aide to navigation.

Because the author has a background in both software engineering and biology, the goal of this thesis was to combine both disciplines to explore a biologically inspired approach to an open problem in artificial intelligence decision making, specifically the ability of an autonomous vehicle to find a path through an unexplored environment. In the natural world, bats have been observed using a variety of senses to seemingly create an internal map as an aid to navigating through their environment. Studying the published research around this phenomena inspired the development of two software systems to explore the same concept as it could be applied to autonomous vehicle pathfinding. The first was an intelligent agent, Chiroptera, capable of real-time of synthesis of sensor data into a 3D map sufficient for performing SLAM operations within an unknown, obstacle filled environment. To test Chiroptera's capabilities, a drone simulator called Bat Cave was developed. Bat Cave provided a robust sandbox in which different sensor configurations and autonomous navigation strategies could be validated.

I.1 Related Work

I.1.1 Drone Control

Unmanned aerial vehicles (UAV) are colloquially referred to as *drones*. Off-the-shelf and bespoke drones are finding increasing use as a front-line tool in wilderness environments. While once they were exclusively the purview of military and intelligence organizations, today anyone can buy or build an inexpensive quadcopter outfitted with a digital camera or other sensors, controlled via a dedicated remote control or smartphone app. Trail and ecological monitoring has traditionally required teams of workers to conduct extensive on-site sampling or measurements. Monitoring via aerial and satellite photography provides a less manpower-intensive alternative, but the usefulness of the collected data can be limited by the camera resolution or unfavorable weather conditions. Drones equipped with high-resolution cameras have been deployed to map vegetation, land cover, and the potential impact of recreational use on protected environments

(Ancin-Murguzur, 2019), while non-camera-based sensors have found use in conducting wildlife observations (Roberts, 2020).

Drones in flight are directly controlled by a human or software based intelligent agent. Manual flight control offers a high degree of flexibility, as human pilots' general intelligence allows them to quickly adapt to new situations. A commercial videography drone that was intended to capture documentary film footage was used for an impromptu search-and-rescue operation in the mountains of Pakistan. The human pilot was able to manually fly the drone around the area in which a climber was lost, successfully locating, and making contact with the climber faster than a ground-based search and rescue team (McRae 2019). Despite their flexibility, however, manually controlled flights are subject to limitations, including radio control distance and signal obstruction. The aforementioned mountain rescue would likely not have been successful if the missing climber was inside a cave or otherwise obstructed such that the drone would not be able to enter the enclosed space without losing contact with the pilot. In cases where a drone needs to be flown outside of human control range, a skilled human pilot is not available, or a high degree of precision maneuvering is required, autonomous solutions must be considered.

I.1.2 Autonomous Aerial Vehicle Navigation

Autonomous flight for general mapping or environmental monitoring applications can involve a solution as conceptually simple as flying a drone through a preset path of GPS waypoints. A predefined path can be revisited for future surveys, offering a means to capture time series imagery from a consistent set of locations (Afghah, 2019). Data from external sensors provides the flight computer with information on the drone's state relative to the environment. A typical sensor suite would include an inertial measurement unit (IMU) to determine the drone's acceleration and orientation, barometric or pressure sensor based altimeter to measure the altitude above sea level (ASL), a downward facing fixed laser or ultrasonic range finder to track altitude relative to the ground (AGL), and global positioning system (GPS) to provide waypoint positional data. This sensor suite is sufficient for aerial survey and mapping operations but is limited in its use when obstacles are present along the flight path or GPS signals are not consistently available due to obstructing biomass such as thick tree canopies or artificial structures.

When GPS is not sufficient, machine vision offers an alternate means of navigating wilderness environments under a tree canopy. Vision-based systems operate by extracting features of interest from a video stream without human assistance. Promising work has been conducted in leveraging machine learning algorithms to achieve autonomous flight through forest trails by training a single camera system to recognize the boundaries of a trail, a pattern recognition problem that in some cases can be difficult for humans as well (Giusti, 2016). Automatic feature extraction can prove challenging when operating in conditions of low or inconsistent lighting, as well as ambiguous feature boundaries (Back, 2020). As such, vision-based approaches can not necessarily be considered a general solution to autonomous navigation in obstacle-filled environments.

I.1.3 SLAM

SLAM is an active area of AI research that seeks to addresses the problems inherent in navigating environments whose layout is not known a priori by incrementally creating a map of the unknown environment while at the same time determining the vehicle's location within the map (Durrant-Whyte 1996). Camera-based approaches known as visual-SLAM provide a promising approach but still require a visual feature detection and extraction step (Cheng 2022) and are subject to the same environmental challenges as non-SLAM vision solutions (Tourani 2022).

Non-camera approaches to SLAM have leveraged sonar for underwater operations (Westman 2018) and LIDAR (Light Detection and Ranging) for self-driving cars (Elhousni 2020). LIDAR is widely used for UAV-based mapping and navigation. It generates a so-called "point cloud," a collection of tens- to hundreds- of thousands of discrete points in 3D space which represent the surface area of an object. Point clouds are generally analyzed post-process, as mapping and survey applications require the ability to differentiate the point cloud into discrete objects and structures. A number of open source software libraries are available for this task, including Open3D[1] and PCL[2]. These libraries store the point cloud in octrees: three dimensional volumes modeled as hierarchical branching data structures in which each parent node has eight child nodes, each of which represents a sub-region within the volume as shown in Figure 1. Point cloud data is stored in the octree's leaf nodes, with each leaf node containing one or more points. Depending on the depth of the octree and resolution of the leaf nodes, some of the detail of the point cloud may be lost. However, octrees are computationally efficient as their hierarchical structure allows points to be found or inserted without requiring a full exploring of the octree.

[1] https://www.open3d.org
[2] https://pointclouds.org

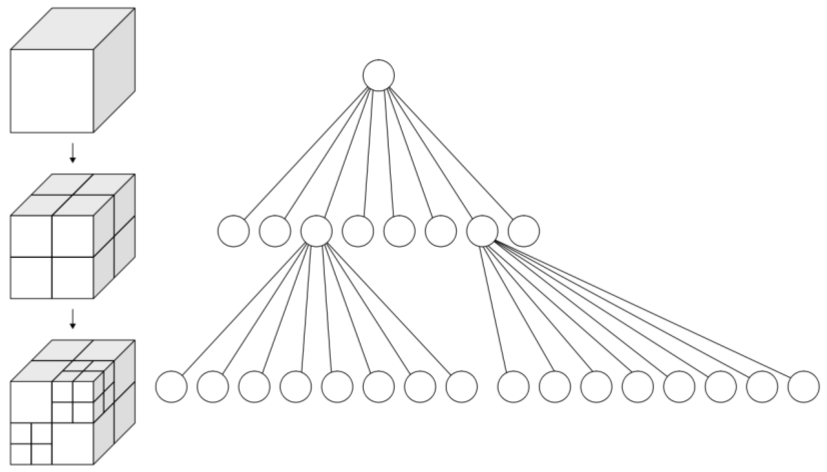


Figure 1: Octree structure and subdivision[3]

Efficient search operations have made octrees the data structure of choice for use in indoor navigation of autonomous vehicles (Rodenberg 2016) and flight navigation of a video game AI (Rabin 2017) where sufficient navigational accuracy can be achieved at a resolution lower than that provided by the point cloud raw data. However, these examples presuppose *a priori* knowledge of the geometry of the environment that is being explored.

I.1.4 Biological Analogues to SLAM

When we consider non-vision based approaches to SLAM, the natural world can serve as a source of inspiration. The term *biomimicry* has been coined to describe the process of examining nature to solve technological problems. Evolution has provided biological intelligent agents with a keen ability to function in unknown environments, as demonstrated by human drone pilots. A biological intelligent agent does not necessarily

[3] Image courtesy of https://en.wikipedia.org/wiki/Octree

have to refer to a human, however. A potential solution to the problem of autonomous navigation of a challenging environment via an intelligent agent could be reframed as something akin to attempting to mimic the observed behavior of an animal like a bat.

The bat (order *chiroptera*) is a creature known to be adept at navigating gracefully through natural environments. Bats do not rely exclusively on their sense of sight but supplement their visual system with additional environment observations sourced through echolocation, olfactory signals, and a sense akin to an internal magnetometer (Holland 2006). While much is not fully understood about exactly how different species of bats navigate their specific environments, it appears that bats can switch between different navigational strategies for different tasks (Genzel, 2018). Research also indicates that bats can keep a large-scale map of their environment in memory, using it as an aid to navigation (Tsoar, 2011).

Chapter II:

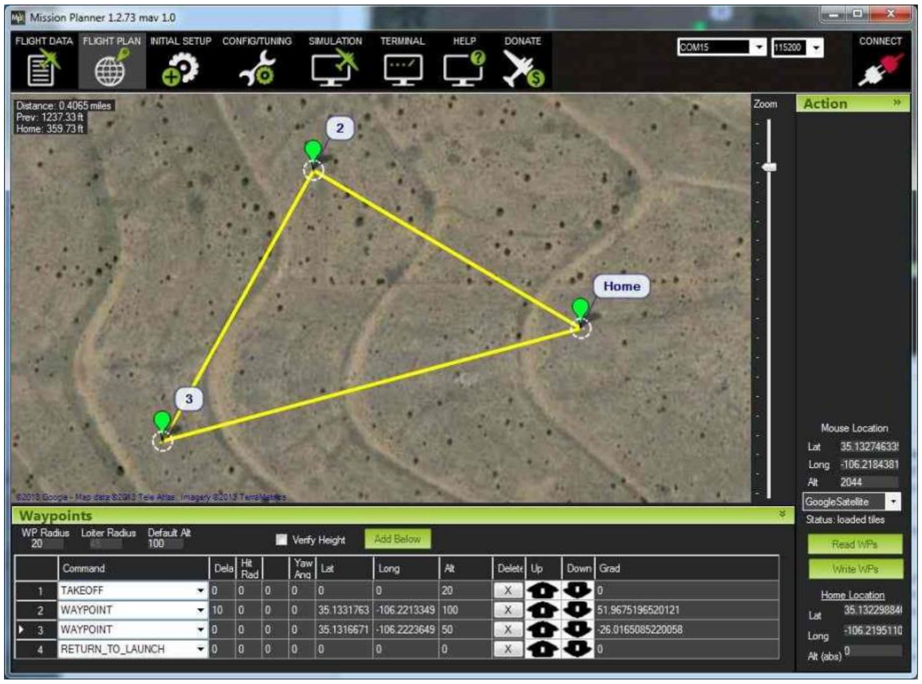
Project Design and Architecture

The primary goal of this thesis was the development of a software based intelligent agent capable of accurately navigating to a given point in an unknown environment. The intelligent agent, Chiroptera, is capable of performing SLAM operations without the use of a camera, relying instead on data captured by a suite of sensors including LIDAR, downward facing ultrasonic range finder, altimeter, and GPS. The LIDAR and ultrasonic range finder data is used to build a graphical representation of the environment, such that Chiroptera can find a path from point to point and make flight control decisions necessary to move along that path. Chiroptera is ultimately intended for use on physical drones, so it has been designed to be compatible with the existing open source MAVLink[4] communication protocol, an interface and messaging protocol used by drone flight controllers, ground stations, and autopilots. By implementing a subset of the MAVLink command language, Chiroptera is compatible with open source MAVLink based flight controllers such as the ArduPilot Copter[5] multi-rotor UAV controller, as well as associated drone hardware such as the PixHawk[6] flight controller.

Off the shelf drone controllers typically include a mission planner which allows users to plan, save, and execute autonomous flights. ArduPilot's mission planner shown in Figure 2 includes a robust graphical interface and integration with Google maps

[4] https://www.mavlink.io
[5] https://www.ardupilot.org/copter
[6] https://pixhawk.org

allowing flight destinations, also known as *waypoints*, to be specified with a simple mouse-click.



Distance: 0.4065 miles
Prev: 1237.33 ft
Home: 359.73 ft

Mouse Location
Lat 35.13274633!
Long -106.2184381
Alt 2044
GoogleSatellite
Status: loaded tiles

Home Location
Lat 35.13229884(
Long -106.2195110
Alt (abs) 0

WP Radius 20
Loiter Radius
Default Alt 100
Verify Height

	Command	Dela	Hit Rad		Yaw Ang	Lat	Long	Alt	Delete	Up	Down	Grad
1	TAKEOFF	0	0	0	0	0	0	20	X			0
2	WAYPOINT	10	0	0	0	35.1331763	-106.2213349	100	X			51.9675196520121
3	WAYPOINT	0	0	0	0	35.1316671	-106.2223649	50	X			-26.0165085220058
4	RETURN_TO_LAUNCH	0	0	0	0	0	0	0	X			0

Figure 2: ArduPilot mission planner

To allow for more time spent focusing on Chiroptera, I did not develop a graphical interface for interactive flight planning and modification, opting instead to define waypoints and other mission commands in newline-delimited ASCII text files which Chiroptera loads and parses upon application startup. The specific format of the mission file will be described in Chapter 3.

While conducting a mission, Chiroptera must be able to convert sensor data into an in-memory geometric representation of its environment. As its focus is on navigation and not detailed survey operations, Chiroptera's internal map only needs a resolution high enough to facilitate obstacle avoidance. Likewise, it isn't necessary to differentiate between different objects in the environment; it's enough for Chiroptera to conceptually know that a given region of space if empty and traversable, while another region has an obstruction that must be avoided. Chiroptera's must continuously update its map as additional sensor data is received. If additional obstacles are detected, a new path to the current waypoint may need to be determined mid-flight.

As a proverbial "brain in a jar," Chiroptera does not know or care if it is operating in a simulated or real-world environment. As such, a secondary goal of this thesis was the development of an environment and drone flight simulator, Bat Cave, in which Chiroptera's mapping and navigation capabilities could be tested. Bat Cave generates a user-defined 3D obstacle filled environment (also known as a *scene*), as well as a virtual drone and sensor data including GPS and altimeter, external LIDAR, and ultrasonic range finder. For this project I chose to base Bat Cave's simulated drone on a S500 quadcopter as shown in in Figure 3. The S500 provides a versatile off the shelf platform that can be tasked for different types of flight operations (Babcock 2023) and is easily integrated with PixHawk flight controllers[7]. As we will see in Section 3, the physical characteristics of the S500 frame will be used to determine the resolution of Chiroptera's octree map.

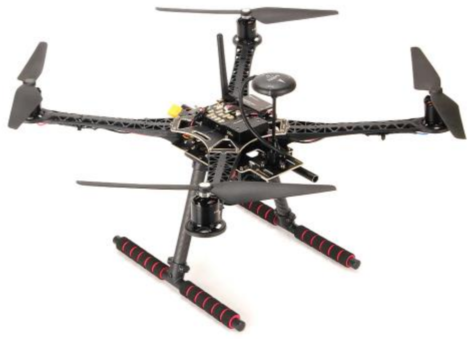


Figure 3: An S500 quadcopter frame

[7] https://docs.px4.io/main/en/frames_multicopter/holybro_s500_v2_pixhawk4.html

Bat Cave's virtual S500 drone is controlled by a custom physics engine called Fleder that I developed specifically for this project. Fleder, a shorted version of *fledermaus*, the German word for bat, implements simulated forces and a real-time collision detector allowing Bat Cave's sensors to interact with the environment geometry in a way analogous to their real-world counterparts. The sensors-environment interaction facilitates the generation of a point-cloud representation of the scene. The sensor data is both visualized within Bat Cave and sent to Chiroptera for use in its mapping and navigation operation.

Like the ArduPilot mission planner, Bat Cave implements a graphical interface that allows a user to monitor the drone's status during flight, as well as start, pause, and reset the simulation.

Communication between Chiroptera and Bat Cave's simulated drone is facilitated via named pipes into which ASCII text formatted command and telemetry messages are written. The communication architecture is shown in Figure 4. To ensure that neither application is blocked during command or simulation execution, both Chiroptera and Bat Cave create new threads in which Read operations can be executed.

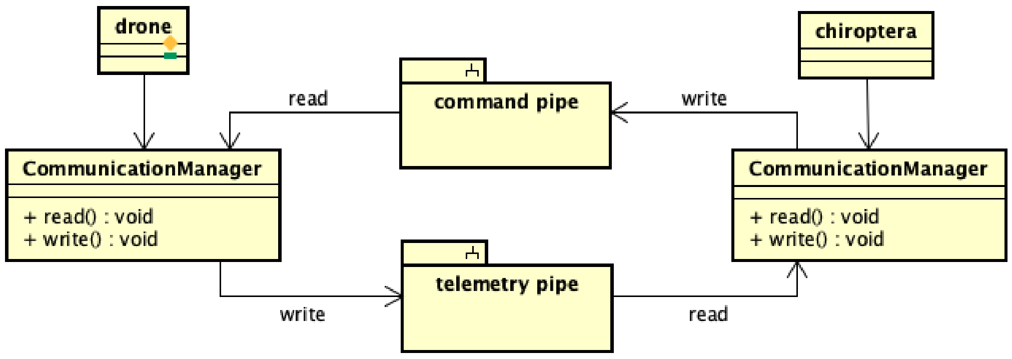


Figure 4: Communication between Chiroptera and Bat Cave's simulated drone

Chiroptera and Bat Cave exist in a feedback loop, in which a message from one application leads to a state change in the other application, and vice versa. Figure 5 illustrates the following high level sequence of events when Chiroptera issues a command to Bat Cave. Section III and IV will describe each of these steps in detail.

1. Chiroptera reads the drone state information from the telemetry pipe.
2. Chiroptera executes the current command, passing the drone state data as a parameter, and returning a drone control instruction.
3. Chiroptera writes the drone control instruction to the command pipe.
4. Bat Cave reads the drone control instruction from the command pipe.
5. Bat Cave updates the simulation.
6. Bat Cave writes the updated drone state to the telemetry pipe.

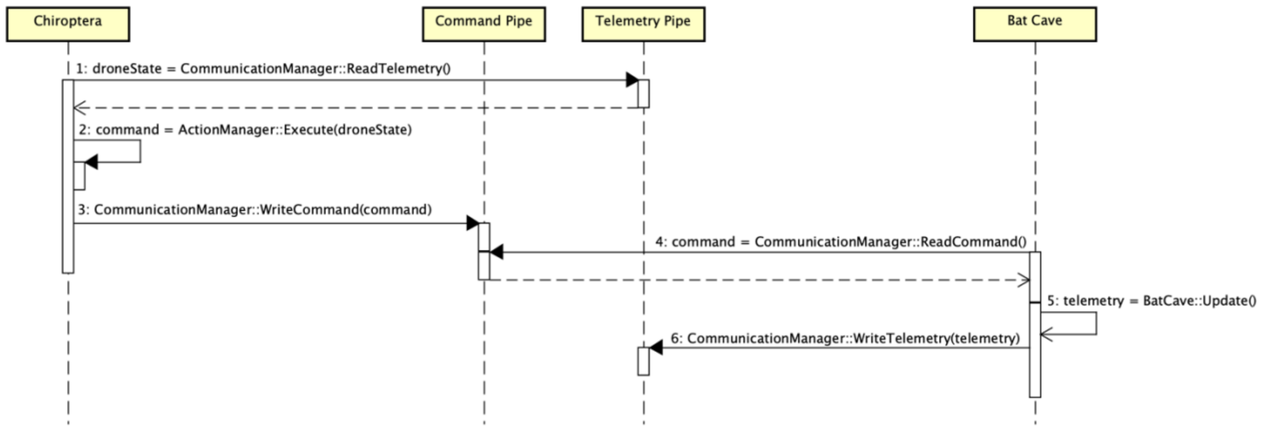

Chiroptera
Command Pipe
Telemetry Pipe
Bat Cave
1: droneState = CommunicationManager::ReadTelemetry()
2: command = ActionManager::Execute(droneState)
3: CommunicationManager::WriteCommand(command)
4: command = CommunicationManager::ReadCommand()
5: telemetry = BatCave::Update()
6: CommunicationManager::WriteTelemetry(telemetry)

Figure 5: Chiroptera command execution and dispatching

Chapter III:

The Chiroptera Intelligent Agent

Chiroptera is conceptually not unlike a video game intelligent agent, something I have developed several of in the past. At the highest level, Chiroptera's *raison d'être* is to navigate an autonomous drone from point to point while conducting SLAM operations facilitated by data received from a suite of external sensors. As a pre-loaded series of flight commands is executed, collected sensor data is used to construct an internal 3D map of the drone's surroundings from which flight operation decisions can be made.

Chiroptera is written in C++11. As it is ultimately intended for use on physical hardware, no third-party libraries are used in the interest of keeping its memory footprint as lean as possible.

The class diagram in Figure 6 illustrates Chiroptera's core architecture.

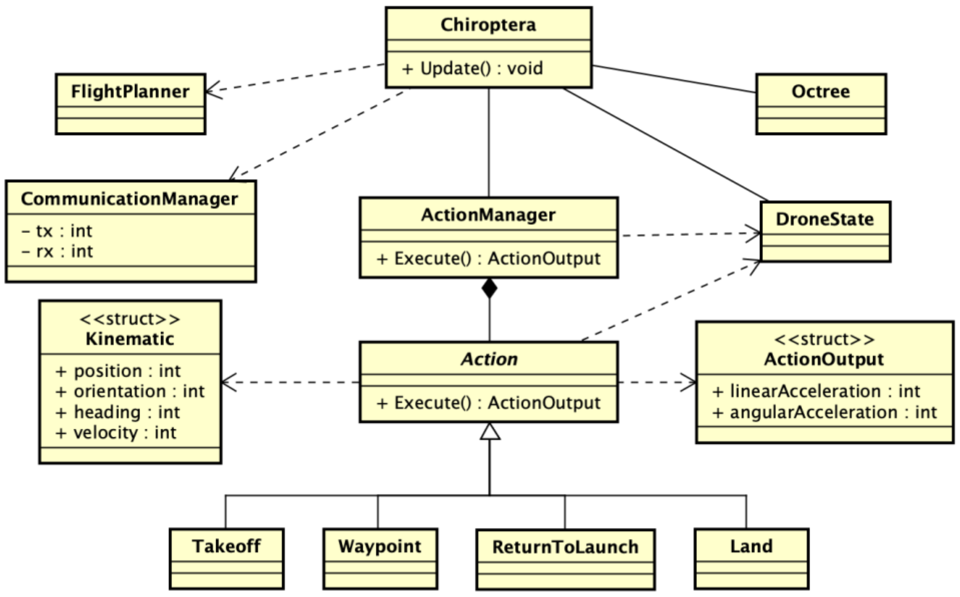


Figure 6: Chiroptera class diagram

Chiroptera is architected as a set of narrowly focused classes, each responsible for one aspect of drone control. The namesake class is responsible for initializing the helper classes and executing Chiroptera's main event loop. The event loop is detailed in section III.2. A `FlightPlanner` class is responsible for loading and parsing the mission command file. As each command is parsed, the `FlightPlanner` returns an Action which is added to an ActionManager. The `Action` and `ActionManager` classes are detailed in section III.1.3.

As the main event loop executes, the `CommunicationManager` described in section II reads telemetry data from the drone and writes commands back to the drone. Section III.1.5 describes the message format used to encode telemetry and command data. Telemetry data is stored in the `DroneState` class, described in section III.1.2. The `Octree` class implements the octree data structure used to store the environment map and perform pathfinding operation. Mapping and pathfinding are described in sections III.1.6 and III.1.7.

III.1 Implementation

III.1.1 Motion

Chiroptera's motion algorithms are based on the kinematic equations of classical mechanics. Starting from a dead stop, Chiroptera will need to calculate the three-dimensional velocity and acceleration vectors necessary to move the drone to its requested position. The final velocity of an object, v_f, can be calculated as the sum of the initial velocity of the object plus the object's acceleration scaled by the elapsed time. This is expressed as the kinematic equation:

$$v_f = v_0 + at$$

where v_0 is the initial velocity of the object, a is the acceleration, and t is time.

Once in motion, adjustments to the drone's velocity will be necessary to change direction or slow down and come to a stop. By definition, acceleration is the change in velocity with respect to time. Solving for a, we have:

$$a = (v_f - v_0) / t$$

Chiroptera uses this equation to calculate the acceleration required to update the drone's velocity as required by the current maneuver. In this equation, time t acts as a scaling factor and can vary based on the desired magnitude of acceleration. Initiating motion from a dead stop can take place over a short interval and small value of t, while gradually slowing down to approach a destination requires a larger value of t. Time intervals do not necessarily need to be hardcoded in advance; rather, we will see in the Action section below that arbitrary values of t can be defined on an action by action basis. It is important to note that values of t appropriate for use in the Bat Cave simulator may not be suitable for use in physical drones. This consideration will be explored further in Section VII.1.

Rather than using a third party physics library which contains unnecessary functionality, Chiroptera directly implements the minimum necessary set of position, motion, and transformation vector and matrix data and operations. These vector and matrix classes will also be used by the Fleder physics engine described in Section IV. A base `Vector3` class is used to model both the {x, y, z} coordinates of an object in 3D world space, as well as a force vector that parameterizes desired changes in motion. The latter is referred to as a steering force (Reynolds, 1999). A `Kinematic` class (Millington, 2009) was implemented to model Chiroptera's current and desired kinematic properties:

```
CLASS Kinematic
  position         Vector3
  orientation      Vector3
  heading          Vector3
  velocity         Vector3
  angularVelocity  Vector3
ENDCLASS
```

III.1.2 Drone State

Chiroptera implements a DroneState class to keep track of its current and target `Kinematic` states, as well as references to the most recently received set of LIDAR data and the navigation graph. A reset flag can be toggled by a message from Bat Cave, instructing Chiroptera to return to its initial state and restart its current flight.

```
CLASS DroneState
  current          Kinematic
  target           Kinematic
  LIDAR            Vector3[]
  graph            Octree
  reset            boolean
ENDCLASS
```

III.1.3 Actions

Chiroptera's core functionality is built around the concept of taking an action in response to a predetermined flight plan or unanticipated change in drone state. An action can represent a kinematic event, a change in internal drone state, or even no behavior at all. Actions are inherited from the `Action` base class whose attributes include the drone's current and desired state, as well as virtual initializer and execution methods. The execute method returns an `ActionOutput` object containing the target steering vector and angular acceleration.

```
CLASS ActionOutput
  action              string
  linearAcceleration  Vector3
  angularAcceleration Vector3
ENDCLASS


CLASS Action
  # properties
  current     Kinematic
  target      Kinematic
  state       DroneState
  is_complete boolean

  # methods
  Constructor()
  virtual Init(DroneState)
  virtual Execute(DroneState) : ActionOutput
ENDCLASS
```

Actions represent individual instructions that change the drone's state. The core set of motion actions described in Table 1 are based on a subset of steering behaviors for autonomous entities originally described by Craig Reynolds (Reynolds, 1999).

Table 1: Steering Actions

Action	**Description**
Seek	Steers the drone towards a specific position in world space.
Arrive	Identical to Seek but directs the drone to slow down as it approaches the target, eventually coming to a stop at the target position.
Alignment	Adjusts the drone's orientation until it is pointing straight towards a specified position in world space.

Obstacle Avoidance	Maneuvers the drone through an obstacle filled environment by dodging anything its path. This applies to obstacles between the drone and its immediate goal, it is not a large scale pathfinding solution.
Path Following	Steers the drone along a predetermined path. The path is determined by the algorithm described in section III.1.7

The implementation of each steering action is extended from the `Action` base class. Each action child class defines necessary attributes and implements custom constructor, initialization, and/or execution methods that contain the logic necessary to fulfill the action. In the `Arrive` action, for example, attributes define the maximum speed and acceleration of the action. A `time_to_target` value is used to scale the resulting acceleration value as described in the previous section, and a `slow_radius` denotes the distance at which Chiroptera should being slowing down. During Arrive initialization the `slow_radius` is defined as one-third the distance from the initial position to the target position. Section VII.1 will discuss alternate future approaches to determining this parameter.

```
CLASS Arrive EXTENDS Action
  # properties
  max_speed          float
  max_acceleration   float
  time_to_target     float
  slow_radius        float

  # methods
  constructor(latitude, longitude, altitude)
  Init(DroneState)
  Execute(DroneState) : ActionOutput
ENDCLASS
```

A full description of all implemented Actions and their attributes can be found in Appendix 2.

Actions can be composed into more complex behaviors. A series of actions can be executed in sequence, or a given action can execute given the current drone state conditions. In the `Waypoint` action, for example, Chiroptera combines the `Align` and `Arrive` actions to adjust the drone’s heading to and move to the requested position in a single command:

```
CLASS Waypoint:
  align        Align
  arrive       Arrive
  is_aligning  boolean

  METHOD Waypoint::Execute(droneState)
    IF isAligning THEN
      IF align is complete THEN
        is_aligning := false
        arrive.Init(droneState)
      ELSE
        actionOutput = align.Execute(droneState)
      END IF
    ELSE
      IF arrive is complete THEN
        Set Waypoint.isComplete := true
      ELSE
        actionOutput = arrive.Execute(droneState)
      END IF
    END IF
    Return actionOutput
  ENDMETHOD
ENDCLASS
```

Actions are initialized and executed by an `ActionManager` class. The `ActionManager` stores the actions that are to be executed as a linked list. The active action is denoted by a pointer to a node within the linked list.

```
CLASS ActionManager
```

```
  # properties
  actions_list:  Action[]
  active_action: Action
  # methods
  ExecuteAction(DroneState) : ActionOutput
ENDCLASS
```

Action execution will be described below in the Execution section.

III.1.4 Commands

Actions are invoked via an API that implements a subset of the ArduPilot Copter interface (ArduPilot 2024). Commands are classified as either Navigation commands used to control the motion of the drone (e.g. travel to the given latitude and longitude), DO commands used to change the drone’s internal state without changing the drone’s position (e.g. change max speed), or Conditional commands which delay DO commands until some condition is met (e.g. the drone adjust yaw by a given number of degrees). As per the MAVLink spec, up to one Navigation and one DO or Conditional command can be executing at a time. A full dictionary of implemented commands can be found in Appendix 2.

III.1.5 Communications

Chiroptera communicates with Bat Cave via named pipes as described in Chapter 2. A `CommunicationManger` class is responsible for opening and closing the telemetry input (`rx`) and command output (`tx`) file streams and reading from/writing to buffers. As no more than one instance of the `CommunicationManager` is required, the class is implemented using a Singleton pattern:

```
CLASS CommunicationManager
```

```
  # properties
  rx: unsigned integer
  tx: unsigned integer

  # methods
  Read()
  Write()
  GetTelemetry()
ENDCLASS
```

Messages are sent and received as newline-delimited ASCII strings. Each line begins with a code which identifies the type of data the line contains. Floating point values are formatted to 4 places left of the decimal point (3 digits plus an option sign) and two digits to the right of the decimal point.

Drone instructions posted to Bat Cave are ASCII strings up to 128 byte in length, with a format shown in Table 2. Identifier 0 can be up to 32 characters long. Identifiers 1-4 are each 24 characters long. Each line includes a newline delimiter. Identifiers 1 and 2 are the drone kinematic instruction data. Identifiers 0, 3 and, 4 denote data that is displayed or visualized in Bat Cave as an aide to debugging Chiroptera's behavior.

Table 2: Chiroptera instruction format

ID	Content	Format	Description
0	Command	String	The currently executing command
1	Linear Acceleration	3 floats, space delimited	The desired steering vector
2	Angular Acceleration	3 floats, space delimited	The desired angular acceleration vector
3	Drone position in SVO	3 floats, space delimited	The world space position of the SVO node the drone occupies
4	Current waypoint position in SVO	3 floats, space delimited	The world space position of the octree node the current waypoint occupies

Telemetry and sensor data received from Bat Cave is up to 256 bytes in length. Telemetry will include either a single reset command or multiple lines describing the current drone state. As with the instruction data, each line includes a newline delimiter. Drone state telemetry will contain both drone kinematic data and acquired sensor data. Identifiers 1-3 are 24 character long strings of kinematic data. Identifier 4 consists of a single floating point AGL value. Identifier 5 is a 24 character string, and can appear multiple times with each line representing acquired LIDAR data. Each line of LIDAR data represents an individual laser beam known as a *channel*. The LIDAR unit organizes channels vertically, and the scan resolution of the sensor is directly related to the number of channels available. Section IV.1.3 will describe the channel configuration implemented by Bat Cave.

Table 3: Bat Cave telemetry and data format

ID	Content	Format	Description
0	Reset	integer	The number 0. Signals Chiroptera to reset to its initial state.
1	Drone position	3 floats, space delimited	The drone position in world space.
2	Drone orientation	3 floats, space delimited	The drone orientation in world space.
3	Linear velocity	3 floats, space delimited	The drone’s current linear velocity in meters / second.
4	AGL	Floating point	Altitude in meters above ground level as measured by the altimeter
5	LIDAR data	3 floats, space delimited	A 3D point in world space. There can be multiple lines of LIDAR data, where each line represents a channel.

III.1.6 Mapping

As we saw in Chapter 2, Chiroptera treats its environment as a volume of discrete 3D regions called octants. If a point cloud data point is determined to exist with a given leaf-node octant, the octant's node type is set to filled and the original data point is discarded. Figure 7 shows a side-by-side of a point cloud versus its voxel representation. The latter is a lower-resolution representation of the environment, but Chiroptera does not require the level of fine detail or object recognition called for by engineering or survey applications. Rather, real-time flight decisions can be based strictly on knowing that some kind of object occupies a certain position in world space.

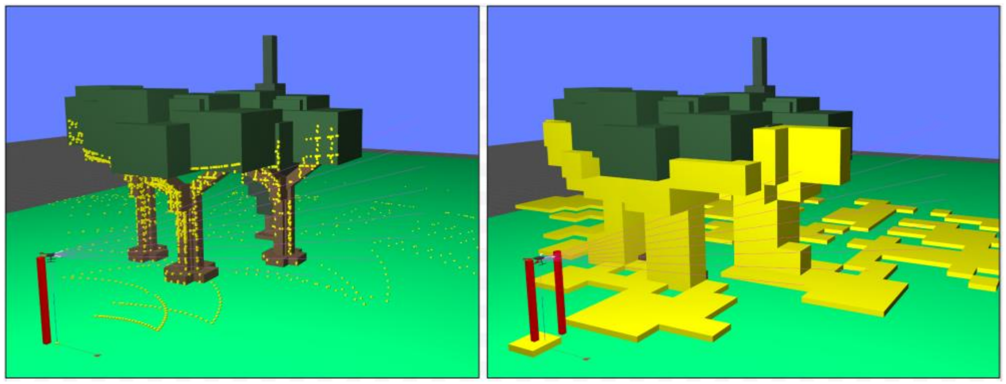


Figure 7: Point cloud visualized alongside its voxel representation.

III.1.7 Localization and Pathfinding

To find a path through the octree, its nodes need to be connected into a traversable graph suitable for analysis by a shortest-path pathfinding algorithm like Dijkstra's (1959) or A* (Hart 1968). Rodenberg et al. (2016) detail several historical approaches to this problem. As Chiroptera's octree is modeled as a pointer-representation, I implemented a combination of the lookup table solution described by Payeur (2006) combined with the

Namdari et al. (2015) approach in which neighboring nodes are joined during octree subdivision. For each octant, a list of face, edge, and vertex connections is stored. As illustrated in Figure 8 each octant can have up to 26 possible connections. The image on the left shows the 8 connections that share the same z-coordinate as the octant, while the image on the right represents the other 18 connections that are on a layer above or below the octant. The C++ implementation of the lookup table and node connections can be found in Appendix 4.

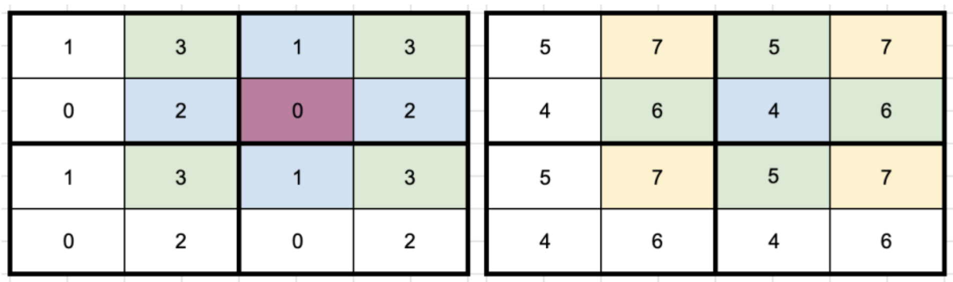


Figure 8: Node 0 shares 6 faces (blue), 12 edges (green), and 8 vertices (yellow)

With the octree regions now joined into a traversable graph, the A* algorithm is used to find a path from the drone's current position to a given goal node. A* is a computationally efficient search algorithm that has its origins in the Shakey project, an early attempt at providing an autonomous robot with capability of navigating an obstacle filled environment (Kuipers 2017).

Chiroptera's A* implementation is based on Hart et al's (1968) described execution of an evaluation function:

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

where *f(n)* is the estimated cost of the optimal path through a node *n*, based on the sum of *g(n)*, the cost of travel from the current node to node *n*, and *h(n)*, a heuristic that estimates the cost from node *n* to the goal node. Chiroptera's cost function *g(n)* returns an estimate of the distance between two nodes based on the type of connection they share as shown in Figure 9.

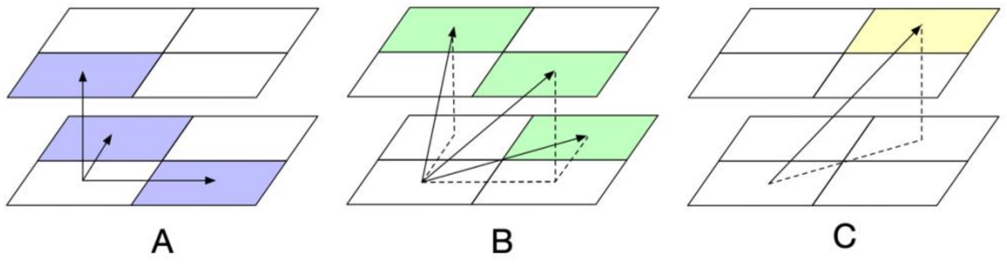


Figure 9: Directions of travel between connected nodes.

Nodes that share a common face (Figure 9.A) are considered to have a distance of 1.0. Nodes that share an edge (Figure 9.B) would be separated by a distance of $\sqrt{(1 + 1)} = \sqrt{2} \approx 1.4$, while nodes that share a single vertex (Figure 9.C) would be separated by a distance of $\sqrt{(\sqrt{2} + 1)} \approx 1.6$. For ease of calculation these values are defined in code as the integers 10, 14, and 16 (see Appendix 4.1).

The heuristic *h(n)* is based on the Euclidian distance between *n* and the goal node. To introduce a bias in favor of nodes that are at the same altitude as the goal node, we add an offset equal to the difference between the altitude and *n* and the altitude at the goal node.

A* is popular as its use of a weighted graph and heuristic guidance guarantee that it will return the shortest path between two points. This result is predicated on having complete *a priori* knowledge of the search space prior to algorithm execution. Chiroptera's pathfinding is designed to work in environments where the characteristics of the search space are not known in advance, however, but rather are updated over the course of the mission.

```
METHOD Pathfind::Execute(droneState)
  IF current_waypoint IS null THEN
    current_waypoint := Pathfind.NextWaypoint()
  ENDIF
  IF current_waypoint IS complete THEN
    next_waypoint := Waypoint.NextWaypoint()

    IF next_waypoint IS filled THEN
      Pathfind::AStar(droneState);
      current_waypoint := Pathfind.NextWaypoint()
    ELSE
      current_waypoint := next_waypoint;
    ENDIF
  ENDIF
  return current_waypoint.Execute(droneState);
ENDMETHOD
```

At each execution of the path traversal, Chiroptera's `Pathfind` action looks ahead to see if the next node in the path has been flagged as filled. If the path is clear, Chiroptera will command the drone to continue moving along the path. If the next node is now known to be obstructed, Chiroptera will execute another A* pass to find a new path around the obstruction. Assuming a valid path exists, the result is a guaranteed path from start to destination, but without the A* guarantee that the final path will be the shortest path between the locations. Chiroptera's A* implementation is guaranteed to find a valid path as long as one exists. If a path does not exist, Chiroptera will return an error and end the simulation. Options for handling this scenario in a physical drone will be discussed in section VII.1.

III.2 Execution

Upon launch, Chiroptera goes through a setup phase, then enters its primary event loop. The setup phase consists of first loading and validating the flight plan. As mentioned in Chapter 2, the flight plan is an ASCII text file with the .flt extension, containing a series of commands in the MAVLink format.

After loading the flight plan file, Chiroptera parses each line. Comment lines are denoted by a starting `#` character. If a line contains an unknown command or insufficient number of parameters for the given command, Chiroptera logs an error and exits. As each line is parsed, an `Action`-derived object is returned and added to the `ActionManager`.

After parsing the command file, Chiroptera then initializes the `CommunicationManager` and opens the named pipes. If either named pipe cannot be opened Chiroptera logs an error and exits. If the names pipes are opened successfully a thread is spawned off to handle Read operations from the telemetry pipe without blocking Chiroptera's main event loop.

With communication channels established, the final setup step is the initialization of the octree. The root node is recursively subdivided until it reaches a specified resolution, and the nodes are connected into a traversable graph as described above. Chiroptera is now ready to begin flight operations.

Flight operations are conducted within an event loop that executes five times per second and consists of a telemetry and state update phase, followed by a command execution phase.

```
METHOD Chiroptera::Update:
  // update state phase
  telemetry := CommunicationManager.GetTelemetry()
  drone_state := Chiroptera.update_drone_state(telemetry)
  Chiroptera.update_octree(telemetry)

  // execute command phase
  actionOutput := ActionManager.execute(drone_state)
  command := Chiroptera.encode_command(actionOutput)
  Chiroptera.send_command_to_drone(command)
ENDMETHOD
```

The loop begins by reading and parsing the telemetry payload data. As shown previously in Table 3, the telemetry data includes both drone flight information and the latest LIDAR data. The telemetry data is used to update the drone's kinematic state. Octree regions which contain the LIDAR and ultrasonic altimeter data are determined and their status is set to filled.

After the drone's state has been updated, the new state is passed as a parameter to the execution of the currently active command. The execute method first checks if the drone state's reset flag has been set. If it has, the `ActionManager` is reset to its initial conditions. If the drone state has not been reset, the `ActionManager` then checks if an action is currently active. Recall that the `ActionManager` stores the actions as a linked list. If there is no active action, the active action is set to the root of the action list. The active action is then initialized, taking the drone state as a parameter.

As each action completes execution the `active_action` is updated to point to the next node in the list. If there are no more actions to execute, the `ActionManager` returns an empty `ActionOutput` which Chiroptera interprets as the conclusion of the flight.

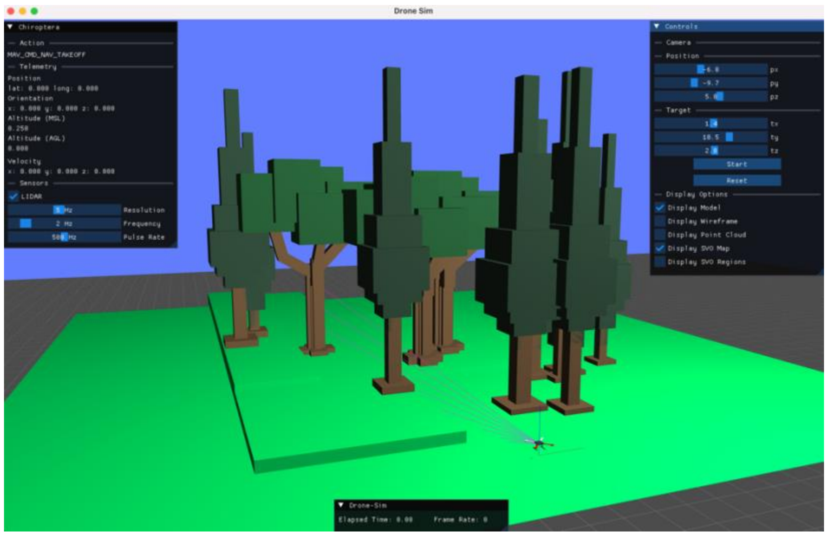
```
METHOD ActionManager::Execute(drone_state)
  IF drone_state.do_reset THEN
    reset(drone_state)
    return
  END IF

  IF NOT active_action THEN
    active_action = actions_list
    active_action.Init(drone_state)
  END IF

  IF active_action.is_complete THEN
    IF actions_list.next THEN
      active_action = actions_list->next
    ELSE
      return empty action
    END IF
    active_action.Init(drone_state)
  END IF

  return active.Execute(drone_state)
ENDMETHOD
```

Recall that the output of an action is an `ActionOutput` data structure containing two `Kinematic` objects representing a steering vector and angular acceleration. The final step of Chiroptera's Update method is to encode the action output into an ASCII string of the format shown in Table XXX. The encoded message is then written to the command pipe, from which Bat Cave will read and act on it.

Chapter IV:

The Bat Cave Environment and Drone Flight Simulator

Bat Cave is a voxel-based flight simulator within which Chiroptera can pilot a virtual quadcopter outfitted with an array of sensors through a 3D environment containing obstacles which must be avoided. A scene description language facilitates the building of different test environments using a library of primitive scene objects including a topologically varied ground plane and variety of trees. The custom physics engine, Fleder, implements the minimum necessary functionality to calculate and apply forces and detect collisions. To assist in visualizing sensor data, Bat Cave includes options to render the raw sensor data point cloud as well as the voxel representation, and octree region boundaries. A graphical interface (Figure 10) includes panels that display drone telemetry, simulation frame rate, and slider bars to adjust LIDAR, octree, and virtual camera settings. The user interface will be described in Section V.



```
▼ Chiroptera
— Action
MAV_CMD_NAV_TAKEOFF
— Telemetry
Position
lat: 0.000 long: 0.000
Orientation
x: 0.000 y: 0.000 z: 0.000
Altitude (MSL)
0.250
Altitude (AGL)
0.000
Velocity
x: 0.000 y: 0.000 z: 0.000
— Sensors
✓ LIDAR
5 Hz      Resolution
2 Hz      Frequency
500 Hz    Pulse Rate
```

```
▼ Controls
— Camera
— Position
-6.0      px
-9.7      py
5.0       pz
— Target
1.4       tx
10.5      ty
2.8       tz
Start
Reset
— Display Options
✓ Display Model
  Display Wireframe
  Display Point Cloud
✓ Display SVO Map
  Display SVO Regions
```

```
▼ Drone-Sim
Elapsed Time: 0.00     Frame Rate: 0
```

Figure 10: Bat Cave scene and user interface

Bat Cave is written in C++11. A 3D graphics renderer was implemented via the OpenGL version 3.3 API[8]. As OpenGL is a low level hardware API, C++-specific function pointers and bindings were generated using the Glad[9] loader-generator. Windowing and rendering contexts are managed by the GLFW[10] OpenGL utility library. Graphical user interface panels and displays are implemented via the C++ ImGUI[11] library.

The class diagram in Figure 11 illustrates Bat Cave and Fleder's core architecture.

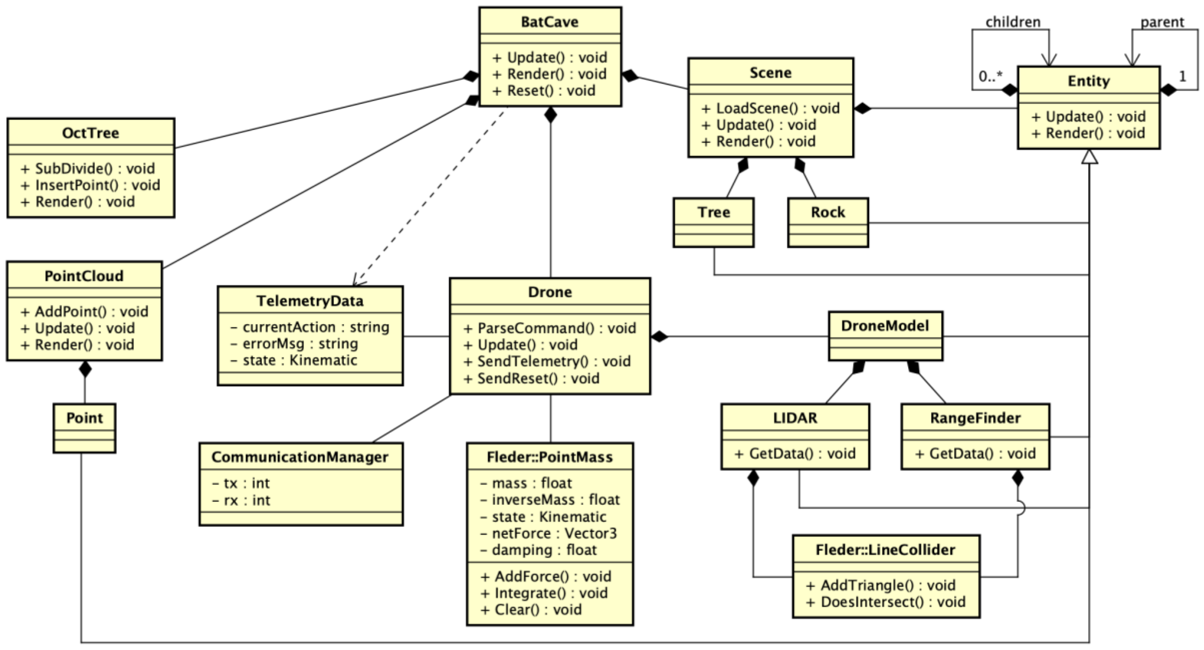


Figure 11: Bat Cave and Fleder UML class diagram

[8] https://www.opengl.org
[9] https://glad.dav1d.de
[10] https://www.glfw.org
[11] https://www.dearimgui.com

Like Chiroptera, the Bat Cave namesake class is primarily responsible for initializing and facilitating updates by the classes that implement simulation functionality. The structure and execution of Bat Cave's main event loop is detailed in section IV.2. The `Scene` class creates and manages the environment in which the drone operates. An abstract `Entity` class implements the core functionality required by all objects in the scene, including defining model data, rendering, and updating object attributes as the simulation runs. The `Scene` and `Entity` classes are described in Section IV.1. While OpenGL was used to render the simulation, Bat Cave's `Scene` and `Entity` classes were designed to be graphics API agnostic. As such, this class diagram intentionally omits the OpenGL integration classes.

Section IV.1.2. describes Fleder and its role in calculating drone motion dynamics and sensor interaction with environment objects. Bat Cave delegates all responsibility for drone operation to the `Drone` class detailed in section IV.1.3. Data acquired by the sensors is stored in the `PointCloud`. A simplified version of Chiroptera's `Octree` displays the voxel representation of the acquired data and provides visual insight into the environment graph Chiroptera uses for navigation. The `PointCloud` and `Octree` are described in section IV.1.4. The `Drone` communicates directly with Chiroptera via the `CommunicationManager` described in section IV.1.5.

IV.1 Implementation

IV.1.1 Scene

Bat Cave scenes are defined via a scene description language which provides instructions for placing voxel-based objects knows as *entities* in the scene. Each `Entity` has a unique identifier and includes a 3D voxel model, the geometry of which is described in a local coordinate system centered on the origin coordinate { 0, 0, 0 }. Complex entities can be composed as aggregates of simpler entities via a parent-child relationship. Each entity in an aggregate holds a reference to its parent entity, as well as a list of references to any child entities that have been added to the aggregate.

```
CLASS Entity
  # properties
  id: unsigned integer
  next_id: static unsigned integer

  model: Model
  parent: Entity;
  children: Entity[]

  # methods
  virtual Update()
  virtual AddToCollider()
  AddChild()
  Reset()
ENDCLASS
```

An entity’s transformation state is updated on each frame of the simulation, and any transformations are propagated down to the child entities as needed. A reset method is available to return the entity to its initial transformation state. To make an entity detectable by a sensor a method is provided to register an entity with Fleder’s collision detector. The specifics of collision detection are described below but suffice it to say that a necessary pre-requisite is the conversion of an entity's vertex data from its local coordinate system to world space coordinates.

Environment entities include a ground plane, rocks, and different species of trees. Trees are modeled as aggregated entities with the tree trunk as the root entity and the branches and leaf canopies as parent/child entities as shown in Figure 12, and described by the following scene syntax:

```
ground 10.0 10.0
oaktree
translate 0.0 3.0 0.0
```

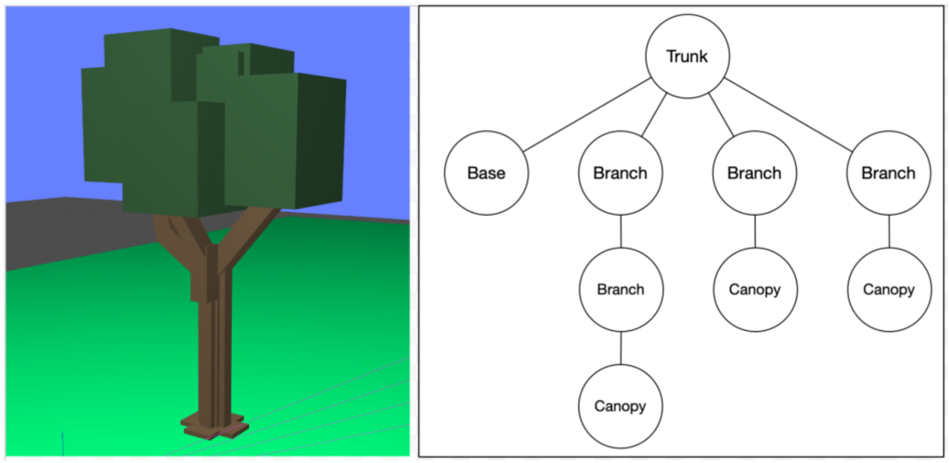


Figure 12: A tree modeled as an aggregated entity

In addition to transformations explicitly defined in the scene description, tree entities have their overall dimensions randomly scaled across a range of .75x – 1.25x, which allows for size variability to be introduced into the simulated groves. Figure 13 shows two different renders of the same scene description file.

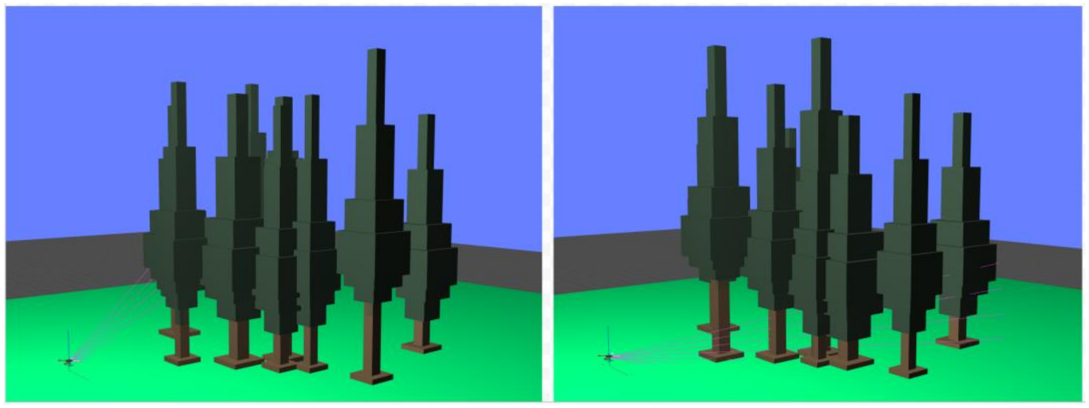


Figure 13: Random variation in the size of rendered trees

Full details of the scene description language is found in Appendix 3.

IV.1.2 The Fleder Physics Engine

Fleder was envisioned as a cross-platform, rendering-framework agnostic dynamic motion and collision physics simulator. Just as with Chiroptera, the necessary vector and matrix data structures and transformation operations were implemented from the ground up, rather than relying on any third party libraries. Compatibility with Bat Cave's OpenGL renderer is handled via an adapter class which provides methods to convert Fleder geometry to OpenGL geometry and vice versa.

Dynamic Motion: Fleder models the virtual drone as a point of specified mass with position, velocity, and acceleration kinematic attributes. The kinematic attributes can be modified through the application of dynamic forces to the point mass. Newton's second law of motion tells us that:

$$F = ma$$

where the net force F is equal to an object's mass m multiplied by the object's acceleration a. The S500 drone has a mass of 782g[12]. As described in Chapter 3, Chiroptera commands include the linear acceleration necessary to update the drone's motion. Fleder's point mass implementation is inspired by the particle implementation class described by Millington (2007).

```
CLASS PointMass
  # properties
  mass: float
  inverse_mass: float
  kinematic: Kinematic
  net_force: Vector3
  damping: float

  # methods
  AddForce(Vector3 force)
  Integrate(float deltaTime)
  ResetAccumulator()
ENDCLASS
```

The net force is a vector that represents the sum of all forces applied to the object. As commands are received from Chiroptera, Fleder updates the net force to represent the requested new linear acceleration. In the integration method the net force is multiplied by the pre-calculated inverse of the drone's mass, with the resulting value becoming the drone's current acceleration as per Newton's second law. The acceleration is scaled by the time interval and the product added to the drone's velocity. The velocity is then constrained by a damping factor which can be thought of as a representation of air resistance.

```
METHOD PointMass::Integrate(deltaTime):
  PointMass.position += PointMass.velocity * deltaTime;
```

[12] https://holybro.com/products/s500-v2-kit

```
  PointMass.acceleration =
    (PointMass.net_force * PointMass.inverse_mass);

  PointMass.velocity += (PointMass.acceleration * deltaTime);
  PointMass.velocity *= (PointMass.damping * deltaTime);
ENDMETHOD
```

Force Generators. Forces applied to an object can be constant, such as gravity, or variable such as the thrust generated by a drone's spinning rotors. Fleder implements a physics engine abstraction known as a `ForceGenerator` (Millington 2007), a class which obfuscates the calculation of a force's magnitude such that Fleder's `PointMass` integrator can apply the resulting force without needing to know the specifics of how the magnitude of the force was determined.

```
CLASS ForceGenerator
  # properties
  force: Vector3

  # methods
  virtual UpdateForce(PointMass p, float deltaTime)
ENDCLASS
```

Recall from section III.1.3 that Chiroptera returns an `ActionOutput` containing the linear acceleration necessary to adjust the drone's velocity. Looking again at Newton's second law, force is proportional to acceleration. A `MotorForceGenerator` class extended from the base `ForceGenerator` provides a means to add the acceleration returned by Chiroptera to the net force acting on the drone's `PointMass`. As shown in section IV.1.2.DynamicMotion, the `PointMass` integrator will determine the final acceleration to apply based on the net force and mass of the drone.

Collision Detection. Fleder implements a line-triangle collider for use by the sensors, inspired by the efficient real-time intersection algorithm described by Ericson (2005). Given a line segment defined by two points and a target triangle, Ericson's algorithm calculates a point-normal representation of the plane in which the triangle exists then tests if the line segment intersects the plane. If the line intersects the plane, the algorithm then checks if the intersection point is found within the original triangle.

```
CLASS LineCollider
  # properties
  entries: Triangle[]

  # methods
  AddTriangle(Vector3 vertices[3])
  DoesIntersect(Vector3 line[2], Vector3 r) : boolean
ENDCLASS
```

Recall that Bat Cave entities include a method to convert their local polygon coordinates to world-space coordinates. Fleder's `LineCollider` class declares a list of world-space `Triangles` which is populated as environment entities are added to the scene. As each polygon is added to the list, the triangle's normal is calculated and stored alongside the vertex data. On each sensor update the list of triangles is checked against a given line defined by two points. If an intersection is detected the distance from the drone to the intersection point is stored. Upon completion of testing the triangle list, the closet point of intersection to the drone, if any, is returned. Full C++ source code for the `LineCollider` can be found in Appendix 4.

IV.1.3 Drone

Bat Cave's simulated drone is composed of a 3D model of the S500 quadrotor frame, simulated LIDAR and ultrasonic rangefinder sensors, and hooks to the communication manager and Fleder's point mass and colliders.

Model: The drone frame model is implemented as a unique scene entity whose position and rotation properties are updated on each frame of the simulation. As the drone model's position in world space is updated, the position property is used as a source of data for two simulated sensors. The drone's { x, y } coordinates serve as a floating point representation of the latitude and longitude, and the { z } coordinate is used to represent the ASL as measured by the altimeter.

The LIDAR and ultrasonic range finder sensors are implemented as extensions of a sensor base class. The sensor base class defines attributes for the frequency of sensor readings and the elapsed time since the last reading as well as virtual function declarations to measure the sensor data and update the sensor model entity.

```
CLASS Sensor : extends Entity
  # properties
  frequency: float
  elapsed_time: float

  # methods
  virtual Update()
  virtual GetSensorData()
ENDCLASS
```

LIDAR: Bat Cave implements a LIDAR that is mounted to a gimbal on the front of the drone allowing for a 90 degree arc to be swept in front of the drone (Figure 15.A) at a default frequency of 2 Hz. The LIDAR emits 8 channels spread across a 30 degree vertical arc with an effective horizontal range of 30 meters. Mounting the LIDAR housing at an upward pitch angle (Figure 14.B) corrects for the drone frame's downward pitch when the drone is in flight (Figure 14.C).

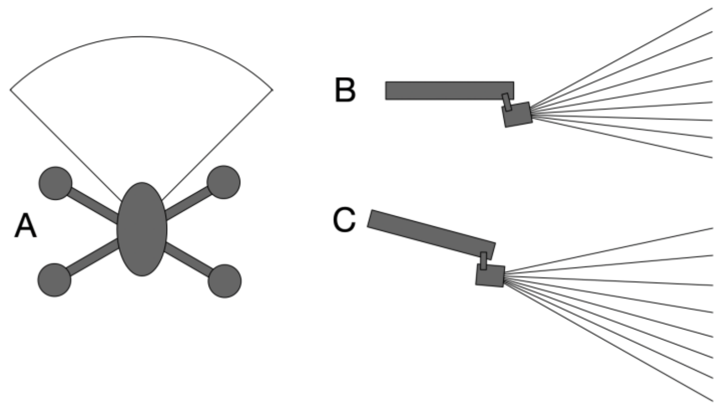


Figure 14: Area scanned by the drone LIDAR

The LIDAR has a updates at a default frequency of 10 Hz. On each update the 8 channels are passed to the line collider as described previously, returning the closest intersection point of each channel with an environment entity. These points are the basis of the SLAM point cloud as shown in Figure 15.

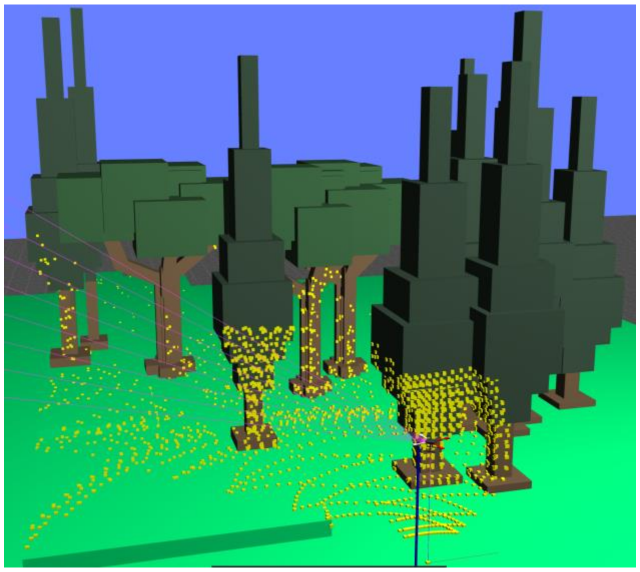


Figure 15: Generating a point cloud from LIDAR data.

Ultrasonic range finder: The downward facing range finder is mounted to the underside of the drone frame. Bat Cave assumes that the range finder will be stabilized by a gimbal that can respond to the drone's pitch angle, ensuring that the range finder always points directly downward. The range finders emit a single beam with a refresh rate of 20 Hz. The beam is passed to the line collider as described in the LIDAR section, with the returned intersection point representing the AGL. The AGL is added to the point cloud as seen in Figure 16 but is also used by Chiroptera to determine when the drone has reached its specified take off altitude, as well as to make altitude adjustments when terrain following is enabled.

IV.1.4 Point Cloud and Octree

Bat Cave stores point cloud data for visualization in an octree variation called a sparse voxel octree (SVO). An SVO is an octree in which a region is subdivided only if it contains a filled leaf node. As shown in Figure 16, rendering an SVO presents a clear visualization of where the collected data is located without the visual clutter of largely empty 3D lattice.

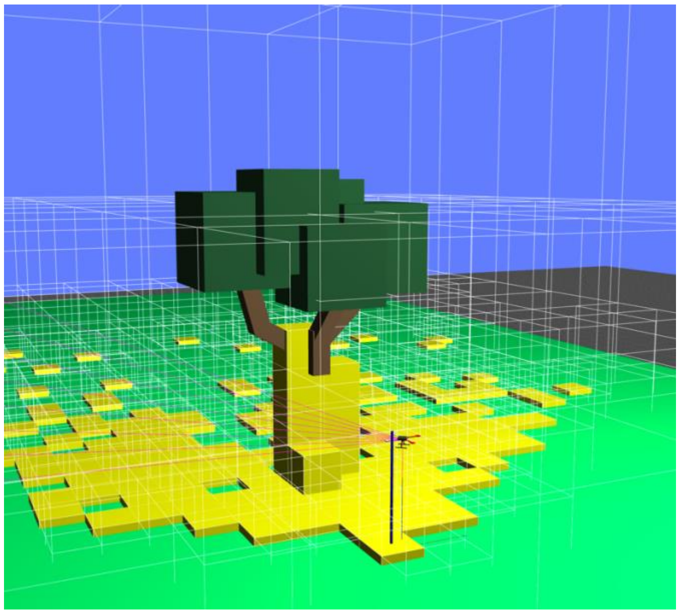


Figure 16: SVO visualization of octree data

IV.1.5 Communications

Bat Cave's `CommunicationManger` implementation is functionally identical to the Chiroptera `CommunicationManager` described in Chapter 3, except that the input (`rx`) channel receives commands and the output (`tx`) channel posts drone telemetry and sensor data. The communication manager is an association of the Drone class, as the communicated data either represents either a command issued to the drone, or drone state and sensor data communicated as telemetry. As with Chiroptera, a separate thread is spawned to handle read operations, so as to not block execution of the main program loop. The command and telemetry message formats are identical to those described in Chapter 3.

IV.2 Execution

Upon launch, Bat Cave goes through an initialization phase before starting its primary event loop. The initialization phase begins by setting up the scene. Bat Cave loads the scene description file then parses each line. If a line contains an unknown identifier or transformation, Bat Cave will log an error then exit. As each line is parsed the appropriate entity is added to the scene registry.

After loading the scene, the drone is initialized. A force generator to simulate the net force produced by the drone's motors is instantiated and attached to the drone model's body. The scene registry is then traversed, and references to each entity are added to the altimeter and LIDAR colliders. The `CommunicationManager` is instantiated as described in Chapter 3. Finally, the octree is instantiated. Unlike Chiroptera which subdivides its octree to the defined resolution upon initialization, Bat Cave's SVO as described in section IV.1.3 is subdivided to a resolution of 1 at the start.

The Bat Cave main loop handles updating the scene, drone, and sensors. The update rate of the simulation is between 30 and 60 frames per second, depending on the complexity of the scene. More complex scenes contain more entities to check collisions against, as well as larger volumes of point cloud data leading to update and rendering slowdowns. Several optimization possibilities are discussed in Section VII.

Bat Cave can exist in one of three possible execution states, *play, pause,* or *reset*, each of which has an effect on the execution of the update method. The initial state is set to *pause* to allow the initial simulation conditions to be examined prior to initiating the mission.

```
METHOD BatCave::Update(float delta_time):
    IF GetSimState() == SIM_PAUSE THEN
```

```
        return
    ELSE IF GetSimState() == SIM_RESET THEN
        ResetSimulation()
        drone.SendReset();
        return
    ENDIF

    fleder.Start();

    drone.Update(delta_time);
    fleder.RunPhysics(delta_time);

    models.Update(delta_time);
    sensors.Update(delta_time);

    elapsed_time += delta_time;
    frame_rate = 1./ delta_time;

    IF elapsed_time > 0.5 THEN
        drone.SendTelemetry();
    ENDIF
ENDMETHOD
```

The update loop begins by checking the simulation state. If the simulation state is *pause*, the loop exits immediately. If the state is *reset*, Bat Cave resets the octree, clears the point cloud, and returns the drone to its initial kinematic state. A reset signal is sent to Chiroptera as described in section III.1.2.

If the simulation is not paused or reset, the physics and entity states are updated. Fleder first clears the net forces applied to the drone and prepares to calculate the dynamics for the current frame. The drone's update method is then invoked.

```
METHOD Drone::Update(delta_time)
    command = parse_command();

    set_thrust(command->linear_acceleration);
    set_angular_acceleration(command->angular_acceleration);
ENDMETHOD
```

The drone update begins by reading and parsing the command payload. Refer to Table 2 in Section III.1.V for the format of the command data. The provided linear and angular accelerations are used to update the drone's state and are passed to Fleder for use in the force calculations described in section IV.1.2. The drone, lidar, and ultrasonic range finder model position and rotation attributes are then updated prior to rendering.

With the new drone and sensor kinematic attributes set, the LIDAR and ultrasonic range finder data is acquired. The sensors leverage the `LineCollider` described in section IV.1.2 to determine if the sensor channel(s) intersects with an environment entity. If an intersection is detected, the point of intersection is added to the point cloud data.

After all updates have completed the simulation timer is updated the new drone state is sent to Chiroptera. Data is written to the telemetry pipe at a frequency of ~5 Hz.

Chapter V:

User Interface

This section describes the command line and graphical interfaces used to execute and interact with Chiroptera and Bat Cave.

V.1 Chiroptera

Chiroptera is executed as a command line application:

```
./chiroptera <mission_filename>
```

The application takes as an argument the name of an ASCII text file with the .flt extension, the contents of which describe a sequence of MAVLink commands as described in section III.1.4.

```
# fells-test.flt
MAV_CMD_NAV_TAKEOFF 1.5
MAV_CMD_PATHFIND 5.0 10.0 2.0
MAV_CMD_NAV_LAND 0.0 0.0
```

In addition to the command file, Chiroptera includes an environment settings header file in which the dimensions and resolution of the of the octree can be specified. The octree is defined by two vertices representing the bottom-left and top-right of the region. The header file can be found in Appendix IV.1.

V.2 Bat Cave

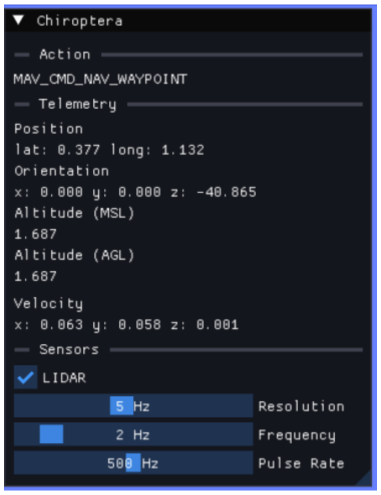
Like Chiroptera, Bat Cave is executed from the command line, passing the name of the environment file as a parameter:

```
./batcave <scene_filename>
```

Launching Bat Cave opens the interface shown in Figure 10 at the start of Section IV. User interface panels display data about the current state of the simulation, as well as controls to start, stop, or change the simulation parameters.

V.2.1 Drone Telemetry

The top left panel shown in Figure 17 displays the telemetry data being transferred to/from Chiroptera. At the top of the panel is the currently active command Chiroptera has issued to Bat Cave. If Chiroptera returns an error, it will be shows below the displayed action. Just below the command is the drone telemetry being sent to Chiroptera. The telemetry panel also includes the drone’s measured ASL. Chiroptera does not use the ASL for any actions at this time, but the value is provided for the user as a comparison to the AGL value returned by the downward facing range finder.



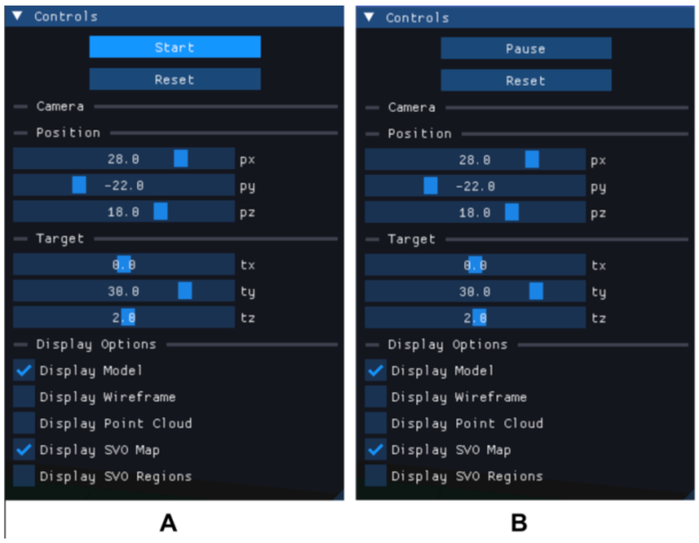
```
▼ Chiroptera
— Action
MAV_CMD_NAV_WAYPOINT
— Telemetry
Position
lat: 0.377 long: 1.132
Orientation
x: 0.000 y: 0.000 z: -40.865
Altitude (MSL)
1.687
Altitude (AGL)
1.687
Velocity
x: 0.063 y: 0.058 z: 0.001
— Sensors
✓ LIDAR
5 Hz      Resolution
2 Hz      Frequency
500 Hz    Pulse Rate
```

Figure 17: Chiroptera command and drone telemetry panel

The LIDAR subpanel offers slider bars that can be used to modify the LIDAR sensor settings. The resolution value is not reflected in Chiroptera but is provided to offer a visual insight into what the octree would look like at different levels of subdivision. The octree resolution can be modified in the `env.h` file described in section V.1.

V.2.2 Execution Control and Rendering Options

The top-right panel provides controls to start, pause, or reset the simulation. Bat Cave launches in a paused state. Pressing the Start button (Figure 18.A) will begin execution of the mission and the button text will change to read Pause (Figure 18.B). The Reset button resets the simulation to its initial state as described in section IV.2.



▼ Controls

Start / Pause

Reset

Camera

Position

28.0	px
-22.0	py
18.0	pz

Target

0.0	tx
30.0	ty
2.0	tz

Display Options

- ✓ Display Model
- Display Wireframe
- Display Point Cloud
- ✓ Display SVO Map
- Display SVO Regions

Figure 18: Start, Pause, and Rest buttons

Two sets of slider bars located below the Start and Reset buttons can be used to change the view of the simulation by adjusting the position of the virtual camera. Finally, a group of check boxes provides options to change the way the scene is rendered. By default, the scene models and SVO are displayed. Options are available to render the scene in wireframe, show/hide the point cloud, and show/hide the SVO regions and filled leaf nodes.

Chapter VI.

Experimental Results

Development of Chiroptera and Bat Cave followed an iterative process, in which new capabilities added to Chiroptera were validated in Bat Cave before moving on to the next step. Sample Chiroptera flight plans were developed to test each new decision-making capability, steering behavior, and associated MAVLink commands. These test flights were conducted in sample scenes consisting of a 30 cubic meter navigable region containing increasingly complex obstacle layouts.

VI.1 Flight Commands and Localization

The first four test flights were conducted in an open scene consisting of a ground plane and no obstacles. The Bat Cave GPS provided latitude and longitude data relative to the drone's starting position of { 0.0, 0.0 } while the downward facing ultrasonic rangefinder provided a measurement of the AGL. Both sensors were updated at a 10 Hz frequency. This value is sufficient for navigating the simulation environment at a speed of 1 m/s. Sensor updates sufficient for high speed maneuvering are outside the scope of this work but are addressed in section VII.

VI.1.1 Takeoff and Landing

The most basic test of the Chiroptera system involves issuing a takeoff command, holding position for a specified number of seconds, then landing successfully. This test validates that the range finder is correctly measuring the AGL above the ground plane.

```
# test-takeoff-landing.flt
```

```
1 MAV_CMD_NAV_TAKEOFF 2.0
2 MAV_CMD_NAV_DELAY 2
3 MAV_CMD_NAV_LAND 0.0 0.0
```

VI.1.1 Arrival

Following a successful takeoff and landing, the next operation validated is a straight-line flight to a specified destination while maintaining altitude as shown in Figure 19.

```
# test-arrive.flt
1 MAV_CMD_NAV_TAKEOFF 2.0
2 MAV_CMD_NAV_DELAY 2
3 MAV_CMD_NAV_WAYPOINT 0 0.0 10.0 0.0
4 MAV_CMD_NAV_LAND 0.0 0.0
```

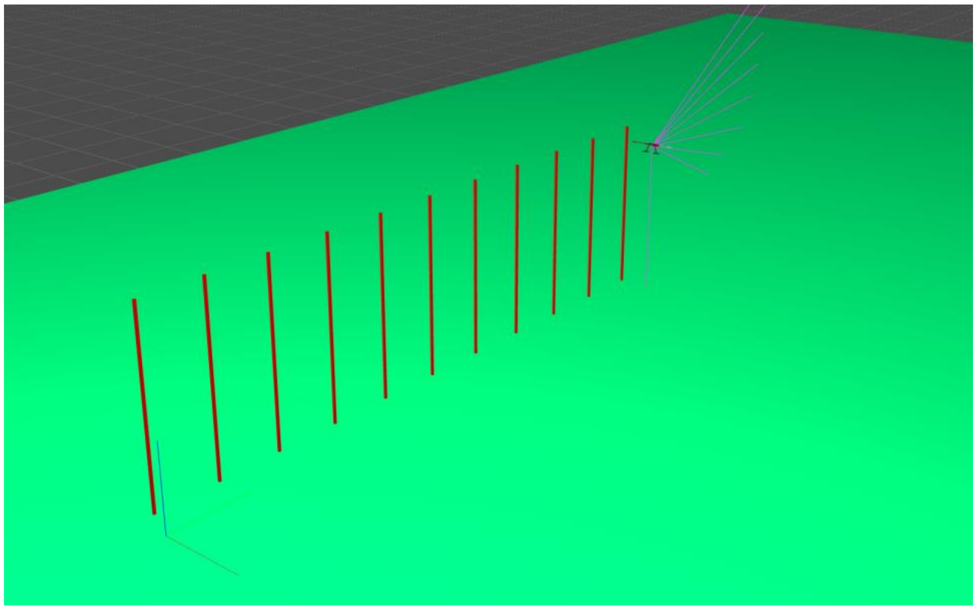


Figure 19: Arrival at a waypoint

VI.1.2 Arrive at Destination then Return Home

After flying to a specified destination, we validate that the drone can align and return to its home location (Figure 20).

```
# test-return.flt
1 MAV_CMD_NAV_TAKEOFF 2.0
2 MAV_CMD_NAV_DELAY 2
3 MAV_CMD_NAV_WAYPOINT 0 0.0 10.0 0.0
4 MAV_CMD_NAV_RETURN_TO_LAUNCH
5 MAV_CMD_NAV_LAND 0.0 0.0
```

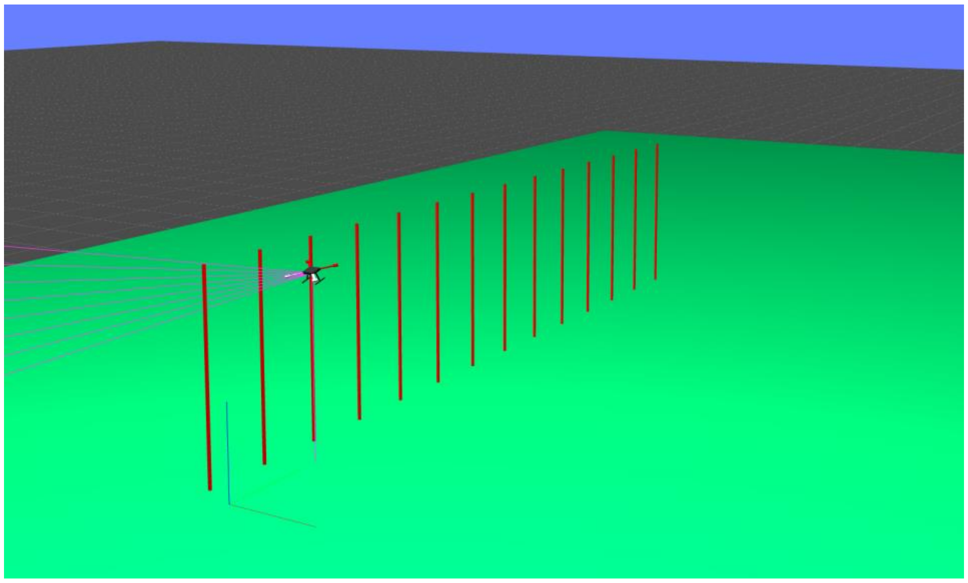


Figure 20: Drone travels to a waypoint then returns to its starting point

VI.1.3 Travel to Multiple Waypoints

In this test, the drone aligns and flies in a clockwise direction to two coordinates before returning to its starting position (Figure 21).

```
#  test-multiple-waypoints.flt
1 MAV_CMD_NAV_TAKEOFF 2.0
2 MAV_CMD_NAV_DELAY 2
3 MAV_CMD_NAV_WAYPOINT 0 -4.0  6.0 0.0
```

```
4 MAV_CMD_NAV_WAYPOINT 0  3.0 12.0 0.0
6 MAV_CMD_NAV_RETURN_TO_LAUNCH
7 MAV_CMD_NAV_LAND 0.0 0.0
```

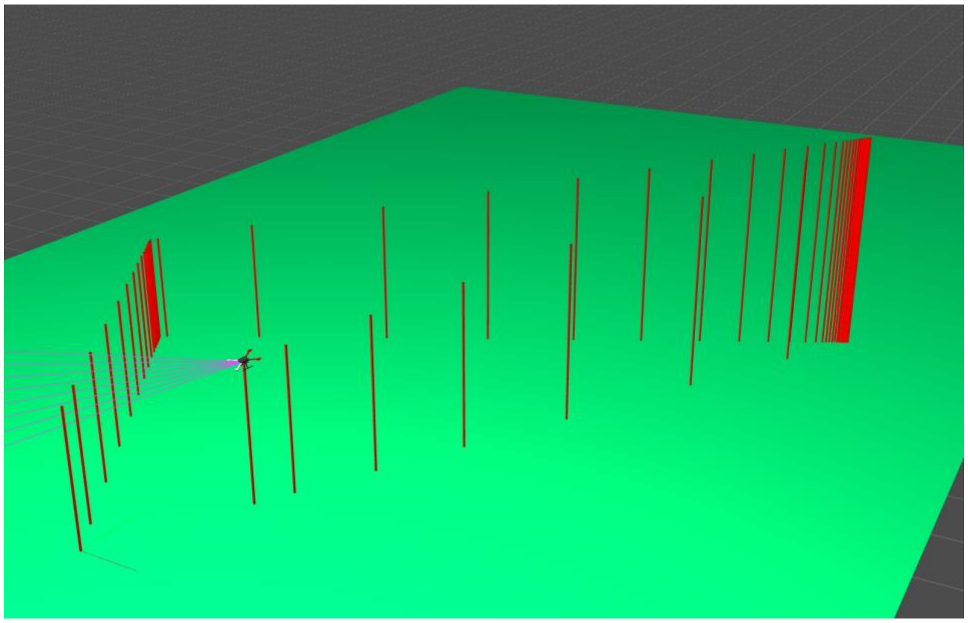


Figure 21: Drone travels to multiple waypoints.

VI.2 Terrain Following

With the flight commands and localization in place, it was now possible to instruct Chiroptera to maintain a consistent AGL. The flat ground plane was expanded to model a terraced landscape. The waypoints previously defined in the test-multiple-waypoints flight plan were now located at different Terrain ALTs.

As in the previous test, the `test-multiple-waypoints.ftl` flight was executed. As expected, the AGL remained consistent throughout the entire flight as shown in Figure 22.

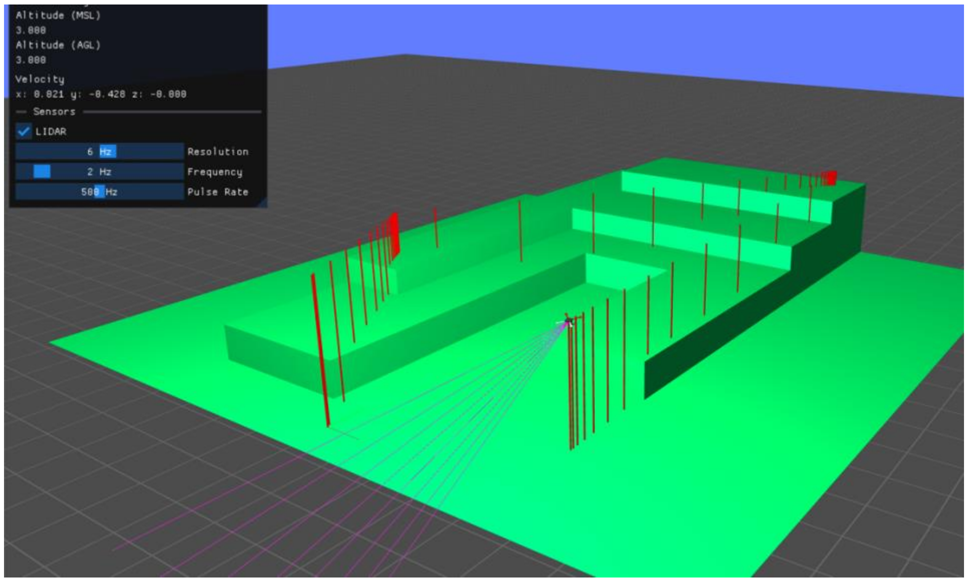

```
Altitude (MSL)
3.000
Altitude (AGL)
3.000
Velocity
x: 0.021 y: -0.428 z: -0.000
— Sensors
✓ LIDAR
6 Hz     Resolution
2 Hz     Frequency
500 Hz   Pulse Rate
```

Figure 22: Drone AGL is not set to follow terrain.

To validate terrain following, `test-multiple-waypoints.`ftl was modified to enable the `TERRAIN_FOLLOW` flag.

```
# test-terrain-follow.flt
1 MAV_CMD_DO_TERRAIN_FOLLOW 1
2 MAV_CMD_NAV_TAKEOFF 2.0
3 MAV_CMD_NAV_DELAY 2
4 MAV_CMD_NAV_WAYPOINT 0 -4.0  6.0 0.0
5 MAV_CMD_NAV_WAYPOINT 0  0.0 17.0 0.0
6 MAV_CMD_NAV_WAYPOINT 0  4.0  4.0 0.0
7 MAV_CMD_NAV_RETURN_TO_LAUNCH
8 MAV_CMD_NAV_LAND 0.0 0.0
```

The AGL variation due to terrain following is shown in Figure 23.

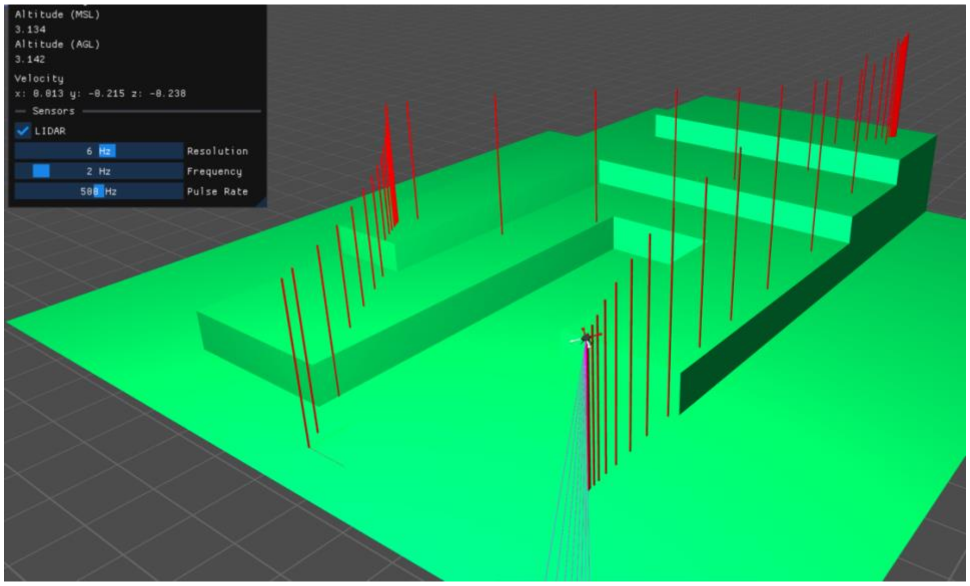

```
Altitude (MSL)
3.134
Altitude (AGL)
3.142
Velocity
x: 0.013 y: -0.215 z: -0.238
— Sensors
✓ LIDAR
6 Hz      Resolution
2 Hz      Frequency
500 Hz    Pulse Rate
```

Figure 23: Drone AGL is adjusted to follow the terrain.

VI.3 SLAM

Chiroptera's mapping and flight planning can be tricky to test, as pathfinding is not necessarily deterministic even when conducted in the same scene. Variation in pathfinding is a result of the initial pathfinding solution's dependence on the results of the initial LIDAR scan of the environment, while subsequent pathfinding decisions will be made relative to the drone's position at that time. If the LIDAR scan is biased in one direction the initial pathfind may favor a path that has more obstacles whose positions are initially unknown to Chiroptera as they were not mapped in the initial scan. To limit this uncertainty the drone is commanded to adjust its yaw from +/- 45 deg then back to its initial heading, effectively instructing it to "look both ways" before beginning pathfinding.

Unlike an A* search of an environment whose layout is known fully in advance, the adaptive nature of Chiroptera's pathfinder means that there can be more than one correct way in which to navigate to a waypoint, but the final path might not be the shortest. In Figure 24, we see two different pathfinding solutions to a scene consisting of four trees with a waypoint on the far side. Plotting a path between or around the trees while maintaining a consistent altitude are both valid solutions to arrive at the given waypoint.

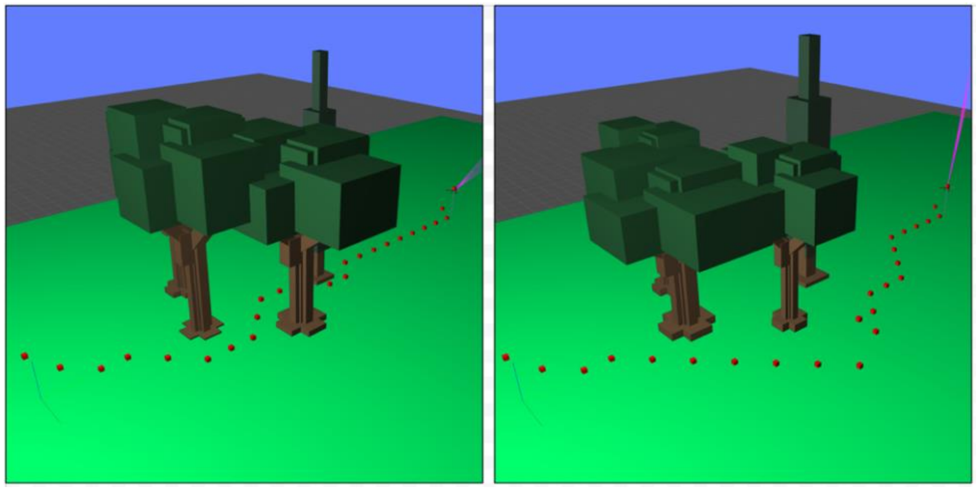


Figure 24: Two different pathfinding solutions to the same scene and flight plan.

In the case of a horizontal obstruction, the 3D interconnectivity of Chiroptera's octree facilitates altering the drone's altitude to find a path as shown in Figure 25. This example illustrates a scenario in flying over the obstacle is the only option. In other cases, flying over an obstacle may not be necessary, but could represent the shortest path. If terrain following is enabled, vertical pathfinding takes precedence when determining the drone's altitude. This precedence is offset by Chiroptera's A* heuristic biasing pathfinding in favor of nodes that are close in altitude to the drone's target altitude as described in section III.1.7.

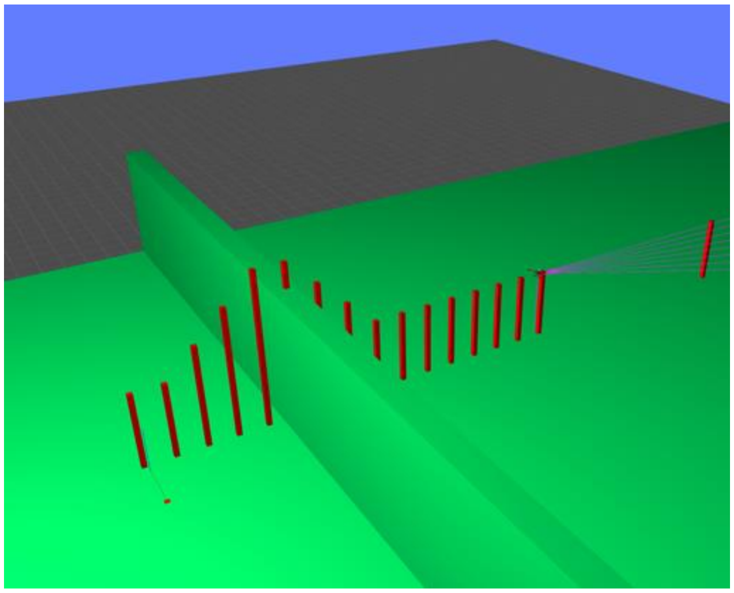


Figure 25: Plotting a path over a horizontal obstruction.

In Figure 26.A we see the drone navigate through a hole in a wall, then adjust its altitude to fly over a subsequent wall. The second wall was not initially visible, and after passing through the hole the drone noticeably stopped for a moment to reconsider its path. To confirm that the drone was capable of flying over or under a horizontal obstruction, the height of the back wall was increased for the next test. As shown in Fig 26.B Chiroptera now calculated a path that flew under the back wall.

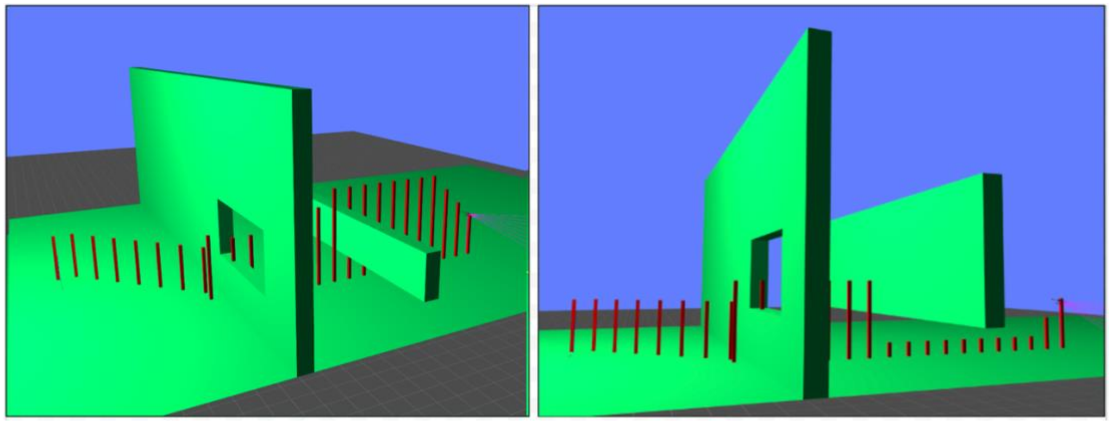


Figure 26: A & B: Plotting a path through a hole in the environment.

VI.4 Simulating a Real Environment

As this project was partly inspired by the watching bats fly through the groves of the Middlesex Fells[13], a small part of that environment was modeled as a scene in Bat Cave. A topographic map of the area around the Winchester Reservoir shown in Figure 27.A was the inspiration for the scene shown in Figure 2.B. This wilderness environment provides the backdrop for the ultimate test of Chiroptera's pathfinding and navigation ability. In these examples the positive y-axis is considered to point north. The octree encompasses a volume of 50 m^3 at a resolution level of 6, so each leaf node will represent a ~ .75 m^3 region. Environment textures were sourced from OpenGameArt.org[14].

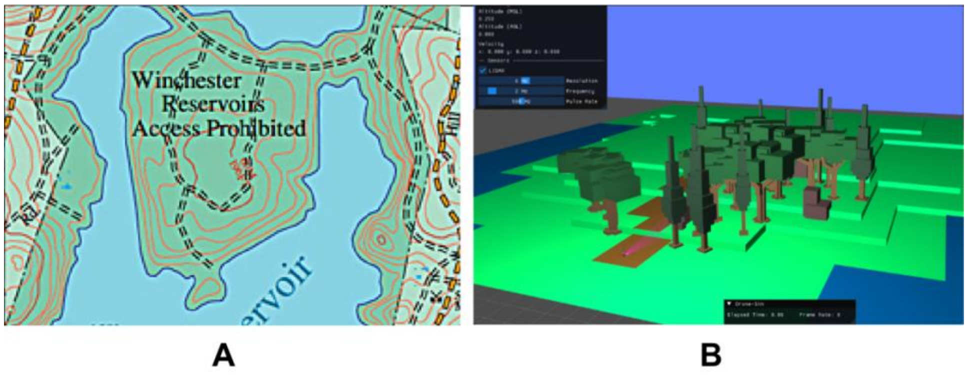


Figure 27: A. Middlesex Fells topographic map and B. Bat Cave representation

Two different missions were tested in which the goal was to navigate the drone from the starting point at the bottom of the map to the fork in the path. The first mission plan explicitly specified active terrain following and multiple waypoints along the path.

```
# test-fells-manual-01.flt
1 MAV_CMD_DO_TERRAIN_FOLLOW 1
2 MAV_CMD_NAV_TAKEOFF 2
3 MAV_CMD_NAV_DELAY   1
```

[13] https://www.mass.gov/locations/middlesex-fells-reservation
[14] https://opengameart.org/

```
4 MAV_CMD_NAV_WAYPOINT 0 -0.5 5.0 2.0
5 MAV_CMD_NAV_WAYPOINT 0 -2.0 10.0 2.0
6 MAV_CMD_NAV_WAYPOINT 0 1.0 15.0 2.0
7 MAV_CMD_NAV_WAYPOINT 0 1.5 20.0 2.0
```

In the first test the waypoints were deliberately chosen to create a flight path that follows the ground path as shown in Figure 28. Three intermediate waypoints were set, with the final waypoint located ~20 meters from the drone's initial position. The elapsed time from takeoff to arrival at the destination waypoint was ~27 seconds.

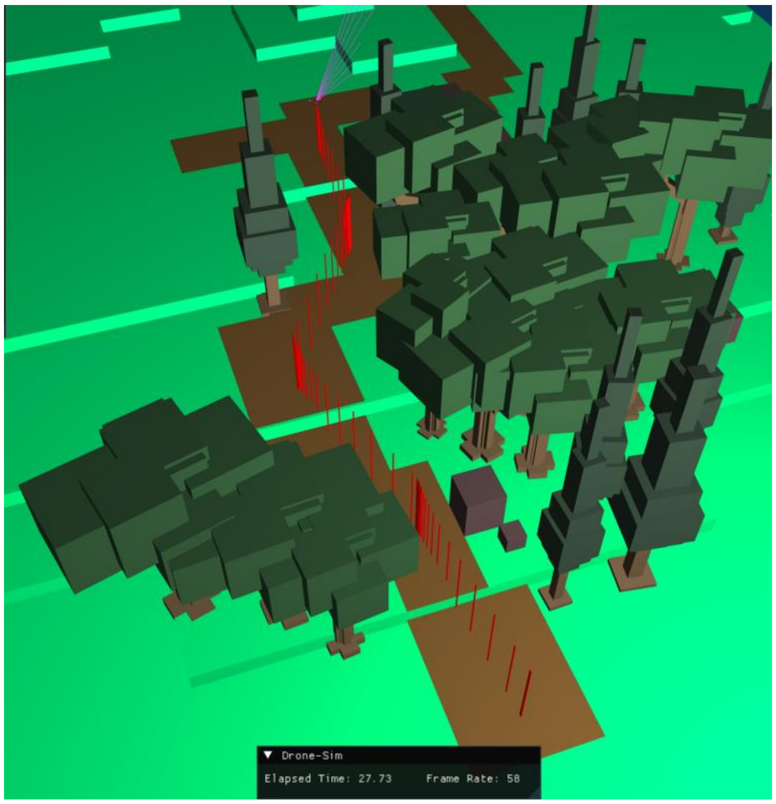


Figure 28: Manually defined waypoint guiding Chiroptera to the fork in the path

In the second test, the manual waypoints were replaced by a single `MAV_CMD_PATHFIND` command, the destination of which matches the destination of the final manually defined waypoint in `test-fells-manual-01.ftl`.

```
# test-fells-pathfind-01.flt
MAV_CMD_NAV_TAKEOFF 2
MAV_CMD_CONDITION_YAW 45.0
MAV_CMD_CONDITION_YAW -45.0
MAV_CMD_NAV_DELAY   1
MAV_CMD_PATHFIND 1.5 20.0 2.0
```

The second test begins with commands to adjust the drone's yaw and "look both way" as described in section VI.3. This side to side scan took ~30 seconds to complete. The initial octree map is shown in Figure 29.

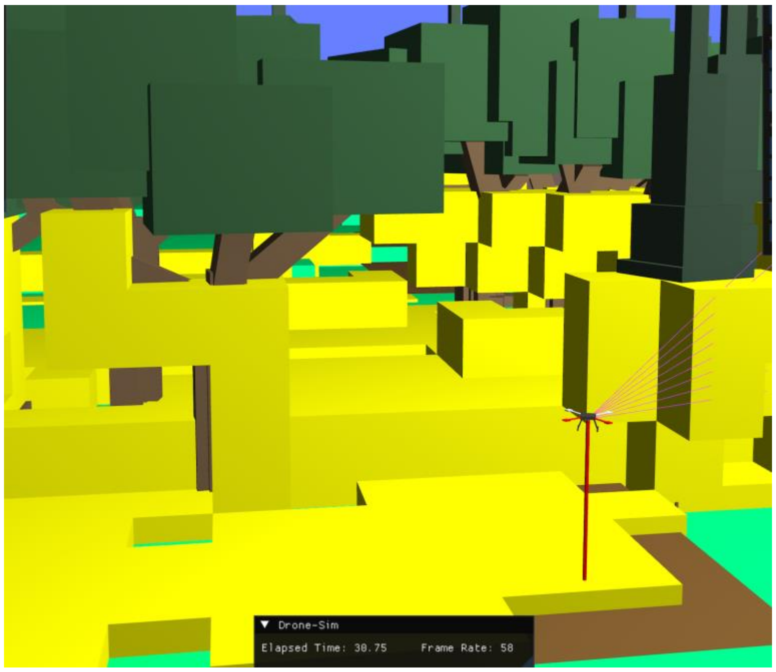
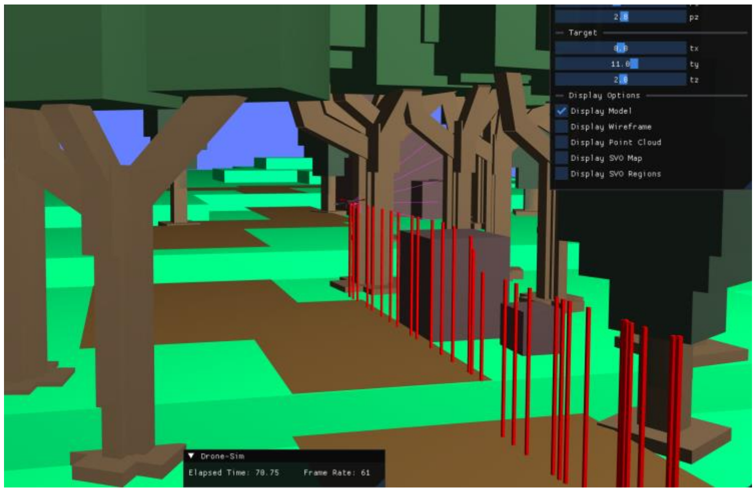


Figure 29: Octree map after a side-to-side scan from the drone's initial position

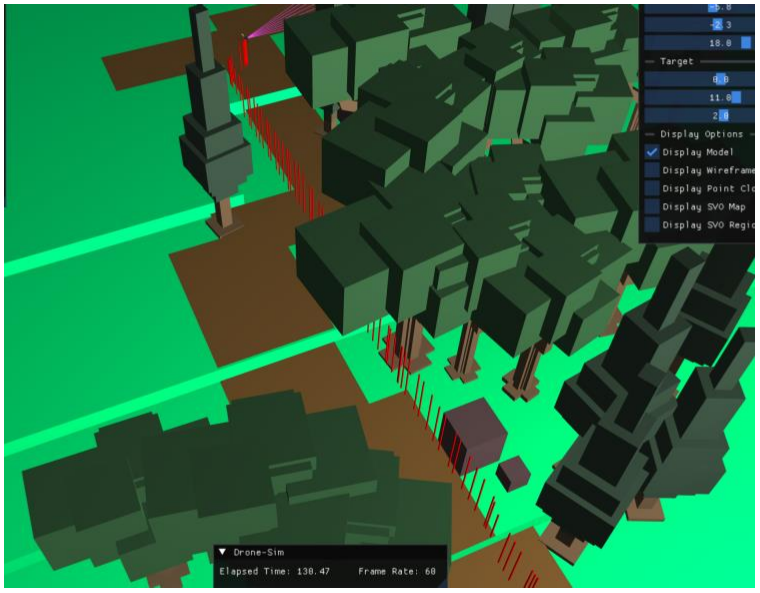
Based on the initial scan, Chiroptera found a nearly straight path to the goal position. Compared to the manually defined waypoints that followed the foot path, Chiroptera's flight path closely hugged the tree line as shown in Figures 30.



2.0 pz
0.0 tx
11.0 ty
2.0 tz
Elapsed Time: 78.75 Frame Rate: 61

Figure 30: Finding a path that closely hugged the tree line

Figure 31 shows the complete flight path, which was straighter and more direct than the manually defined path. Chiroptera required ~130 seconds to conduct this mission.



-2.3
18.0
0.0
11.0
2.0
Elapsed Time: 130.47 Frame Rate: 60

Figure 31: Chiroptera’s calculated route to the fork in the foot path

To test Chiroptera’s ability to navigate through off-trail, the preceding tests were expanded to include a new target position halfway up the right-hand path, roughly ~30 meters from the drone’s initial position as shown in Figure 32. In the manual test two additional waypoints were defined, and the elapsed time from takeoff to arrival at the new target coordinate was ~43 seconds.

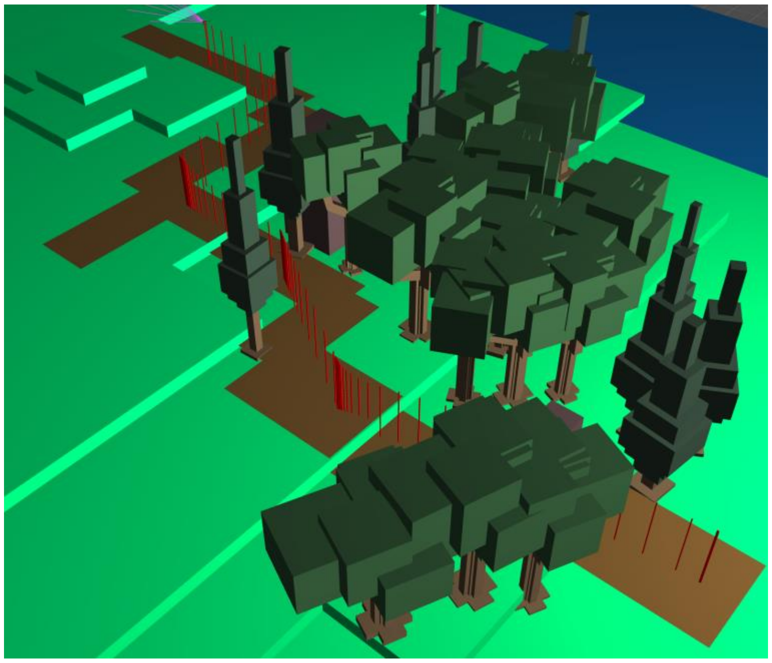


Figure 32: Expanding the Fells test to include a further target waypoint.

In the previous pathfinding test, Chiroptera again was instructed to make an initial scan of the environment. As before, this scan took ~30 seconds to complete. Unlike the previous pathfinding test in which the destination coordinate was due north of the starting position, the new destination coordinate was located north-northeast of the start. As shown in Figure 33, Chiroptera almost immediate turned northeast and traveled into the forest. In Figure 34 the drone can be seen exiting the forest on the far side after finding a path that took it between the trees.

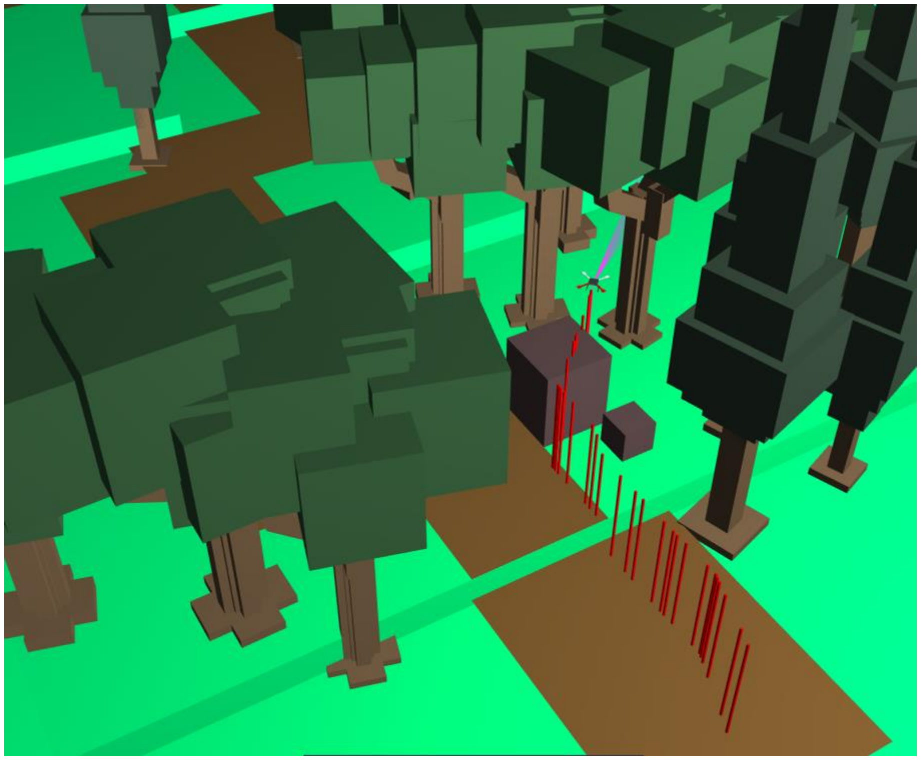
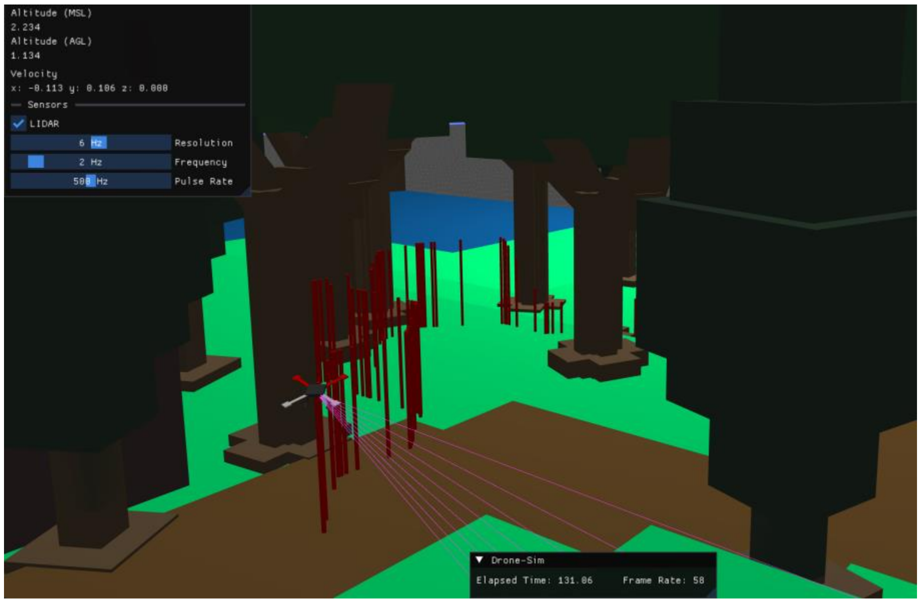


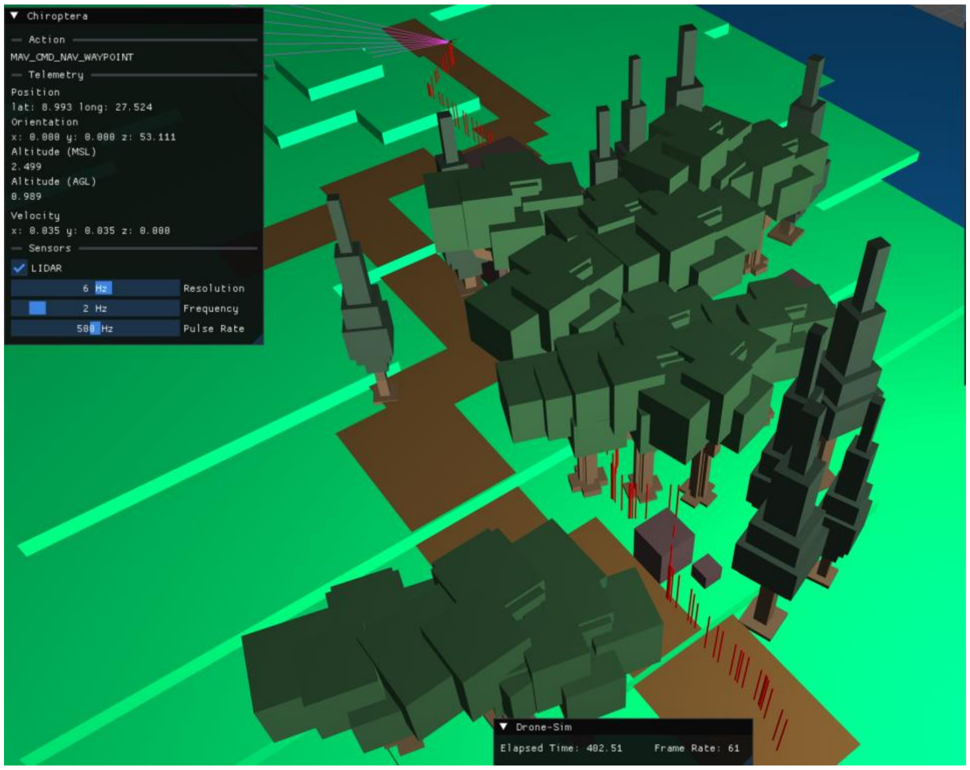
Figure 33: Chiroptera navigating off path and into the forest.



```
Altitude (MSL)
2.234
Altitude (AGL)
1.134
Velocity
x: -0.113 y: 0.106 z: 0.000
— Sensors ——————————
✓ LIDAR
6 Hz      Resolution
2 Hz      Frequency
500 Hz    Pulse Rate
```

```
▼ Drone-Sim
Elapsed Time: 131.06      Frame Rate: 58
```

Figure 34: Chiroptera exiting in the forest.



```
▼ Chiroptera
— Action ———————————————
MAV_CMD_NAV_WAYPOINT
— Telemetry ————————————
Position
lat: 8.993 long: 27.524
Orientation
x: 0.000 y: 0.000 z: 53.111
Altitude (MSL)
2.499
Altitude (AGL)
0.989
Velocity
x: 0.035 y: 0.035 z: 0.000
— Sensors ——————————————
✓ LIDAR
      6 Hz        Resolution
      2 Hz        Frequency
    500 Hz        Pulse Rate
```

```
▼ Drone-Sim
Elapsed Time: 402.51     Frame Rate: 61
```

Figure 35: Chiroptera's direct route through the forest to reach the far side

Chiroptera required ~6.5 minutes to complete this flight, the path of which is shown in Figure 35. After the opening ~30 second scan of the environment, the initial pathfinding operation took ~3 minutes. Given that the octree regions representing the environment beyond the area mapped by the initial scan were treated as unfilled, Chiroptera's A* operation would consider them all to be traversable nodes resulting in a much larger set of candidate paths to examine.

As Chiroptera traveled through the forest it was observed to be recalculating a path every few seconds. Subsequent pathfinding operations took progressively less time to complete, as the more detailed map environment available to Chiroptera allowed the A* implementation to discard octree nodes that were now known to be blocked.

Chapter VII:

Conclusions and Future Work

The goal of this thesis was the development of two software applications to test the feasibility of non-camera based SLAM operations. The first application, Chiroptera, has demonstrated that autonomous pathfinding through unknown environments whose topography is modeled via a moderate-resolution voxel octree is a viable approach to this challenge. The test flights in section VI.3 and VI.4 in particular show that if a path is available Chiroptera is guaranteed to find it, though it may take significant time to do so. It should be noted that in the section VI.4 manually defined flights the topography of the scene was fully visible to the user and reasoned decisions regarding waypoint positioning could be made in advance. The distance between waypoints was ~5 m, while the size of Chiroptera's octree leaf nodes was ~.75m^3. The manually defined path allowed the drone to maintain speed for longer stretches versus the automated path, though the latter can be improved by enhancing Chiroptera's path following algorithm such that it does not come to a complete stop at each intermediate waypoint. However, automated pathfinding allowed the drone to successfully fly directly through the forest, a task that would have been difficult for a manually defined path. Without knowing precise tree positions and branch structures, a human would not be able to accurately define waypoints through the forest suitable for a safe flight to be conducted.

Chiroptera's pathfinding algorithm demonstrated a critical vulnerability, however, when traversing a diagonal connection between two nodes which are obstructed on either side as shown in Figure 36. As we saw in section III.1.7, Chiroptera looks ahead in the pathfinding node list to check if any nodes have been recently flagged as filled. This functionality should be expanded to check the condition of adjacent leaf nodes when a diagonal move is made to ensure that Chiroptera does not attempt to fly the drone through an occupied space.

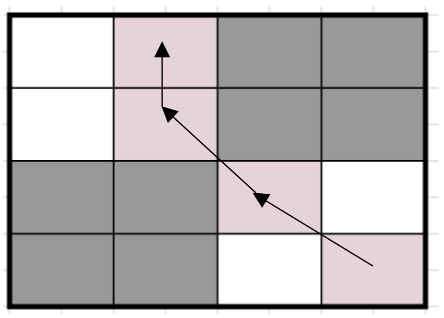


Figure 36: A seemingly valid path that could result in colliding with an obstruction

The second software application, Bat Cave, proved to be a suitable environment in which to test Chiroptera's capabilities and work to resolve the described algorithmic difficulties. Debugging Chiroptera's behavior was made difficult by the black-box nature of the application. As Chiroptera was conceptually designed to not know or care if it was operating in a simulated or real environment, the only way to visualize its internal behavior was by attempting to mimic its octree structure in Bat Cave, then plot its current drone state and target as commands were received. No good way was found to visualize pathfinding paths as they were being calculated, a challenge that is critical to address in future work, described in section VII.1.1.

Out of the need to prioritize development time towards the pathfinding algorithm, the voxel-based environment features were modeled at a low-resolution and level of detail. Bat Cave is technically capable of supporting more complex models, and its ability to simulate non-camera sensor interactions with environmental features is a significant and unique contribution of this thesis.

VII.1 Future Work

VII.1.1 Pathfinding Optimizations and Debugging

Chiroptera currently executes an A* search on the full octree. As it does not have *a priori* knowledge of the topography of its environment, nodes in the octree graph that are not immediately scanned are initially considered empty. As such, the initial pathfinding has been observed taking as long as ~3 minutes when attempting to find a path to a point ~30 meters away. A possible solution to optimize pathfinding efficiency would be to conduct the A* search through an SVO instead of a full octree. This would require a modification to the graph connections and A* algorithm heuristic such that it can determine when to search a node on a higher or lower level of the graph.

Concurrent to algorithmic improvement is the need to visualize the pathfinding process. A debugging message sent to Bat Cave as the pathfind algorithm is executing would potentially allow a user to see the calculation of the path in real time. This would allow modification of the A* implementation and heuristic, while also calling attention to potential bugs in the implementation of the octree and graph structure.

VII.1.2 Integrating Chiroptera with Hardware

The ultimate goal of Chiroptera's development is integrating with a physical drone and testing non-camera SLAM in a real wilderness environment. Chiroptera currently mimics a number of MAVLink commands, so rather than producing an

`ActionOutput` that is intended for use by Bat Cave, Chiroptera could directly output MAVLink XML-based messages[15]. When working with custom Actions that return an acceleration vector, an adapter could be developed that would translate the `ActionOutput` to a suitable MAVLink command(s). Serializing Chiroptera's output in a MAVLink compatible message would provide Chiroptera the ability to issue commands directly to a PixHawk-based flight controller.

VII.1.3 Bat Cave Enhancements

Bat Cave currently takes a brute force approach to creating and rendering object displayed in a scene. Inefficiencies in the implementation become apparent in larger scenes containing many environment objects, and by extension, large point clouds, as the frame rate of the simulation drops below 30 frames per second. There may be options available to optimize scene geometry, for example, displaying point cloud data as two-dimensional points projected atop the three-dimensional scene, rather than modeling each point as a three-dimensional object. Instancing larger models such as trees and more complex rock structures may serve to save memory by allowing a single copy of the object geometry to be stored, rather than duplicating the full set of vertices and edges each time an object is added to the scene.

Beyond geometry optimization, Bat Cave's user interface can be enhanced to provide an experience similar to ArduPilot's Mission Planner shown in section II. One approach would be the replacement of text-based telemetry output with animated dials. Likewise, the camera controls can be modified to provide the more familiar pan-tilt-zoom

[15] https://mavlink.io/en/guide/xml_schema.html

capability found in 3D modeling and animation applications and linked to mouse movement to allow a more interactive experience.

The user interface could incorporate an interactive mission planner that allows the user to interactively build Chiroptera flight plans, replacing the process of manually creating a mission text file. This interactive mission builder could provide the ability to update commands and Chiroptera’s flight plan during the mission.

Fleder’s point mass simulation rules are sufficient for modeling the drone’s behavior during point-to-point flights. Implementation of a full rigid body physics system would allow for the simulation of more complex maneuvers and flight dynamics. A rigid body system in which multiple motor force generators are attached to the drone in positions corresponding to the positions of motors on the physical unit. In addition to the net forces currently applied to the drone, the net torque generated by the motor force generators would provide the ability to simulate banking maneuvers, making Bat Cave and Fleder a more realistic toolbox for testing drone flight dynamics.

Appendix 1.

Glossary

Altimeter

A sensor that measures the altitude[16] of the drone above whatever feature lies directly below it.

Altitude above Ground Level (AGL)

The altitude of the drone relative to whatever object is directly below it as measured by a downward facing laser or ultrasonic range finder.

Altitude above Sea Level (ASL)

The altitude of the drone relative to the mean sea level of the world as measured by the altimeter.

A*

A pathfinding algorithm that finds the shortest path through a directed graph.

ArduPilot Copter

A suite of open source tools including hardware drivers, drone controllers, and ground control software for use by autonomous multirotor vehicles.

[16] The different definitions of altitude are consistent with the ArduPilot specification

Biomimicry

A technique for solving technical challenges by applying biologically inspired processes.

Flight Computer

A software based intelligent agent that determines a flight path, plots a trajectory, and controls the drone in flight.

Flight Controller

Hardware and software system responsible for monitoring sensors, motors, and communication, and controlling the motion of the drone in flight.

Global Positioning System (GPS)

A network of satellites in geosynchronous orbit that provide positional information.

Intelligent Agent

An entity that leverages sensor data to collect information about its environment, then uses this information to decide how to interact with the environment.

Inertial Measurement Unit (IMU)

An electronic sensor that leverages a series of gyroscopes and acceleration sensors to determine an object’s orientation.

Light Detection and Ranging (LIDAR)

A technique for determining the distance to an object by targeting the object with a pulse of laser light and measuring the time necessary for the reflected light to be returned to a sensor.

Octree

A hierarchical branching volumetric data structure in which each node has eight children.

Point Cloud

A collection of individual point in three-dimensional space that define the shape of an object or volumetric environment.

PixHawk

An open-hardware platform for autonomous drone flight controllers.

Quadcopter

An aerial drone which flies using a system of four rotors, each controlled by their own individual motor.

Relative Altitude (REL)

The altitude relative the drone’s origin point.

Simultaneous Localization and Mapping (SLAM)

Construct and update a map of an unknown environment while localizing the vehicle's position within the map in real time.

Sparse Voxel Octree (SVO)

An octree that efficiently stores voxel data by only allocating nodes for data as it is needed.

Steering Force

A three-dimensional force vector that parameterizes desired changes in motion.

Target Altitude (TALT)

The desired altitude specified to Chiroptera.

Terrain Altitude (Terrain ALT)

The height above sea level of a natural or artificial terrain feature.

UAV

An uncrewed aerial vehicle capable of controlled flight via a remote ground station, or fully autonomous onboard computer.

Voxel

A data point representing a volume within a grid in three-dimensional space.

Appendix 2.

Chiroptera Actions and MAVLink Commands

Actions

Steering Behaviors

Align: The drone adjusts its yaw until its heading is pointing towards the given coordinate.

Parameter	Description
Latitude	The target latitude. If this is 0, the drone will hold at its current latitude. Floating point.
Longitude	The target longitude. If this is 0, the drone will hold at its current longitude. Floating point.

Arrive: Approach the given coordinates. Slowing down when and come to a complete stop upon arrival.

Parameter	Description
Latitude	The target latitude. If this is 0, the drone will hold at its current latitude. Floating point.
Longitude	The target longitude. If this is 0, the drone will hold at its current longitude. Floating point.
Altitude	Optional. The target altitude. If this is blank, the drone will hold at its current altitude. Floating point.

Seek: Approach the given coordinates but do not come to a stop. Continuously re-align to point at the target position.

Parameter	**Description**
Latitude	The target latitude. If this is 0, the drone will hold at its current latitude. Floating point.
Longitude	The target longitude. If this is 0, the drone will hold at its current longitude. Floating point.
Altitude	Optional. The target altitude. If this is blank, the drone will hold at its current altitude. Floating point.

Delay: Take no action for given number of integer seconds.

Parameter	**Description**
Duration	Duration in seconds to wait. Integer.

Orient: The drone adjusts its yaw until it reaches the given orientation in degrees.

Parameter	**Description**
Orientation	The target orientation in +/- degrees. Floating point.

Pathfind: Use A* to navigate the environment to the given coordinates. Re-calculate a path if newly discovered obstacles are found to block the route.

Parameter	**Description**
Latitude	The target latitude. If this is 0, the drone will hold at its current latitude. Floating point.
Longitude	The target longitude. If this is 0, the drone will hold at its current longitude. Floating point.
Altitude	Optional. The target altitude. If this is blank, the drone will hold at its current altitude. Floating point.

Commands

As described in Chapter 3, Chiroptera implements a subset of the MAVLink command protocol[17]. The following MAVLink commands are currently supported.

Navigation Commands

Navigation commands are used to specify the movement of the drone.

MAV_CMD_NAV_TAKEOFF. The drone climbs straight up to the specified altitude. If this command is invoked when the drone is already in flight, the drone will ascend to the newly specified altitude above ground level (AGL).

Parameter	**Description**
Altitude	The desired AGL in meters. Floating point.

MAV_CMD_NAV_DELAY. The drone will remain stationary at its current position until the specified number of seconds has elapsed.

Parameter	**Description**
Time	Delay in seconds expressed as an integer

MAV_CMD_NAV_ALIGN. The drone will adjust its yaw until its heading points to the latitude and longitude specified in floating point representation.

Parameter	**Description**
Latitude	The target latitude. If this is 0, the drone will hold at its current latitude. Floating point.

[17] https://mavlink.io/en/services/command.html

Longitude	The target longitude. If this is 0, the drone will hold at its current longitude. Floating point.

MAV_CMD_NAV_WAYPOINT. Composition of the `Align` and `Arrive` actions. Align and navigate in a straight line to the specified latitude, longitude, and altitude above ground level (AGL). The drone will hold position for the specified delay time before proceeding to the next command.

Parameter	Description
Delay	Hold time at waypoint in seconds expressed as an integer
Lat	The target latitude. If this is 0, the drone will hold at its current latitude. Floating point.
Lon	The target longitude. If this is 0, the drone will hold at its current longitude. Floating point.
Alt	The target AGL. If this is 0, the drone will maintain its current AGL. Floating point.

MAV_CMD_NAV_LAND. The drone will land at either its current position or at the specified latitude and/or longitude coordinates.

Parameter	Description
Lat	The target latitude. If this is 0, the drone will land at its current latitude.
Lon	The target longitude. If this is 0, the drone will land at its current longitude.

MAV_CMD_NAV_RETURN_TO_LAUNCH. The drone will return to its origin point and then land. This command does not take any parameters.

Parameter	Description

-	-

DO Commands

Do commands are executed immediately and perform an action that changes the drone's flight state without changing the motion of the drone.

MAV_CMD_DO_CHANGE_SPEED. Set the maximum speed of the drone.

Parameter	**Description**
Type	0: ground speed, 1: climb rate, 2: descent rate
Speed	The target speed in meters / second. Floating point.

MAV_CMD_DO_TERRAIN_FOLLOW. The drone will climb or descend as necessary to maintain an AGL consistent with the altitude specified by the MAV_CMD_NAV_TAKEOFF command.

Parameter	**Description**
Enable	Enable (1) or disable (0) terrain following. Terrain following is off by default.

Conditional Commands

MAV_CMD_CONDITION_YAW. Set a new yaw value for the drone.

Parameter	**Description**
Degrees	+/- 0-360
Speed	Yaw adjustment speed in degrees / second. Integer.

Appendix 3.

Bat Cave Scene Description Language

Bat Cave scenes are stored in newline delimited ASCII text files with the extension `.scn`. Lines beginning with a `#` are treated as comments and ignored by the parser.

Primitives

Ground

A flat plane generally used to model flat ground or the surface of water.

Parameter	**Description**
X	The size of the plane in the X-dimension. Floating point.
Y	The size of the plane in the Y-dimension. Floating point.

Rock

A rock or boulder than can be placed above ground. Currently modeled as a box.

Parameter	**Description**
X	The bounds of the rock in the X-dimension. Floating point.
Y	The bounds of the rock in the Y-dimension. Floating point.
Z	The bound of the rock in the Z-dimension. Floating point.

OakTree

A tree with a large canopy and several branches.

Parameter	Description
-	-

PineTree

A tree with a narrow canopy and no branches.

Parameter	Description
-	-

Transformations

Translate

Translates the preceding entity to the given coordinates relative to the entity's center of geometry.

Parameter	Description
X	The new coordinate in the X-dimension. Floating point.
Y	The new coordinate in the Y-dimension. Floating point.
Z	The new coordinate in the Z-dimension. Floating point.

Rotate

Rotates the preceding entity by the given degrees around the entity's center of geometry.

Parameter	Description
X	The rotation in degrees in the X-dimension. Floating point.
Y	The rotation in degrees in the Y-dimension. Floating point.
Z	The rotation in degrees in the Z-dimension. Floating point.

Scale

Scales the preceding entity by the given scale factors relative to the entity's center of geometry.

Parameter	Description
X	The scale factor in the X-dimension. Floating point.
Y	The scale factor in the Y-dimension. Floating point.
Z	The scale factor in the Z-dimension. Floating point.

Attributes

Color

Assigns the given red, green, and blue color values to the preceding entity's fragment shader.

Parameter	Description
R	The value of the Red color channel. Integer 0-255.
G	The value of the Green color channel. Integer 0-255.
B	The value of the Blue color channel. Integer 0-255.

Appendix 4.

Selected Source Code

Full source code is available in a private repository on GitHub[18]. Repository access is available upon email request[19].

Appendix 4.1 Octree

```
/***********************************************************************
 * Environment stores general Chiroptera configuration settings.
 * Currently used only to define the bounds and resolution of the
 * octree
 **********************************************************************/
#ifndef ENVIRONMENT_H
#define ENVIRONMENT_H

#define SVO_BOTTOM_LEFT Point3{-25.f, -1.f, -.5f}
#define SVO_TOP_RIGHT Point3{25.f, 49.f, 49.5f}
#define SVO_RESOLUTION 6

#endif

/**
 *  Connection stores a directed connection and associated cost
 *  between two nodes
 */
#ifndef SVO_CONNECTION_H
#define SVO_CONNECTION_H

class Connection {
    void *from;
    void *to;
    int cost;

public:
    Connection(void *from, void *to, int cost) :
        from{from}, to{to}, cost{cost} { }

    void* GetFromNode() { return this->from; }
```

[18] https://www.github.com/alexshopov
[19] Email: shopov.alex@gmail.com

```
    void* GetToNode() { return this->to; }
    float Cost() { return this->cost; }
};


#endif

/**
 *  BoundingBox stores the minimum and maximum bounds of a region
 *  and a map of vertices to region faces
 */

#ifndef SVO_BOUNDING_BOX_H
#define SVO_BOUNDING_BOX_H

#include "../vector.h"
typedef struct Vector3 Point3;

struct BoundingBox {
    /**
     *
     *   5---7
     *  /   /|
     * 4---6 |
     * |   | 3
     * |   |/
     * 0---2
     */
    Point3 verts[8];

   // faces mappings. We'll use this to find sibling nodes
   int faces[6][4] = {
       {0, 1, 5, 4}, // left
       {2, 3, 7, 6}, // right
       {4, 5, 7, 6}, // top
       {0, 1, 3, 2}, // bottom
       {2, 0, 4, 6}, // front
       {3, 1, 5, 7}, // back
    };

    // min = bottom-left-front, max = top-right-back
    BoundingBox(Point3 min, Point3 max) {
        verts[0] = min;
        verts[1] = {min.x, max.y, min.z};
        verts[2] = {max.x, min.y, min.z};
        verts[3] = {max.x, max.y, min.z};

        verts[4] = {min.x, min.y, max.z};
        verts[5] = {min.x, max.y, max.z};
        verts[6] = {max.x, min.y, max.z};
        verts[7] = max;
    }
};

#endif

/*************************************************************************
 *  The octree is a hierarchical data structure consisting of a parent node
 *  and 8 child nodes, each representing a region in space defined by a
 *  minimum and maximum vertex
 ************************************************************************/

#ifndef SVO_OCTREE_H
#define SVO_OCTREE_H

#include <vector>
#include "connection.h"
#include "bounding-box.h"
#include "../env.h"
#include "../vector.h"

/**
 * The cost to travel from one node to another based on the type of connection
 * between the nodes
 */
#define COST_SHARED_FACE   10
#define COST_SHARED_EDGE   14
#define COST_SHARED_VERT   16

enum NodeType {
    REGION, EMPTY, FILLED
};

enum NodePosition {
    BottomLeftFront = 0,
    BottomLeftBack,
    BottomRightFront,
    BottomRightBack,
    TopLeftFront,
    TopLeftBack,
    TopRightFront,
    TopRightBack
};

/**
 * A map of vertices, faces, and edges shared between a node and its neighbors
 */
struct ConnectionMap {
    int verts[7];
    int faces[3];
    int edges[3];
    int corners;
};

class Octree {
    int resolution = SVO_RESOLUTION;
    unsigned int level = 0;
    Point3 min,   // bottom-left-front
           midpt, // center
           max;   // top-right-back
    Octree *parent;
    Octree *children[8] {};

    Point3 findMidpt();
    bool containsPoint(Point3 p);

    // graph construction
    void setConnections();
    bool findSharedVertices(Octree *node,
                            Point3 p,
                            std::vector<Octree*> &shared);
    void sharedFaces(int faces[], std::vector<Octree*> &sharedVerts);
    void sharedEdges(int edges[], std::vector<Octree*> &sharedEdges);

public:
    unsigned int id;
    NodeType nodeType = REGION;
    Position position;

    std::vector<Connection*> connections;
    int nodeIndex;
    float cost;
    float estimatedCost;
    BoundingBox *boundingBox;

    Octree(Octree *parent, Position pos, Point3 minpt, Point3 maxpt);
    ~Octree();

    void Subdivide();
    Octree* GetChild(int idx);
    void InsertPoint(Point3 p);
    Octree* Find(Point3 p);
    void ConnectionTo(Octree *to, int cost);
    Point3 GetMidPt() { return midpt; }
```

```
};

#endif

/**
 * octree.cpp
 */
#include "octree.h"

/**
 * Predefine the face, edge, and vertex connections for each of the 8
 * nodes in the octree
 */
ConnectionMap neighbors[8] = {
    {
        {0, 1, 2, 3, 4, 5, 6},
        {1, 2, 4},
        {3, 5, 6},
        7
    },
    {
        {1, 0, 2, 3, 4, 5, 7},
        {0, 3, 5},
        {2, 4, 7},
        6
    },
    {
        {2, 1, 0, 3, 4, 6, 7},
        {0, 3, 6},
        {1, 4, 7},
        5
    },
    {
        {3, 1, 2, 0, 5, 6, 7},
        {1, 2, 7},
        {0, 5, 6},
        4
    },
    {
        {4, 1, 2, 0, 5, 6, 7},
        {5, 6, 0},
        {7, 1, 2},
        3
    },
    {
        {5, 1, 3, 4, 0, 6, 7},
        {4, 7, 1},
        {6, 0, 3},
        2
    },
    {
        {6, 2, 3, 4, 5, 0, 7},
        {4, 7, 2},
        {5, 0, 3},
        1
    },
    {
        {7, 2, 3, 4, 5, 6, 1},
        {5, 6, 3},
        {4, 1, 2},
        0
    },
};

/**
 * Constructor
 */
Octree::Octree(Octree *parent, Position pos, Point3 minpt, Point3 maxpt) :
    position{pos}, parent{parent}, min{minpt}, max{maxpt}
{
    if (this->parent)
        this->level = parent->level + 1;

    this->boundingBox = new BoundingBox(min, max);

    if (level == this->resolution)
        this->nodeType = EMPTY;

    this->midpt = this->findMidpt();
}

/**
 * Destructor
 */
Octree::~Octree() {
    if (this->children[0]) {
        for (Octree *child : this->children) {
            delete child;
        }
    }
}

/**
 * Subdivide the octree into eight child nodes
 */
```

```
void Octree::Subdivide() {
    // bottom layer
    this->children[BottomLeftFront] =
        new Octree(this,
                   BottomLeftFront,
                   this->min,
                   this->midpt);

    this->children[BottomLeftBack] =
        new Octree(this, BottomLeftBack,
                   {this->min.x, this->midpt.y, this->min.z},
                   {this->midpt.x, this->max.y, this->midpt.z});

    this->children[BottomRightFront] =
        new Octree(this,
                   BottomRightFront,
                   {this->midpt.x, this->min.y, this->min.z},
                   {this->max.x, this->midpt.y, this->midpt.z});

    this->children[BottomRightBack] =
        new Octree(this,
                   BottomRightBack,
                   {this->midpt.x, this->midpt.y, this->min.z},
                   {this->max.x, this->max.y, this->midpt.z});

    // top layer
    this->children[TopLeftFront] =
        new Octree(this,
                   TopLeftFront,
                   {this->min.x, this->min.y, this->midpt.z},
                   {this->midpt.x, this->midpt.y, this->max.z});

    this->children[TopLeftBack] =
        new Octree(this,
                   TopLeftBack,
                   {this->min.x, this->midpt.y, this->midpt.z},
                   {this->midpt.x, this->max.y, this->max.z});

    this->children[TopRightFront] =
        new Octree(this,
                   TopRightFront,
                   {this->midpt.x, this->min.y, this->midpt.z},
                   {this->max.x, this->midpt.y, this->max.z});

    this->children[TopRightBack] =
        new Octree(this,
                   TopRightBack,
                   this->midpt,
                    this->max);

    for (auto *c : this->children) {
        c->setConnections();
    }
}

/**
 * Define a new connection from *this to another node
 */
void Octree::ConnectionTo(Octree *to, int cost) {
    // only connect nodes on the same level until we
    // work out SVO navigation later
    if (to->level != this->level) return;

    this->connections.push_back(new Connection{(void*)this, (void*)to, cost});
}

/**
 * Find all octrees that share a common vertex with *this
 */

bool Octree::findSharedVertices(Octree *node, Point3 p,
                                std::vector<Octree*> &shared) {
    for (Octree *child : node->children) {
          if (child && child->containsPoint(p)) {
              if (child->nodeType == REGION) {
                   this->findSharedVertices(child, p, shared);
              } else {
                  if (child != this) {
                      shared.push_back(child);
               }
            }
        }
    }
}

/**
 * Find all octrees that share a common face with *this
 */
void Octree::sharedFaces(int faces[], std::vector<Octree*> &sharedVerts) {
    for (int i = 0; i < 3; ++i) {
        this->ConnectionTo(this->parent->children[faces[i]],
                           COST_SHARED_FACE);
    }

    for (auto *s : sharedVerts) {
        Position pos = s->position;
        if (pos == faces[0] || pos == faces[1] || pos == faces[2]) {
            this->ConnectionTo(s, COST_SHARED_FACE);
            s->ConnectionTo(this, COST_SHARED_FACE);
        }
    }
}

/**
 * Find all octrees that share a common edge with *this
 */
void Octree::sharedEdges(int edges[], std::vector<Octree*> &sharedEdges) {
    for (int i = 0; i < 3; ++i) {
        this->ConnectionTo(this->parent->children[edges[i]],
                           COST_SHARED_EDGE);
    }

    for (auto *s : sharedEdges) {
        Position pos = s->position;

        if (pos == edges[0] || pos == edges[1] || pos == edges[2]) {
            this->ConnectionTo(s, COST_SHARED_EDGE);
            s->ConnectionTo(this, COST_SHARED_EDGE);
        }
    }
}

/**
 * Define the graph connections between this octree and its
 * neighbors/parents/children
 */
void Octree::setConnections() {
    Octree *root = this;
    while (root->parent) {
        root = root->parent;
    }

    ConnectionMap *cm = &neighbors[this->position];

    std::vector<Octree*> shared;
    this->findSharedVertices(root,
                         this->boundingBox->verts[cm->verts[0]], shared);
    this->sharedFaces(cm->faces, shared);

    for (int i = 1; i < 7; ++i) {
        this->findSharedVertices(root,
                             this->boundingBox->verts[cm->verts[i]],
                             shared);
    }
    this->sharedEdges(cm->edges, shared);
}

/**
 * Insert a point by recursively traversing the octree down to the
 * leaf layer
 */
void Octree::InsertPoint(Point3 p) {
    if (this->containsPoint(p)) {
        if (this->nodeType == FILLED) return;

        if (this->nodeType == REGION) {
            if (!this->children[0])
                this->Subdivide();

            for (Octree *child : this->children) {
                if (child) child->InsertPoint(p);
            }
        } else {
            this->nodeType = FILLED;
        }
    }
}

/**
 * Return the child node at the given index
 */
Octree* Octree::GetChild(int idx) {
    if (idx < 0 || idx > 8) return nullptr;

    return this->children[idx];
}

/**
 * Find the geometric midpoint of the octree
 */
Point3 Octree::findMidpt() {
    return (this->min + this->max) / 2.f;
}

/**
 * Determine if the octree contains the given point
 */
bool Octree::containsPoint(Point3 p) {
    return p.x >= this->min.x && p.x <= this->max.x &&
           p.y >= this->min.y && p.y <= this->max.y &&
           p.z >= this->min.z && p.z <= this->max.z;
```

```
}

/**
 * Find the region containing a given point by recursively traversing the
 * octree down to the leaf layer
 */
Octree* Octree::Find(Point3 p) {
    Octree *node = nullptr;

    if (this->containsPoint(p)) {
        node = this;

        if (this->nodeType == REGION) {
            for (Octree *child : this->children) {
                Octree *tmp = child->Find(p);
                if (tmp != nullptr) {
                    node = tmp;
                    break;
                }
            }
        }
    }

    return node;
}
```

Appendix 4.2 Pathfind

```
/*********************************************************************
 * Action is a header-only class that is extended into specific
 * action behaviors.
 * Inspired by the invoked action approach taken by Ian Millington in
 * Artificial Intelligence for Games (2009)
 *********************************************************************/
#ifndef ACTION_BASE_CLASS_H
#define ACTION_BASE_CLASS_H

#include <iostream>
#include "action-utils.h"
#include "../vector.h"
#include "../telemetry/telemetry.h"

typedef struct Vector3 Point3;
typedef struct Vector3 Euler;

struct ActionOutput {
    std::string action = "...";
    Vector3 linearAcceleration;
    Euler angularAcceleration;
    Point3 node{-1.f,-1.f,-1.f};
    Point3 target{-1.f,-1.f,-1.f};
};

class Action {
    bool isComplete = false;

protected:
    Kinematic drone;
    Kinematic baseTarget; // the ultimate target for this action
    Kinematic target;     // temporary target used when avoiding an obstacle

    DroneTelemetry *telemetry = nullptr;

    void SetIsComplete(bool isComplete) { this->isComplete = isComplete; }
    void SetNewTarget(Kinematic newTarget) { this->target = newTarget; }

public:
    Action *next;

    Action() = default;
    virtual ActionOutput Execute(DroneTelemetry *droneTelemetry) = 0;
    virtual void Init(DroneTelemetry *droneTelemetry) { isComplete = false; }
    bool IsComplete() { return isComplete; };
};

#endif

/************************************************************************
 * Align the drone to a given set of { x, y } coordinates
 ************************************************************************/

#ifndef ACTION_ALIGN_H
#define ACTION_ALIGN_H

#include "action.h"

class Align : public Action {
    bool isSetHeading;
    float maxAngularAcceleration = 3.f;
    float maxAngularSpeed = 0.5f;

    float targetRadius = 0.25;
    float slowRadius = 2.0f;

public:
    Align(float latitude, float longitude) {
        this->target.position.x = latitude;
        this->target.position.y = longitude;
        this->isSetHeading = true;
    }

    void Init(DroneTelemetry *telemetry);
    ActionOutput Execute(DroneTelemetry *telemetry);
};

#endif

#include <iostream>
#include <cmath>
#include "align.h"

void Align::Init(DroneTelemetry *telemetry) {
    Action::Init(telemetry);

    this->targetRadius = 0.1f;
    this->maxAngularSpeed = 90;
    this->maxAngularAcceleration = 1.f;
    this->slowRadius = 45;

    this->drone.position = telemetry->GetPosition();
    this->drone.orientation = telemetry->GetOrientation();

    Vector3 nCurrent = this->drone.position;
    Vector3 nTarget = this->target.position - nCurrent;
    float theta = atan2(nTarget.y, nTarget.x);
    float nOrientation = (theta * 180 / M_PI);

    this->target.orientation.z = nOrientation;
}

ActionOutput Align::Execute(DroneTelemetry *telemetry) {
    ActionOutput out;
    out.action = "MAV_CMD_CONDITION_YAW";

    this->drone.orientation = telemetry->GetOrientation();
    float rot = this->target.orientation.z - this->drone.orientation.z;
    if (abs(rot) < targetRadius) {
        this->SetIsComplete(true);
        return out;
    }

    float targetSpeed;
    if (rot > this->slowRadius) {
        targetSpeed = this->maxAngularSpeed;
    } else {
        targetSpeed = this->maxAngularSpeed * rot / this->slowRadius;
    }

    out.angularAcceleration.z = targetSpeed;

    return out;
}

/************************************************************************
 * Travel to a given set of coordinates. Slow down and come to a stop at
 * the target destination
 ************************************************************************/
#ifndef ACTION_ARRIVE_H
#define ACTION_ ARRIVE _H

#include "action.h"

class Arrive: public Action {
    float maxSpeed;
    float maxAcceleration;
    float timeToTarget;
    float targetRadius;
    float slowRadius;

public:
    Arrive(float latitude, float longitude, float altitude);
    void Init(DroneTelemetry *telemetry);
    ActionOutput Execute(DroneTelemetry *telemetry);
};

#endif

/************************************************************************
 * Waypoint is a composition of Align and Arrive. Align to the given point
 * then travel to the target position, coming to a complete step when the
 * drone arrives
 ************************************************************************/

#ifndef ACTION_WAYPOINT_H
#define ACTION_WAYPOINT_H

#include "action.h"
#include "align.h"
#include "arrive.h"
#include "../svo/octree.h"
```

```
class Waypoint : public Action {
    Align *align;
    Arrive *arrive;

    bool isAligning = true;

public:
    Octree *node;

    Waypoint(int delay, float latitude, float longitude, float altitude);
    Waypoint(int delay, float latitude,
             float longitude, float altitude, Octree *node);
    void Init(DroneTelemetry *telemetry);
    ActionOutput Execute(DroneTelemetry *telemetry);
};

#endif

/*************************************************************************
 * Heuristic used by the A* algorithm. Returns a cost equal to the
 * geometric distance between the current and target nodes with an
 * additional offset added to represent any required change in altitude. This
 * offset is intended to bias the A* search in favor of nodes that are at the
 * drone’s current altitude.
 ************************************************************************/

#ifndef SVO_HEURISTIC_H
#define SVO_HEURISTIC_H

#include "./octree.h"

class Heuristic {
    Octree *goal;

public:
    Heuristic(Octree *goal) : goal{goal} { }

    float EstimateFrom(Octree *from) {
          Point3 a = from->GetMidPt();
          Point3 b = this->goal->GetMidPt();

          float deltaZ = abs(b.z - a.z);

          return (b - a).Magnitude() + deltaZ;
    }
};

#endif
```

```
/***********************************************************************
 * PathfindList is a helper class that contains a list of nodes the need
 * to be search or have already been search by the Pathfind A* implementation.
 * The use of a dedicated pathfind list class was adapted from an approach
 * described in Millington’s Artificial Intelligence for Games (2009)
 **********************************************************************/

#ifndef ACTION_PATHFIND_LIST_H
#define ACTION_PATHFIND_LIST_H

#include <vector>
#include "../svo/octree.h"
#include "../svo/node.h"
#include "../svo/heuristic.h"

class PathfindList {
    std::vector<Node*> entries;
public:
    inline void operator +=(Node *node) {
        this->entries.push_back(node);
    }
    inline void operator -=(Node *node) {
        std::vector<Node*>::iterator it;
        it = find(entries.begin(), entries.end(), node);
        this->entries.erase(it);
    }
    int Size();

    Node* ShortestEstimated();
    bool Contains(Octree *node);
    Node* Find(Octree *node);
};

#endif

/**
 * pathfind-list.cpp
 */
#include "pathfind-list.h"

int PathfindList::Size() {
    return this->entries.size();
}

Node* PathfindList::ShortestEstimated() {
    float shortest = MAXFLOAT;
    Node* out = nullptr;
    for (auto *n : this->entries) {
        if (n->estimatedTotalCost < shortest) {
            out = n;
            shortest = n->estimatedTotalCost;
        }
    }

    return out;
}

bool PathfindList::Contains(Octree *node) {
    for (auto *n : this->entries) {
        if (n->node == node) return true;
    }

    return false;
}

Node* PathfindList::Find(Octree *node) {
    for (auto *n : this->entries) {
        if (n->node == node) return n;
    }

    return nullptr;
}

/************************************************************************
 * Pathfind. Perform an A* search of a directed graph and return a set of
 * nodes representing a path from an origin to destination node.
 *
 * The original A* algorithm was described by Hart, et al in A Formal Basis
 * for the Heuristic Determination of Minimum Cost Paths (1968).
 * This implementation is based on that paper as well as contributions by
 * Millington (Artificial Intelligence for Games, 2007)
 ************************************************************************/

#ifndef ACTION_PATHFIND_H
#define ACTION_PATHFIND_H

#include <vector>
#include "action.h"
#include "waypoint.h"
#include "pathfind-list.h"
#include "../svo/heuristic.h"

struct NodeRecord {
    Octree *node;
    Connection *connection;
    float costSoFar;
    float estimatedTotalCost;
};

class Pathfind : public Action {
    Heuristic *heuristic;
    Node *startNode;

    std::vector<Waypoint*> waypoints;
    Waypoint *currentWaypoint = nullptr;
    Waypoint* nextWaypoint();

public:
    Pathfind(float latitude, float longitude, float altitude);
    void Init(DroneState *state);
    ActionOutput Execute(DroneState *state);
};

#endif

/**
 * pathfind.cpp
 */
#include "pathfind.h"

Pathfind::Pathfind(float latitude, float longitude, float altitude) {
    this->target.position.x = this->baseTarget.position.x = latitude;
    this->target.position.y = this->baseTarget.position.y = longitude;
    this->target.position.z = this->baseTarget.position.z = altitude;
}

void Pathfind::Init(DroneState *state) {
    Action::Init(state);

    this->waypoints.clear();
    this->currentWaypoint = nullptr;

    state->targetPosition = this->target.position;

    Octree *start = state->graph->Find(state->GetPosition());
    Octree *end = state->graph->Find(this->target.position);
    this->heuristic = new Heuristic(end);

    start->GetMidPt().Println();
    end->GetMidPt().Println();

    this->startNode = new Node{};
```

```
this->startNode->node = start;
this->startNode->connection = nullptr;
this->startNode->costSoFar = 0.f;
this->startNode->estimatedTotalCost = heuristic->EstimateFrom(start);

PathfindList open;
PathfindList closed;

open += this->startNode;

Node *current;
// iterate through nodes
while (open.Size()) {     // find the smallest element in the open list
    current = open.ShortestEstimated();

    // if current is the goal, we're done
    if (current->node == end) {
        break;
    }

    // pop current off open and add to closed
    open -= current;
    closed += current;

    // loop through outgoing connections
    for (auto *connection : current->node->connections) {
        if (!connection)
            continue;

        // get cost estimate to end node
        Octree *endNode = (Octree*)connection->GetToNode();
        if (!endNode)
            continue;

        // if endNode is blocked, move on to the next one
        if (endNode->nodeType == FILLED)
            continue;

        float endNodeCost = current->costSoFar + connection->Cost();

        Node *endNodeRecord = nullptr;
        float endNodeHeuristic = 0.f;

        // if closed contains end-node
        if ((endNodeRecord = closed.Find(endNode))) {
            if (endNodeRecord->costSoFar <= endNodeCost)
                continue;
            closed -= endNodeRecord;
            endNodeHeuristic =
                endNodeRecord->estimatedTotalCost –
                    endNodeRecord->costSoFar;
        } else if ((endNodeRecord = open.Find(endNode))) {
            if (endNodeRecord->costSoFar <= endNodeCost)
                continue;

            endNodeHeuristic =
                endNodeRecord->estimatedTotalCost –
                    endNodeRecord->costSoFar;
        } else { // unvisited node, add to list
            endNodeRecord = new Node{};
            endNodeRecord->node = endNode;
            endNodeHeuristic = heuristic->EstimateFrom(endNode);
        }

        // we need to update the node
        endNodeRecord->costSoFar = endNodeCost;
        endNodeRecord->fromNode = current;
        endNodeRecord->connection = connection;
        endNodeRecord->estimatedTotalCost =
           endNodeCost + endNodeHeuristic;

        // throw it on the pile
        if (!open.Contains(endNode)) {
            open += endNodeRecord;
        }
    }
}

// we’ll return an error but how else can we gracefully handle not
// finding a path to the target node?
if (current->node != end) {
    std::cout << "No Path Found" << std::endl;
    return;
} else {
    std::cout << "Found Path" << std::endl;
}

// nodes were added from end -> start so now we need to reverse the order
// of the list
// add current to path
Octree *from = current->node;
Point3 pt = from->GetMidPt();
this->waypoints.push_back(new Waypoint{0, pt.x, pt.y, pt.z, from});

while (current->fromNode->node != start) {
        from = (Octree*)(current->connection->GetFromNode());
        pt = from->GetMidPt();
        this->waypoints.push_back(new Waypoint{0, pt.x, pt.y, pt.z, from});
        current = current->fromNode;
    }
}

Waypoint* Pathfind::nextWaypoint() {
    auto *next = this->waypoints.back();
    this->waypoints.pop_back();
    return next;
}

ActionOutput Pathfind::Execute(DroneState *state) {
    ActionOutput out;
    out.action = "MAV_CMD_NAV_PATHFIND";

    if (this->currentWaypoint == nullptr) {
        if (this->waypoints.size() == 0) {
            this->SetIsComplete(true);
            return out;
        }

        this->currentWaypoint = this->nextWaypoint();
        this->currentWaypoint->Init(state);
    }

    if (this->currentWaypoint->IsComplete()) {
        if (this->waypoints.size() == 0) {
            this->SetIsComplete(true);
            return out;
        }

        auto prevWaypoint = this->currentWaypoint;

        this->currentWaypoint = this->nextWaypoint();
        this->currentWaypoint->node->GetMidPt().Println();
        this->currentWaypoint->node->PrintId();

        /**
         * the next node has become occupied since we first scanned it.
         * Reinitialize the action and conduct a search for a new path.
         */
        if (this->currentWaypoint->node->nodeType == FILLED) {
            this->Init(state);
            return out;
        }
        this->currentWaypoint->Init(state);
    }

    out = this->currentWaypoint->Execute(state);
    return out;
}
```

Appendix 4.3 Line Collider

```
/************************************************************************
 * The LineTriangle Collider is based on the line-triangle intersection
 * algorithm described by Ericson in Real Time Collision Detection,
 * Chapter 5.3. Ericson’s algorithm is invoked as part of a larger routine
 * that tests a set of triangles and in the event of multiple collisions
 * returns the triangle closest to the origin point of the line.
 ***********************************************************************/

#ifndef LINE_TRIANGLE_COLLIDER_H
#define LINE_TRIANGLE_COLLIDER_H

#include <vector>
#include "vector.h"

namespace fleder {

typedef Vector3 Point3;

/**
 * Point-normal representation of the plane that encapsulates a triangle
 */
struct Plane {
    float d;
    Vector3 n; // normal

    Plane() = default;
    Plane(Point3 a, Point3 b, Point3 c) {
        Vector3 v0, v1;
       v0 = b - a;
       v1 = c - a;

       n = v0.Cross(v1);
       n = n.Normalize();

       d = n.Dot(a);
    }
};

/**
 * 3-Vertex plus normal representation of a triangle
 */
struct Triangle {
    Point3 a, b, c;
    Vector3 n; // normal
    Plane plane;

    Triangle(Point3 a, Point3 b, Point3 c) : a{a}, b{b}, c{c} {
        plane = Plane{a, b, c};
        n = plane.n;
    }
};

} // end namespace

class LineCollider {
    std::vector<Triangle> entries;
    bool intersectLinePlane(Point3 a, Point3 b, Triangle tri, Point3 &r);

public:
    bool DoesIntersect(Point3 a, Point3 b, Point3 &r, Vector3 &n);
    void AddTriangle(Point3 a, Point3 b, Point3 c);
};


#endif

/**
 * line-collider.cpp
 */
#include "line-collider.h"

using namespace fleder;

float scalar_triple(Vector3 a, Vector3 b, Vector3 c) {
    return (a.Cross(b)).Dot(c);
}

void LineCollider::AddTriangle(Point3 a, Point3 b, Point3 c) {
    this->entries.emplace_back(a, b, c);
}

/**
 * Given two point, find and return the closest intersecting triangle
 */
```

```
bool LineCollider::DoesIntersect(Point3 a, Point3 b, Point3 &r, Vector3 &n) {
    bool intersects = false;
    float mag = MAXFLOAT;

    for (Triangle tri : this->entries) {
        Point3 pt;
        if (this->intersectLinePlane(a, b, tri, pt)) {
            intersects = true;

            Vector3 vr = pt - a;
            float vrmag = vr.Magnitude();
            if (vrmag < mag) {
                mag = vrmag;
                r = pt;
                n = tri.plane.n;
            }
        }
    }

    return intersects;
}

/**
 * Implementation of Ericson’s line-plane collision detector
 */
bool LineCollider::intersectLinePlane(Point3 a, Point3 b,
                                      Triangle tri, Point3 &r) {
    Vector3 ab = b - a;
    float t = (tri.plane.d - tri.plane.n.Dot(a)) / tri.plane.n.Dot(ab);

    if (t >= 0.0f && t <= 1.0f) {
        // check if point is inside triangle
        Vector3 pa = tri.a - a;
        Vector3 pb = tri.b - a;
        Vector3 pc = tri.c - a;

        float u = scalar_triple(ab, pc, pb);
        if (u < 0.0f) return false;

        float v = scalar_triple(ab, pa, pc);
        if (v < 0.0f) return false;

        float w = scalar_triple(ab, pb, pa);
        if (w < 0.0f) return false;

        r = (ab * t) + a;
        return true;
    }
    return false;
}
```

References

Afghah, F., Razi, A., Chakareski, J., & Ashdown, J. (2019). Wildfire Monitoring in Remote Areas using Autonomous Unmanned Aerial Vehicles. In arxiv.org.

Ancin-Murguzur, F. J., Munoz, L., Monz, C., & Hausner, V. H. (2019). Drones as a tool to monitor human impacts and vegetation changes in parks and Protected Areas. *Remote Sensing in Ecology and Conservation, 6*(1), 105-113.

ArduPilot Dev Team (2024), ArduPilot Copter, ArduPilot. https://ardupilot.org/copter/index.html

Babcock, Judson T (2023). System Identification of an S500 Quadrotor UAV. Department of Aeronautics, Defense Technical Information Center

Back, S., Cho, G., Oh, J., Tran, X., & Oh, H. (2020). Autonomous UAV trail navigation with obstacle avoidance using Deep Neural Networks. *Journal of Intelligent & Robotic Systems, 100*(3-4), 1195-1211.

Bouziani, M., Amraoui, M., & Kellouch, S. (2021). Comparison assessment of Digital 3D models obtained by drone-based LIDAR and drone imagery. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XLVI-4/W5-2021*, 113-118.

Jun Cheng, Liyan Zhang, Qihong Chen, Xinrong Hu, and Jingcao Cai. 2022. A review of visual SLAM methods for autonomous driving vehicles. Eng. Appl. Artif. Intell. 114, C (Sep 2022).

Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik*. **1**: 269-271.

H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part I," in IEEE Robotics & Automation Magazine, vol. 13, no. 2, pp. 99-110, June 2006

M. Elhousni and X. Huang, "A Survey on 3D LIDAR Localization for Autonomous Vehicles," 2020 IEEE Intelligent Vehicles Symposium (IV), Las Vegas, NV, USA, 2020, pp. 1879-1884

Elmokadem, T., & Savkin, A. V. (2021). A method for autonomous collision-free navigation of a quadrotor UAV in unknown tunnel-like environments. *Robotica, 40*(4), 835-861

Ericson, C. (2005). Real-Time Collision Detection (1st ed.). CRC Press.

Genzel, D., Yovel, Y., Yartsev, M. (2018). Neuroethology of Bat Navigation. *Current Biology*, 28, R997-R1004

Giusti, A., Guzzi, J., Ciresan, D. C., He, F., Rodriguez, J. P., Fontana, F., . . . Gambardella, L. M. (2016). A machine learning approach to visual perception of forest trails for Mobile Robots. *IEEE Robotics and Automation Letters*, *1*(2), 661-667.

Hart, P. E., Nilsson, N. J. & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4, 100-107.

L. Huang, "Review on LIDAR-based SLAM Techniques," 2021 International Conference on Signal Processing and Machine Learning (CONF-SPML), Stanford, CA, USA, 2021, pp. 163-168.

Juniper, A. (2018). *The Complete Guide to Drones: Choose + build + photograph + race*. New York, NY: Wellfleet Press.

Kuipers, B., Feigenbaum, E. A., Hart, P. E., & Nilsson, N. J. (2017). Shakey: From Conception to History. AI Magazine, 38(1), 88-103.

Lee, S. Y., Du, C., Chen, Z., Wu, H., Guan, K., Liu, Y., . . . Liao, W. (2020). Assessing safety and suitability of old trails for hiking using ground and drone surveys. *ISPRS International Journal of Geo-Information, 9*(4), 221.

McRae, J.N.; Gay, C.J.; Nielsen, B.M.; Hunt, A.P. (2019) Using an Unmanned Aircraft System (Drone) to Conduct a Complex High Altitude Search and Rescue Operation: A Case Study. Wilderness Environ. Med., 30, 287–290.

Millington, I. (2007). Game Physics Engine Development (1st ed.). CRC Press.

Millington, I & Funge, J. (2009). Artificial Intelligence for Games (2nd ed.). CRC Press.

Namdari, M. H., Hejazi, S. R., and Palhang, M. (2015). Mcpn, octree neighbor finding during tree model construction using parental neighboring rule. 3D Research, 6(3):1–15.

Payeur, P. (2006). A computational technique for free space localization in 3- d multiresolution probabilistic environment models. Instrumentation and Measurement, IEEE Transactions on, 55(5):1734–1746.

Petrlik, M., Krajnik, T., & Saska, M. (2021). LIDAR-based stabilization, navigation and localization for uavs operating in dark indoor environments. *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*.

Rabin, S. (Ed.). (2017). Game AI Pro 3: Collected Wisdom of Game AI Professionals (1st ed.). A K Peters/CRC Press.

Reynolds, C.W. (1999). Steering Behaviors For Autonomous Characters. In *Proceedings of the Game Developers Conference*, Vol. 1999. Citeseer, 763–782

Roberts, B., Neal, M., Snooke, N., Labrosse, F., Curteis, T., & Fraser, M. (2020). A bespoke low-cost system for radio tracking animals using multi-rotor and fixed-wing Unmanned Aerial Vehicles. *Methods in Ecology and Evolution, 11*(11), 1427-1433.

Rodenberg, O & Verbree, Edward & Zlatanova, Sisi. (2016). INDOOR A* PATHFINDING THROUGH AN OCTREE REPRESENTATION OF A POINT CLOUD. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences. IV-2/W1. 10.5194/isprs-annals-IV-2-W1-249-2016.

Smolyanskiy, N., Kamenev, A., Smith, J., & Birchfield, S. (2017). Toward low-flying autonomous MAV Trail Navigation using Deep Neural Networks for environmental awareness. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Tan, C. H., Ng, M., Shaiful, D. S., Win, S. K., Ang, W. J., Yeung, S. K., . . . Foong, S. (2018). A smart unmanned aerial vehicle (UAV) based imaging system for inspection of Deep Hazardous Tunnels. *Water Practice and Technology, 13*(4), 991-1000.

Tsoar, A., Nathan, R., Bartan, Y., Vyssotski, A., Dell'Omo, G., and Ulanovsky, N. (2011). Large-scale Navigational Map in a Mammal. Proc. Natl. Acad. Sci. 108, E718–E724.

Tourani A, Bavle H, Sanchez-Lopez JL, Voos H. Visual SLAM: What Are the Current Trends and What to Expect? Sensors (Basel). 2022 Nov 29;22(23):9297.

Weldon, W. T., & Hupy, J. (2020). Investigating methods for integrating unmanned aerial systems in search and rescue operations. *Drones, 4*(3), 38.

E. Westman, A. Hinduja and M. Kaess, "Feature-Based SLAM for Imaging Sonar with Under-Constrained Landmarks," 2018 IEEE International Conference on Robotics and Automation (ICRA), Brisbane, QLD, Australia, 2018, pp. 3629-3636.

Yue, X., Zhang, Y., Chen, J., Chen, J., Zhou, X., & He, M. (2024). LIDAR-based SLAM for robotic mapping: state of the art and new frontiers. Industrial Robot: the international journal of robotics research and application.