Advancements in the 'Multi-Agent Exploration for Three-dimensional Space' Framework

Andreas Biegel, Kasper Johan Holmgaard Nielsen, Thomas Boel Stubkjær
Department of Computer Science, Aalborg University, 9220 Aalborg, Denmark,
Email: {abiege18, kjhn18, tstubk18}@student.aau.dk

***Abstract*—Multi-agent Exploration Simulator for 3D Space (MAES3D) is an open-source framework developed in Unity for testing and comparing exploration and coverage algorithms (Biegel et. al). This paper proposes to further expand upon the framework by addressing shortcomings and implementing new features missing from the prior iteration of the framework. These changes include an improvement of the sensing capabilities of agents allowing them a longer sensing range, a visualization of the explored areas, a more configurable map generator, and an improved camera and user interface. Furthermore, a novel exploration algorithm called Dual-Stage Viewpoint Planner (DSVP) is translated from 2D using a single robot to 3D with multiple robots. Experiments are conducted to compare the performance of DSVP and the already implemented algorithms named Random Ballistic Walk (RBW) and Local Voronoi Decomposition (LVD). The results of the experiments show that RBW is outperformed by both LVD and DSVP, with DSVP as the concise winner. Finally, a performance test shows that MAES3D is able to simulate at least 10 agents simultaneously.**

***Index Terms*—Continuous Space, Random Ballistic Walk, Local Voronoi Decomposition, Dual-Stage Viewpoint Planner, Swarm Robotics**

I. Introduction

Exploration algorithms are typically evaluated using a purpose-built simulator designed specifically for that algorithm. This means that there is no practical way of comparing algorithms as they are each developed with different assumptions about hardware capabilities [1]. Furthermore, the general-purpose simulators that exist are either too complex in nature and require extensive configuration and computational resources [2, 3], or do not have support for 3D environments and agent movement [4].

This led to the development of Multi-agent Exploration Simulator for 3D Space (MAES3D). A verbatim copy of the project report for MAES3D developed in the previous semester is available in Appendix B. The main objective of MAES3D is to remove the overhead required to implement and compare algorithms. This is done by having a built-in map generator, prefabricated agents with the capability for inter-agent communication, a light detection and ranging (LiDAR) system updating a local exploration map, and a set of tasks they can perform e.g. moving, while only exposing the user to a controller for interacting with the agents when implementing an algorithm. The source code is available on GitHub at https://github.com/DeagleBiegel/MAES3D and a video demonstration of the framework is available on Youtube at https://youtu.be/b33ph1WrYB4.

The purpose of this paper is to further improve the capabilities of MAES3D by implementing new features presented and discussed in future work by Biegel et al. in Appendix B and a new novel algorithm by Zhu et. al [5] called Dual-Stage Viewpoint Planner.

The paper is structured as follows. In section II, we describe the relevant background information regarding the current implementation of MAES3D and a novel algorithm we considered. Next, in section III, we discuss the shortcomings of MAES3D and solutions. In section IV, we describe the features and algorithm that will be implemented in MAES3D. In section VI, we conduct a comparison of the existing algorithms and the new algorithm. Then, in section V, we conduct a performance test of the framework. Finally, in section VII, we summarize our findings and discuss potential avenues for future work in section VIII.

II. Background Information

This section will outline relevant background information to ensure a better understanding of the subject matter, which includes the terminology used, the framework we built on, other solutions from the bibliography, and a novel exploration algorithm we considered.

A. Terminology

This section will describe some key concepts and terms used throughout the paper.

A **voxel** is a representation of a pixel in a three-dimensional space. The word is derived from Volume and Pixel. And they are commonly used in computer graphics to represent a value in a three-dimensional grid.

Exploration is the set of voxels an agent has seen during traversal and consists of voxels that are either a wall or in the free space.

Coverage is the set of voxels in free space that has been physically occupied by an agent when relocating to a new location.

B. Existing Solutions

Multi-agent Exploration Simulator (MAES) by Andreasen et al. [6] is a framework for simulating coverage and exploration algorithm in a two-dimensional continuous space.

The framework is able to generate two different kinds of environments named *building maps* and *cave maps*. Building maps feature hallways and rooms to resemble a building, and

cave maps that resemble a cave-like structure using irregular shapes. Within these two types of maps, multiple different two-dimensional exploration and coverage algorithms were simulated, and then later the results of the said simulation were compared.

These algorithms included both algorithms for continuous space, as well as algorithms that operate in discrete grid-based space. Here, they converted all algorithms to continuous space to enable the possibility of comparing the algorithms.

Autonomous Robots Go Swarming (ARGoS) [3] is a robot simulator developed to simulate large swarms of robots with support for various physics engines. ARGoS is built to be highly modular allowing a user to add new functionality in the form of sensors, actuators, individual robot components, means of communication, and environment dynamics such as wind and water flow.

Gazebo [2] is a tool for robot simulation, offering a set of features allowing the user to create a virtual world in which they can simulate robot behavior. Another key feature of Gazebo is a library that allows the user to exchange data between clients in the form of nodes, which can be used to distribute the workload on the same computer or multiple computers.

C. Multi-agent Exploration Simulator for 3D

MAES3D by Biegel et al. [7] in Appendix B is a framework for simulating coverage and exploration algorithms in a three-dimensional continuous space using swarm robotics with inter-agent communication inspired by MAES [6]. The three points of interest the framework has to comprise are (a) the map generator, (b) the interface for agents, and (c) algorithms for coverage and exploration.

(a) The map generator uses a cellular automaton (Appendix B, Section 3.1.3) utilizing Moore neighborhood to smoothen out a randomly filled three-dimensional grid of voxels, a predefined amount of times. The size of the map is adjustable by the user, meaning the width, height, and depth can be changed with a lower limit of 30 and an upper limit of 100 voxels in each dimension.

(b) The agents mimic the behavior of underwater drones. It can only move in the direction the agent is facing and therefore has to rotate when changing direction. They use a modified version of the Look Compute Move (LCM) Model [8] by also incorporating a communicate phase making it the Look Compute Communicate Move (LCCM) Model (Appendix B, Section 3.4). Each agent has a controller with a set of instructions that can be called to manipulate it (e.g. telling it where to go) when implementing an algorithm.

(c) The native novel algorithms implemented in MAES3D are Random Ballistic Walk (RBW) and Local Voronoi Decomposition (LVD). In RBW [9], the agent moves continuously in a straight line and rotates a random amount of degrees when colliding with an object (Appendix B, Section 3.3.1). LVD [10] works on the principle of divide-and-conquer by decomposing the agent's viewable space into Voronoi regions based on the agent's proximity to a set of voxels. The agents explore the map by moving to the nearest unexplored voxel within their own Voronoi region. However, they move to the nearest least recently visited occlusion point (a point that breaks the line of sight) iff all voxels in their region are explored (Appendix B, Section 3.3.2).

D. Dual-stage Viewpoint Planner

Dual-stage Viewpoint Planner (DSVP) by Zhu et al. [5] is a two-dimensional exploration algorithm. The algorithm is split into two stages, each with a different goal.

The first stage is called the Exploration Stage. The goal of this stage is to explore the nearby terrain to find the next position the agent should move to. This is done by finding frontiers in the current view of the agent where each frontier is a point on the border between an explored region to an unexplored region. The subset of frontiers that are in the direction the agent last traveled will be stored as local frontiers whereas the rest will be stored as global frontiers. Within an area around the agent called the "planning horizon", random points are sampled with a bias towards the local frontiers to construct a rapid-exploration random tree (RRT) [11] with the agent's current position as the root.

For each node in the local RRT a gain is calculated that is an approximation of the unexplored area that would be explored if the agent was positioned at the node in question using the equation:

$$Gain(V) = VoxelGain(V) \cdot e^{-dist(V)\cdot\lambda_1},$$

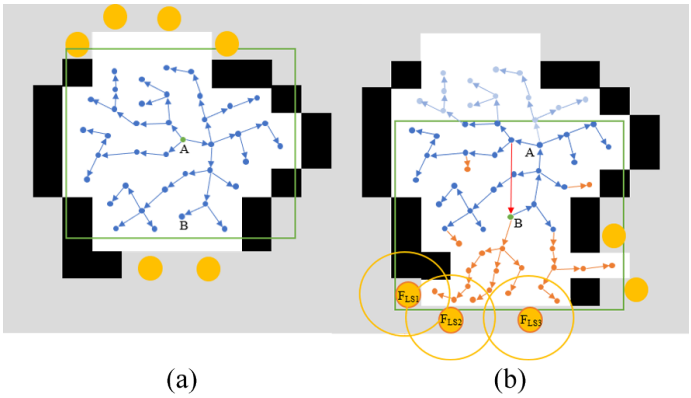
where $VoxelGain(V)$ is the amount of unexplored area that is explored from the node, $dist(V)$ is the distance from the agent's current position to the node, and λ_1 is a parameter that penalizes travel distance.

The local RRT is searched to find the branch within the planning horizon that contains the nodes that give the highest cumulative gain using the equation:

$$Gain(B) = \sum_{V_i^j \in B_i} Gain(V_i^j) \cdot e^{-DTW(B_i)\cdot\lambda_2},$$

where V_i^j represents the ith node in branch B_i, $DTW(B_i)$ is the similarity between the last selected branch and branch B_i which reflects the current movement direction, and λ_2 is a parameter that penalises change in movement direction.

This branch represents the path within the planning horizon, that will approximately explore the biggest area of unexplored space, and thus the last node in the branch is selected as the next destination. Once the agent has moved to the node, it is set as the new root of the local RRT, and the exploration stage is run again, expanding the local RRT. An example of this RRT expansion and pruning can be seen on Figure 1
For each iteration of the exploration stage, the viewpoints of every branch with positive gain will be stored as a vertex in a global Rapidly-exploring Random Graph (RRG) [11]. Every time a vertex is added, a connection between it and the closest existing vertex will be created. Furthermore, if any existing vertices meet both of the following conditions a connection between that vertex and the new vertex will also be created.



(a) (b)

Fig. 1: Exploration stage. (a) shows the RRT as a blue graph and local frontiers from the previous iteration as yellow circles. Unexplored space is represented by grey, walls as black, and explored space as white. The green square represents the planning horizon of that iteration. (b) shows the RRT expansion where orange dots are the newly added nodes and light blue dots are the pruned nodes [5].

$$\begin{cases} D_E(v, v_{new}) < \delta \\ D_G(v, v_{new})/D_E(v, v_{new}) > \gamma \end{cases},$$

where Euclidean distance is represented by D_E and the shortest distance between the new vertex and the nearest vertex is represented by D_G. The two parameters δ and γ are used to restrict how many connections that are added to the graph. δ is a fixed length that connections have to stay below and γ is a factor to ensure that new connections have to be within a certain amount of the already established connection.
If no frontiers can be generated in the exploration phase the agent will switch to the Relocation Stage. In this stage, the agent will find the global frontier closest to its current position and set it to its next position. The agent will then utilize the RRG to move towards that global frontier. When the agent has arrived at the destination it will switch back to the exploration stage.

The algorithm terminates when no new frontiers can be generated and there are no global frontiers the agent can relocate to.

III. Shortcomings of MAES3D and Solutions

The current implementation of MAES3D leaves a lot of features to be desired. This section will discuss the shortcomings of MAES3D mentioned in future work of the first report in Appendix B, and also shortcomings thought of after the report was publicized. Furthermore, solutions will be proposed to alleviate the mentioned shortcomings.

A. Sensing the Environment

Currently, the agents sense the environment by raycasting to every cell that is on the edge of a sphere with the radius of the agent's observation range. While this ensures that every cell within the observation range will intersect with one of the rays it also means that the number of rays required increases with the surface area of the sphere with a radius of the agent's observation range. Furthermore, additional checks are performed for each cell as a ray intersects to ensure that it is not obstructed by other parts of the environment. The computational expense of these operations means that a balance needs to be found between an acceptable observation range and computational overhead that will not slow down the simulation. During testing, it was found that this balance was reached with an observation range of 5 voxels.

Given that the environments can reach a maximum of 100^3 voxels, this 5-voxel observation range poses a considerable limitation to the agent's sensing ability.

This problem can be addressed by optimizing the raycasting algorithm to allow for an increased number of rays to be projected without a decrease in performance. This would allow us to develop a method of sensing that is based on a LiDAR system with an increased range where the observation resolution at longer distances is still acceptable due to the increased number of rays.

B. Movement

The current implementation of the agent's movement does not reflect the behavior of movement for real-world drones. As of right now, they do not possess the ability to accelerate and decelerate, meaning they reach a top speed instantaneously and come to a halt immediately when reaching their destination. Furthermore, the start-up and stop time of the rotation when turning is also instantaneous.

To improve on this, we would like to add a way for the agent to accelerate and decelerate when moving and rotating. For moving, the agent generates speed over time when starting to move, holding a constant speed, and decreases the speed when the agent is at a set distance to the destination allowing enough time for stopping. The same rule should apply when rotating. Here, the agent's rotation speed increases, holding a constant speed, and decreases when it is nearing the specified angle and coming to a halt.

C. Visualisation of Explored Areas

There is currently no way to see what part of the map has been explored. This makes it a guessing game whether or not the agent is heading toward the last unexplored areas when the exploration is almost complete. MAES visualizes what has been explored by highlighting tiles the agent has seen using a green color, and a blue color to highlight what an individual agent has seen when it is selected, as seen on the video demonstration of MAES available on Youtube at [4].

Replicating what MAES does can obstruct the user's vision of the agents in three-dimensional space – highlighting voxels in free space using a semi-transparent color around the agent(s). Another idea is to highlight the faces of walls that have been explored. However, having multiple layers of pathways blocking the vision can make it near impossible to find the walls that have been highlighted or vice versa in the center of the map.

Thus, we think the most elegant solution to this problem is to do the inverse of what MAES does. This means highlighting

voxels in free space that have yet to be explored with a semi-transparent color. The unexplored voxels in free space are easily distinguished from explored voxels and do not obstruct the user's vision of agents as they do not reside inside the semi-transparent colored voxels.

D. Map Generator

The map generator in MAES3D works by randomly deciding whether each voxel in the map grid should be free or a wall and performing iterative smoothing on this random grid to achieve a cave-like structure. Due to the highly randomized nature of this process along with the culling of smaller, unconnected pockets, the maps that are generated can vary greatly in size from the specified map size. E.g. a map generated in a 100^3 might end up with a resulting map that fits inside a 50^3 grid.

When creating a map, the generator itself uses some variables that result in the map structure being denser with fewer open areas the bigger the map got. These variables include the fill ratio of the initial map before smoothing and the amount of smoothing iterations a map went through before being used for the simulation.

In this generator, the final generated map is entirely dependent on the initial random distribution of voxels and the number of smoothing iterations. The pair of values that control the chance of a voxel being filled and the number of smoothing iterations have a narrow range of producing acceptable maps which greatly limits the amount of control a user can have.

When smoothing a map, it was decided to use the Moore neighborhood method [12] with 13 or less being the number of neighboring walls needed to remove a wall, and 14 or more to add a wall. Both the method and the number of neighboring walls needed to add/remove a wall while smoothing, are variables that could be determined by the user to create a structure more to their liking.

Creating a new map generator designed to always utilize the entire available map grid and offer the user more configurability over the generation will allow the user to generate a more varied set of maps with a more predictable output. This will be done by generating and connecting spheres in the map grid where the parameters that control the sphere generation are configurable by the user.

Furthermore, the only way to generate a specific map more than once is to input the same parameters, such as size and seed, both times. If a map is generated with a random seed it is currently not possible to extract the seed meaning that generating that same map on-demand is not possible. This can present itself as a problem if a user wishes to see the effects of a change in an exploration algorithm by running it on the same map without having to remember the exact parameters.

To solve this, a file format will be designed that allows a user to store a specific map after generation on the disk so it can be loaded for later simulations.

E. Camera and User Interface

There is currently no way to interact with the simulation while it is running. A user can only stop or start a new simulation overriding the previously running simulation. Also, the camera is static with an overview of the map and cannot be manipulated, e.g. following an agent or rotating around the map.

The only way to actually navigate the map is to enter the scene view in the Unity application while a simulation is running. Here, you can move the camera using the built-in key binds (W, A, S, and D) and hold the right mouse button down to rotate the camera. This poses some problems, such as the UI is not visible and that functionality is not supported when the framework is running outside of Unity as a stand-alone application.

To solve this, we would like to look at ways to improve the user experience when spectating the exploration, while simultaneously making it easier to observe specific agents when testing. A solution is to have a fixed point on the center of the map and agent(s) the camera can rotate around. There should also be displayed additional information in the UI if an agent is selected, which e.g. can be the ID of the agent, where it currently is, and customized information parsed from the chosen algorithm.

F. A Pathfinding*

The current implementation of the A* Pathfinding algorithm works for what it currently has to do. The algorithm is used when an agent has to move from A to B and there is not a direct line of sight from the current position. The only algorithm currently using A* is LVD, which only moves to voxels it can physically see, and occasionally have to navigate around the corner of a wall to reach said voxel. This means that current paths calculated using A* are not complex. However, they do become more complex when we want to explore frontier-based algorithms such as DSVP. Here, the agent has to potentially navigate from one side of the map to the other side, making the paths more complex and thus taking a long time to calculate.

This poses a problem, as the calculation made with A* is done on the main thread in a single cycle, meaning the update loop is halted and therefore freezes the framework until a valid path is found.

Here, we would like to explore alternatives such as using coroutines and multi-threading. Having the algorithm execute on a separate thread using multi-threading breaks the deterministic behavior of the framework, as the algorithm is executed outside the bounds of the fixed update loop and the time to calculate a path is dependent on the performance of the processor. Thus, running the same scenario on two different setups can produce different outcomes. A way to keep the framework deterministic is to use A* as a coroutine. Here, the main loop of A* runs a predetermined amount of times on the main thread and then yields control back to Unity. This is repeated every time fixed update is called and gives the illusion of multi-threading. Therefore, two different setups will produce the same path in the same amount of fixed update cycles – keeping it deterministic.

IV. Our Proposed MAES3D Features

This section will go through the design choices for the solutions for the shortcomings mentioned in section III, as well as how they are implemented in MAES3D.

A. LiDAR-based sensing

A new method for sensing the environment has been developed that emulates a LiDAR system. In a 2D setting, LiDAR works by projecting beams through evenly spaced points in a circle around the agent and measuring the distance of impact with an object. This approach will be modified to 3D by projecting beams through approximately evenly spaced points in a sphere around the agent. This will result in an implementation where it is not guaranteed that every cell within the desired observation range will intersect a ray. However, by optimizing the raycasting algorithm and calculation, the number of rays that are projected can be raised enough that the loss of precision of individual rays can be compensated for by the increase in the number of rays.

To place the approximately evenly spaced points on a sphere, a method has been implemented that maps a Fibonacci lattice onto a sphere based on the methods described by Martin Roberts [13]. First, the number of desired projection beams is mapped as evenly spaced points on a unit square using the Fibonacci lattice:

$$t_i = (x_i, y_i) = \left(\left\{ \frac{i}{\phi} \right\}, \frac{i}{n} \right) \text{ for } 0 \leq i \leq n$$

where ϕ is the golden ratio, n is the desired number of rays, and $\{\frac{i}{\phi}\}$ is the factional part of $\frac{i}{\phi}$. This Fibonacci lattice works by placing the first point at $(0, 0)$ and displacing the next point ϕ along the x-axis (and using the modulo operator to wrap it to be within the $[0, 1]$ interval) and $\frac{1}{n-1}$ along the y-axis effectively placing each point in a distinct row. As ϕ is the most irrational number it not only ensures that two x_i values are never equal, it also maximizes the minimum nearest neighbor distance and avoids the x values clumping together [14].

These points can be mapped from the 2D cartesian coordinate system, represented by the coordinates (x, y), onto the surface of a sphere in spherical coordinates, represented by the angles (θ, ϕ), with the formula:

$$(x, y) \rightarrow (\theta, \phi) : (2\pi x,\ arccos(1 - 2y))$$

The resulting point can further be converted into a point in the 3D cartesian coordinate system, represented by the coordinates (x, y, z) using the formula:

$$(\theta, \phi) \rightarrow (x, y, z) : (cos(\theta)sin(\phi),\ sin(\theta)sin(\phi), cos(\phi))$$

How the points are mapped from a unit square onto a sphere can be seen on Figure 2.
Transposing these points so that the agent is in the center of the sphere, rays are projected from the position of the agent through each of the points thus creating a set of evenly spaced beams around the agent.

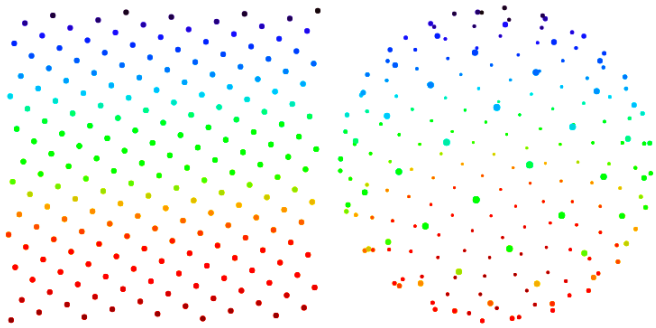


Fig. 2: Fibonacci lattice of 200 points mapped onto a square and a sphere [15]

Like the old method for sensing, the projected beams are based on the grid traversal algorithm proposed by Amanatides & Woo [16], however, as opposed to the old method, the rays are now projected past their target point, continuing until a wall is intersected.

B. Directional sensing

The methods described in subsection IV-A simulate a 360° field of view (FOV) sensor. Sensors with a less than 360° FOV can also be implemented by utilizing the intermediate representation of the points as spherical coordinates represented by the angles (θ, ϕ). A sensor with a less than 360° FOV is emulated by only using the points where the θ and ϕ of the angular representation are both less than the desired FOV.

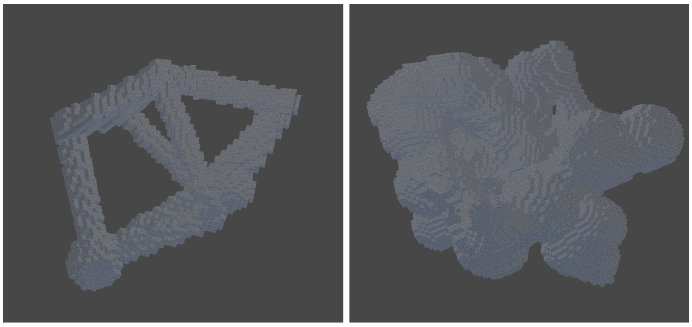
C. Map Generator

The ability to further customize the old map generator has been added. The user is now able to set the fill and smoothing variables, where the old values are used as default in case the user does not specify any.

Additionally, a new map generator has been designed with the goal of giving the user more control of the resulting map by exposing more variables. This map generator works by generating and carving out spheres at randomized positions within the map grid. These spheres are generated until a certain percentage of the voxels within the map grid have been carved out. When this percentage of voxels has been carved out of the map grid, the generated spheres are connected by creating tunnels between nearby spheres.

Continuously generating spheres in the grid until a certain percentage has been carved out ensures that the maps that are generated adequately utilize the map size that was selected by the user addressing one of the main issues of the old map generator.

Furthermore, giving the user the ability to control the threshold also allows the user to use this to control the openness of the generated maps. I.e. a low threshold will result in sparsely placed centers connected by long corridors whereas a high threshold will result in larger, more interconnected areas. The intricacies of the generated maps can further be adjusted by allowing the user to control the range of the radius of the



(a) (b)

Fig. 3: Maps generated by the new map generator. (a) shows a map generated with a low fill percentage and small spheres. (b) shows a map generated with a higher fill percentage and larger spheres.

generated spheres as well as the number of tunnels that are connected to each sphere.

As compared to the old map generator, the enhanced customization options in the new map generator, allow the user to test the exploration algorithms in a much more diverse set of environments as shown on Figure 3. This allows the user to gain a deeper insight into scenarios a specific exploration algorithm excels in.

D. Map Exporting and Importing

To allow a user to reuse a map in multiple simulations without having to remember the specific configuration of the generator such as the seed, the dimensions, and the generator parameters, the user is now able to export the map that has been generated to a file that can be stored on the disk in MAES3DMap (M3DM) format. The M3DM format is a novel format designed specifically for storing MAES3D maps. The datastream of an M3DM file can be split into two distinct parts, the header, and the data section. A summary of the contents of an M3DM file can be observed on Figure 4

The header consists of 3 bytes or 24 bits. As MAES3D supports sizes in each dimension in the range of 30 to 100 the size of each dimension will be stored as an unsigned 7-bit number with a maximum value of 127. The remaining 3 bits in the header define the data offset.

The data section contains the data that describes the layout of the map. Since the map in MAES3D is represented as a 3-dimensional array of boolean values, the data section will consist of a sequence of bits that depicts a flattened representation of this 3-dimensional array. That means that an M3DM file that depicts a $50\times100\times75$ map of a total of 350,000 cells will also have a data section of at least 350,000 bits. Since files are stored in bytes instead of individual bits, it is highly likely that the number of bits necessary to represent the map does not fit neatly into any number of bytes.

For example, a map of size $35\times35\times35$ will contain 5,359 cells. This can not be neatly encapsulated in any number of bytes and will be stored in 670 bytes at a total of 5,360 bits. This leaves one leftover bit in the data section that is not a part of the actual data.

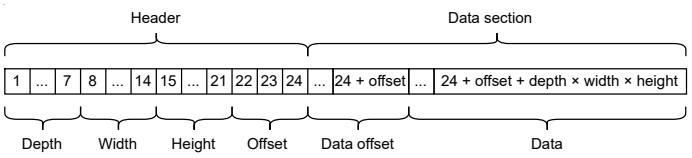


Fig. 4: MAES3DMap (M3DM) file specification

The leftover bits are prefixed to the data as 0 values and the number of leftover bits in the data section is stored as a 3-bit unsigned integer in the offset section of the file header. The map import process will read the header to decode the size of the map and the number of data offset bits. Since MAES3D supports sizes in each dimension in the range of 30 to 100 it is first validated if each size value is within this range. The importer then makes sure that the data section contains exactly $\frac{depth\times width\times height+offset}{8}$ bytes before the map rebuilding process is started. If there is a discrepancy between the expected number of bytes in the data section and the actual number of bytes in the data section, it indicates that the file is not a valid M3DM file and that the map cannot be constructed.

If the preliminary checks are cleared the map grid will be constructed from the data section. First, the number of bits indicated in the header will be ignored and any subsequent bit is placed into a 3-dimensional array with the sizes indicated in the header in the order. Lastly, to ensure that the map is compatible with MAES3D the edges of the map are filled with walls, and any smaller pockets that may be present are filled in. Although every map generated by or exported from MAES3D already fulfills the purpose of the last step, it is still performed to ensure compatibility with maps that are generated by external tools.

E. Camera and User Interface

In this new iteration of MAES3D, multiple camera features have been implemented to better the interaction between the user and the simulation whilst it is running. A new panel with buttons has been added, the purpose of this panel is to move the view between a general point of the cave and each agent within the cave. In addition, keybindings for the numerical keys have been added to change the camera position between the agents and also the escape key for the cave view.

Functionality for camera rotation has been added to the mouse, which is accessible by holding down the right mouse button. This allows the user to rotate the camera 360 degrees around the target. The user can use the left mouse button to select an agent as the target for the camera.

When an agent is selected through any means, a panel with information about the agent and the algorithm it is running is displayed to the user. This panel allows for some basic information like the current position, target position, and current speed, as well as space for some custom information that can be tailored to the specific algorithm in use by the agent.

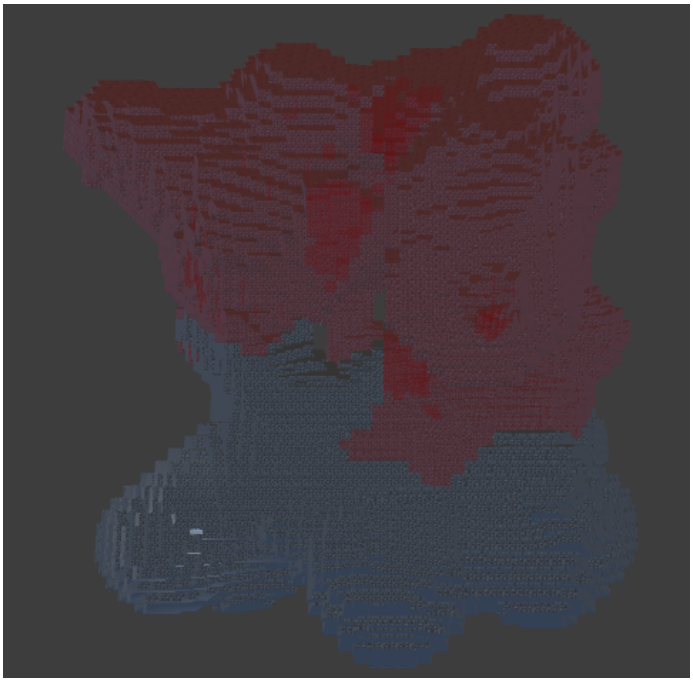


Fig. 5: An overview of a cave with agents visible in the bottom left. The red mesh shows unexplored voxels. The parts not covered in red have been explored by the agents.

Furthermore, a toggle-able option of showing all unexplored tiles has been added to help visualize what is left to be explored in the selected simulation.

F. Visualisation of Unexplored Areas

In subsection III-C we talked about the possible ways to visualize what has been explored and vice versa. Here, we have added a way to visually identify areas that have yet to be explored. This is done by giving every unexplored voxel a semi-transparent red tint as seen on Figure 5. The red tint is achieved by adding a mesh with a translucent red surface to the scene whose shape is modified to reflect what the agents have yet to observe. As modifying a mesh is a computationally expensive process, the entire map grid is split into multiple meshes where only the meshes that represent a voxel that has been observed will be updated. For example, the exploration visualization in a map of size 100^3 will be composed of 64 meshes of size 25^3.

The first time a cell is observed by an agent, the position of the cell is stored in a shared list. Every 10th fixed update, the meshes that correlate to each point within the shared list are updated to reflect the newly observed cells.

G. Alternative A* Pathfinding Methods

The problem listed in subsection III-F described the issues of the current implementation of the A* pathfinding algorithm. Here, we have implemented two additional methods in addition to the current existing method for calculating a path using A*.

The first method takes advantage of multi-threading and is an asynchronous method called `MoveToAsync()`. Here, the path is calculated on a new thread. The second method is a coroutine method called `MoveToCoroutine()`. In Unity, a coroutine method allows you to spread a task across several frames, the method can be paused in the middle of the execution and return control back to Unity, and then continue where it stopped on the next frame [17]. In other words, the path is calculated across several frames, or in this case fixed updates. In both methods, the agent enters a `busy` state to indicate it is currently calculating a path, and changes to `moving` state when a path is found. The already existing method called `MoveTo()` is the same as before, and calculates the path in a single go on the main thread.

This gives the user the freedom of choice to use any of the three methods and not limit them to only one. Furthermore, the number of iterations executed per update for `MoveToCoroutine()` is adjustable by the user in the UI with the default setting being 100 iterations each time a fixed update occurs.

H. DSVP in Three-dimensional Space

For the DSVP algorithm to work in three-dimensional space, changes to the two-dimensional aspect of the algorithm had to be implemented. This refers to the local RRT, the global RRG, and the planning horizon being expanded out in the third dimension. Furthermore, since the algorithm is originally designed for a single agent, additional behavior is needed to allow the agents to effectively aid each other in exploring the map.

When generating nodes for the local RRT, we have to consider not only the width and depth but also the height. Since every node generated should be within the planning horizon, the horizon itself should be expanded into three dimensions. With the planning horizon expanded, we can simply check if the generated node's x, y, and z coordinates are within the planning horizon in relation to the agent's coordinates.

To allow multiple agents to run the DSVP parallel to each other while exploring the cave in an efficient manner, an adaptation was made to the DSVP algorithm. This adaption refers to changing the algorithm from the LCM model to the LCCM model. This change allows the agents to periodically communicate with other agents within view/range of each other.

Since MAES3D already has an implemented pathfinding algorithm, this was used instead of the method described in the original DSVP paper [5]. However, the goal is still to reach the same destination as the original DSVP algorithm would. Therefore, the local RRT is extended into a global RRT, instead of using a global RRG. The global RRT will be used to find a global frontier within the path of the agents, the same way the original DSVP did with a global RRG.

The DSVP algorithm is split into two sub-algorithms: Exploration which can be observed in Algorithm 1, and Relocation which can be observed in Algorithm 2.

The purpose of the *Exploration* algorithm is to find the next destination for an agent. This is done by generating an RRT

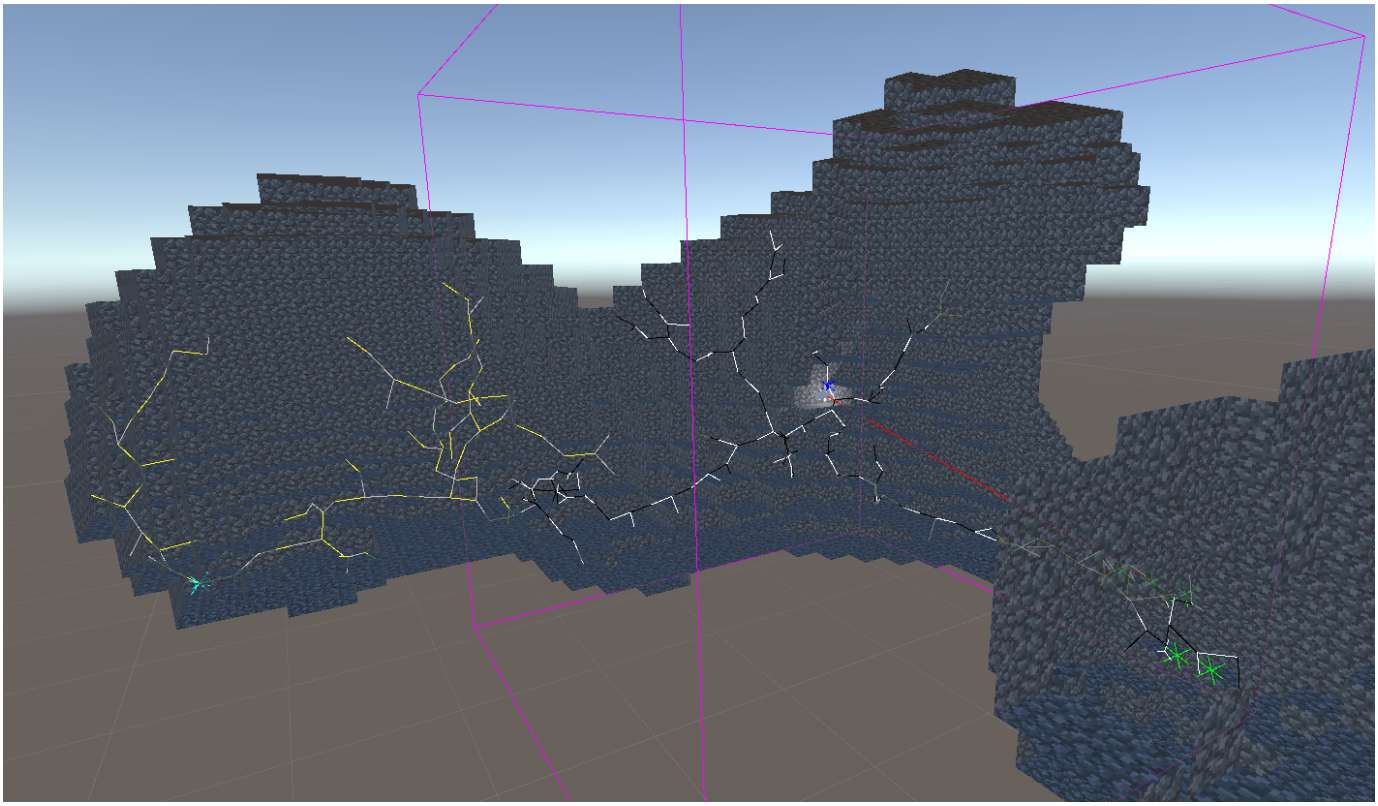


Fig. 6: Illustration of the DSVP algorithm calculating the agent's destination. The yellow and white tree in combination with the black and white tree forms the global RRT, with the origin point in the cyan cross. The black and white tree itself represents the local RRT, with its origin in the blue cross. The purple cube represents the planning horizon. The green crosses represent local frontiers, with the red line pointing at the selected frontier for the next destination.

with a basis in the agent's current location A_{pos}, the local frontiers F_L, as well as the prior iteration of the RRT. Then afterward calculate the gain of each branch B_i in the new tree, with the ultimate goal of traversing towards the node of that branch.

Algorithm 1 Exploration

Set root position A_{pos} and local frontiers F_L
Update $F_{L\_selected}$
$RRT_L \leftarrow DynamicExpansion()$
$BestGain \leftarrow 0$
for i from 1 to $RRT_{L_{max}}$ **do**
    Compute $Gain(B_i)$
    if $Gain(B_i) > BestGain$ **then**
        $Gain(B_i) \leftarrow Gain(B_i)$
        $BestBranch \leftarrow B_i$
    end if
end for
$PreviousTree \leftarrow CurrentTree$

The purpose of the *Relocation* algorithm is to find the global frontier closest to the agent, and then find the node in the global RRT closest to that frontier. First, we set a variable to be positive infinite. Then we traverse through all global frontiers in F_G, where F_{Gmax} is the number of global frontiers, and check their distance to the agent. The distance will be stored in $Dist$ if the value is smaller, and the global frontier is stored in F_{best}.

Once the closest global frontier is found, we iterate through all nodes in the global tree, and find the node closest to said global frontier.

Algorithm 2 Relocation

$Dist \leftarrow \infty$ and $F_{best} \leftarrow NULL$
for i from F_{Gmax} to 1 **do**
    if $F_{Gi\_Dist} < Dist$ **then**
        $Dist \leftarrow F_{Gi\_Dist}$
        $F_{best} \leftarrow F_{Gi}$
    end if
end for
$Dist \leftarrow \infty$ and $N_{best} \leftarrow NULL$
for i from RRT_{Gmax} to 1 **do**
    if $RRT_{Gi\_Dist} < Dist$ **then**
        $Dist \leftarrow RRT_{Gi\_Dist}$
        $N_{best} \leftarrow RRT_{Gi}$
    end if
end for

V. Performance Test

The performance of the framework is tested and evaluated in this section by performing benchmarks. The Benchmarks are

Agents	5	10	15	20
Time to complete	30:07	30:23	32:13	36:22
Updates	705,973	688,614	682,114	576,215

TABLE I: The time it takes to complete a 30-minute simulation and number of updates using the DSVP algorithm

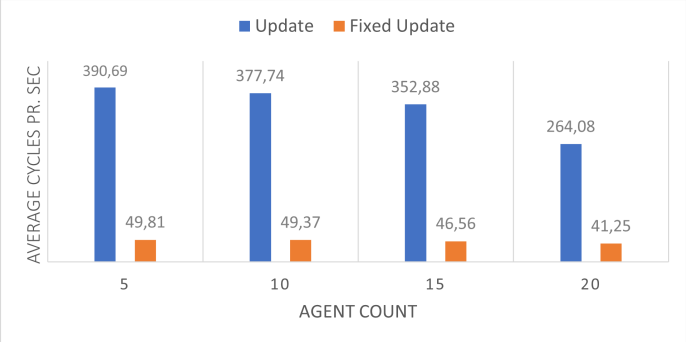


Fig. 7: Average cycles pr. second for update and fixed update for 5, 10, 15, and 20 agents using the DSVP algorithm

conducted on an identical map with the size of 100×100×100 voxels using various numbers of agents (5, 10, 15, 20) with the DSVP algorithm running for 30 simulated minutes. The computer used for the benchmarks has an i7-9700k [18] with a max turbo frequency of 4.9 GHz as the Central Processing Unit (CPU) and 16 GB of DDR4 ram at 3200 Mhz.

The matrix used for evaluation is the time-to-complete 90,000 update cycles in non-simulated minutes, and average updates and fixed updates a second. The main logic of the agents is executed using `FixedUpdate()` in Unity, which is a native method called 50 times a second, resulting in 90,000 calls over 1,800 simulated seconds. The components that do not require any determinism, such as the controller for the camera, are updated every frame using `Update()` in Unity.

The time-to-complete 90,000 fixed update cycles in non-simulated minutes and the total number of update cycles using 5, 10, 15, and 20 agents can be seen in Table I. And the average cycles pr. second for update and fixed update can be seen on Figure 7 with the respective agent count.

The ideal number of fixed updates is 50 cycles a second. Here, we can see on Figure 7 that 5 and 10 agents are close to 50 cycles/seconds, however as the agent count increases it starts to waver off.

The biggest bottleneck is the communication between agents. The local map for an agent is iterated through, by using a for-loop, each time it has to communicate with another agent. The bottleneck becomes exponentially worse as the agent count increases and the number of times map sharing occurs are:

$$1 + 2 + 3 + ... + Agent\ count - 1,$$

also known as triangular numbers [19]. In other words, the 10th agent has to share its local map with 9 other agents, and the 9th agent has to share its local map with 8 other agents as it already had communication with the 10th agent. This leads to the following exponential function:

$$y(x) = \frac{x^2}{2} - \frac{x}{2}.$$

Therefore, the total number of times map sharing occurs for e.g. 20 agents is 190 times. The map size used for the benchmarks was 100×100×100 voxels, meaning the total number of iterations that occurs are $1,000,000 \times 190 = 190,000,000$ when all agents are visible to each other.

The majority of the additional 6 minutes and 22 seconds spent for 20 agents, see Table I, is the byproduct of having the communication on the main thread halting everything else until it is completed.

VI. Experiments

This section will describe the setup used and the results of the experiments we conducted to evaluate the algorithms. The algorithms (LVD, RBW, DSVP) are tested on 10 different maps with three different sizes (i.e. 30 maps in total divided into 3 different sizes) using 2, 5, and 10 agents. They terminate after 30 minutes or 100% exploration has been achieved. The settings for the simulation are:

- **Duration:** 30 minutes
- **Agents:** { 2, 5, 10 } agents

The new map generator, Sphere Connections, is used and the properties of the maps we generate for the experiments are:

- **Fill Ratio:** 20%
- **Sphere Radius:** 3 - 8 voxels
- **Sphere Connections:** 3 connections
- **Smoothing Iterations:** 5 iterations
- **WxHxD:** { 50^3, 75^3, 100^3 } voxels

We also used the method that calculates the path for A* using a coroutine with the following setting:

- **Coroutine Iterations (A*):** 100 iterations

The performance matrix used to evaluate the algorithms is the average exploration time over 10 maps with the same size for a given algorithm. The exploration time is the time it takes to explore all free voxels at least once. The results of the experiments can be seen on Figure 8, 9, and 10.

The results for the experiments using maps with the dimensions of 50×50×50 voxels with 2, 5, and 10 agents for each algorithm can be seen on Figure 8. Here, we can see that all of the algorithms are able to complete exploration, with the exception of LVD using 2 agents and RBW using 2 and 5 agents, which reached an average of respectively 99.9%, 99.7%, and 97.7% exploration.

The results for the experiments using maps with the dimensions of 75×75×75 voxels with 2, 5, and 10 agents for each algorithm can be seen on Figure 9. Here, we can observe that DSVP has a tendency to outperform the other two algorithms, with the LVD being a steady second place. Surprisingly, compared to 50×50×50 and 100×100×100, LVD and RBW show similar results when there are 2 agents.

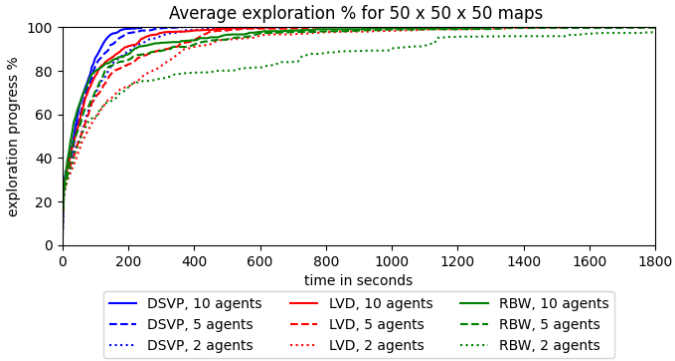


Fig. 8: A comparison of the algorithms in a 50×50×50 map with 2, 5, and 10 agents using exploration % as performance matrix over 30 minutes. The average result of 10 seeds.

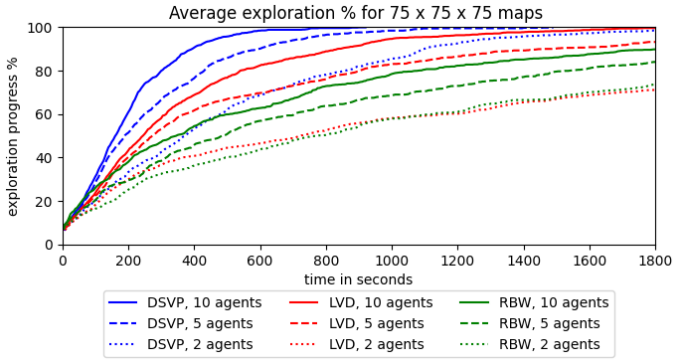


Fig. 9: A comparison of the algorithms in a 75×75×75 map with 2, 5, and 10 agents using exploration % as performance matrix over 30 minutes. The average result of 10 seeds.

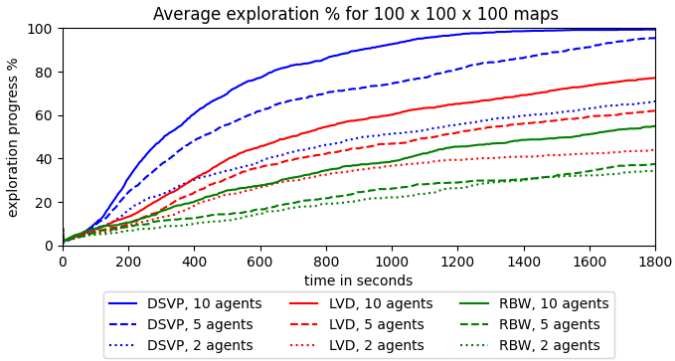


Fig. 10: A comparison of the algorithms in a 100×100×100 map with 2, 5, and 10 agents using exploration % as performance matrix over 30 minutes. The average result of 10 seeds.

The results for the experiments using maps with the dimensions of 100×100×100 voxels with 2, 5, and 10 agents for each algorithm can be seen on Figure 10. Here, we can see that RBW and LVD are closely aligned using only 2 agents, however, LVD starts to perform better as the agent count increases compared to RBW. DSVP is consistently better, compared to LVD and RBW, and is the only algorithm able to terminate by reaching 100% exploration with an agent count of 10. Furthermore, DSVP is almost able to achieve 100% exploration using only 5 agents, whereas LVD is close to 60% and RBW 30% exploration.

VII. Conclusions

MAES3D is a framework for implementing, testing, and comparing coverage and exploration algorithms in a three-dimensional space. MAES3D comes with the ability to generate random seeded maps, and tools for implementing new algorithms using an agent's sensing, communication, and movement abilities. MAES3D has also received several improvements, such as an improved sensing range, acceleration and deceleration, an entirely new map generator, and a better user interface with interactability making it more a streamlined experience to work with. In addition to this, a three-dimensional adaptation of a novel exploration algorithm, DSVP, was implemented in MAES3D. Furthermore, performance tests were conducted and showed that MAES3D is able to support at least 10 agents using DSVP as the algorithm. But, the performance starts to waver off as the agent count increases because of the increased complexity of inter-agent communication. Experiments showed that RBW had consistently worse performance than both LVD and DSVP. LVD came in 2nd place, whereas DSVP showed the best performance in all scenarios beating both LVD and RBW.

VIII. Future Work

In this section, we propose possible avenues for future work that can be implemented to further improve upon the capabilities of MAES3D.

A. Optimized Communication

As described in section V, the most significant bottleneck for the performance is communication. Optimizing this would allow the simulations to run faster without the stuttering caused by the communication computation. This improved communication would, instead of comparing every cell of the agents that have seen each other's exploration maps, only evaluate the cells that have been newly observed since the two agents last communicated. This would require every agent to keep a list of cells for each other agent containing the cells that have been observed. When two agents get into communication range of each other only the lists associated with those two agents will be evaluated and emptied. This means that consecutive communication phases between agents would compare significantly fewer cells.

The computational overhead of the current implementation is directly related to the size of the environment whereas the

next new implementation is only dependent on how much an agent has explored. This means that the optimized communication would possibly also allow for map sizes larger than what is currently possible without it impacting performance.

B. Building Environment

A possible avenue for future work is to introduce a completely new environment. In MAES [4], you can also construct maps that resemble a building. Here, we would like to do something similar, however, instead of only having hallways and rooms we could also introduce staircases that lead to new floors. This would require us to implement a new type of agent with aerial behavior that can navigate the said environment, as this is based on land and not underwater, which we will describe in subsection VIII-C.

C. Aerial Drones

In addition to the underwater drone movement that the agents possess, we would like for the drones to have an alternative movement pattern that resembles the movement of an aerial drone. This will result in the agent having the ability to move in all cardinal directions, in combination with vertical movement, all without having to rotate the direction the drone is facing. Such movement patterns would be useful when simulating above-water scenarios such as the *Building Environment*.

D. Spawn Position

All agents start at empty voxels closest to the origin point (0, 0, 0) when creating a new simulation. However, when simulating more specific maps, the user might want the agents to start in specific locations, or distributed throughout the map.

A solution to this would be implementing a way to give the user the ability to set one or multiple start locations or select that the agents should be scattered randomly throughout the map.

E. File Encoding

As described in subsection IV-D, the current implementation of M3DM files treats the bits of the file directly as voxels in the map, making the data section of e.g. a $50\times50\times50$ map always be 50^3 or 125,000 bits. As the files consist of long strings of successive identical bits run-length encoding (RLE) [20] can be used to greatly reduce the size of an M3DM file. In RLE, instead of each bit indicating a single voxel, a group of a fixed number of bits will indicate the number of successive identical voxel values where the value of a voxel will be swapped for every group. For example, using a group size of 3 the bits 110011001 will consist of the groups 110, 011, and 001. This will get decoded as 6 solid voxels, 3 free voxels, and 1 solid voxel.

Acknowledgments

References

[1] J. Tang, H. Duan, and S. Lao, "Swarm intelligence algorithms for multiple unmanned aerial vehicles collaboration: A comprehensive review," *Artificial Intelligence Review*, vol. 56, no. 5, pp. 4295–4327, 2023.

[2] Gazebo, "Gazebo," https://gazebosim.org/home, 2023.

[3] ARGoS, "Argos: Large-scale robot simulations," https://www.argos-sim.info/, 2023.

[4] Maes, "A trailer showcasing the capabilities of the maes tool." https://www.youtube.com/watch?v=Yjd1599WfPc, 2021.

[5] H. Zhu, C. Cao, Y. Xia, S. Scherer, J. Zhang, and W. Wang, "Dsvp: Dual-stage viewpoint planner for rapid exploration by dynamic expansion," https://frc.ri.cmu.edu/~zhangji/publications/IROS_2021.pdf, 2021.

[6] M. Z. Andreasen, P. I. Holler, M. K. Jensen, and M. Albano, "Comparison of online exploration and coverage algorithms in continuous space," https://www.scitepress.org/Papers/2022/109759/109759.pdf, 2022.

[7] A. Biegel, K. J. H. Nielsen, and T. B. Stubkjær, "Maes3d: Multi-agent exploration simulator for three-dimensional space," https://projekter.aau.dk/projekter/files/512993579/MAES3D.pdf, 2023.

[8] P. Flocchini, G. Prencipe, and N. Santoro, *Computational Models*. Cham: Springer International Publishing, 2012, pp. 7–16. [Online]. Available: https://doi.org/10.1007/978-3-031-02008-7_2

[9] M. Kegeleirs, D. G. Ramos, and M. Birattari, "Random walk exploration for swarm mapping," *Towards Autonomous Robotic Systems*, p. pages 211–222, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-23807-0

[10] J. G. M. Fu, T. Bandyopadhyay, and M. H. Ang, "Local voronoi decomposition for multi-agent task allocation," https://ieeexplore.ieee.org/abstract/document/5152829?casa_token=9TK4M5C2-6MAAAAA:g9GDUmiKGZS8Y3DxtitoOjT7srPF_ovDc7npPagAUOEQxa40wsn74VDTEOWxcTyYmEl2OGFtVHw, 2009.

[11] S. M. LaValle and J. J. Kuffner, "Rapidly-exploring random trees: Progress and prospects: Steven m. lavalle, iowa state university, a james j. kuffner, jr., university of tokyo, tokyo, japan," *Algorithmic and computational robotics*, pp. 303–307, 2001.

[12] N. Johnston, "Game of life explanation," https://conwaylife.com/wiki/Neighbourhood, 2022.

[13] M. Roberts, "Evenly distributing points on a sphere," https://extremelearning.com.au/evenly-distributing-points-on-a-sphere/, 2020.

[14] Á. González, "Measurement of areas on a sphere using fibonacci and latitude–longitude lattices," *Mathematical Geosciences*, vol. 42, pp. 49–64, 2010.

[15] A. Sch, "Fibonacci lattices," https://observablehq.com/@meetamit/fibonacci-lattices, 2023.

[16] J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing," *Proceedings of EuroGraphics*, vol. 87, 1987.

[17] U. Documentation, "Coroutines," https://docs.unity3d.com/Manual/Coroutines.html, 2023.

[18] Intel, "Intel® core™ i7-9700k processor," https://www.intel.com/content/www/us/en/products/sku/186604/intel-core-i79700k-processor-12m-cache-up-to-4-90-ghz/specifications.html, 2023.

[19] Wolfram, "Triangular number," https://mathworld.wolfram.com/TriangularNumber.html, 2023.

[20] C. S. F. Guide, "Run length encoding," https://www.csfieldguide.org.nz/en/chapters/coding-compression/run-length-encoding/, 2019.

Appendix A
Methodology

For this section, we want to talk about how we worked on the project during the 9th and 10th semesters, while also mentioning what changed.

A. 9th Semester

The beginning of the 9th semester marked the start of the development of MAES3D. Since the majority of the group had no experience with Unity, the first part of the project revolved around learning and familiarizing ourselves with the engine. In essence, we tried to implement as much as possible, with no knowledge of the complexity and implementation time of the tasks, as we were learning the ins and out of Unity while figuring out what was possible and vice versa.

B. 10th Semester

For this semester, an initial list of features was devised, mainly consisting of features discussed in the future works section of the 9th semester. The familiarity and knowledge of working with Unity gained during that semester allowed us to estimate the complexity of implementing the proposed features. This was done using planning poker where each feature was discussed and given an estimation in the form of a number in the Fibonacci sequence. These estimations were used to create a rough but flexible schedule for development, reminiscent of a Gantt chart. This schedule included a 2-day buffer every two weeks to allow for certain features to go slightly over the allotted time and a designated period for discussing the work. While the group consists of 3 project members, only 2 tasks were designated overlapping time frames. This allowed two members to, at all times, engage in pair programming where it was deemed most beneficial as well as swapping out what members worked on which features. A Trello board was used to keep track of the designation of tasks marking them as ”to do” and ”doing”, and also for marking features as ready for code review before being considered done.

APPENDIX B

MULTI-AGENT EXPLORATION FOR THREE-DIMENSIONAL SPACE

– The following section is a verbatim copy of the project report developed in the previous semester found at [7] –

MAES3D

Multi-Agent Exploration Simulator
for Three-Dimensional Space

AALBORG UNIVERSITY
STUDENT REPORT

Project Report
cs-22-ds-9-10

Aalborg University
Department of Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Department of Comp. Science
Aalborg University
https://www.aau.dk

Title:
MAES3D

Scientific Theme:
Distributed Systems

Project Period:
SW9, Fall 2022

Project Group:
cs-22-ds-9-10

Participant(s):
Andreas Biegel
Kasper Johan Holmgaard Nielsen
Thomas Boel Stubkjær

Supervisor(s):
Michele Albano

Copies: 1

Page Numbers: 43

Date of Completion:
January 20, 2023

Multi-Agent Exploration Simulator for Three-dimensional space (MAES3D) is a framework for simulating three-dimensional exploration and coverage algorithms in a cave-structured environment. The work consisted in implementing a randomised cave generator and an agent that can sense its environment, share information with other agents and move according to inputs from an interface. This interface can be utilised to implement exploration and coverage algorithms that dictate how and where the agent should move. Using this interface, Random Ballistic Walk (RBW) and a version of Local Voronoi Decomposition (LVD) modified to work in three dimensions were implemented in MAES3D. The performance of RBW and LVD was evaluated by letting them explore identical environments showing that LVD performed better than RBW.

Table of Contents

1. Introduction

Exploration in swarm robotics refers to the task of traversing an initially unknown environment for the purpose of gaining new information. This is also referred to as Online Terrain Coverage (OTC) meaning that pre-processing the map beforehand is not possible. Applications range from surveillance [10], search & rescue [4] [10] to mapping environments [12]. In some of the applications, it is crucial to finish the task in a minimal amount of time (e.g. search & rescue). Therefore, a tool for simulating exploration algorithms is crucial to compare them against each other.

Popular simulation solutions include ARGoS [3] and Gazebo [6]. However, these solutions can be difficult to set up and configure. The reason behind the difficulty in setup and configuration is due to heavy emphasis on modularity and customisability. Both modularity and customisability increase the complexity of the simulation for the user. Furthermore, the requirements of the computing power can change vastly based on the simulation, in some cases even requiring multiple computers or even clusters. Another solution is the Multi Agent Exploration Simulator (MAES) tool [2], which simplifies testing and comparing two-dimensional exploration and coverage algorithms by developing a framework where the algorithms can act on common ground. ARGoS and Gazebo both support three-dimensional scenarios as a result of their extensive configurability, MAES on the other hand only support two-dimensional scenarios.

This project proposes a framework for simulating three-dimensional exploration and coverage algorithms in a more accessible manner by expanding on the philosophy of MAES. The framework will support agent movement in continuous space, with data collection being based on a grid structure. Information sharing between agents will be based on a limited vision restriction, meaning information will not be able to travel through walls or extended distances. Furthermore, a seed-based cave generator is included to ensure that algorithms can be compared to each other based on similar cave layouts. The framework will be tested by implementing a modified three-dimensional version of Random Ballistic Walk and Local Voronoi Decomposition.

2. Problem Analysis

An analysis of the problem will be conducted in this chapter. The analysis will look into existing solutions for simulating multi-agent exploration, and existing two-dimensional algorithms. Furthermore, a problem statement will be derived based on the findings.

2.1 Existing Solutions

This section will go into some of the existing solutions within the branch of distributed exploration using multiple agents.

2.1.1 ARGoS

ARGoS (Autonomous Robots Go Swarming) [3] is a tool that allows a user to simulate a large number of robots in complex environments aimed at the study of swarm robotics. ARGoS provides a solution that can simulate any type of robot in any type of environment using a highly modular approach. This means that a user can configure the robot's abilities, using a wide range of virtual motors and sensors, and environmental dynamics such as wind and water flow, and as such the physics simulation of ARGoS is a fundamental part of bringing the simulations closer to a real scenario. The highly modular nature of ARGoS also means that it requires an extensive configuration process to get started involving the configuration of a robot, the specific environment, and the physics that are applied.

2.1.2 Gazebo

Gazebo [6] is open-source software that allows users to create and simulate complex environments for robots. It is commonly used for testing and evaluating robot behaviour and performance. There are various physics libraries (e.g. for gaming, robotics, and science), as Gazebo is designed on the premise that there is not a single physics engine that is universally best for all simulation contexts. Furthermore, it provides numerous sensor models designed to generate realistic data from simulation environments. They can also be configured at run time to mimic specific real-world

sensors. It also provides a library for exchanging data between clients (i.e. nodes) on the same computer or multiple computers.

2.1.3 MAES

The paper [2] proposes a framework called Multi-Agent Exploration Simulator (MAES) to test and compare different exploration and coverage algorithms, some of which are used for grid-based exploration, in an unknown two-dimensional continuous space. The paper utilised algorithms such as Random Ballistic Walk and Local Voronoi Decomposition within MAES. The algorithms were tested on two different types of maps, representing a cave and a floor of a building. The performance of the algorithms was evaluated based on the amount of the map covered within a set time.

MAES is not developed as a tool to simulate the physics and capabilities of robots but instead as a framework for running grid-based exploration algorithms in a continuous space to compare them on a common ground. As such MAES provides a module for generating environments and a robot that has been preconfigured in regard to its movement and sensing capabilities. The configuration of MAES thus mainly consists of providing an exploration algorithm that can control the agents.

2.2 Algorithms

This section will look at two approaches to the problem of two-dimensional exploration and coverage in unknown environments.

2.2.1 Local Voronoi Decomposition for Multi-agent Task Allocation

The paper [5] proposes a two-dimensional exploration algorithm, Local Voronoi Decomposition (LVD), that utilises the principle of divide and conquer. There is a heavy emphasis on the divide part of the strategy, which is done by decomposing the agent's viewable space into Voronoi regions; the agents can determine their Voronoi region by using local information and other agents within line of sight. The conquer part of the strategy is simplistic and uses a greedy approach. The agent moves to the nearest unexplored cell within its Voronoi region, and chooses from an ordered list if there is more than one candidate. The agent uses occlusion points (i.e, corners of obstacles that breaks the line of sight) to find unexplored cells when all cells, in the current region, are explored. A comparison between LVD, Brick&Mortar, and Ant algorithm showed that LVD produced the best results in a simulation, having the lowest number of cycles elapsed for full exploration, and is close to the ideal time (i.e. lowest amount of steps required for full exploration). Furthermore, the algorithm brought overlapping of cells to a minimum; ranging from 1.72% to 20.20% depending on the type of map.

2.2.1.1 Voronoi Diagram

A Voronoi diagram, seen in Figure 2.1, is a partition of a plane with `n` points into convex polygons (i.e. regions). The convex polygons are formed using generating points (also refereed to as seeds or sites), and every point inside a polygon is closer to its generating point than to any other of the generating points [11].

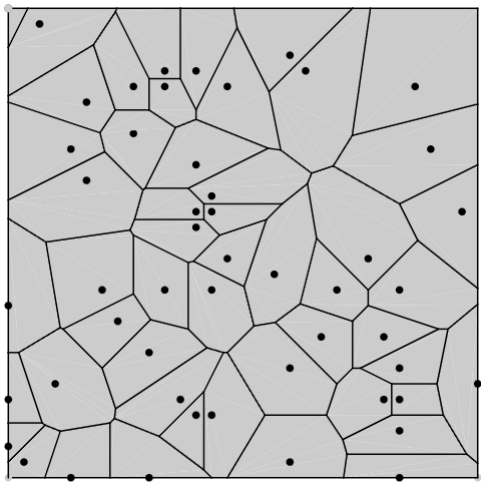


Figure 2.1: Illustration of a plane partitioned into convex polygons forming a Voronoi diagram using the black dots as generating points [11]

2.2.2 Random Walk

Random walk describes a set of approaches where an agent explores an environment by randomly changing the rotation and moving straight ahead. Thus, the agent is required to detect collision (i.e, colliding with a wall or an agent), able to rotate, and move straight ahead. There are several variations of random walks; Brownian motion, correlated random walk, Lévy Walk, Lévy taxis and random ballistic walk. A comparison between the aforementioned random walks showed that random ballistic walk (RBW) produced the best results in a simulation having a better coverage of the environment. In RBW, the agent continuously moves in a straight line until colliding with either an obstacle or another agent [9].

2.3 Problem Formulation

In this section, we describe the nature of the problem. Tools such as ARGoS (Section 2.1.1) and Gazebo (Section 2.1.2) provide an extensive library for simulating a wide range of robots in a large number of scenarios using a highly configurable setup process. In contrast, MAES (Section 2.1.3) simplifies this configuration process by only focusing on developing an interface for bringing two-dimensional grid-based exploration algorithms into a continuous space by providing a preconfigured robot and generation of environments. However, we were unable to find a similar framework that has the same functionality in a three-dimensional space.

"How do we make a framework for simulating three-dimensional exploration and coverage algorithms using one or more agents in continuous space?"

The variations of Random Walks are described in section 2.2.2. Here, the forefront Random Walk algorithm is the Random Ballistic Walk for exploration. This algorithm is also used as a baseline exploration algorithm for comparison with other algorithms in section 2.1.3. Section 2.2.1 proposed the Local Voronoi Decomposition algorithm making use of local information to explore. This algorithm is also used in section 2.1.3 together with Random Ballistic walk. Thus, the two aforementioned algorithms will be implemented to test the framework:

"How do we implement Random Ballistic Walk and Local Voronoi Decomposition in the framework?"

3. Design

This chapter will go through the design of each part of the simulator, starting with the cave generator, agents, and algorithms. This will culminate into an architecture and finally go through the design of the user interface.

3.1 Cave Generation

This section will describe the design principles for generating a cave in a three-dimensional space. The cave will be structured in a grid populated by voxels. A Cellular Automaton will be applied upon the grid to carve out spaces, i.e, removing voxels from the grid, so it resembles a cave-like structure.

3.1.1 Voxel

A voxel, derived from the word Volume and Pixel, represents a pixel with volume in a three-dimensional space and will be used as a wall in the grid (Figure 3.1). They are constructed using vertices, corners of a triangle, connected with lines. There are eight vertices in total forming twelve triangles; two triangles for each side of the voxel. The direction of the connection of the vertices, forming the triangle, decides whether the texture faces outwards or vice versa. Thus, for the texture to be outwards, the connections of the vertices have to be clockwise. For example, (0, 0, 0) → (0, 0, 1) → (1, 0, 0) → (0, 0, 0) and (0, 0, 1) → (1, 0, 0) → (0, 0, 0) → (0, 0, 1) to draw a texture on one side of the voxel.

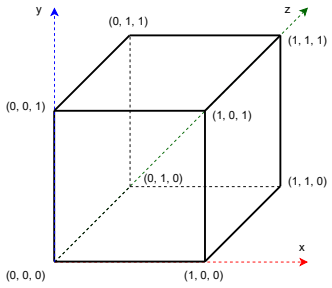


Figure 3.1: Illustration of a voxel

3.1.2 Grid

To keep track of the cave we will create a grid of voxels. The grid will be created based on a seed either generated or decided by the user. The seed will be implemented to ensure the ability to test different configurations on the same map. The grid will contain the state of the voxels within. The voxels can be three different states, *Unexplored*, *Wall*, or *explored*. When creating the grid, each voxel will have a predetermined chance of being either *full* or *empty*. The last state, explored, will be utilised during the simulation. The *empty* state will be replaced with the *explored* when an agent explores the area.

3.1.3 Cellular Automaton

Cellular Automation refers to the evolution of cells in a grid, based on a rule set and step-based intervals. Cellular automation is often used in a two-dimensional environment, but can be applied to three-dimensional environments as well [7].

For each step in the step-based intervals, the Cellular automaton iterates through all cells in an environment, observing each cell and its neighbours. Neighbourhoods can be described in many ways, however the two most common ones are Van Neumann Neighborhoods and Moore neighbourhoods, which can be observed in Figure 3.2. Van Neumann Neighbourhoods consist of the cells sharing sides, meaning in a two-dimensional grid it refers to the cells above, below, and on each side. Moore neighbourhoods consist of the same cells as Van Neumann, with the addition of the diagonal cells counted as neighbours as well [8].

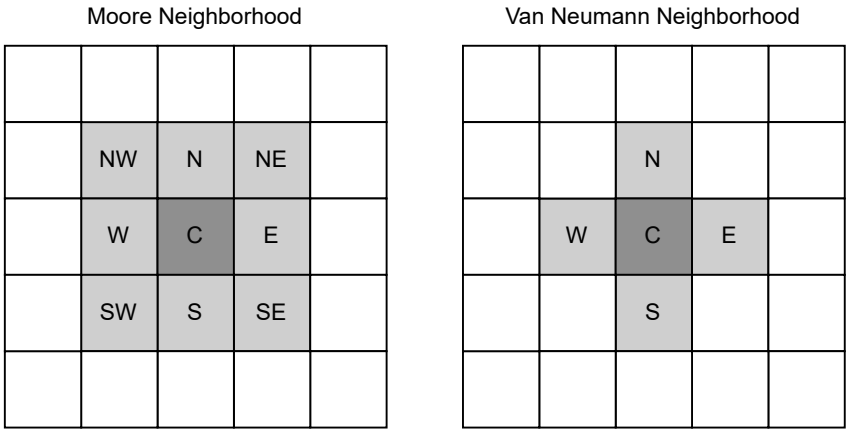


Figure 3.2: The figure displays the neighbours of C from the perspective of Moore Neighbourhood and Van Neumann Neighbourhood respectively

With the information gathered from that environment, a new environment is created

one cell at a time. Thus, the state of the cell either stays the same, changes from filled to unfilled, or vice versa. Once all cells of the first environment have been iterated through, and the second environment has been created, the new environment then replaces the first environment and the step is complete. If the first environment was changed as it was iterated through. The later cells would be adjusted based on changed information, which will result in a chain reaction of either filled or empty cells.

In this project, a cellular automaton will be applied in a three-dimensional grid-based environment. The starting environment will contain a three-dimensional grid containing the predefined amount of voxels. These voxels will be randomly distributed throughout the grid. The rule we would like to apply to the cellular automaton is:

- To fill the voxel if more than half of the neighbour spaces are filled
- To empty the voxel if half or less of the neighbour spaces are filled

This rule will ensure that over multiple iterations, the grid will smooth out and resemble a cave-like structure.

3.2 Agents

This section will go through the design of the agent. This includes the movement of the agent, how the agent sees the environment, and finally how the agents communicate with each other.

3.2.1 Movement of the Agent

The two main mediums for 3D exploration are underwater and in the air. For our simulation, it has been decided that underwater will be the target medium, and thus the interface for moving the agents will reflect what is possible underwater. For this purpose, the agents will move like a submarine. This means that the agents will be able to move forwards in relation to the current rotation of the agent, and turn in place or control the yaw to allow for control on the X- and Y-axis. Furthermore, the agents will be able to control the depth and vertical position, to allow for movement on the Z-axis.

We do not use Unity's provided physics tools for moving the agent due to the non-deterministic and unreliable nature of this. While Unity's physics system can be implemented in a way that makes it deterministic, unreliableness is still a factor. This presents a problem as we want to ensure that any movement the agents perform is executed exactly as instructed. For this reason, we have implemented our own basic functionality for moving the agent. This is a trade-off in the precision of movement over the realism of movement. Furthermore, this means that Unity's collision detection tools are not available to us, therefore, we have also implemented our own

functionality for this.

3.2.2 Sensing the Environment

To sense the walls of the environment, the exploration agents will be able to observe walls that are within line of sight within the agent's observation range.

Often, agents used for mapping purposes will use LIDAR (Light Detection and Ranging) or image processing technology to map their surroundings. However, as the purpose of MAES is to test exploration algorithms and not a robot's sensing abilities, this process is simplified.

Instead of simulating a LIDAR system, where a set number of evenly spaced beams are projected and their collision point detected resulting in a point cloud, a raycast is projected from the agent to each cell that is on the edge of the sphere around the agent with a radius of the observable range. This means that at least one raycast will pass through each cell that is within the observable range. Any cell that a raycast passes through is marked as explored and if the raycast is obstructed the cell at the position of the raycasts collision is marked as a wall. Furthermore, any cell the agent moves through is marked as covered

While Unity does have built-in raytracing tools, the need for finding each cell a raytrace intersects and the opportunity for optimising the raytracing for grid traversal, a customised raytracing solution will be developed based on the grid traversal methods proposed by Amanatides & Woo [1].

This means that as the range is increased, the resolution of the environment detection is not proportionally lowered as would be the case of simulating a LIDAR system. Instead the number of beams that are projected increases with the observation range as the amount of cells on the edge of the sphere increases. For example, when the agent has an observation radius of 5, 360 beams will be projected and when the agent has an observation radius of 6, 528 beams will be projected.

In addition to detecting the observable cells, the agent will also keep track of what other agents are within line of sight. Due to this implementation of sensing being computationally expensive, the agents take turns sensing each cycle, one at a time.

3.2.3 Communication

To be able to share the information an individual agent has gathered, the agents will need to be able to communicate with each other. The communication will involve

distributing the map of cells an agent has discovered while moving through the environment with other agents that are within view. This process can be generalised into two distinguishable processes: Keeping a register of which agents an agent has seen and merging an agents maps with the maps of agents it has seen.

The register of agents an agent has seen is directly available from the sensing as explained in section 3.2.2. The merging consists of iterating through every cell of the map and checking whether any unobserved cells have been observed by an agent from the register. This merging results in a computationally expensive process that gets more expensive as the environment gets bigger and more agents are added to the simulation.

3.2.3.1 Tags

In two-dimensional multi-agent exploration, dropping tags is also used as a communication option. These can be used to deposit information and allow agents to communicate with the tag as if it was a stationary agent. These tags will not be implemented as there is no logical way for these to work with three-dimensional movements in mind without falling to the ground or floating to the top in the environment we are simulating.

3.3 Algorithms

This section will go through the design choices and pseudo code for the RBW and LVD algorithms and the required changes to make them work in a three-dimensional space.

3.3.1 Random Ballistic Walk

When creating RBW, we want to mimic the algorithm the paper in section 2.1.3 used, albeit in a three-dimensional space meaning touching the floor or ceiling is also an option, as seen in Algorithm 3.1. To do this, each agent starts facing a random direction. When the simulation begins all agents will begin to move in their facing direction until they collide with a wall, ceiling, or floor. When colliding with a wall, the agent will rotate a random amount between 90 to 270 degrees on the vertical axis, and continue in that direction. When colliding with the floor or ceiling, the agent will pitch a random amount between 15 and 45 degrees perpendicular to the object that the agent collided with, and continue in the same direction as prior to the collision.

```
1 move
2 if the agent touches a wall then
3     rotate a random amount between 90 and 270 degrees
```

```
else if the agent touches the floor or ceiling then
    pitch a random amount between 15 and 45 degrees perpendicular to
    the object collided with
end if
go to 1
```

Algorithm 3.1: Random Ballistic Walk

3.3.2 Local Voronoi Decomposition

The algorithm for LVD can be seen in Algorithm 3.2 and is split up into two steps: Get Region Step and Navigation Step. The Get Region Step is the divide part of the strategy, where the closest voxels are assigned to an agent within the Field of View (FoV) of said agent to form a region. After that, the algorithm uses a greedy approach to explore the voxels within its region in the Navigation Step. The agent traverses to the closest unexplored voxel, and if there is more than one, it goes through them in an ordered list (e.g. north, east, south, west, up, down). The algorithm enters Search Mode if there are no unexplored voxels within the region.

```
Get Region Step //Divide part of the strategy
for all cells within view do
    if the voxel is nearer to itself as compared to all other agents
    within view then
        mark the voxel as within region
    end if
end for
Navigation Step //Conquer part of the strategy
if there is just one nearest unexplored cell that is within region
then
    move to it
else if there are more than one nearest unexplored voxel then
    move in an ordered list, eg. [North, East, South, West, Up, Down]
else
    Search Mode
end if
go to 1
```

Algorithm 3.2: Local Voronoi Decomposition from [5] configured to three-dimensional space

The algorithm for Search Mode can be seen in Algorithm 3.3 and is used for finding unexplored voxels by traversing to or swinging around occlusion points. Occlusion points are places that break the line of sight and the agent can therefore gain new information by visiting them. The Search Mode is exited when unexplored cells appear in the agent's region. The agent prioritises occlusion within points that have not yet been visited during Search Mode. In case all the occlusion points have already been visited, the agent traverses to the least recently visited. If there is only one

occlusion point and it has recently been visited, or no occlusion points at all, the agent traverses to the boundary of its region which does not coincide with any wall or obstacle. The agent stores all the occlusion points it has seen, as it needs to keep track of which is the least recently visited one, but it only considers the occlusion points within its FoV when figuring out where to go.

```
if at any time an unexplored voxel appears within region then
    exit Search Mode and go back to Algorithm 1
end if
if there is at least one occlusion point which has not been visited
during the Search Mode then
    move to the nearest occlusion point which has not been visited
else if all occlusion points have already been visited then
    move to the least recently visited occlusion point
else if there are no occlusion points or if the only occlusion point
has just been visited then
    move to the nearest Voronoi boundary which does not coincide with
    any wall or obstalce
end if
```

Algorithm 3.3: Search Mode from [5] configured to three-dimensional space

3.4 Architecture

Each computational cycle of MAES3D follows the principles of the Look Compute Move (LCM) model and is inspired by the architecture of MAES. In LCM, every cycle is divided into 3 distinct phases that are executed independently by each agent; the look phase, compute phase, and move phase. However, since this model is catered towards single-agent use cases and non-communicating agents, the phase of an agent communicating with other agents is added, creating the Look Compute Communicate Move (LCCM) model. The communication phase is placed after the compute phase to accommodate the possibility of allowing exploration algorithms to control the sent messages, and before the move phase to ensure that any agent seen in the look phase is still within line of sight when a potential message is sent.

The execution of the cycle will be handled by an overarching module, the simulation itself. In the simulation, each phase of the LCCM cycle is handled by its own distinct module. These modules consist of one exploration manager responsible for handling the look phase and one communication manager responsible for handling the communication phase. The move phase will be handled by each individual agent internally and the compute phase will be handled by the instance of an exploration algorithm that each agent contains. An illustration of the components of the simulation and how they are connected can be seen in Figure 3.3.

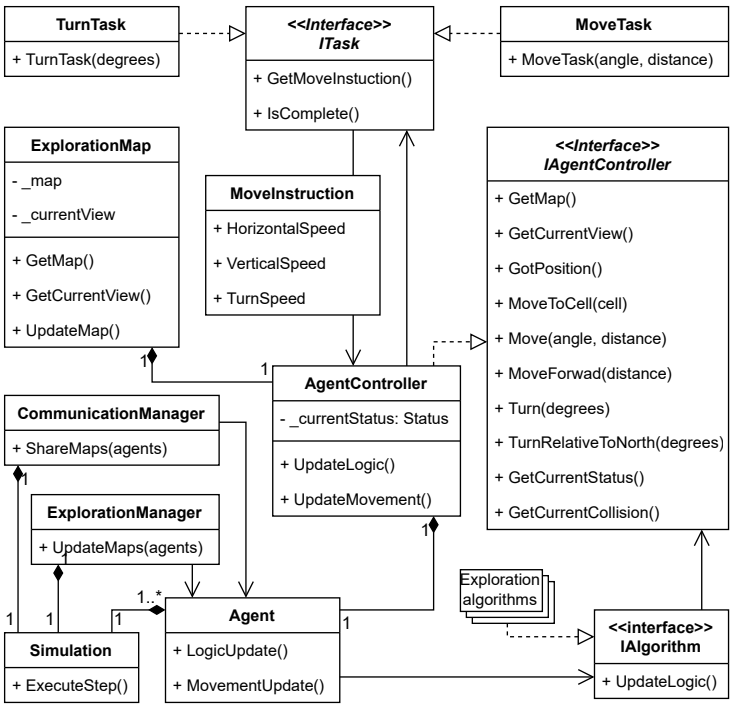


Figure 3.3: Class diagram of the simulation

3.4.1 Look phase

The first phase of the Look-Compute-Move-Communicate model is the Look phase. This phase senses the environment, the information which lays the foundation of what the exploration algorithm in question bases its decisions on in the compute phase. Sensing the environment refers to detecting the cells within view and observation range of the agent as well as whether these cells contain a wall or not. An illustration showing the components used in the look phase can be seen in Figure 3.4.

The process of the agents making observations is handled by the exploration manager. The exploration manager is a property of the simulation itself and is instantiated when a new simulation is built and thus decides how every agent senses its environment. As every agent has the same sensing capabilities, having one module that handles the sensing of every agent allows us to precompute and store the raytracing lines described in section 3.2.2 once, instead of for each agent.

Each cycle, the exploration manager scans the area around an agent, and any cell that is within range and within view of the agent has that cell marked as explored in its exploration map. The exploration map itself is a property of the individual

agent and keeps a record of not only every cell the agent has observed and covered but also whether it has observed a wall or a free space on this cell. Furthermore, it stores a representation of what cells are currently in its view. Both the record of every observed cell and the representation of the cells currently in view is stored in a three-dimensional array, like the environment map itself. Besides the cells in view, the exploration map also keeps track of what agents are within view of the agent.

The exploration map keeps track of everything the exploration algorithm needs to compute the next position of the agent. The exploration algorithm can access this through various methods provided by the agents `IAgentController` interface.

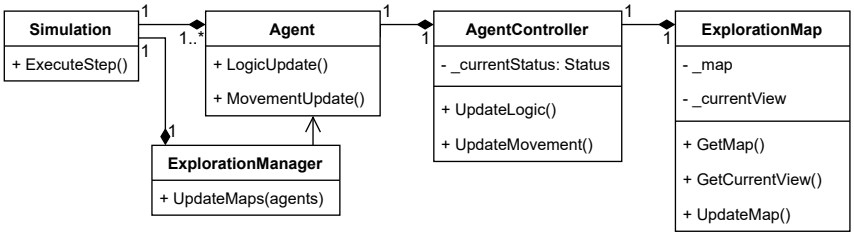


Figure 3.4: Class diagram of the components used in the look phase

3.4.2 Compute Phase

In the compute phase, an agent will make use of the information gathered in the look phase to make a decision on where to move in the move phase.
As MAES3D is intended to be a simulation for testing exploration algorithms, the compute phase does not have a distinct implemented process. Instead, a framework is set up that gives an exploration algorithm access to the information that is needed to make a decision about where to move next. An illustration showing the components used in the computer phase can be seen in Figure 3.5.

An exploration algorithm has access to its associated agent's information through the methods exposed by the `IAgentController` interface. This interface allows an exploration algorithm to access not only the current view of the agent, through the `GetCurrentView()` method, but also the entire map of what that agent has discovered so far through the `GetMap()` method. An exploration algorithm will use the information received from these methods to make a decision on where the agent should move next depending on that specific exploration algorithm.

When an exploration algorithm has decided on the agent's next location it will command the agent to move to that location. An exploration algorithm controls its associated agent through `IAgentController`s various move and turn tasks such as `MoveUp()`, `MoveDiagonal()`, and `Turn()` all of which take appropriate parameters for

deciding for example how far an agent should move. Furthermore, `IAgentController` also exposes a method, `MoveToCell()`, which strings together movement tasks that reach the desired cell.

The methods mentioned are implemented by an agent inheriting the `IAgentController` interface, defining how they are performed. Giving the exploration algorithm access to only the methods exposed by the interface opens the possibility of later defining agents that move or sense differently, by changing the implementation of the methods, while keeping the exploration algorithms compatible.

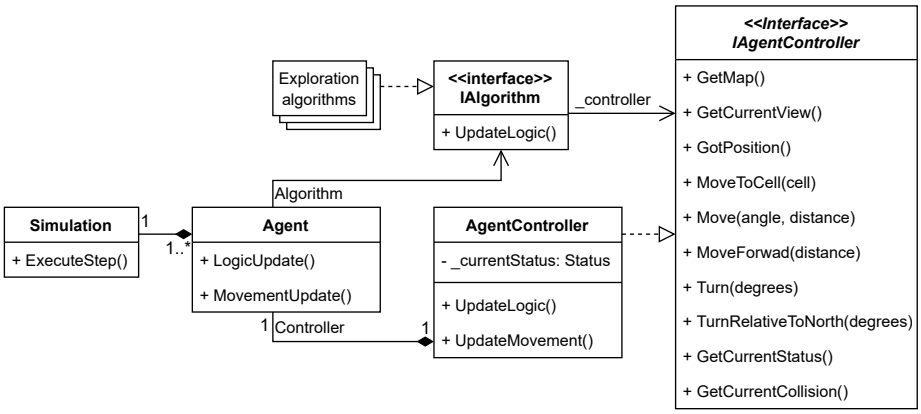


Figure 3.5: Class diagram of the components used in the compute phase

3.4.3 Communicate Phase

Sharing exploration maps refer to the inter-agent communication that may occur when two or more agents are within line of sight of each other as well as within communication range. Here agents will share the information about the environment gathered in previous cycles thus distributing knowledge among agents, thus giving each agent more information to use in the compute phase. An illustration showing the components used in the communication phase can be seen in Figure 3.6.

The process of the agents sharing information is handled by the communication manager. Like the exploration manager, the communication manager is a property of the simulation itself and is instantiated when a new simulation is built and decides how the agents share their independently collected maps.

As explained in section 3.2.3 the process of sharing maps can be split into two distinct processes: Keeping a register of which agents have seen which other agents and merging the maps of agents that have seen each other. Because of the computationally expensive nature of the merging, this process will be run at a fixed interval whereas

the computationally inexpensive process of keeping a register will be run every cycle

Every cycle, the communication manager uses the agents discovered in the look phase to keep a register of which agents have seen which other agents since the last time the communication manager called a map merge. When a map merge is called, the exploration manager updates each agent's map by merging it with the map of the agents that it has seen since the last merge was called.

Separating the functionality of the communication manager into one computationally inexpensive action that will be run every cycle and one computationally expensive action that will be run at a fixed interval ensures that the simulation will run at the desired speed most of the time. This, however, does mean that there is a chance that two agents will share data with each other that was acquired after they observed each other.

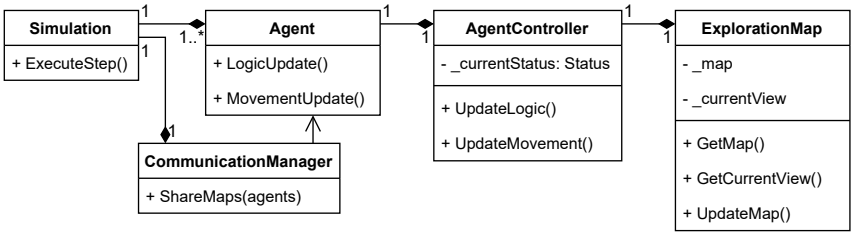


Figure 3.6: Class diagram of the components used in the communication phase

3.4.4 Move Phase

In the move phase, an agent will act upon the decision made in the compute phase.

When one of the previously mentioned move or turn methods is called from an exploration algorithm, the associated agent will have to act upon this command to move. How an agent reacts to one of these calls depends on the implementation of the method within an `AgentController` class however there is a basic structure of these methods. An illustration showing the components used in the move phase can be seen in Figure 3.7.

When a movement method is called from an exploration algorithm, the agent will translate that call into a task that is an instance of the `ITask` interface that will be stored by the agent. In each movement phase the agent will query the task for how it should move in this cycle using the `GetMoveInstruction` method that is exposed by the `ITask` interface. While the task itself keeps track of how far an agent needs to move and how much it has to turn, the `GetMoveInstruction` gives the agent information about how that movement is interpreted on a cycle-per-cycle basis in the

form of individual instructions that itself is an instance of the `MoveInstruction` class.

Each move instruction tells the agent how much to, in this cycle, move the agent in the relative x and y direction as well as how much the agent should turn.

Once a task is complete, meaning the agent has processed enough move instructions to exhaust the task of the remaining distance, the agent will discard the task and set its status to idle. This indicates to the exploration algorithm that the agent has completed the previous task it was given and is ready to take on a new task.

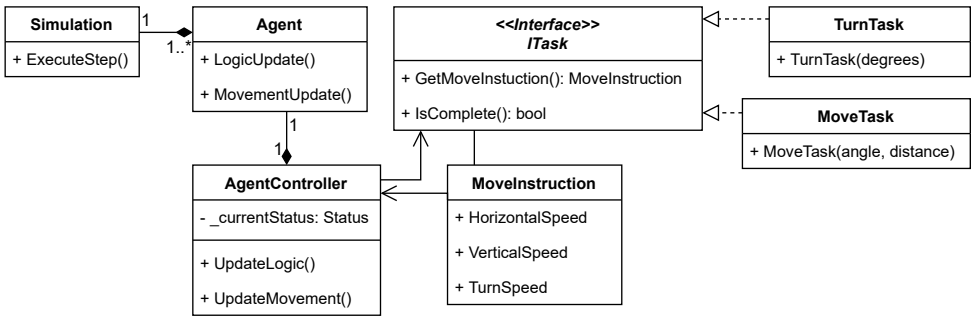


Figure 3.7: Class diagram of the components used in the move phase

3.5 User Interface

This section will talk about the design of the User Interface (UI) of the project. The UI will be utilised by the user to set the properties of the simulation. Furthermore, the UI should give some interactive functionality that the user can interact with for a more pleasant user experience.

The UI should be split up into three different segments:

1. Properties for the Simulation
2. Properties for the Map Generation
3. User interactivity

To give a clear representation of the UI, a wireframe is constructed. The wireframe in 3.8 should be considered while reading through the next 3 sections.

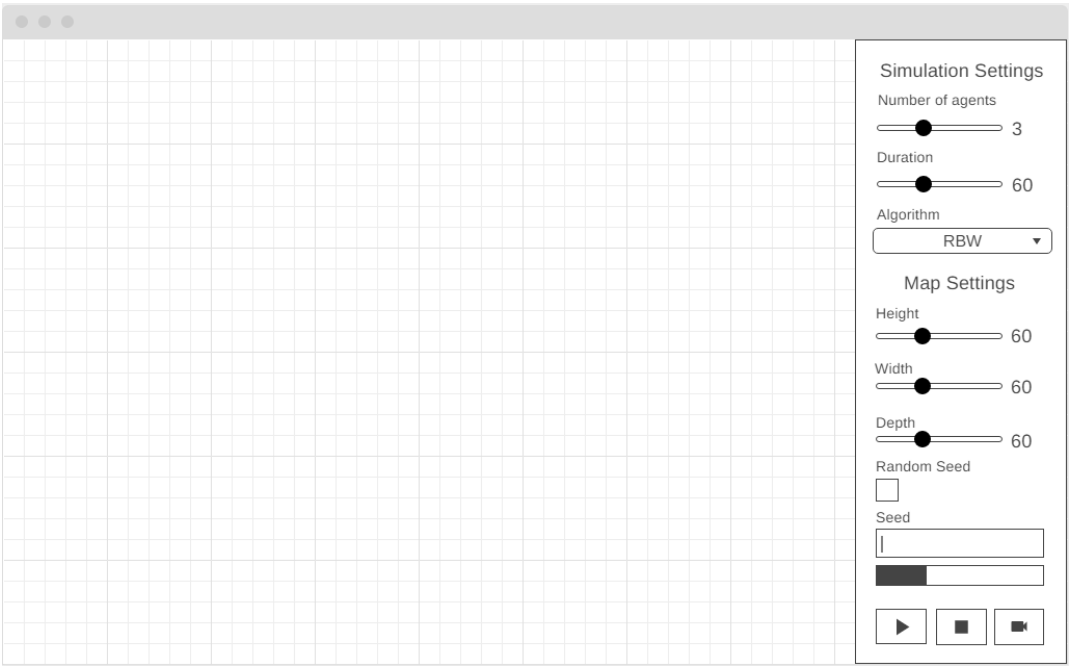


Figure 3.8: Illustration of the wireframe for the UI

3.5.1 Simulation

The purpose of the simulation segment is to give the user the possibility to choose the properties of the simulation. These properties are as follows:

- The number of agents
- The duration of the simulation
- The algorithm applied to the agents

The *number of agents* and the *duration* of the simulation will be represented by a slider, with boundaries of 1-10 agents and 1-120 minutes. These sliders give the user the opportunity for customising the simulation, while still keeping the simulation within a sensible manner. Selecting the *algorithm* will be represented by a drop-down menu of all the algorithms available to the simulator, with RBW as the default value. The reason behind choosing RBW as the default value is that it is the baseline algorithm you can use to compare with other algorithms.

3.5.2 Map Generation

The map generation segment will contain all the properties needed for creating a map suitable for simulation. These properties are as follows:

- Map Height
- Map Width
- Map Length
- Random seed indicator
- Seed input

The *height*, *width*, and *length* of the map will be represented by sliders with boundaries of 30 to 100. The boundaries are based on what values will give a possible explorable map, without creating a map big enough to impact performance. The *random seed indicator* will be represented by a checkbox, which generates a random seed used to create a random map if checked. Otherwise, it will enable the *seed input*, if not checked, which is represented by a text box where the user can insert a custom seed of their own.

3.5.3 User Interactivity

The user interactivity segment will contain buttons that the user can interact with as well as the progress of the simulation: These buttons are as follows:

- Start the simulation
- Force stop the simulation
- Switch viewpoint of the simulation

The *start simulation* button will begin a simulation with all predefined properties of both the simulation and the map generation. In the case of starting a simulation with undefined properties, the user will be able to press the *force stop* button, which will stop and delete the simulation. The last button, *change viewpoint* will loop between cameras to give the user a better overview of the simulation and make it possible to follow how the agents act within the environment.

4. Implementation

This chapter will go through the implementation of each part of the simulator. Each section will include code listings with some of the implementation and explanatory text to accompany it. Furthermore, an illustration of the product will be given showing the implementation is working.

4.1 Cave Generation

This section will describe how we implemented the topics of section 3.1 to generate a three-dimensional cave map.

4.1.1 Filling and smoothing the map-data

A three-dimensional boolean array is used to contain the data about each voxel, either a wall or an empty space represented by a true or false respectively. This array is then randomly filled through the `PopulateVoxelMap()` function displayed in Listing 4.1. On lines 3-7, `PopulateVoxelMap()` uses either a predefined seed or generates a seed based on the current time and date. On lines 9-11 three nested loops are created representing the height, width, and depth of the chunk. On lines 12-14, we set the current voxel to be a wall if it is at the edge of the map. If this is not the case, we then on lines 15-21 generate a number between 0 and 100 using the seeded random number generator. This number is then used to determine if the current voxel in `voxelMap[x,y,z]` should be a wall or an empty space. In our case, we found that creating a wall if the number was below or equal to 52 would lead to a cavelike structure after smoothing. This variable can, however, be increased to get a more close and more clustered cave structure, or decreased to get a more open and empty cave.

```
1 void PopulateVoxelMap () {
2
3     if (useRandomSeed) {
4         seed = System.DateTime.Now.ToString();
5     }
```

```
6
7       Random.InitState(seed.GetHashCode());
8
9       for (int y = 0; y < ChunkHeight; y++) {
10          for (int x = 0; x < ChunkWidth; x++) {
11              for (int z = 0; z < ChunkDepth; z++) {
12                  if (x == 0 || x == ChunkWidth - 1 || y == 0 || y ==
                    ChunkHeight - 1 || z == 0 || z == ChunkWidth - 1) {
13                      voxelMap[x, y, z] = true;
14                  }
15                  else {
16                      if (Random.Range(0, 100) <= 52) {
17                          voxelMap[x, y, z] = true;
18                      }
19                      else {
20                          voxelMap[x, y, z] = false;
21                      }
22                  }
23              }
24          }
25      }
26  }
```

Listing 4.1: The PopulateVoxelMap() function which filles the voxelMap based on a seed

To create a cave-like map based on the voxel map created prior, we need to smooth out the voxel map. To do this we implement a cellular automaton, seen in Listing 4.2. On line 3, we create a clone of `voxelMap`. On lines 5-7, we iterate through all voxels in `voxelMap`. On line 9, we check how many neighbors the current voxel has based on a three-dimensional *Moore Neighborhood*. On line 11-16, we change the current voxel to be full if there are more than 13 neighbours and empty if there is less than or equal to 13 neighbours. The reason behind 13 is that it is half the amount of maximum neighbours using the Moore Neighborhood method. Finally, on line 21, `voxelMap` is replaced with the clone created prior.

```
1   void SmoothenVoxelMap() {
2
3       bool[,,] tempMap = (bool[,,]) voxelMap.Clone();
4
5       for (int y = 1; y < ChunkHeight - 1; y++) {
6           for (int x = 1; x < ChunkWidth - 1; x++) {
7               for (int z = 1; z < ChunkDepth - 1; z++) {
8
9                   int neighbourWallCount = GetNeighbourWallCount(x, y, z);
10
```

```
11                if (neighbourWallCount > 13) {
12                    tempMap [x, y, z] = true;
13                }
14                else {
15                    tempMap [x, y, z] = false;
16                }
17            }
18        }
19    }
20
21    voxelMap = tempMap;
22 }
```

Listing 4.2: The PopulateVoxelMap() is a cellular automaton which smoothes out the array voxelMap

4.2 Agents

This section will display how the agent's movement is implemented, as well as the tasks an agent can fulfil.

4.2.1 Sensing the environment

The implementation of an agent's sensing of the environment can be seen in Listing 4.3. On line 5 the set of precomputed relative observation lines is iterated though. These observation lines are calculated at startup and are the lines that go from an agent to every cell in the edge of the sphere around the agent with a radius of the observable range as described in section 3.2.2. On line 6 the method `GetObservedCellsOnLine()` is called with the agent's position and the observation line as arguments. This method traverses the observation line relative to the agent's position using the methods proposed by Amanatides & Woo [1] and returns all the cells that intersected the line. Each of these observed cells, they are checked against the voxel map of the current configuration to see if they correlate to a wall or a free space and are marked accordingly in the agent's exploration map. If the cell is a free space it is also marked as visited in the exploration manager's internal map which is used to keep a global record of what cells have been explored in order to report statistics to the user.

```
1 private void UpdateMap(SubmarineAgent agent) {
2     agent.Controller.ExplorationMap.ResetCurrentView();
3     Vector3 agentPosition = agent.Controller.GetPosition();
4
5     foreach (ObservationLine observationLine in _observationLines) {
```

```
6         List<Cell> observedCells = GetObservedCellsOnLine(agentPosition,
          observationLine);
7
8         foreach (Cell cell in observedCells) {
9             if (_voxelMap[cell.x, cell.y, cell.z] == true) {
10                agent.Controller.ExplorationMap.UpdateCell(cell,
                  CellStatus.wall);
11            }
12            else {
13                agent.Controller.ExplorationMap.UpdateCell(cell,
                  CellStatus.explored);
14
15                if (_exploredMap[cell.x, cell.y, cell.z] == false) {
16                    _exploredMap[cell.x, cell.y, cell.z] = true;
17                    _exploredTiles++;
18                }
19            }
20        }
21    }
22
23    Cell currentCell = Utility.CoordinateToCell(agentPosition);
24    agent.Controller.ExplorationMap.UpdateCell(currentCell,
      CellStatus.covered);
25    SimulationSettings.progress = (_exploredTiles*100)/_explorableTiles;
26 }
```

Listing 4.3: Implementation of an agent sensing the environment

4.2.2 Example of a task

When the `Move()` method is called from an exploration algorithm, it sets the agent's current task to be an object of the class `MovementTask`. The constructor for the `MovementTask` class can be seen in Listing 4.4. This constructor takes two parameters from the call of the exploration algorithm that is the total distance the agent should move and the vertical angle of the movement as well as one parameter which is the speed the agent is able to move. This distance and angle are converted to a horizontal and a vertical distance as well as a direction modifier that indicates if the agent should move up or down on the y-axis. These values are then used to calculate a normalised vector that equates to the direction the agent should move in.

```
1 public MovementTask(float angle, float targetDistance, float speed) {
2     float angleInRad = Mathf.Deg2Rad * Mathf.Abs(angle);
3
4     float forwardDistance = targetDistance * Mathf.Sin((Mathf.Deg2Rad * 90)
```

```
    - angleInRad) / Mathf.Sin(90);
5   float verticalDistance = targetDistance * Mathf.Sin(angleInRad) /
    Mathf.Sin(90);
6
7   if (angle >= 0) {
8       _verticalDirectionModifier = 1;
9   }
10  else {
11      _verticalDirectionModifier = -1;
12  }
13
14  _directionVector = new Vector2(forwardDistance, verticalDistance *
    _verticalDirectionModifier).normalized;
15  _targetDistance = targetDistance;
16
17  _speed = speed * Time.fixedDeltaTime;
18  _traveledDistance = 0;
19 }
```

Listing 4.4: Construct for the `MovementTask` class

Every cycle the agent calls the `GetInstruction()` method of its current task. The `GetInstruction()` method for the `MovementTask` class can be seen in Listing 4.5. This method returns an instance of the class `MoveInstruction` that indicated how much the agent should move in this cycle. On line 4 it is checked whether the movement of this cycle would go past the target distance. If not, a `MoveInstruction` is returned that indicates that the agent should move along the direction vector at full speed. If the movement would go past the target distance, a `MoveInstruction` is returned that indicates that the agent should move along the direction vector only the remaining distance of the task.

```
1  public MoveInstruction GetInstruction() {
2      float _remainingDistance = _targetDistance - _traveledDistance;
3
4      if (_traveledDistance + _speed < _targetDistance) {
5          _traveledDistance += _speed;
6          return new MoveInstruction(_directionVector.x * _speed,
           _directionVector.y * _speed, 0);
7      }
8      else {
9          _isComplete = true;
10         return new MoveInstruction(_directionVector.x * _remainingDistance,
           _directionVector.y * _remainingDistance, 0);
11     }
12 }
```

Listing 4.5: The `GetInstruction()` function of the `MovementTask`

4.2.3 Applying movement to the agent

The `ApplyMovement()` is responsible for making an agent move according to the current task. The method takes a `MoveInstruction` and applies it to the agent. `MoveInstruction` is a simple object consisting of three properties: `VerticalSpeed`, `HorizontalSpeed` and `TurnSpeed` each of which tell the robot how much to move in the current cycle depending on the current task. On line 3 the horizontal and vertical speeds of the instruction are translated into a velocity vector in world space. On line 5 a sphere cast is projected along the velocity vector starting from the agent's position. This sphere cast returns true if a sphere would intersect another object, such as the map and other agents, if moved from the position of the agent along its velocity. This sphere cast works as collision detection. If a collision is detected the code on lines 6-19 is executed and lines 7-15 sets the collision type of the agent according to the normal vector of the surface where the collision was detected and then cancels the current task of the agent. If a collision is not detected the code on lines 22-24 is executed where the velocity is added to the agent's position and the agent is rotated according to the turn speed of the instruction.

```
1  private void ApplyMovement(MoveInstruction instruction) {
2
3      Vector3 vel = _transform.forward * instruction.HorizontalSpeed +
       _transform.up * instruction.VerticalSpeed;
4
5      if (Physics.SphereCast(GetPosition(), 0.4f, vel.normalized, out
       RaycastHit hit, vel.magnitude)) {
6          CollisionType colType;
7          if (hit.normal.y < Mathf.Sin(Mathf.Deg2Rad * -45)) {
8              colType = CollisionType.ceiling;
9          }
10         else if (hit.normal.y > Mathf.Sin(Mathf.Deg2Rad * 45)) {
11             colType = CollisionType.floor;
12         }
13         else {
14             colType = CollisionType.wall;
15         }
16
17         _currentCollision = colType;
18         _currentTask = null;
19         _currentStatus = Status.Idle;
20     }
```

```
21      else {
22          _transform.position += vel;
23          _transform.rotation *= Quaternion.Euler(0, instruction.TurnSpeed, 0);
24          _currentCollision = CollisionType.none;
25      }
26  }
```

Listing 4.6: Implementation of applying movement to an agent

4.3 Algorithms

This section will go through how the agent tasks are utilised to replicate three-dimensional exploration algorithms.

4.3.1 Random Ballistic Walk

The implementation of the logic for RBW can be seen in Listing 4.7. The method uses a switch-statement for determining what to do in each scenario. The agent changes the angle of direction if it collides with the floor (15 to 45 degrees) or ceiling (-15 to -45 degrees). Secondly, the agent rotates between 90 and 270 degrees if it collides with a wall. Otherwise, the agent defaults to continue moving in the direction it already had.

```
1  public void UpdateLogic()
2  {
3      if (_controller.GetCurrentStatus() == Status.Idle)
4      {
5          switch (_controller.GetCurrentCollision())
6          {
7              case CollisionType.floor:
8                  _directionAngle = 45; // Random.Range(0, 90)?
9                  _controller.Move(_directionAngle, 0);
10                 break;
11             case CollisionType.ceiling:
12                 _directionAngle = -45; // Random.Range(0, -90)?
13                 _controller.Move(_directionAngle, 0);
14                 break;
15             case CollisionType.wall:
16                 _controller.Turn(Random.Range(90, 270));
17                 break;
18             default:
19                 _controller.Move(_directionAngle, 0);
20                 break;
21         }
22     }
```

```
23 }
```

Listing 4.7: Implementation of the logic for Random Ballistic Walk

4.3.2 Local Voronoi Decomposition

The implementation of the logic for LVD can be seen in Listing 4.8. The `destination` variable on line 5 is a `Cell` and stores where the agent should traverse to. The variable `visibleAgents` on line 7 stores the other agents within the FoV of the agent itself, the variable `currentView` on line 8 stores the cells currently visible from the agent's position, and the variable `map` on line 9 stores the agent's map. The agent enters exploration mode on line 11. Here, the agent tries to find a cell within its own region. The agent enters search mode on line 15 if the destination is still `null` on line 13 after exiting exploration mode. Here, it tries to find an occlusion point to traverse to. At last, the agent moves to the destination on line 25 if the destination is not `null` on line 23.

```
1  public void UpdateLogic()
2  {
3      if (_controller.GetCurrentStatus() == Status.Idle)
4      {
5          Cell destination = null;
6
7          List<Vector3> visibleAgents = _controller.GetVisibleAgentPositions();
8          List<Cell> currentView = _controller.GetVisibleCells();
9          CellStatus[,,] map = _controller.GetCurrentView();
10
11         destination = ExplorationMode(currentView, map, visibleAgents);
12
13         if (destination == null)
14         {
15             destination = SearchMode(currentView, map, visibleAgents);
16
17             if (_occlusionPoints.ContainsKey(destination))
18             {
19                 _occlusionPoints[destination]++;
20             }
21         }
22
23         if (destination != null)
24         {
25             _controller.MoveToCell(destination);
26         }
27     }
```

```
28 }
```

Listing 4.8: Implementation of the logic for Local Voronoi Decomposition

4.4 The Application

An illustration of the application can be observed in Figure 4.1. The image shows 5 agents moving around in a cave with the dimensions of 50 × 50 × 50 voxels. Each agent's trajectory throughout the cave is represented using a white line. The right-hand side of the figure shows the UI containing all the properties (e.g. height, width, and depth) for creating the simulation, as well as the duration and progress of the currently running simulation.

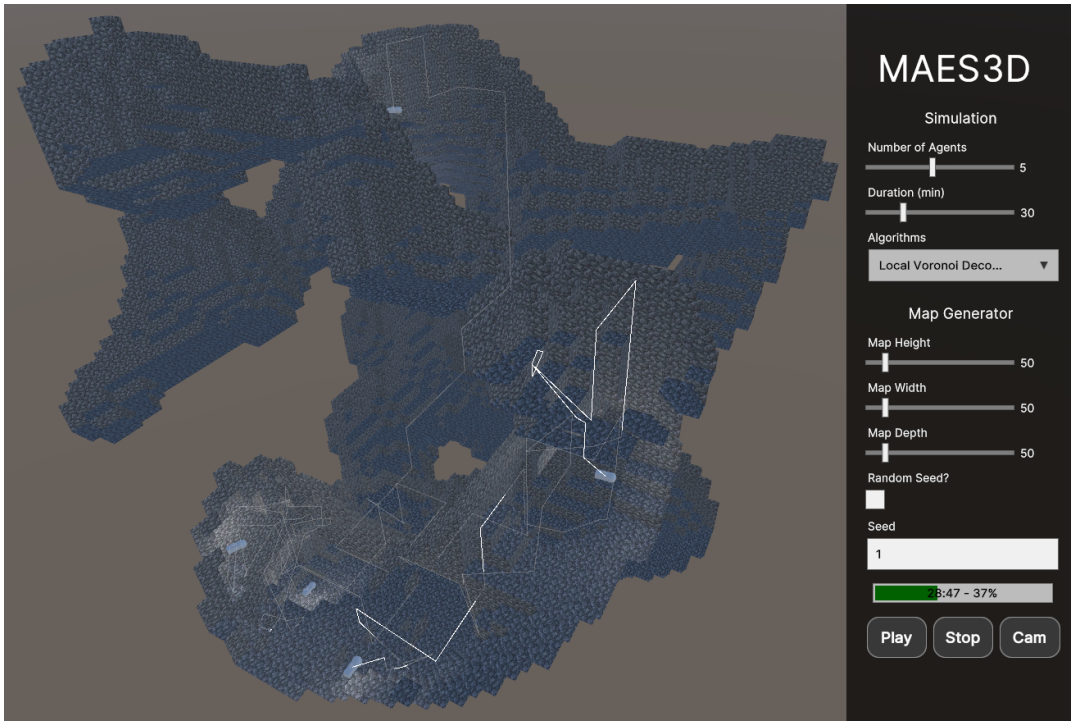


Figure 4.1: Illustration of agents exploring a cave on the left side constructed from the UI on the right side

5. Experiment

This chapter entails the setup used for the experiments, the results for LVD and RBW using exploration time as a metric, and a comparison of the two algorithms.

5.1 Experiment Setup

For the experiment, we want to create a replicable test environment, where we can test both the RBW and LVD algorithms. This environment will consist of a map generated with the dimensions of 50 × 50 × 50 voxels, and a predefined seed for each simulation. The simulations will run for a duration of 30 minutes using 5 agents. The agents will spawn in the voxel nearest to (0, 0, 0) with enough space to form a cross pattern. The performance matrix we use for evaluation is the exploration time. The exploration time is the time it takes to explore all the free voxels in an environment at least once; the total number of free voxels for each seed can be seen on Table 5.1. A snapshot of the accumulated exploration is taken every 10 seconds and is used for comparison.

Seed	1	2	3	4	5
Free voxels	13,871	11,976	16,261	12,891	14,787

Table 5.1: Free voxels in the seeds used for the experiments

5.2 RBW Results

The results for RBW can be seen on Figure 5.1. The Y-axis shows the exploration in percentage and the X-axis shows the time in minutes. The algorithm reached an exploration of 91%, 94%, 92%, 98%, and 94% for each corresponding seed resulting in an average of 93.8% exploration. The algorithm has a hard time navigating through narrow hallways or dead ends due to the randomness when hitting a wall or the floor and ceiling, which Seed 1 mostly comprised of and is why it performed the worst. Furthermore, in Seed 4 the agents spawn close to a large room without any small corridors nearby, this results in the agents discovering a lot of the cave quickly at first and then evening out at the end.

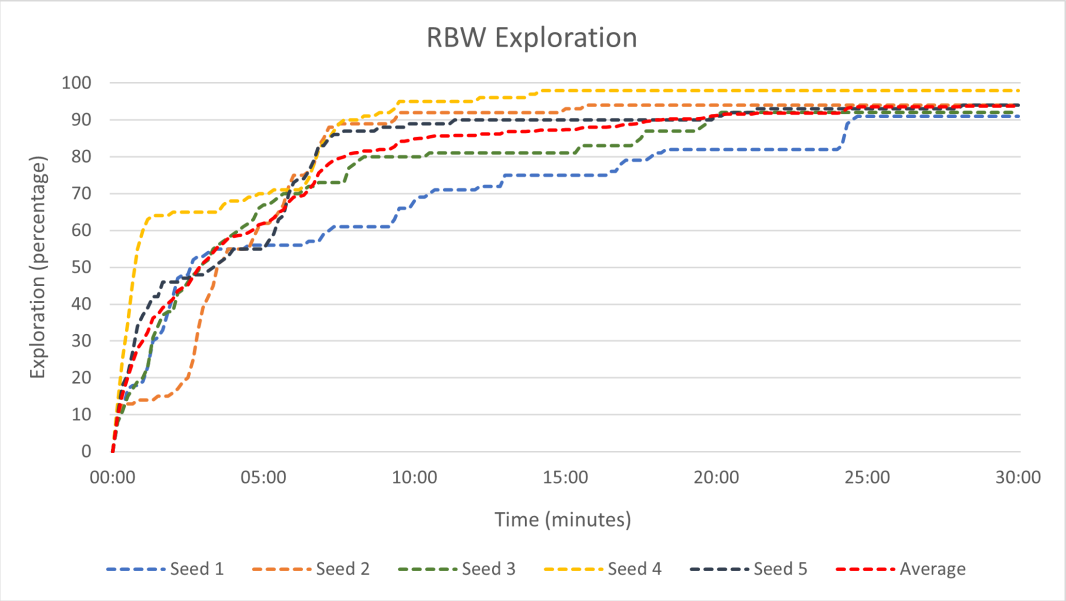


Figure 5.1: Results of the RBW algorithm in 5 different scenarios

5.3 LVD Results

The results for LVD can be seen on Figure 5.2. The Y-axis shows the exploration in percentage and the X-axis shows the time in minutes. The algorithm could not find the remaining free voxels within 30 minutes for Seeds 3 and 5. Here, the algorithm spent the remaining 11 minutes and 20 seconds for Seed 3, and 13 minutes and 40 seconds for Seed 5 in search mode trying to find the last free voxels. The algorithm has a hard time navigating out of dead ends due to the limited range and number of occlusion points. Here, we see the same problem for Seed 1 as in the previous section 5.2 and is further discussed in section 6.1. The algorithm, like RBW, performed the best in Seed 1 because of the openness of the cave and the lack of dead ends.

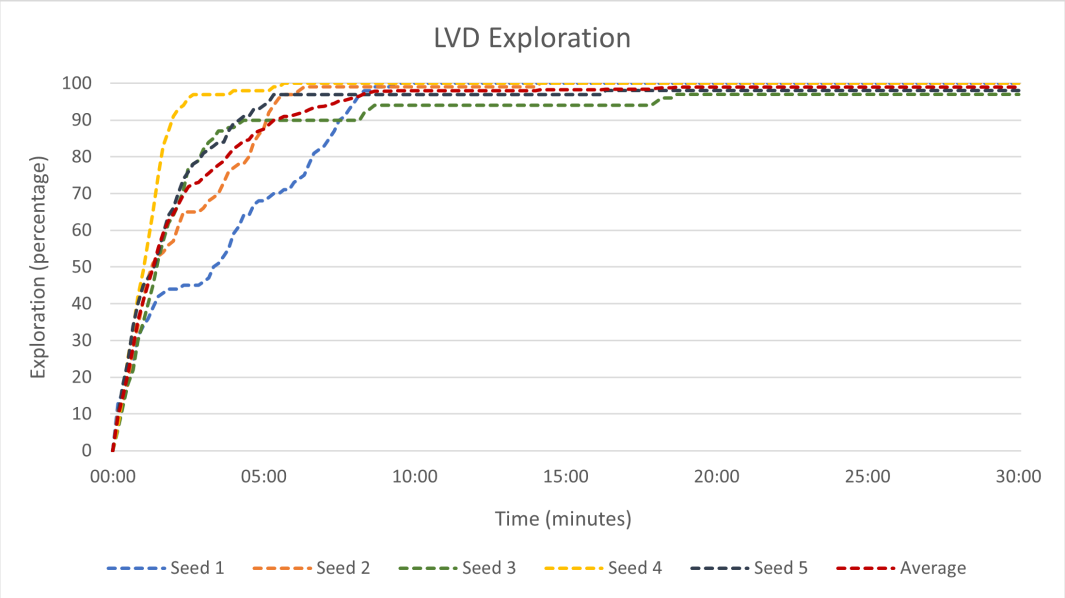


Figure 5.2: Results of the LVD algorithm in 5 different scenarios

5.4 Comparison of LVD and RBW

A comparison of the average exploration over 30 minutes for RBW and LVD can be seen on Figure 5.3 for the five seeds. The Y-axis shows the exploration in percentage and the X-axis shows the time in minutes. LVD reached an average exploration of 99% and RBW reached an average exploration of 93.8% at 30 minutes. LVD reached 99% at 18 minutes and 40 seconds whereas RBW reached 93.8% at 28 minutes and 10 seconds. When looking at the time it took for both algorithms to reach 90% exploration, it took LVD 5 minutes and 20 seconds, and RBW 18 minutes. In terms of exploration, LVD is faster than RBW by a wide margin.

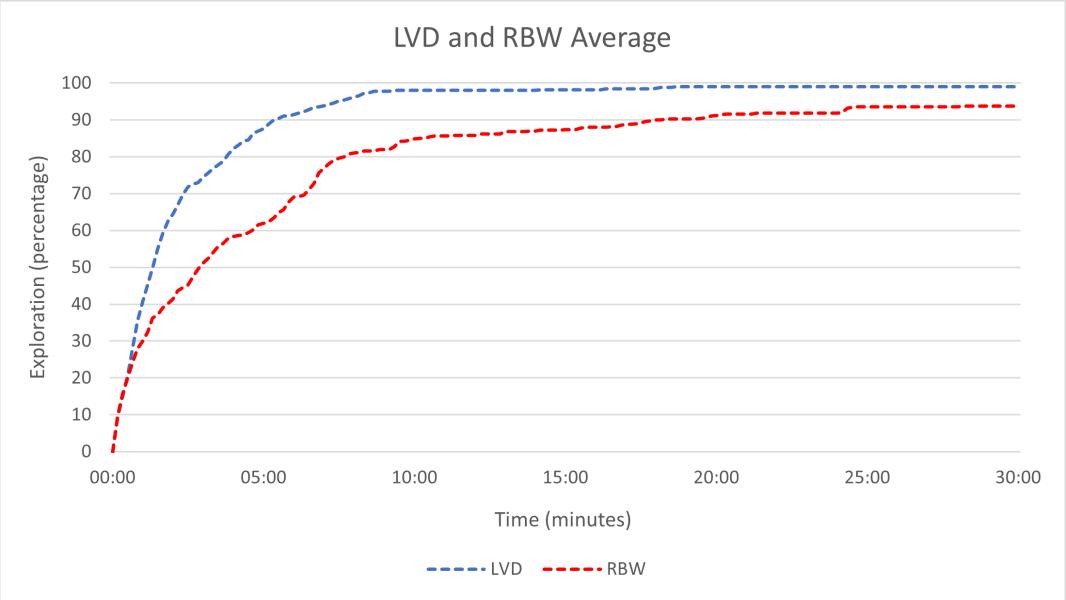


Figure 5.3: A comparison of the results for LVD and RBW

6. Discussion

In this chapter, we will discuss the shortcomings of our solution. These shortcomings are issues that occur when generating maps at a certain size, the movement of the agent, and what is lacking when comparing the algorithms.

6.1 LVDs Relevance in Three-dimensional Space

This section will discuss problems erupting when implementing LVD in a three-dimensional environment. The paper [5] does not explicitly state the range of the observable region of an agent. However, the figures show illustrations of agents having an infinite range. In MAES3D, the radius of the observed area is finite. Therefore, the agents have to be within a finite range of each to be a factor in the decision-making of which voxel the agent will move to, as the closest agent will control it. The agents used a radius of 5 voxels in the experiments, meaning agents further than 5 voxels away did not have an impact on the agent's movement.

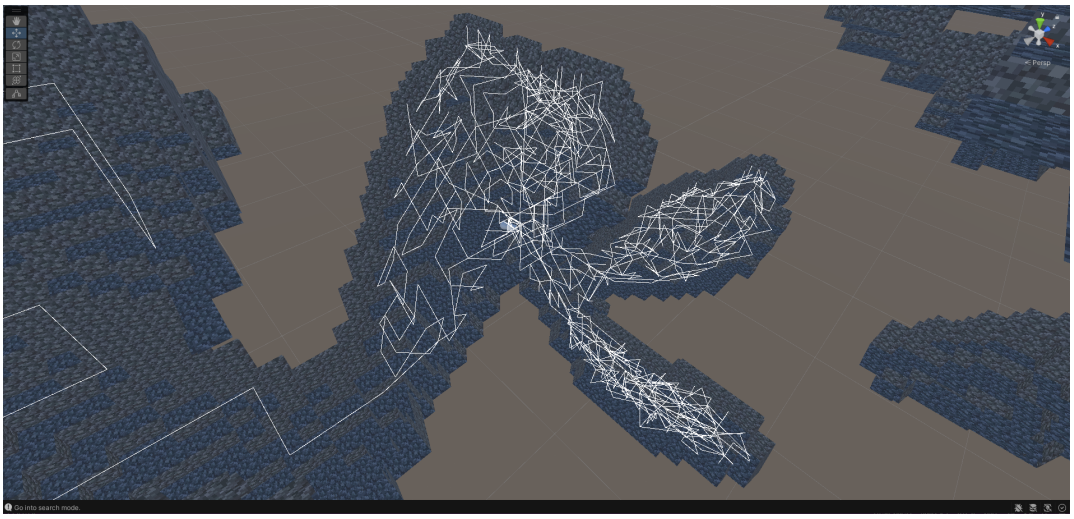


Figure 6.1: Trajectory of an agent traveling to occlusion points in a dead end

Another problem that often occurred can be seen on Figure 6.1. The agent's trajectory can be seen when it encounters a dead end. Here, the agent continuously visits the many occlusion points within view until it finds a way out. This results in the agent spending most of the time traversing occlusion points along the wall, instead of exploring new areas. The excess amount of occlusion points is a byproduct of the way the cave is structured using cubes; creating a lot of points that break the agent's line of sight. [2], solves this issue by combining occlusion points within some distance of each other to be the same.

6.2 Measurement of Progression

In 2.3 it was mentioned that both exploration and coverage algorithms should be able to be simulated within our framework. However, due to time being our most pressuring factor, it was decided not to fully implement coverage as a means of measuring the progression of the simulation. To measure the progression we currently use a factor of all explored voxels divided by all explorable voxels. Since we intended to implement coverage, we are also able to compute the factor of all covered voxels by all coverable voxels, which is the same as explorable voxels. However, since the simulator does not have any coverage algorithms implemented, we did not implement a way of showing the percentage of covered voxels. If coverage algorithms were to be implemented, all the requirements to implement coverage as a measurement of progression is already implemented.

6.3 Dynamic Map Generation

In the current implementation of the program, the map is generated to fit $100 \times 100 \times 100$ voxels. This results in the implementation working as intended for this scenario. However, reductions in the dimensions of the map result in the map being generated with fewer coherent tunnels and more isolated holes.

The worst-case scenario is if the map gets too small, the smoothing algorithm in Listing 4.2 will turn the map into a solid block with no holes or tunnels at all. A possible solution to this problem would be making a dynamic map smoothing algorithm that changes the amount of smoothing iterations, and the number of needed neighbors for a smoothing operation to be executed.

6.4 Agent Movement

The current implementation of the agent movement simulates the movement of a submarine. This means, as explained in section 3.2.1, that the pattern of the agent's movement is based on the *look*, *compute*, *communicate*, *move* model explained in

section 3.4, where the *move* part is based on the agent turning to face the direction it is supposed to go, and then moving in a forward motion towards the target destination. This method of moving results in the agent spending time turning from facing the position it is looking to face the direction of the target destination. This is time spent not progressing the exploration at all since turning is happening while the agent is standing still.

To prevent spending a lot of time turning, the movement of the agent could be implemented to simulate a drone instead of a submarine. This implementation of movement will focus on moving the agent along the cardinal directions and the altitude represented respectively by the (x,z) plane, and along the y-axis.

7. Reflections

In this chapter, we will reflect on some of the decisions taken throughout the project in relation to Unity and Github.

7.1 Unity

In this project, it was decided to use Unity as the engine for our simulator. Unity is an engine primarily used for game development in two- and three-dimensional space. We chose Unity not only because of our prior experience with C# and that the MAES project was created in Unity. But also because of the built-in functionality such as sphere casting and UI Builder.

7.2 GitHub

GitHub is a repository hosting service, which allows multiple developers to contribute, change, view and use public code written by other developers. GitHub was used as our repository for the project. GitHub provided the project with an overview of the project and changes made within it. Furthermore, GitHub gave us the ability to version control our project.

8. Conclusion

The problem analysis in chapter 2 analyses the problem of simulating three-dimensional exploration and coverage algorithms. This was done by looking at three other simulators namely Gazebo, ARGoS, and MAES. Furthermore, the algorithms *Random Ballistic walk* (RBW) and *Local Voronoi Decomposition* (LVD) were explored, which ultimately resulted in the two following problem formulations:

"How do we make a framework for simulating three-dimensional exploration and coverage algorithms using one or more agents in continuous space?",

and

"How do we implement Random Ballistic Walk and Local Voronoi Decomposition in the framework?".

To solve said problem formulation, design choices for the agent's abilities, the environment generation, the user interface, and a three-dimensional version of RBW and LVD were specified in chapter 3 resulting in an architecture based on the look-compute-communicate-move model for agents described in section 3.4. Based on the design and architecture, an implementation was created and explained in chapter 4. In chapter 5 the solution was tested on five different cave layouts using both RBW and LVD. Furthermore, the results were compared and discussed. The shortcomings of the solution were later discussed in chapter 6.

Based on the problem formulation, the experiments, and the discussion, we conclude that a framework for simulating exploration algorithms was created. This model is able to simulate every phase of the LCCM model by specifying agent capabilities and an interface for exploration algorithms. Using the framework both coverage and exploration algorithms can be executed as shown by implementing RBW and LVD. Furthermore, we can also conclude from the experiments that our implementation of LVD outperforms RBW in a three-dimensional space.

9. Future Work

This chapter will display some of the ways our project can be further developed or improved.

9.1 Minor Bugs

This section will talk about a few minor bugs that were not a priority to be fixed within the time restrictions of the project. These bugs will not render the program unusable as a whole, however, some scenarios are not able to be simulated due to these bugs

9.1.1 Map Size

As mentioned in section 6.3, changing the default size of the map too much might result in the map not being suited for a simulation. An adaptive implementation of `SmoothenVoxelMap()` observed in listing 4.2 should include different implementations based on different map sizes.

9.1.2 Unintentional Movement of Agents

Sometimes during the simulation of the LVD algorithm, we observe an agent moving back and forth within the same voxel space. An example of this can be observed on Figure 9.1. The figure illustrates the path of an agent with white lines. Within the red circle, two small lines can be observed protruding out. These two small lines represent the path the agent took when it was continuously moving back and forth.

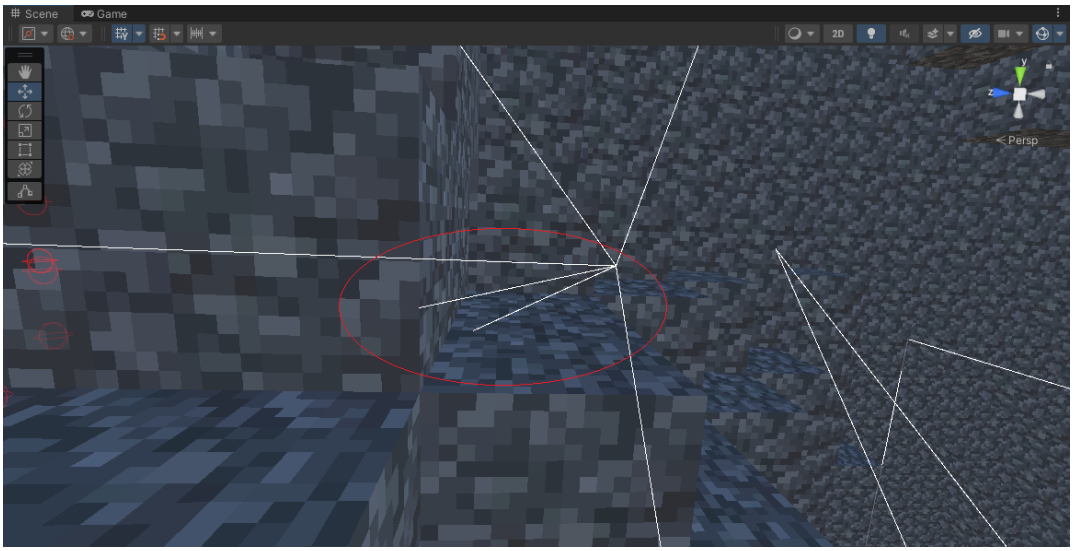


Figure 9.1: The figure displays the case where an agent moves back and forth in place during the LVD simulation

Another unintentional movement happens under the use of the RBW algorithm. Even though we use sphere casting to prevent agents from leaving the boundary of the map, we still sometimes observe agents leaving the map resulting in the simulation crashing.

9.2 Agent Movement

Section 6.4 talks about the current implementation of the agent movement, and how it requires the agent to turn in different directions before being able to move. An adaptive implementation to the agent's movement would be to, in contrast to moving like a submarine, implement a movement based on a drone's moving patterns. This implementation will reduce the amount of time used on arbitrary movements such as turning, thereby improving the overall performance of the simulation. This alternative agent should be developed to implement the same interface as the current agent does. This would ensure that any exploration algorithm developed for an agent with one type of movement would also work on any other type of agent.

9.3 Sensing the Environment

As described in section 3.2.2, the sensing of the agents is based on raytracing to every cell in the edge of a sphere with the radius of the agent's observation range. This ensures that every cell within the observation range is accounted for in every cycle. However, this also means that as the observation range gets bigger, so does

the computational expense of the look phase which ultimately limits the possible observation range without lowering the performance of the simulation. Developing an additional method for sensing the environment that works in a similar way to a LIDAR system would, while potentially lowering the precision of the sensing, allow an agent to sense its environment with a far greater, or infinite range

As a bi-product of the sensing being handled by the exploration manager, the LIDAR-based approach can be developed as an alternative exploration manager that interacts with the rest of the simulation in a similar way to the current exploration manager. This would allow a user to easily switch between the two approaches when setting up a scenario.

9.4 Communication

The current implementation of inter-agent communication requires agents to be within line of sight of each other to share their knowledge of the environment with each other. Providing the user with more options for how the communication is performed would allow testing exploration algorithms for agents that have different communication capabilities. This configuration could include whether the agents use global communication, resulting in one single shared exploration map, or local communication as the current implementation uses but with greater configuration options such as deciding the maximum range of a message and whether it requires line of sight or can partially or fully blocked by the environment.

Furthermore, allowing an agent to send and receive customized messages, instead of it being limited to map sharing, would open the possibility of implementing exploration algorithms that uses more specialised inter-agent communication to delegate and synchronize the exploration.

9.5 Coverage

As mentioned in section 6.2 the current implementation of the project is only taking exploration into account when displaying the progress of the simulation. With the intention of simulation both exploration and coverage algorithms, an adaptation of the implementation could be to add a choice between displaying either coverage or exploration as the metric of progression. Since an algorithm is focused on optimising either coverage or exploration, the project could consider the intended goal of the algorithm, and then display the progress of the simulated algorithm based on the aforementioned goal.

9.6 Runtime User Interface

The main function of the current interface is to configure the parameters of a simulation. Developing a secondary user interface focused on providing a user with information about the current state of the simulation while it is running. This interface could consist of graphs representing exploration and coverage over time, as well as agent-specific information such as the current task, target position of the task, amount of exploration done by that single agent, etc. Additionally, the exploration algorithm could provide information to this user interface making debugging the exploration algorithms easier as well as displaying information that is algorithm specific, such as whether an agent is in search mode or exploration mode for LVD.

9.7 Visualisation

Currently, the only visual representation of the progress of the exploration is in the form of the percentage of the explorable cells that have been explored and a line indicating where agents have moved within the environment. One addition that would greatly increase the visual representation of the exploration progress would be highlighting the cells that have been explored. In the case of walls, this could be done by changing the colour of the cells that have been explored. In the case of free cells, the problem is harder to solve as colouring the otherwise transparent cells would result in the explored cells obstructing the visibility of the environment itself.

Bibliography

[1] John Amanatides and Andrew Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing". In: *Proceedings of EuroGraphics* 87 (1987).

[2] Malte Z. Andreasen et al. *Comparison of Online Exploration and Coverage Algorithms in Continuous Space*. On-line: https://ieeexplore.ieee.org/abstract/document/5152829?casa_token=9TK4M5C2-6MAAAAA:g9GDUmiKGZS8Y3DxtitoOjT7srPF_ovDc7npPagAUOEQxa40wsn74VDTEOWxcTyYmEl2OGFtVHw. 2022. Accessed: 29/11/2022.

[3] ARGoS. *ARGoS: Large-scale robot simulations*. On-line: https://www.argos-sim.info/. 2023. Accessed: 18/01/2023.

[4] Ettore Ferranti, Niki Trigoni, and Mark Levene. "Brick & Mortar: an on-line multi-agent exploration algorithm". In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. 2007, pp. 761–767. DOI: 10.1109/ROBOT.2007.363078.

[5] James Guo Ming Fu, Tirthankar Bandyopadhyay, and Marcelo H. Ang. *Local Voronoi Decomposition for multi-agent task allocation*. On-line: https://ieeexplore.ieee.org/abstract/document/5152829?casa_token=9TK4M5C2-6MAAAAA:g9GDUmiKGZS8Y3DxtitoOjT7srPF_ovDc7npPagAUOEQxa40wsn74VDTEOWxcTyYmEl2OGFtVHw. 2009. Accessed: 29/11/2022.

[6] Gazebo. *Gazebo*. On-line: https://gazebosim.org/home. 2023. Accessed: 18/01/2023.

[7] Nathaniel Johnston. *Cellular automaton*. On-line: https://conwaylife.com/wiki/Cellular_automaton. 2022. Accessed: 19/01/2023.

[8] Nathaniel Johnston. *Game of Life Explanation*. On-line: https://conwaylife.com/wiki/Neighbourhood. 2022. Accessed: 19/01/2023.

[9] Miquel Kegeleirs, David Garzón Ramos, and Mauro Birattari. "Random Walk Exploration for Swarm Mapping". In: *Towards Autonomous Robotic Systems* (2019), pages 211–222. ISSN: 0302-9743. DOI: 10.1007/978-3-030-23807-0. URL: https://doi.org/10.1007/978-3-030-23807-0.

[10] Schranz M et al. “Swarm Robotic Behaviors and Current Applications”. In: *Frontiers in Robotics and AI* (2020). ISSN: 0302-9743. DOI: 10.3389/frobt.2020.00036. URL: https://doi.org/10.3389/frobt.2020.00036.

[11] Inc. Wolfram Research. *Voronoi Diagram*. On-line: https://mathworld.wolfram.com/VoronoiDiagram.html. 2022. Accessed: 19/01/2023.

[12] Jie Zhao, Jian Jiang, and Xizhe Zang. “Cooperative Multi-Robot Map-building based on Genetic Algorithms”. In: *2006 IEEE International Conference on Information Acquisition*. 2006, pp. 360–364. DOI: 10.1109/ICIA.2006.306026.