Utrecht University

Automated Testing Agent Movement through 3D Environments with Omnidirectional Movement

Stefan Hoekzema

Master Thesis

Supervisors: Dr. S.W.B. (Wishnu) Prasetya
Dr. J. (Julian) Frommel

July 2024

ABSTRACT

Auto-navigation and exploration are crucial for performing automated tests. Existing automated testing tools can do this well in 2D games, but there is no tool yet for 3D games with omnidirectional movement. We created a new automated testing framework, built on top of the existing iv4XR framework, that uses an Octree to store the game world. This Octree data-structure can be used for pathfinding and can be dynamically updated based on observations. We show the method works by performing automated tests for the game Space Engineers and compare the speed and memory performances with a baseline voxel grid and 3D nav-grid. We conclude the Octree to be superior in both overall speed and memory performance in most use cases.

Contents

1 INTRODUCTION

Modern games have complex user interfaces and game interactions, with often a continuous 2D or 3D virtual world. This makes it hard to test them. Automated testing tools for games are a bit behind compared to automation in other fields. Most games are tested with the use of ‘play-testing’ [30, 31], where human players are asked to play the game and run through a bunch of test scenarios to find and report issues. This manual testing process takes a lot of time and costs a lot of money. Automated play testing is a way to replace human play-testers by letting an agent take control of the game and simulating human play-testing for testing purposes. Although this cannot replace the need for human play-testing in regards to evaluating user experience, it can replace puzzle solvability and reachability tests, as well as reduce the need for human play-testers for finding bugs.

To perform automated tests, auto-navigation and exploration are crucial. Some games have their own build in auto navigation system. However, this can not always be used for testing purposes. Which is where automated testing tools come in. Automated testing tools provide developers a way to write tests for their games. More than unit tests, it allows the testing of player interactions and movement.

These tools work well for 2D games as well as 3D games that take place on the ground, which can be seen as just 2D with added elevation. Some games, however, take place in full 3D with omnidirectional movement, like with a jetpack, drone or in space. From what we found, these automated testing tools do not yet provide a way to efficiently navigate such 3D environments.

The objective of the thesis is to design and create an automated testing tool capable of navigating in a 3D environment with omnidirectional movement. The tool aims to perform more efficiently than straightforward extensions of 2D testing approaches into 3D. The performance of the developed tool will be evaluated and compared against two baselines to demonstrate its effectiveness and superiority.

To achieve this objective, we propose a 3D space navigation method using an Octree data-structure that can be expanded and updated based on observations. We expand the iv4XR [32] framework with this navigation method and use it to perform automated tests.

We develop an automated testing framework for such 3D games and show that it works by implementing it for the game Space Engineers [15]. We build this framework on top of the already existing automated testing framework iv4XR, which has an implementation for Space Engineers (SE) that can already handle testing SE in 2D.

The rest of the thesis is structured as follows: First, in Section 2, we give an overview of important background information and literature. In addition to that, we will introduce some related work about automated testing, pathfinding, and exploration. In Section 3, we explain the underlying logic of our method, explaining the Octree data structure, how we adapted it to work with exploration in mind and how the agent can do pathfinding with it. In Section 4, we go over our preliminary 2D Quadtree demo implementation. We used this demo to test the method in a simpler use case before implementing it into the more complicated automated testing framework (iv4XR) on a complicated continuous 3D game (Space

Engineers). Section 5 goes over how we implemented both the data-structure, updating the data-structure, pathfinding and exploration. We will also go over some test methods we made that can be used for automated testing. Then, in Section 6, we explain how we test the performance of our method, compare it with two baseline methods and show the results. We also explain the possibilities the method provide for testing and show it working for the game Space Engineers. Finally, we draw our conclusions from the results and discuss the limitations and future work in Section 7.

2 BACKGROUND

2.1 Automated Game Testing

Software quality testing is increasingly more important to all businesses. The gaming industry is no exception. Testing individual functions is easily done using Unit Tests, however more complicated scenarios are still done by use of 'play-testing' [30, 31]. Automated testing techniques, like Model-based Testing (MBT) [6, 40, 41], Search-based Testing (SBT) [10, 24] and Symbolic testing [1, 36], help with this. They are, however, not directly applicable to computer games, because of their fine-grained level of interactivity [34]. Because of this, a level of abstraction has to be applied.

MAuto [39] is an automated testing tool that makes use of the record & replay technique. It is a testing tool for games on mobile (android) devices. MAuto uses image recognition to record test cases, which can then be automatically replayed for testing purposes. This makes it so that a single recorded test can then be used repeatedly during development to validate if the tested part still works. These tests, however, cease to work if the layout of buttons on screen has been changed, because the recorded actions then do not line up anymore. Simulating a press on the top-left corner of the screen to press the back button does not work anymore if the back button has been moved to the top-right. So although record & replay is powerful and easy to implement, it is inflexible.

GameDriver[1] is another automated testing tool, build for extended reality systems and games. Just like MAuto, it also works with record & replay testing and has the same drawbacks. It can be integrated to Unreal Engine and Unity, and can also be used for automated testing of virtual reality games.

Unity also has their own testing framework[2]. This framework allows users to script their tests, but is mostly for unit testing or simply automatically testing if the game runs or not. It cannot replace human play-testing.

Icarus [29] is an advanced automation testing tool for point and click adventure games, optimized for the Visionaire Studio game engine[3]. It represents the game state as a list of actions and keeps track of reward values for these actions. Using these reward values, it learns how to solve puzzles. This works great for coarse grained games, with few possible actions and states, but for fine-grained precision games, like continuous 2D or 3D games, not so much.

The Intelligent Verification/Validation of Extended Reality Systems (iv4XR) project[4] is an open source agent-based automated testing framework for extended reality systems, which includes computer games [32, 33]. It uses Ablib[5], also known as iv4XR-core, which is a Java library providing intelligent agents for automated testing. One of the games it has a plugin for, to perform automated tests, is Space Engineers (SE)[6] [15]. With the plugin, iv4XR can run synchronously with SE, retrieving world state observations from the game and sending instructions to the game. Space Engineers is a sandbox game about engineering, construction, exploration, and survival in space and on planets. In SE, the player can switch between flying mode and walking mode. Walking can be done when there is gravity or when using the magnetic boots in zero-gravity. Flying with the jetpack works the same with or without gravity and allows the player to move freely in all directions. Iv4XR works with Belief Desire Intent (BDI) agents [13]. For navigation, the game world is divided into a navigation mesh [16], which is converted into a nav-graph. On this graph, a graph-based path finding algorithm, like Dijkstra [7], can be used. The path planning algorithm implemented in Aplib is A* [12]. The iv4XR SE plugin[7] has an implementation for 2D navigation on the ground, but not for 3D navigating in space.

2.2 3D Path Planning & Exploration

The problem of finding optimal paths in 3D space is computationally more complex than in a 2D plane. Most path planning algorithms, such as Dijkstra's algorithm [7], A* [12] and D* [38] focus on optimizing the path distance, but seldom consider data effectiveness in high abstraction, creating a bottleneck on algorithmic complexity when applied to large 3D space [43]. In addition, most path planning algorithms used for 3D games are just 2D path planning algorithms, but with added elevation. These work fine for most use cases, but not for all of them, such as space simulations with omnidirectional movement. Using an efficient spatial partitioning data-structure helps reduce this complexity from continuous to discrete space.

A Quadtree is a hierarchical data structure for 2D spaces. It makes use of recursive decomposition of space [35]. Adding hierarchy makes searching large 2D spaces faster because it allows it to skip a large number of nodes when it already determined that the parent did not contain whatever it is searching for. It also reduces memory usage because many adjacent same-labeled nodes can be combined into one node.

Octrees are the 3D version of Quadtrees [18]. Octrees are a way to add hierarchy to 3D voxel spaces by joining the 8 voxels in a $2 \times 2 \times 2$ grid together into a single node and recursively joining 8 of those together again until all voxels are part of the Octree. Each Octree node then has a link to their parent-node, to their children nodes and a label to denote whether they are empty, filled, or mixed. Just like Quadtrees can be used for hierarchical path planning for 2D quad grids, Octrees allow hierarchical path planning for continuous 3D voxel grids.

One way to apply Octrees for 3D path planning, is with Sparse Voxel Octrees (SVO) [3]. Different from a typical Octree data structure, is that in an SVO, the data is stored in Morton Code order [26] in memory. The Morton code order flattens the entire 3D Octree

[1]https://www.gamedriver.io/
[2]https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html
[3]https://www.visionaire-studio.net/
[4]https://github.com/iv4XR-project

[5]https://github.com/iv4XR-project/aplib
[6]https://www.spaceengineersgame.com/
[7]https://github.com/iv4XR-project/iv4XR-se-plugin

into a linear, one-dimensional array. Instead of having all Octree nodes have 8 children, SVO combines 64 voxels in a 4x4x4 grid together into a single leaf-node. The labels of all 64 leaf-node voxels are stored as a 64-bit integer, where every bit represent if the corresponding voxel is filled or not. Knowing the label of the leaf-node, then, is as simple as checking for the 64-bit integer value. If this value is 0, then all voxels are empty, so the leaf-node is empty. If this value is 0×FFFFFFFFFFFFFFFF or -1, then all voxels are filled. Any other value means the leaf-node is mixed. Using pointers to link to the voxels makes memory usage vary between 32-bit and 64-bit operating systems. To control memory usage, a subnode index (which ranges from 0 to 63) is used instead.

Octomap [42] is a technique that also make use of Octrees. However, as compared to SVO for pathfinding, Octomap is designed with exploration in mind. It is used for automated robot exploration in 3D environments by detecting the world through a depth camera, which updates its world-view with more data from different viewing positions. It is however mostly used for mapping 3D worlds from exploration data.

3D pathfinding techniques are also developed and used for underwater scenarios. Mangeruga et al. [23] developed a 3D pathfinding algorithm for underwater navigation for divers. The algorithm first analyses a 3D underwater model and turns it into a voxel grid. Given a number of Points Of Interest (POI), it then calculated the best paths between them, using an A* variation, after which it determines the best path to visit as many of them as possible, using DFS. Heuristics are used to take oxygen usage, air decompression cost and distance into account. Kulkarni and Lermusiaux [21] made a 3D path planning algorithm for in the ocean, that takes the ocean currents into account in optimizing long distances to travel. Fairfield et al. [9] developed a Simultaneous Localization And Mapping (SLAM) method to explore underwater caves and tunnels for an autonomous hovering underwater vehicle. It uses sonar sensors to detect obstacles and maps them to a 3D voxel grid. Then it combines nodes with the same label into bigger nodes, making it a more memory efficient Octree. Where SVO and octomap used Octrees for pathfinding alone, Fairfield et al. additionally used it for exploration. To update the Octree during exploration, instead of replacing the entire Octree, they only remake the branches that were updated and copy the references to the other nodes.

In the field of robotics, 3D pathfinding is done for planning paths through the air with the robot's constraints in mind. Tuovenen et al. [37] made a way for exploring 3D indoor environments with a payload constrained autonomous micro-aerial vehicle (MAV). Dornhege and Kleiner [8] made an algorithm for autonomous rescue robots, with as goal to explore a 3D environment to rescue buried victims. It uses sensors to detect the world and structures it voxels into a hierarchical octomap. It extends the well known 2D frontier-based exploration method towards 3D environments by introducing the concept of voids—unexplored volumes in 3D that are occluded or enclosed by obstacles.

All in all, there is quite a lot of research done in 3D path planning, with many using Octrees in some way. However, these methods would not work for automated testing with unknown worlds. Exploration is a big part of automated testing, and most path planning implementations assume the world is already known. And the ones that focus on exploring an unknown world, are from a different research field, like robotics. "*In robotics, the emphasis is often on the motion of a complicated robotic system in a relatively simple environment. In games, the opposite is true. From a path planning perspective, the entity can often be modelled as a simple vertical cylinder, while the environment can be very complicated with tens of thousands of obstacles*" [27]. There is a distinct lack of 3D pathfinding methods for games and the ones that do exists, do not take exploration into consideration and thus cannot directly be used for automated testing purposes.

2.3 Space Engineers and iv4XR

Before we go into our methodology, first we will explain in more detail how the original 2D iv4XR project works. The project, like stated before, works with the Belief Desire Intent (BDI) agency and represents the world as a navigation graph (NavGrid). The implementation of the tests using BDI is done by use of Goals and Tactics. The Goals represent the desires, the Tactics represent the intent and the agent state represents the belief. Using Tactics, you can create Goals and using Goals, you can create tests for the automated testing agent. Tests can simply be done by loading in a game world and creating a test agent, then giving the agent a GoalStructure to complete and finally activate the update loop.

2.3.1 *Goals and Tactics.*

When you run a test, you first create an agent and give it a GoalStructure. GoalStructures can be combined using combinators like SEQ, FIRSTof, ANYof. A primitive GoalStructure is composed of a Goal to complete, for example getting in front of a button, and a Tactic on how to work towards completing said Goal. GoalStructures can be given a budget, denoting the maximum amount of time it is allowed to spent on trying to complete it. If no budget is specified, it is set to infinity. Just like GoalStructures can be combined, so can Tactics. A primitive Tactic consists of an Action. An Action can be seen as a pair of *guard* and *effect*. The *guard* is a predicate over the agent's state, and the *effect* part is a program that reads the agent's state and the Action's own state to produce a proposal [2].

As an example. If the goal is to get close to a target location, you create a GoalStructure that is solved if the current agent position is close to the target position. It will try to solve the goal using the `navigateTo()` Tactic. The *guard* of `navigateTo()` is finding a path towards the goal location using A* pathfinding. If it finds one, it passes it to the *effect*, otherwise it fails the *guards* and therefore the Tactic. The *effect* takes the path from the *guard* and moves along the path by giving movement instructions to the agent. After the agent has moved, it will check if the new agent position is close enough to the target position in the Goal. If so, it will succeed the primitive GoalStructure, otherwise it will reapply the Tactic until either the Goal is completed or the *guard* fails.

For combination GoalStructures (e.g. SEQ), it won't simply succeed as with a primitive GoalStructure. A SEQ, needs all GoalStructures inside it to succeed in order to succeed. It fails once any of them fails. A FIRSTof will try all GoalStructures inside it in order and succeed when any of them succeed. If fails if all of them fail. An ANYof will try all GoalStructures inside it in random order and succeed when any of them succeed. It fails if all of them fail. A REPEAT repeats the GoalStructure inside until it succeeds or the budget runs out. Finally, a SUCCESS always succeeds and a FAIL always fails. Tactics, in addition to having primitive Tactics, also have combinator Tactics. Just like the GoalStructures, it also has the SEQ, FIRSTof and ANYof combinators. Unique to Tactics, though, is the ABORT Tactic, which when selected, aborts the entire current primitive GoalStructure.

2.3.2 *Agent state updates.*

After the agent has been created and given a GoalStructure to complete, the update loop begins. If no Goal is currently being pursued, or the current Goal is marked as completed, failed or aborted, then the next Goal will be selected based on the GoalStructure combinator logic. Otherwise, it will work on completing the current active Goal. This starts by observing the game environment. This results in a world object model (*WOM*) with information of the agent and all blocks within the viewing range of the agent. Then this observation is merged with the observation of the previous iteration (except if it was the first observation). After this, it checks whether some blocks disappeared and, if so, removes them both from the *WOM* and the NavGrid (navigational 2D grid). Then it checks for new blocks and adds them to the NavGrid. For the dynamic door block, it additionally checks if it is open or closed and sets the corresponding blocking state in the NavGrid. Updating the NavGrid is done using the `addObstacle()` and `removeObstacle()` methods. Finally, it applies the current active primitive Goal. It selects a primitive Tactic based on the Tactic combinator logic to apply this iteration. This, as explained before, goes through the *guard*, and if it succeeds, the *effect* and then the Goal. If the Goal succeeded, then the Goal is marked as completed. If the *guard* failed and there is no next Tactic in the logic, then the Goal is marked as failed. If the current Tactic was the ABORT Tactic, then the Goal is marked as aborted. Otherwise, the Goal stays in progress for the next iteration.

The actual movement logic can be found inside some Tactic *effects*. For example, the `navigateToTAC()` Tactic essentially works as follows [17].

```
1  public static Tactic navigateToTAC(Vec3 destination) {
2    return action("navigateTo")
3      .do2((AgentState state) ->
4        (Pair<List<DPos3>, Boolean> queryResult) ->
5      {
6        var path = queryResult.fst;
7        var arrivedAtDestination = queryResult.snd;
8
9        if (arrivedAtDestination) {
10         state.currentPathToFollow.clear();
11         return new Pair<>(state.wom.position,
12                           state.orientationForward()) ;
13       }
14       state.currentPathToFollow = path;
15
16       var next = state.currentPathToFollow.get(0);
17       var nextPos = state.navgrid.getSqCenterPos(next);
18       if ((nextPos - state.wom.pos).lengthSq() <= DIST_THRESHOLD)
19       {
20         state.currentPathToFollow.remove(0);
21         return new Pair<>(state.wom.pos,
22                           state.orientationForward());
23       }
24       CharacterObservation obs = null;
25       obs = yTurnTowardACT(state, nextNodePos, 0.8f, 10);
26       if (obs != null) {
27         return new Pair<>(obs.position(), obs.orientationForward());
28       }
29       obs = moveToward(state, nextPos, 20);
30       return new Pair<>(obs.position(), obs.orientationForward());
31     })
32     .on((AgentState state) -> {
33       if (state.wom == null) return null;
34
35       var agentSq = state.navgrid.projectedLocation(state.wom.pos);
36       var destSq = state.navgrid.projectedLocation(destination);
37       var destSqPos = state.navgrid.getSqCenterPos(destSq);
38       if ((destSqPos - state.wom.pos).lengthSq() <= DIST_THRESHOLD)
39       {
40         return new Pair<>(state.currentPathToFollow, true);
41       }
42       int pathLength = state.currentPathToFollow.size();
43       if (pathLength == 0 ||
44           destSq != state.currentPathToFollow.get(pathLength-1))
45       {
46         var path = state.pathfinder.findPath(state.navgrid,
47                                              agentSq, destSq);
48         if (path == null) return null;
49         path = smoothenPath(path);
50         return new Pair<>(path, false);
51       }
52       else {
53         return new Pair<>(state.currentPathToFollow, false);
54       }
55     })
56     .lift();
57 }
```

The *guard* is defined in lines 32-56 and the *effect* part in lines 2-30. The *guard* first checks if the state has a world object model (*WOM*), meaning it has observed the world (line 33). If not, it fails. Then it calculates the node the destination is in and the node the agent is in. If the distance between the center of the destination node and the agent (not agent node) position is smaller than the distance between nodes (line 38), then the *guard* was passed, and it enables the *effect* with the current path and a `true` value denoting that the destination has been reached (line 41). Otherwise, it checks if there is no path planned yet or if there is a path, if that path leads to a different destination (lines 42-44). If so, then we need to plan a new path, which we do using `findPath()` (line 46), which uses the A* search algorithm. If there is no path to the destination, the guard fails, otherwise it smoothens the path, removing unnecessary intermediate nodes, and passes the path to the *effect* part. If we do not need to plan a path, because the state already has one to the destination (line 43), we skip pathfinding and simply pass the path to the *effect* part (lines 52-53).

The *effect* is responsible for moving along the path. It first checks if the guard returned a *true* value. If so, we reached the destination, the current path stored in the state is cleared, and we return the current agent position and orientation for the Goal to check if we indeed finished the goal (lines 10-11). If not, we need to follow the path. First, we store the path in the state, so that the guard can skip pathfinding the next iteration (line 14). Then, if on the path we reached the next intermediate position, the intermediate position is removed from the path, and we return the position and orientation to the Goal (lines 16-21). If the next position on the path was actually the destination (so not an intermediate position), then the Goal should be cleared with the returned position and orientation. If we are not yet at the next intermediate location, we first check if we need to turn the agent to look in the general direction of movement. The `yTurnTowardsACT()` checks this and, if it needs to, rotates the agent around the y-axis to face the target direction before returning a new updated agent observation (lines 25). If no observation was returned, then we did not rotate, so then we move towards the next intermediate location (line 29). If we did rotate, then we will skip movement for this iteration (lines 26-27). After rotation or movement, we return the new agent position and orientation to the goal.

3 METHODOLOGY

Automated testing is all about testing if things work as intended, however to be able to test most scenarios, the player character needs to be in a certain state. Positioning is crucial for this, so the agent needs to be able to move to the required position to test something or to test the movement itself. Just moving towards the goal location, however, is not enough. There can be obstacles in the way, so the agent needs to be able to sense the world and navigate in it.

This navigation problem can be split into three sub-problems. One, how to find a somewhat optimal path from the current location to some known goal location. Two, how to explore the world if the goal location is unknown in order to find said goal. And three, how to store the world in a memory efficient and path planning time efficient way. Path planning is crucial for world navigation, exploration allows path planning for unknown environments and efficiency makes it usable for large 3D worlds.

3.1 Abstraction

The game world contains too much information, so an abstraction layer is used to only store the important parts. The agent sees the world through this abstraction layer. It uses this layer to find paths, and it must be able to expand the abstract world during exploration. This abstraction layer will be stored in memory, so we want to keep the amount of information stored minimal. For path planning, the agent needs to find a path through the abstract world, so the abstraction layer must be made in a way that allows the agent to do this efficiently.

Space engineers is a 3D game with continuous axes, so every object can be in $\mathbb{R}^3$. To reduce this, we can represent the world as a voxel space, reducing it to $\mathbb{Z}^3$. This works fine for small worlds, however for larger worlds, the memory use is still too big and pathfinding will also be too slow. To improve this, we can make use of a different spatial partitioning data-structure—an hierarchical one. For voxels, the most well known is the Octree [25] data structure. In an Octree, we divide a node into 0 or 8 child-nodes, with every child-node being another sub Octree node with again 0 or 8 children. This way, we can group leaf-nodes with the same occupancy status together into a single, larger, leaf-node, reducing memory usage. An example of an Octree is visualised in Figure 1. The Octree nodes themselves simply store whether they are occupied or not.

An alternative way to implement Octrees is Linear Octal Encoding [11]. Instead of storing both open and blocked Octree nodes, it instead only keeps a list of encoded Octree blocked nodes, without open nodes. For example, if the nodes 2, 7, 12 and 15, from Figure 1, are BLOCKED nodes, then the encoded Octree looks like: $\{12, 15, 2X, 7X\}$. Starting from the root note, every digit, going from left to right, represents in which child-node an object is (so a blocked node). The digits range from 0 to 7 where the first four are at the front in 'z' order with top-left being 0, and the last four in the same

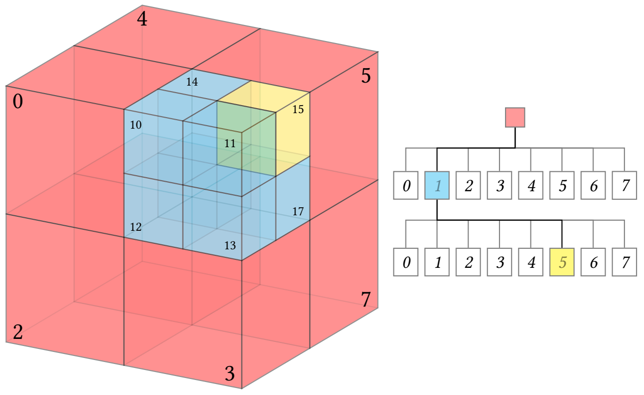


Figure 1: Octree example

order at the back. The X means that every child is BLOCKED. This significantly reduces the memory use even further, because we no longer store open nodes. However, storing only BLOCKED nodes would also make exploration more difficult, as we need a way to known what is and what is not explored yet.

3.2 Pathfinding

For moving through the world, we use a variation of A* that makes use of the Octree data-structure. Because the goal of our pathfinding algorithm is just to get to some location efficiently, we do not care about returning a guaranteed shortest path. A* starts with the starting location (and cost of 0) in its *openSet*, it then iteratively picks the location with the lowest cost from the *openSet*, moves that location to the *closedSet* and adds its neighbours and their cost to the *openSet*, until it reaches the goal location. Then from the goal location, it tracks back the way it came, to get, when inverted, the final optimal path. What makes it different from Breadth First Search (BFS), is that it uses a heuristic to guide itself towards more optimal neighbour choices to inevitably find the goal in less iterations.

This works fine for a normal grid, where getting the neighbours is trivial, but for an Octree we have to adapt it so that it can find the right neighbours. We can either store a list of pointers to the neighbours for every node, or use a function to calculate the neighbours at runtime. Calculating the neighbours is more memory efficient but takes more time to do. The algorithm to get the neighbours from an Octree node works as follows:

Algorithm 1 Octree right neighbour

```
procedure $node.rightNeighbour()$
    if $node \in parent.childrenleft$ then
        $neighbours \leftarrow parent.children[node.code+1].leftEdgeChildrenLeafs()^{\dagger\dagger}$
    else if $node \in parent.childrenright$ then
        $neighbours \leftarrow parent.rightNeighbour()$
    else
        $neighbours \leftarrow null$
    end if
    return $neighbours$
end procedure
```

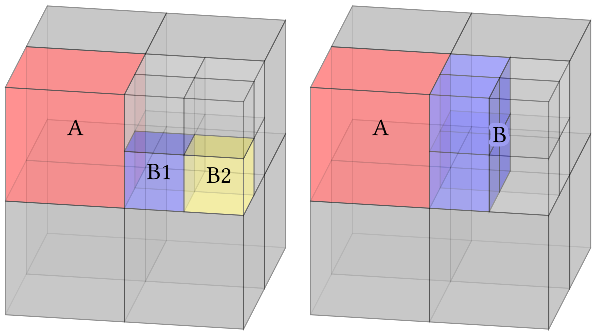


Figure 2: Octree neighbours: A is the only left neighbour of B1, but A has 4 right neighbours in B

As an example, the right neighbour of node **B1**, in Figure 2, is node **B2**. The algorithm checks the code of node **B1** and using the code, knows that its direct parent has the right neighbour as a child, so it can just retrieve it from the parent. The left neighbour of node **B1** is node **A**. The algorithm once again checks the code of node **B**. The code is the same, but because it wants the left neighbour, it knows the direct neighbour cannot give it. Thus, is will recursively call the algorithm on the parent-node, until it either finds a neighbour or it reaches the root node (meaning there is no neighbour).

The right neighbour of node **A** is node **B**. This node, however, is not a leaf-node, so instead of returning node **B** as the right neighbour of node **A**, we return all the leftmost leaf-nodes of node **B**.

Getting the right neighbour of a node, that is on the right side of its parent-node, means that it cannot directly get the neighbour from its parent. Thus, it will recursively call the get neighbour method on the parent until it finds one, that is on the left side of its parent. This one can then return the right neighbour. However, as we want to return a navigable node, we cannot return itself if it is not a leaf-node. To get the actual right neighbour of the original leaf-node, the high order neighbour node needs to return its leftmost leaf-node adjacent to the original leaf-node. For example, in Figure **??**, the right neighbour of node C is node F. Node C first checks B for a right neighbour, which checks node A in turn. Node A then, a child on the left side of node R, finds node D as a right neighbour. Then node D checks the left child adjacent to node B, node E, which return its left child adjacent to node C, node F.

Diagonal neighbours can simply be calculated by combining two or three of the cardinal direction neighbour methods.

Using hierarchical A* [14], which does pathfinding at the lowest resolution and then refines the higher detail levels iteratively, is another possibility. Hierarchical A* can significantly improve performance, however it bloats the memory usage significantly, because it requires knowledge, for every node at every layer, to which neighbouring nodes that can be reached. Just like with A* we can calculate the neighbours instead of storing them, but then it would become slower again and the advantages become unclear.

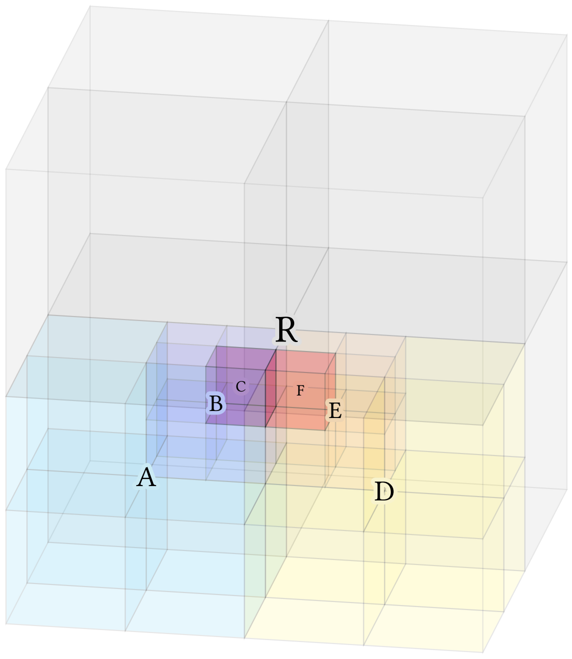


Figure 3: Octree neighbours: Node C cannot find a right neighbour in its parent B, so it checks the parent's parent A, which also cannot find it, so it checks their parent R. Node A finds the right neighbour D in R. Then because B in A is top-right-back, we get E, the top-left-back child of D. Lastly we get the top-left-front child of E, F as the right neighbour of node C.

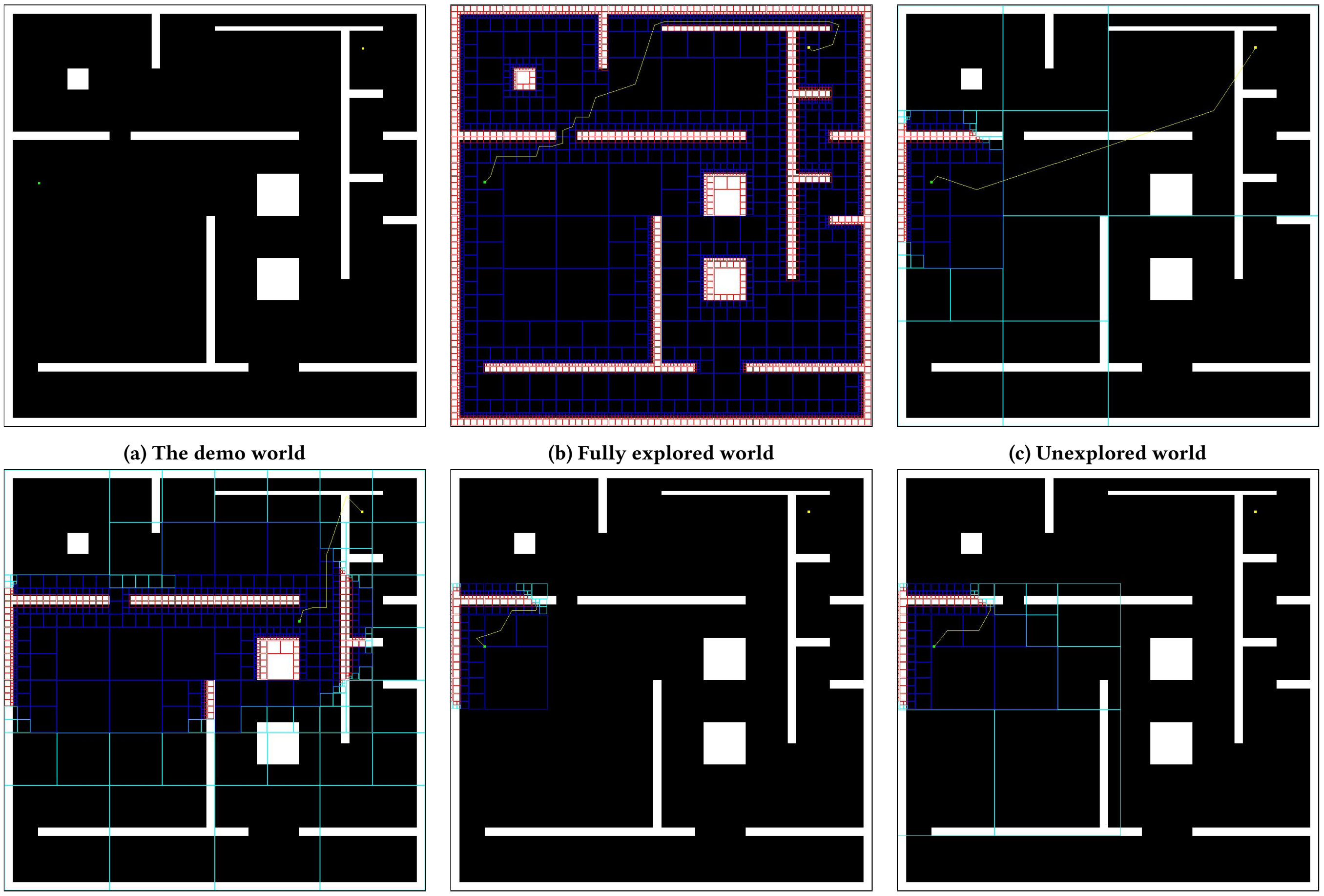
3.3 Exploration

Pathfinding can be done if the goal position is known, otherwise the agent needs to explore to find the goal. The agent has a predefined viewing distance, which is set in a configuration file. Everything in this viewing range, the agent observes in the form of a world object model (*WOM*). The current implementation of iv4XR for SE maps the world as a grid of quads and keeps track of what quads have obstacles in them (BLOCKED) and which ones do not (OPEN). This grid is just stored in a `hashMap`, however, the same cannot be done for the 3D implementation, because it would just be much too big and slow. This is why the Octree structure from Section 3.1 is so important. During exploration, the Octree needs to be expanded and updated. Everything inside the viewing distance needs to be observed and if some object, for example, is destroyed or added, then some nodes in the Octree need to be changed to reflect this change in the game environment. If we explore, then we will inevitably view parts of the world not yet encapsulated into the Octree world representation, so we need to be able to expand the Octree. We expand the Octree by creating a new Octree root-node and making the old root-node a child of the new one.

4 PRELIMINARY 2D IMPLEMENTATION

Before we started on the full 3D Octree implementation for the automated testing framework, we first created a simple 2D Quadtree version. Because our data-structure also needs to be able to be expended and nodes can have an UNKNOWN label in addition

†† leftEdgeChildrenLeafs() recursively goes down the hierarchy, only returning the leaf-nodes on the left edge, neighbouring the original query node. The explanation for how this is done can be found in Section 5.2.



(a) The demo world **(b) Fully explored world** **(c) Unexplored world**

(d) Unexplored world, updated path **(e) Unknown world** **(f) Unknown world, expanded Quadtree**

Figure 4: Quadtree demo

to OPEN, CLOSED or MIXED, we need to make some alterations to a normal Octree. Instead of developing it while immediately integrating it with the complicated iv4XR project for the game Space Engineers, we decided to first implement the Octree idea in a simpler environment. For this, we created a 2D version of the Octree method, which would be a Quadtree. This demo would also confirm if the Octree method would actually work.

The 2D demo is written in Java. The game world is continuous 2D with a black background and white walls, visualized in Figure 4a. The player character is a small green cube, and the goal is visualized as a small yellow cube. Implementing a Quadtree on this world gives us a Quadtree as visualized in Figure 4b. In the image, red squares are BLOCKED Quadtree nodes and blue squares are OPEN Octree nodes. For clarity, only leaf-nodes are visualized. The yellow line is the path from the player character to the goal. This path is calculated using Dijkstra's algorithm. Neighbouring nodes are calculated using the functions explained in Section 4.1.

If the player does not have an infinite viewing distance, then we do not know of obstacles outside the viewing distance. Such a scene is visualized in Figure 4c. In the figure, the cyan squares are UNKNOWN nodes. When the goal location is known, but it is inside an UNKNOWN node in the Quadtree, then the path to the goal avoids known BLOCKED nodes and goes through OPEN or UNKNOWN nodes. When it follows this path, the player automatically updates the Quadtree with new observations (see Figure 4d). The path will also be updated based on the new information and plan around known obstacles. This works well, but we cannot assume to know the size of the world, so we need to be able to expand the Quadtree. With an unknown world scale, the starting Quadtree will contain the full viewing range (see Figure 4e), but when the viewing range exits the Quadtree's bounding box as in Figure 4f, then the Quadtree will expand itself, creating a new root node, with the old root node as one of the children.

The demo simply tests the data-structure's ability to do pathfinding and updates based on observations. We did not implement any automation or testing. Instead, the user has full control over the players (green cube) movement and Quadtree nodes are visualized on screen and updated at runtime. Additionally, using the mouse, we made it so that pressing the left mouse button would place a new wall at the cursors location and pressing the right mouse button would remove any wall under the cursor. This makes it easy to visually debug the Quadtree and the pathfinding.

4.1 Neighbours

Retrieving the neighbours from a Quadtree node is not a trivial task. We implement Algorithm 1 as follows.

```
1  public List<Quadtree> getRightNeighbour(Stack<int> codes)
2  {
3    switch (this.code) {
4      case 0 -> { // top-left
5        return parent.children[1]
6               .leftEdgeChildrenLeafs(codes);
7      }
8      case 1 -> { // top-right
9        codes.push(0);
10       return parent.getRightNeighbour(codes);
11     }
12     case 2 -> { // bottom-left
13       return parent.children.get(3)
14              .leftEdgeChildrenLeafs(codes);
15     }
16     case 3 -> { // bottom-right
17       codes.push(2);
18       return parent.getRightNeighbour(codes);
19     }
20   }
21   return null;
22 }
```

Every directional `get_Neighbour()` method is almost exactly the same, just using different codes. With how often these methods will be called, adding an additional line to check which neighbour we want to get every recursive call would make the program unnecessarily slower. Thus, we separated them into different methods, skipping one line of code for recursive call, for every neighbour, for every node during pathfinding. To get the right neighbour, we check if the current node is on the left side or right side of its parent-node by checking the code (line 3). The code is in Morton code order, so the left side codes are 0 and 2 and the right side codes 1 and 3. If the current node is on its parent's right side, then the parent cannot give the right neighbour, so we recursively call the function on the parent-node (lines 10 and 18). Eventually the node is on its parent left side (except if it is on the edge of the root-node, resulting in zero right neighbours) and it has found a right neighbour. However, we are not done yet, because the node has to be a leaf-node. If it is a leaf-node, we are done. Otherwise, we have to retrieve the children nodes on the left side. If those are not leaf-nodes, then we once again have to get the left side child-nodes. Thus, we call the `leftEdgeChildrenLeafs()` method, which, as the name suggests, gets a list of all leaf-nodes on the far left edge of the current node by recursively going down (lines 5 and 13). But we do not want all of them, because if we recursively went up, then getting all left edge leaf-nodes of the higher up neighbour node also returns the right neighbours of some other nodes. For example, the bottom-right node goes up once to the parent. The parent is on the left side of their parent and gets the right neighbour. The right neighbour gets its two left child-nodes. One of these two is the right neighbour of the original node, but the other is the right neighbour of the top-right node, the top neighbour of the original node. To remember the path up and take the corresponding path down, we added a Stack of codes. Whenever the getRightNeighbour() method is recursively called (so we go up in the hierarchy), we add the opposite code of the current node's code to the stack (lines 9 and 17). The opposite code of 1 for the right neighbour is 0, the code of its left neighbour. Then when leftEdgeChildrenLeafs() is called, we give it this stack.

```
1  public List<Quadtree> getLeftEdgeLeafs(Stack<int> codes)
2  {
3    List<Quadtree> list = new ArrayList<>();
4    if (this.children.isEmpty()) {
5      if (this.label != BLOCKED)
6        list.add(this);
7      return list;
8    }
9    if (!codes.isEmpty()) {
10     list.addAll(children[codes.pop()]
11           .getLeftEdgeLeafs(codes));
12   }
13   else {
14     list.addAll(children[0]
15           .getLeftEdgeLeafs(codes));
16     list.addAll(children[2]
17           .getLeftEdgeLeafs(codes));
18   }
19   return list;
20 }
```

This method, whenever it recursively calls itself (so we go down in the hierarchy), checks and removes one value from the stack. The value it retrieves is the code of the child-node it needs to recursively call (line 10). If the stack is empty, and it still is not a leaf-node, then the original leaf-node was bigger than the right neighbours, so it returns all left children leaf-nodes (lines 14-17). If the node is a leaf-node, it simply returns itself (lines 4-7). Of course, if the final node selected as a neighbour has the BLOCKED label, we do not return it, because it is inaccessible. To keep the demo simple and to increase the ease of visual debugging, we did not implement diagonal neighbours for the demo.

4.2 Updating the Quadtree

Every frame, every object inside the viewing range of the player is observed and added to the Quadtree with the `update()` method. The update method takes the entire list of walls (both in and outside viewing range) and an Ellipse2D (from the *java.awt.geom* package).

```
public void update(List<Wall> walls, Sphere viewingRange)
{
  if (!viewingRange.intersects(this.boundary)) {
    return;
  }
  if (!this.children.isEmpty()) {
    foreach (child in children) {
      child.update(walls, viewingRange)
    }

    if (all children labels == BLOCKED) {
      this.label = BLOCKED;
      children = new ArrayList<>();
    }
    if (all children labels == OPEN) {
      this.label = OPEN;
      children = new ArrayList<>();
    }
    return;
  }
  if (any (wall in walls) contain (this.boundary))) {
    this.label = BLOCKED;
    if (!children.isEmpty()) {
      children = new ArrayList<>();
    }
26        return;
27      }
28      if (!any (wall in walls) intersects (this.boundary))) {
29        this.label = EMPTY;
30        if (!children.isEmpty()) {
31          children = new ArrayList<>();
32        }
33        return;
34      }
35
36      if (boundary.height < 2 * MIN_NODE_SIZE) {
37        this.label = BLOCKED;
38        return;
39      }
40      if (this.label != MIXED) {
41        this.subdivide(this.label);
42        this.label = MIXED;
43      }
44      foreach (wall in walls) {
45        if (wall.intersects(this.boundary)) {
46          foreach (child in children) {
47            child.update(walls, viewingRange)
48          }
49        }
50      }
51      if (all (child in children) label == BLOCKED)) {
52        this.label = BLOCKED;
53        children = new ArrayList<>();
54      }
55    }
```

The method starts at the root-node and recursively goes through the child-nodes. If the current node is not fully or partially inside the viewing range, then it does nothing in that node and its children (lines 3-4). Otherwise, it continues. Lines 6-19 are for nodes with child-nodes, and lines 21-53 for leaf-nodes. When the current node has children, it simply recursively calls the `update()` method for all children (lines 7-8). After that, it checks if all child-nodes have the same label and if so, it makes itself a leaf-node by throwing away the children (overwriting with empty `arrayList`) (lines 11-17). If the current node is a leaf-node, it first checks if it is fully inside any of the walls. If so, it simply becomes a BLOCKED leaf-node (lines 21-24). Then it checks if none of the walls intersect with the node. If so, it simply becomes an OPEN leaf-node (lines 28-31). Note that all nodes start with an UNKNOWN label, and only nodes inside the viewing range will be updated. If neither of these were the case, then it means one or more walls partially intersect with the node, in which case we subdivide the node (if it wasn't already subdivided) (lines 40-42). However, to prevent nodes from becoming too small and taking up unnecessary memory space, lines 36-38 check if the node size after subdivision (so half the current size) will be smaller than the minimum node size. If so, it will simply be labeled as BLOCKED and not subdivide. After subdivision, we loop over all objects and if they intersect the node, we recursively call the `update()` method on all child-nodes (lines 44-47). Finally, once all child-nodes have their label, we once again check if we need to become a leaf-node if all children are BLOCKED (lines 51-53). This happens when the quad contains multiple objects, where none of them fill the entire quad, but they do so together.

4.3 Conclusion

The demo showed that the Quadtree method works in 2D. As the methods are designed with 3D Octrees in mind, it should theoretically also work in 3D with Octrees. It is just a case of changing rectangles and circles to cubes and spheres and adding two additional get neighbour methods. To make it work in 3D, the third dimension simply had to be added. To make it work for automated testing for Space Engineers, it has to be incorporated into the iv4XR project's structure, replacing the NavGrid with additional changes to handle expansion and exploration. Finally, as it was a simple demo, we did not try to optimize its speed and memory efficiency. So this still has to be done for the final version.

5 IMPLEMENTATION

With the demo done and method working in 2D with Quadtrees, we could confidently start on the actual Octree implementation. In this section, we will explain how we implemented the Octree method from Section 3 and how we incorporated it into the automated testing framework iv4XR for the game Space Engineers. For this we will go over the neighbour methods in Section 5.2, the expanding method in Section 5.3, and the agent movement changes in Section 5.4. To make it so that A* pathfinding can be done, which is a node-based pathfinding algorithm, we implemented padding, which we explain in Section 5.5. Then, in Section 5.6, we the changes to the iv4XR state updating method and introduce some new testing functions in Section 5.7. We will also explain how we implemented the VoxelGrid method to use as a baseline. Finally, in Section 5.8 we explain how the VoxelGrid baseline was implemented.

From this point on, whenever we mention a *grid* (in cursive), then we mean the spatial partitioning data-structure. This can reference to the Octree, VoxelGrid or NavGrid, depending on the context.

5.1 Octree

The information an Octree node need to store is as follows:

```
public class Octree {
  public byte label;
  public byte code;
  public Boundary boundary;
  public Octree[] children;
  public Octree parent;
}
```

Octree nodes contain a label, which can be one of four possible types: BLOCKED, OPEN, MIXED or UNKNOWN. This is stored as a byte, which reserves 8 bits of memory. There is no data type which reserves less than that, except for a boolean, which reserves just one bit. Using two booleans instead of a byte would reduce memory usage, however using a byte makes sure that the data is aligned with Java's native data types, which should make retrieving data more efficient. The `code` contains a value from 0 to 7, which indicates which child the node is of its parent-node in Morton code order. The `boundary` stores the position and size of the bounding box and contains comparison functions like `contains()` and `intersects()`. The link to the parent-node is stored in `parent`. The child-nodes are stored in the `children` array. The array will have a size of 8, if it has children, or be set to `null`, if it is a leaf-node. When we subdivide a node, we simply overwrite the null value of the array to a new array. When we combine multiple nodes together, we simply set the array to be null, which gets rid of the references to the child-nodes, freeing up memory.

When observing the world, the agent keeps track of all blocks inside its viewing range. Every new block inside this range is added to the Octree using the following Octree method:

```
1  public void addObstacle(WorldEntity block) {
2    if (this.label == Label.BLOCKED) return;
3
4    var blockBB = blockBB(block.position);
5    if (blockBB.contains(this.boundary)) {
6      this.label = Label.BLOCKED;
7      if (children != null) {
8        children = null;
9      }
10     return;
11   }
12   if (blockBB.intersects(this.boundary)) {
13     if (boundary.size() < MIN_NODE_SIZE) {
14       this.label = Label.BLOCKED;
15       return;
16     }
17     if (this.label != Label.MIXED) {
18       this.subdivide(this.label);
19       this.label = Label.MIXED;
20     }
21     for (Octree node : children) {
22       node.addObstacle(block);
23     }
24     boolean allMatch = true;
25     for (Octree node : children) {
26       if (node.label != Label.BLOCKED) {
27         allMatch = false;
28         break;
29       }
30     }
31     if (allMatch) {
32       this.label = Label.BLOCKED;
33       children = null;
34     }
35   }
36 }
```

The method will only change nodes into BLOCKED nodes, so already BLOCKED nodes are ignored at line 2. In line 4, a boundary is made from the input block, which is then used in line 5 to check if the Octree node is fully inside this block. If so, the node is set to be BLOCKED, and it sets the children array to null, if it wasn't already. The rest of the code is only called if the node is only partially inside the boundary (line 12). To prevent nodes from becoming too small and taking up unnecessary memory space, lines 13-15 check if the node size is smaller than the minimum node size for subdivision. If so, the node will simply be labeled as BLOCKED and not subdivide. Otherwise, the node needs to be subdivided. Lines 17-19 make sure, if it wasn't subdivided already, to subdivide the node. Then it recursively calls addObstacle() for all child-nodes (lines 21-22). Finally, once all children have their label, we check if all children have the same label (BLOCKED). If so, the parent becomes BLOCKED and throws away the children (lines 24-33). This can happen when the Octree node intersects with multiple blocks, where none of them fill the entire boundary, but together they do. Removing obstacles is done with the identical method removeObstacle(), but with the OPEN label instead. This method is called whenever a block is removed within the observation range. Every iteration, the agent observes the world. The blocks that were in the previous observation and not in the current one are removed, but only if they are still inside the observation range, otherwise we know that they are not in the *WOM*, because they are too far away. Whenever a button is pressed, we don't know if the door will be open or closed, so we give the corresponding Octree nodes a UNKNOWN label with another identical method (setUnknown(), but with the UNKNOWN label. The blockBB() method in line 4, creates the padded boundary. More on this in Section 5.5.

The subdivide method in line 18 looks as follows:

```
public void subdivide(byte label) {
  children = new Octree[8];
  float half = boundary.size / 2;
  Vec3 position = boundary.position;
  children[0] = new Octree(
    new Boundary(position, half),
    this, 1, label);
  children[1] = new Octree(
    new Boundary(position + Vec3(half,   0,    0)), half),
    this, 2, label);
  children[2] = new Octree(
    new Boundary(position + Vec3(   0,half,    0)), half),
    this, 3, label);
  children[3] = new Octree(
    new Boundary(position + Vec3(half,half,    0)), half),
    this, 4, label);

  children[4] = new Octree(
    new Boundary(position + Vec3(   0,    0,half)), half),
    this, 5, label);
  children[5] = new Octree(
    new Boundary(position + Vec3(half,    0,half)), half),
    this, 6, label);
  children[6] = new Octree(
    new Boundary(position + Vec3(   0,half,half)), half),
    this, 7, label);
  children[7] = new Octree(
    new Boundary(position + Vec3(half,half,half)), half),
    this, 8, label);
}
```

This method fills the children array with 8 new Octree nodes. All with a boundary half the size of their parent and moved in Morton code order. So the first child has the same lower boundary, with the upper boundary at the center of the parent boundary. The second child has the same y and z coordinates as their parent, but is moved halfway on the x-axis. All children get a reference to their parent-node (this), are given a code with the index the child is in the parent-node's children array. Lastly, the children are given the same label as their parent. Whenever we make a fully new Octree node (so whenever a root node is made), we initiate it with the UNKNOWN label. Then, when we subdivide, we give the children the same label as their parent before we change the parent label into MIXED. We do this, because if for example an OPEN node needs to be subdivided, because the agent placed a block that intersects with the node, then we known that the children are either OPEN or BLOCKED. Setting the new Octree nodes initially to UNKNOWN would be stupid, because the addObstacle() method only sets the overlapping nodes to be BLOCKED, which leaves the other child-nodes with an UNKNOWN label. We can still go over all nodes inside the viewing range and set any UNKNOWN nodes to OPEN, but this is unnecessary extra work.

5.2 Pathfinding

Calculating the neighbours of a node can be done using a function like the following for every direction.

```
public List<Octree> getRightNeighbour(Stack<Integer> codes) {
  switch (this.code) {
    case 0 -> { // top-left-front
      return parent.children[1].getLeftEdgeLeafs(codes);
5        }
6        case 1 -> { // top-right-front
7          codes.push(0);
8          return parent.getRightNeighbour(codes);
9        }
10       case 2 -> { // bottom-left-front
11         return parent.children[3].getLeftEdgeLeafs(codes);
12       }
13       case 3 -> { // bottom-right-front
14         codes.push(2);
15         return parent.getRightNeighbour(codes);
16       }
17       case 4 -> { // top-left-back
18         return parent.children[5].getLeftEdgeLeafs(codes);
19       }
20       case 5 -> { // top-right-back
21         codes.push(4);
22         return parent.getRightNeighbour(codes);
23       }
24       case 6 -> { // bottom-left-back
25         return parent.children[7].getLeftEdgeLeafs(codes);
26       }
27       case 7 -> { // bottom-right-back
28         codes.push(6);
29         return parent.getRightNeighbour(codes);
30       }
31     }
32     return null;
33   }
```

This function is one of 26 neighbour functions: the right neighbour. The function is called from a leaf-node to get that node's right neighbouring leaf-nodes. Depending on what layer the current node is, it can have one or more right neighbours. In line 2, we check what child-node this node is in its parent-node (the code). If it is the first child (code 0, see Figure 1) then line 5-6 is returned. In that case, the right neighbour of node 0 (the top-left-front node) is node 1 (the top-right-front node). However, we are not done yet, because that node could be split into more children, so instead of returning child-node 1, we call `getLeftEdgeLeafs()`, which returns itself if it has no children, and otherwise it recursively calls `getLeftEdgeLeafs()` on all its child-nodes on the left edge (nodes 0,2,4,6). It is even more complicated getting the right neighbour when the current node is on the right edge of its parent. When that is the case, it will recursively call the `getRightNeighbour()` function on its parent until it reaches a parent-node on the left edge. Only then will it call `getLeftEdgeLeafs()`. However, this time, we do not want to get all left edge nodes, because only a part of them actually neighbour the original leaf-node. This is where the stack `codes` comes into play. Whenever we recursively call `getRightNeighbour()`, we add a code to the stack (lines 8, 16, 24 and 32). Then, when the function `getLeftEdgeLeafs()` is eventually called, we pass it to the code stack. This function reads as follows:

```
1  public List<Octree> getLeftEdgeLeafs(Stack<Integer> codes) {
2    List<Octree> list = new ArrayList<>();
3    if (this.children.isEmpty()) {
4      if (this.label == Label.OPEN)
5        list.add(this);
6      return list;
7    }
8    if (!codes.isEmpty()) {
9      list.addAll(children[codes.pop()].getLeftEdgeLeafs(codes));
10   }
11   else if (codePopOverwrite != 0) {
12     list.addAll(codePopOverwriteSwitch(codes));
13   }
14   else {
15     list.addAll(children[0].getLeftEdgeLeafs(codes));
16     list.addAll(children[2].getLeftEdgeLeafs(codes));
17     list.addAll(children[4].getLeftEdgeLeafs(codes));
18     list.addAll(children[6].getLeftEdgeLeafs(codes));
19   }
20   return list;
21 }
```

Whenever the `code` stack is not empty (line 8), it means we are an order higher than the original leaf-node, so we recursively call `getLeftEdgeLeafs()` on only the child with the code from the stack (line 9). Every time we go a layer lower, the stacks get smaller until we are on the same layer as the original leaf-node. The codePopOverwrite line (11) for cardinal directions is always 0, so it never runs line 12. This will come into play for diagonal neighbours. Then lines 15-18 recursively calls the method on all left-edge-children leaf-nodes. If a leaf-node has been reached, then it returns itself, but only if the leaf-node is OPEN (line 3-6). The explore variant of the method also allows UNKNOWN nodes to be returned (otherwise it can never find a path to a UNKNOWN node). Finally, it returns a concatenated list of all leaf-nodes from the recursive calls.

The neighbour method for the six cardinal directions are very similar. Diagonal neighbours are also similar, but a bit more complicated. Take the top-right neighbour method for example.

```
1  public List<Octree> getTopRightNeighbour(Stack<Integer> codes)
2  {
3    switch (this.code) {
4      case 1 -> { // top-left-front
5        return parent.children[1].getTopNeighbour(codes);
6      }
7      case 2 -> { // top-right-front
8        codes.push(2);
9        return parent.getTopRightNeighbour(codes);
10     }
11     case 3 -> { // bottom-left-front
12       return parent.children[1]
13           .getBottomLeftEdgeChildrenLeafs(codes);
14     }
15     case 4 -> { // bottom-right-front
16       return parent.children[1].getRightNeighbour(codes);
17     }
18     case 5 -> { // top-left-back
19       return parent.children[5].getTopNeighbour(codes);
20     }
21     case 6 -> { // top-right-back
22       codes.push(6);
23       return parent.getTopRightNeighbour(codes);
24     }
25     case 7 -> { // bottom-left-back
26       return parent.children[5]
27           .getBottomLeftEdgeChildrenLeafs(codes);
28     }
29     case 8 -> { // bottom-right-back
30       return parent.children[5].getRightNeighbour(codes);
31     }
32   }
33   return null;
34 }
```

In here, instead of having four cases, where the neighbour is simply part of the parent-node, there are only 2 (cases 3 and 7). Also, there are now only two recursive calls of `getTopRightNeighbour()` (cases 2 and 6), leaving four cases that are a bit different. In cases 1 and 5, the method will first take the top neighbour from the parent and then call the `getRightNeighbour()` method from there. In cases 4 and 8, the method will call `getTopNeighbour()` from its direct right neighbour. This brings a problem to the table. If we want to get the top-right neighbour of a node, then it is possible for it to call the `getTopNeighbour()` method from the right neighbour. The `getTopNeighbour()` method eventually calls the `getBottomEdgeLeafs()` method. However, we originally wanted the top-right neighbour only, so when the query node is higher in the Octree hierarchy (bigger) than the leaf-nodes, it can return nodes that do not diagonally neighbour the original query node. To fix this, whenever we call a diagonal neighbour method, we first set the static `codePopOverwrite` value to the code corresponding with the diagonal direction. Cardinal directions have code 0 and will not do anything. Then any directional `get_EdgeLeafs()` method check the `codePopOverwrite` value and if it is not 0, calls the `codePopOverwriteSwitch()` method. This is a giant switch case of all 20 diagonal neighbour codes. Using this switch case, it switched from the wrong `get_EdgeLeafs()` to the right one.

Diagonals like top-right-back are even more complicated, with only one case where it is directly available and one case where it recursively calls itself. The other six cases are a `getTopNeighbour()`, `getRightNeighbour()`, `getBackNeighbour()`, `getTopRightNeighbour()`, `getTopBackNeighbour()` and a `getRightBackNeighbour()`.

Every `get_Neigbour()` method adds some nodes to the neighbours list. The only exception is the root node. If a node is at the very border of the Octree, then there is a direction with no neighbour. To handle such cases, the function recursively calls itself on the parent-node until it reaches the root node. The root node is the only node with a code outside the range [0-7], which results in it reaching the `return null;` line of the method. The `neighbours()` method, which calls all the directional `get_Neighbour()` method to get every neighbour, then checks for `null` return values. Whenever *null* is returned, it ignores it, except if we are doing exploration. If we are doing exploration and retrieving the neighbours of a node, then we allow UNKNOWN nodes in addition to OPEN nodes. And everything outside the root Octree node is UNKNOWN. So whenever a null value is returned, then we create a new temporary UNKNOWN Octree node in the corresponding direction and add it to the neighbours list. This temporary node is automatically thrown away after exploration.

5.3 Exploration

If we explore, then we will inevitably view parts of the world not yet encapsulated into the Octree world representation, so we need to be able to expand the Octree. The following function first checks if the Octree needs to be expanded and, if so, will proceed to do so.

```
1  public Octree checkAndExpand(Boundary range) {
2    if (boundary.contains(range)) return null;
3
4    byte oldcode = getOldCode(); // what child position the old rootnode will have in the new one
5
6    Octree newRoot;
7    Vec3 position = boundary.position;
8    float newSize = boundary.size * 2;
9    float oldSize = boundary.size;
10   switch (oldcode) {
11   case 1 -> newRoot = new Octree(
12     new Boundary(position, newSize),
13     null, 0, Label.MIXED);
14   case 2 -> newRoot = new Octree(
15     new Boundary(position - Vec3(oldSize,0,0), newSize),
16     null, 0, Label.MIXED);
17   case 3 -> newRoot = new Octree(
18     new Boundary(position - Vec3(0,oldSize,0), newSize),
19     null, 0, Label.MIXED);
20   case 4 -> newRoot = new Octree(
21     new Boundary(position - Vec3(oldSize,oldSize,0), newSize),
22     null, 0, Label.MIXED);
23
24   case 5 -> newRoot = new Octree(
25     new Boundary(position - Vec3(0,0,oldSize), newSize),
26     null, 0, Label.MIXED);
27   case 6 -> newRoot = new Octree(
28     new Boundary(position - Vec3(oldSize,0,oldSize), newSize),
29     null, 0, Label.MIXED);
30   case 7 -> newRoot = new Octree(
31     new Boundary(position - Vec3(0,oldSize,oldSize),
32            newSize),
33     null, 0, Label.MIXED);
34   default -> newRoot = new Octree(
35     new Boundary(position - Vec3(oldSize), newSize),
36     null, 0, Label.MIXED);
37   }
38   newRoot.subdivideExpand(this, oldcode);
39   return newRoot;
40 }
```

Line 2 make sure it only expands the Octree if we explore outside the root Octree node boundary. Line 4 calculates in what direction the Octree needs to be expanded and stores that info as a code. This code will be the new code of the old root node and signify what child position it will get in the new root node's children array. The calculation is done using a big string of `if-then-else` statements on the x, y and z-axis differences between the player position (stored in `range`) and the center of the root Octree node (the boundary position). Lines 6-36 then create a new Octree root node, depending on the code calculated before. Then line 38 subdivides this new root node, but does not call `subdivide()`, but instead the `subdivideExpand()`. The method `subdivideExpand()` is similar, but takes the old root node as input, with the old code, and makes it the corresponding child-node of the new root-node. This method looks as follows:

```
public void subdivideExpand(Octree child, byte code) {
  children = new Octree[8];
  child.parent = this;
  child.code = code;
  float half = boundary.size / 2;
  Vec3 position = boundary.position;

  for (byte i = 1; i <= 8; i++) {
    if (code == i) {
      children[i-1] = child;
      continue;
    }
    Boundary bb;
    switch (i) {
      case 1 -> bb =
        new Boundary(position, half);
      case 2 -> bb =
        new Boundary(position + Vec3(half,   0,   0), half);
      case 3 -> bb =
        new Boundary(position + Vec3(   0,half,   0), half);
      case 4 -> bb =
        new Boundary(position + Vec3(half,half,   0), half);
      case 5 -> bb =
        new Boundary(position + Vec3(   0,   0,half), half);
      case 6 -> bb =
        new Boundary(position + Vec3(half,   0,half), half);
      case 7 -> bb =
        new Boundary(position + Vec3(   0,half,half), half);
      default -> bb =
        new Boundary(position + Vec3(half,half,half), half);
    }
    children[i-1] = new Octree(bb, this, i, Label.UNKNOWN);
  }
}
```

This method first sets the old root node's parent to itself (the new root node) and changes it code (lines 3-4). Then line 8 loops over the 8 child codes (the index +1, so 1-8). If the code (`i`) is equal to the old code (`code`), then it adds the old root node as a child (lines 9-11). Otherwise, it creates a new Octree node with half the size and adds it to the children array (lines 13-32). The switch statement is the same as the normal `subdivide()` method.

5.4 Agent Movement

Now that the agent can plan a path to their goal, it just needs a way to travel along said path. This is done using the primitive method `moveAndRotate()`, which takes a 3d movement vector and two rotation scalars. One for rotating the camera horizontally (yaw rotation) and one for rotating the camera vertically (pitch rotation). It also has a roll rotation input value, however it did not work. This was unfixable, as the implementation of this method is in the SE plugin, which was made by the developers of SE for the iv4XR project. The movement vector of `moveAndRotate()` moves the agent relative to its own axis system. This axis system is different from the global axis system and agent orientation. Thus, to move the agent in the right direction, the movement vector needs to be translated from global space to local space. Originally, iv4XR did this by rotating the vector around the y-axis using Rodrigues' rotation formula [5]. This works fine for 2D movement, but in 3D, we cannot easily assume the agent's up orientation is aligned with the y-axis. Thus, we changed it for our 3D implementation to a transformation matrix. Instead of expressing it in terms of sin and cos of angles, we expressed it in terms of dot products between global and local axis vectors. We derived this formula from [4].

$$R = \begin{bmatrix} \hat{x} \cdot \hat{x}' & \hat{x} \cdot \hat{y}' & \hat{x} \cdot \hat{z}' \\ \hat{y} \cdot \hat{x}' & \hat{y} \cdot \hat{y}' & \hat{y} \cdot \hat{z}' \\ \hat{z} \cdot \hat{x}' & \hat{z} \cdot \hat{y}' & \hat{z} \cdot \hat{z}' \end{bmatrix} \tag{1}$$

Using $v' = R * v$, we can rotate a vector from global space, with unit vector coordinates $(\hat{x}, \hat{y}, \hat{z})$, to local space, with unit vector coordinates $(\hat{x}', \hat{y}', \hat{z}')$.

Now to follow a path, the agent gets the next intermediate stop in the path and first checks if it is looking in that general direction. If not, it will rotate around the y-axis using `moveAndRotate()` and not move. If it is, it will not rotate, and it will move towards that direction by passing the transformed vector to `moveAndRotate()`. Both rotation and movement are done by sending a burst of successive rotate or move commands to SE. The burst will stop after a number of times or if it reached the goal. This continues every cycle until the end of the path is reached. Then it will check if the agent indeed is at the desired location to complete a move goal.

5.5 Padding

Paths consist of a list of points that the agent needs to travel through and most path planning algorithms, including the one we use, are graph based and consider the agent to be just a single point. However, the agent is not a single point, but a playable human character with a height, width, and depth. In SE, the agent has a height of 1.8 units, which are synonymous with meters, and a width and depth of 1.0 units or meters. If we do nothing to address the agent's dimensions, then when the agent is travelling along its planned path, it can bump into walls or even get stuck. This is a non-holonomic constraint on the movement. This constraint can be taken into account using corridors [28]. A corridor is a path with a width. The corridor width is as large as obstacles allow (or its specified maximum size). With corridors, the agent will move within the corridor around the path rather than just follow the path. This allows the agent to move more flexibly and allows the agent to avoid collisions more easily (as long as the corridor is wide enough). However, this is too costly and complicated for our program. Corridors allow the agent to handle many non-holonomic constraints, but for our application, where the only non-holonomic constraint we need to solve is the agent's size, it is inefficient. For static worlds, non-holonomic constraints can be considered during the pre-processing phase [19, 20].

A much simpler solution is to just pad all obstacles with extra size, so that we can plan assuming the agent is a single point. When we update the Octree with a new block, instead of using the block's bounding box and making all nodes that are inside it BLOCKED, we instead create a new bounding box, increase its size a bit and then use that. Doing this makes it so that the player can be seen as a single point, because all OPEN nodes have already taken the player size into account. The padding we add to the bounding box is $padding = (agent.height - minimumNodeSize)/2$. First of all, we consider only the height, not the agent width, because one, it is the bigger of the two. And two, because padding is done when the block is added to the Octree, after which the agent's orientation can change. If we first pad one unit in the y-axis and half a unit in the x- and z-axes, and then the agent rotates so that its up-orientation is in the x-direction, then the padding would not be enough. Secondly, we subtract half of the minimum node size. We do this because the agent always travels to the center of a node, so the distance from the center to the edge of the node is extra and can be removed from the padding. Worst case, this distance is half the minimum node size. In Figure 5, the blue line below the block is the normal size from the center. The grid shows the minimal nodes. For the example, we ignore cases where nodes can be combined. The block without padding will cover C5 and D5 fully. C4, D4, C6 and D6 will also be labeled as BLOCKED because we do not subdivide smaller than the minimal node size. When we pad the agent width to it (the red line), then we see that nodes C3 and D3 would also be BLOCKED, meaning the agent cannot travel to their center. However, you can clearly see that the agent can travel to it. So we subtract half the minimum node size (green line) from the padding, which makes C3 and D3 OPEN. Although the example shows it for the agent width, the same logic applies to the agent height, which is the one we use.

Padding is done, whenever a block needs to be processed in the Octree. The methods `addObstacle()`, `removeObstacle()` and `setUnknown()` all call the method `blockBB()` for this (see line 5 in `addObstacle()` in Section 5.1). This method goes as follows:

```
Boundary blockBB(Vec3 pos) {
  float vpadding = (AGENT_HEIGHT - MIN_NODE_SIZE) / 2;
  Vec3 size = new Vec3(1.25f + vpadding,
                       1.25f + vpadding,
                       1.25f + vpadding);
  return new Boundary(pos - size, pos + size);
}
```

Although padding guarantees it so that the pathfinder never finds an impossible path to traverse because of player size, it worsens

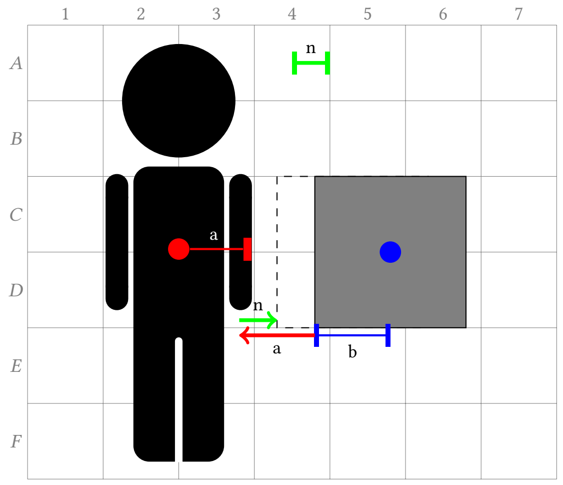


Figure 5: Padding: to make sure the agent can always travel to the center of a node, we pad blocks (with width $2b$) with agent width a and then subtract half the minimum node size n from it.

another issue. This issue exists because nodes can only be subdivided until the minimum node size. If this minimum node size is too big, then small hallways in SE, can disappear from the *WOM*, because the node(s) between the walls will overlap with part of the walls and become entirely blocked. This is especially an issue because of doors. A door is 2.5 units big and will almost always be surrounded by walls (except the front and back). Another issue is that when pathfinding from a location, the pathfinding will be done from the node the agent's center is located. The minimum worst case distance from the edge of a block and the center of the player is half the agent's height (0.9). Worst case, the node will just barely overlap with the block and the block will essentially be larger by the node size (lets call this node rounding). So if we want the agent's center to never be allocated to a blocked node, we need the node size s smaller than half the agent's height 0.9. Now, if we consider padding, this is not enough anymore. Now we need the node size s with padding p smaller than 0.9.

$$\begin{aligned} s + p &< 0.9 \\ s + (0.9 - \frac{s}{2}) &< 0.9 \\ s/2 &< 0 \end{aligned} \tag{2}$$

Solving for s, we get that the node size needs to be smaller than 0, but it also cannot be negative, so this is impossible. This makes sense, because the padding is 0.9 minus half the node size, and we add the node size because we rounded the nodes up. If we round the nodes down, then it becomes possible, however it would bring another problem, namely to guarantee the pathfinder will always find a path without collisions, we would need to add the node size to the padding.

$$\begin{aligned} p &< 0.9 \\ (0.9 - \frac{s}{2} + s) &< 0.9 \\ \frac{s}{2} &< 0 \end{aligned} \tag{3}$$

This is the exact same problem. To actually circumvent this problem, we release the condition of the player never being inside a blocked node. If we instead allow the agent to start pathfinding inside a blocked node and just not allow it to travel to one. Then, only when pathfinding starts in a blocked node, do we need to do something to guarantee the node the player is in cannot find a neighbouring OPEN node on the other side of the wall. This can happen if the nodes are too large, because then the wall and the OPEN area with the agents center in it can belong to a single blocked node, which makes it possible to find OPEN neighbours on both the agents side of the wall and the other side. To guarantee this to never happen for 1 block wide wall (2.5 game units), we need walls to at least collide with two nodes. Worst case for this would be if there is no node rounding going on.

$$\begin{aligned} 2s &< 2.5 + 2p \\ s &< 1.25 + p \\ s &< 1.25 + (0.9 - \frac{s}{2}) \\ 1.5s &< 2.15 \\ s &< 1.43333 \end{aligned} \tag{4}$$

So we only need the node size smaller than 1.43333. However, this will still not guarantee doors to have an OPEN node. To guarantee this, the distance between two neighbouring padded walls needs to be large enough to fit at least one node. We once again consider the worst case scenario of a node entirely being rounded up. The distance between both walls means that we pad twice, but we do not add two times node size because of rounding up. If we round up one wall, then the next wall's rounding depends on the first one. For a node size of 1, worst case for one wall is an extra 1 padding. But the second wall will not add 1, but 0.5, because in a block of 2.5, we can have 3 nodes of size 1. If the first is exactly colliding with the wall, then the second wall will be exactly halfway to the third node. So to calculate the required size, we solve for s in the following formula.

$$\begin{aligned} s &< 2.5 - (s + p + (2.5 \bmod s) + p) \\ s &< 2.5 - (s + 0.9 - \frac{s}{2} + (2.5 \bmod s) + 0.9 - \frac{s}{2}) \\ s &< 2.5 - s - 0.9 + \frac{s}{2} - (2.5 \bmod s) - 0.9 + \frac{s}{2} \\ s &< 0.7 - (2.5 \bmod s) \\ s + (2.5 \bmod s) &< 0.7 \end{aligned} \tag{5}$$

Let's say $r = 2.5 \bmod s$. By definition $0 <= r < s$. Worst case scenario, r approaches s, so then we get $s+s < 0.7$, so $s < 0.35$. Best case scenario, $r = 0$, so $s < 0.7$. Every value below 0.35 guarantees doors have an OPEN node, but between 0.35 and 0.7 are still some possible ranges of values. We ran a simple script to try out all values between them with a step size of 0.01. The possible ranges are in intervals $0 < s < \frac{5}{14}$, $0.36 < s < \frac{5}{12}$, $0.45 <= s <= 0.5 = \frac{5}{10}$ and $0.6 < s <= 0.625 = \frac{5}{8}$. The pattern being sizes between 0 and 5/14, 9/25 and 5/12, 9/20 and 5/10, 9/15 and 5/8. The largest possible value that guarantees an OPEN node for doors is thus 0.625. However, because of floating point arithmetic errors, we decided to use the node size 0.62. Now that doors will always have at least one OPEN node and the node size is small enough to guarantee at least two node thick walls, the problem is fixed for the VoxelGrid data-structure. The node size for Octree nodes are also good, except if the wall is two nodes thick, then those nodes are probably part of a group of 8 BLOCKED nodes, which are then combined as a single larger leaf-node. This makes the wall potentially once again one node wide. This however is worst case alignment and worst case combining, making it very improbable. So for Octrees, whenever we start pathfinding inside a blocked node, we can just temporarily split up the node again, then do the pathfinding like normal from its new smaller node and then after pathfinding recombine the split nodes back together.

5.6 Updating the Octree

All these implementations of pathfinding and exploration for the Octree data-structure are of course so we can use them for testing purposes. The iv4XR program already has such an implementation for this, with its Goals and Tactics. It also has an *AgentState* which holds the NavGrid (2D) and the logic to update it. We split this *AgentState* up into an *AgentState2D* and an *AgentState3D*. The data and methods that they both share, they inherit from the now abstract *AgentState*. The *AgentState3D* is also abstract and holds the generic methods and data for both the *AgentState3DOctree* and the *AgentState3DVoxelGrid*.

The update method inside the AgentState3DOctree works as follows:

```
1  public void updateState(String agentId) {
2    super.updateState(agentId);
3    WorldModel newWom = env().observe();
4    WorldModel gridsAndBlocksStates = env().observeBlocks();
5    for(var e : gridsAndBlocksStates.elements.entrySet()) {
6      newWom.elements.put(e.getKey(), e.getValue()) ;
7    }
8    if (wom == null) {
9      wom = newWom ;
10     grid.initializeGrid(wom.pos, OBSERVE_RADIUS);
11     for(var block : getAllBlocks(gridsAndBlocksStates)) {
12       addToOctree(block);
13     }
14     grid.updateUnknown(centerPos(), OBSERVE_RADIUS);
15   }
16   else {
17     var changes = wom.mergeNewObservation(newWom);
18     Octree newRoot = grid.checkAndExpand(wom.pos);
19     if (newRoot != null) grid = newRoot;
20
21     List<String> tobeRemoved = wom.elements.keySet().stream()
22       .filter(id -> !newWom.elements.containsKey(id))
23       .collect(Collectors.toList());
24     for(var id : tobeRemoved) wom.elements.remove(id) ;
25
26     for(var gridNew : changes) {
27       var gridOld = gridNew.getPreviousState();
28       if (gridOld == null) continue;
29
30       tobeRemoved.clear();
31       tobeRemoved = gridOld.elements.keySet().stream()
32         .filter(blockId -> !gridNew.elements.containsKey(blockId))
33         .collect(Collectors.toList());
34
35       boolean rebuild = false;
36       for (var blockId : tobeRemoved) {
37         var block = gridOld.elements.get(blockId);
38         if (Vec3.dist(block.pos, newWom.pos) < OBSERVE_RADIUS) {
39           grid.removeObstacle(block);
40           rebuild = true;
41         }
42       }
43       if (rebuild) {
44         for (var block : getAllBlocks(gridsAndBlocksStates)) {
45           addToOctree(block);
46         }
47       }
48       else {
49         List<String> tobeAdded = gridNew.elements.keySet().stream()
50           .filter(id -> !gridOld.elements.containsKey(id))
51           .toList();
52         for (var blockId : tobeAdded) {
53           addToOctree(gridNew.elements.get(blockId));
54         }
55       }
56     }
57   }
58 }
```

This update method first updates the environment (`env`). This is done inside the `updateState()` method in the parent class, so line 2 handles this. Next, the World Object Model (*WOM*) and the SE grids and blocks states are observed. The *WOM* holds all important player data. Then lines 5-6 add all grids and blocks states to this *WOM*. In SE, all connected blocks are grouped together in grids. The *WOM* we observe at the start, we name *newWOM*, because we will compare it with the previous one. If this is the first time we run this method, the old *WOM*, henceforth just *WOM*, will be `null`. If so, we set the *WOM* to the *newWOM*, initialize the Octree with a root-node that encompasses the entire viewing range and then add all observed blocks to the Octree (lines 9-12). Lastly, because all Octree nodes are initialized as UNKNOWN and because when we update it with the block states, we only set nodes to the BLOCKED state, this means there are no OPEN nodes yet. `updateUnkown()` fixes this by setting all UNKNOWN nodes inside the viewing range to OPEN (line 14). If this is not the first time we run the method, then we run the rest of the method. First we merge the new and the old *WOM* (line 17). This adds new observed grids and adds new observed blocks to their corresponding grids. Note that it only adds new grids or overwrite old ones with a newer state. The method returns a list of differences between the new and old *WOM*. Next, before we try to add anything to the Octree, we first check to see if the observation radius is still fully inside the root Octree node. If not, we expand the Octree so it is (line 18-19). See Section 5.1 for how this works. Lines 21-24 make sure the *WOM* (now merged) gets rid of any grids that are not in the *newWOM*, meaning they are now outside the viewing range. Lines 26-53 loop over the changes line 17 returned. These changes are in the form of a list of grids of changed (added or removed) blocks. First in the loop, we get the previous state of the changed grid and if it doesn't have one, it means it was added, so we skip the rest of the loop. Otherwise, we get a list of all blocks inside the viewing range in the grid that were in the old state, but not in the new. Then we loop over the removed blocks and check if they are inside the viewing range (lines 36-40). If they are inside, that means the blocks are removed from the SE world and should be removed from the Octree. If they are outside, that means the block is removed from the *WOM*, but probably still exists inside the SE world. It is now outside viewing range, so we do not update the Octree. After checking for all blocks, if any block was removed from the Octree, then we have a small problem, caused by padding. The area we made OPEN by removing the block is padded just like when we added the block, but the Octree nodes that are now OPEN could have intersected with multiple blocks, so it may still need to be BLOCKED. Because of this, we go over all blocks inside viewing range and re-add all of them to the Octree (lines

43-45). This makes sure any nodes that were incorrectly changed from BLOCKED to OPEN, are changed back. This is very expensive, but not a big problem, because it almost never happens. If no block was removed, then we get a list of all blocks that are in the new grid, but not in the old one. So for all newly observed blocks, we add them to the Octree (lines 49-53).

5.7 Testing

With the spatial partitioning data-structure in place, a way to update it based on observations and the ability to plan a path through it and a way to give the agent in SE movement commands, we only need to create the actual test methods that can be used to use them for testing purposes. The GoalStructures we slightly altered are:

(1) closeTo(Vec3 target)
(2) closeToBlock(Function selector, BlockSide side)

And the GoalStructures we added are:

(1) smartCloseToBlock(Function selector, BlockSide side)
(2) exploreTo(Vec3 target)
(3) explore()
(4) exploreToFind(Function selector, BlockSide side)
(5) closeToButton(int buttonNr)

Some goals require some information about the world that is unavailable before running the test. In these cases, instead of simply adding a GoalStructure to the test, we instead add a function that takes the agent state and insert a new GoalStructure after the current one. So instead of a test looking like:

```
GoalStructure G = SEQ(closeTo(door),
                      faceTowards(door)
                      );
```

It will look like:

```
GoalStructure G = SEQ(DEPLOYonce(agent,closeTo(door)),
                      DEPLOYonce(agent,faceTowards(door))
                      );
```

When G is created, all the test goals are created immediately, which is a problem when we want to wait until we need to perform the test goals so that we can observe the world in that instance. So if a goal is to get close to a door, then we do not want to look if there is a door and create a goal to go to that location immediately. Especially considering we create the root GoalStructure G before the agent even gets to observe the game world. So we instead use a DEPLOYonce goal, that when activated does not test anything, but instead creates a new goal using the current state and inserts it after itself. Then the DEPLOYonce goal is solved, and it will switch to the next goal, which is the newly inserted goal. DEPLOYonce removes itself from the goal structure, so it can only activate once. If you want to deploy a goal multiple times (in a REPEAT combinator), then you can use DEPLOY.

The Goal `closeTo()` takes a target location and simply tries to go there using pathfinding.

```
1  public Function<AgentState,GoalStructure> closeTo(Vec3 target)
2  {
3    return (AgentState state) -> {
4      Vec3 targetSq = state.getBlockCenter(target);
5      GoalStructure G = goal()
6        .toSolve((Pair<Vec3,Vec3> posAndOrientation) -> {
7          var agentPos = posAndOrientation.fst ;
8          return (targetSq - agentPos).lengthSq() <= DIST_THRESHOLD;
9        })
10       .withTactic( FIRSTof(navigateToTAC(target), ABORT()) )
11       .lift();
12     return G;
13   };
14 }
```

The function takes the state, which contains the Octree (or another spatial partitioning data-structure). Given a target location, it checks what node it is in and then creates the goal to get close to the center of that node. Lines 7-8 check if the Goal is solved. Line 10 defines what Tactics the GoalStructure uses to solve itself. The Tactic being used is first trying to navigate to the target location. If the Tactic is not enabled (failed the *guard*), then it switches to the ABORT Tactic, which makes the Goal fail. If the Tactic is enabled, then it returns an updated position and orientation, which are used to check if the Goal is solved. The only changes we made to this method from the original iv4XR implementation were taking the center of a 3D node instead of the center of a 2D node in line 4 and changing the inner workings of the `navigateToTAC()` Tactic.

The Tactic `navigateToTAC()` goes as follows:

```
public Tactic navigateToTAC(Vec3 destination) {
  return action("navigateTo")
    .do2((AgentState state)
        -> (Pair<List<DPos3>, Boolean> queryResult) -> {
      var path = queryResult.fst;
      var arrivedAtDestination = queryResult.snd;

      if (arrivedAtDestination) {
        state.currentPathToFollow.clear();
        return Pair<>(state.pos(), state.orientationForward());
      }
      state.currentPathToFollow = path;

      var next = state.currentPathToFollow.get(0);
      var nextPos = state.getBlockCenter(nextNode);
      if ((nextPos - state.pos()).lengthSq() <= DIST_THRESHOLD) {
        state.currentPathToFollow.remove(0) ;
        return Pair<>(state.pos(), state.orientationForward());
      }
      CharacterObservation obs;
      obs = yTurnTowardACT(state, nextNodePos, 0.8f, 10) ;
      if (obs != null) {
        return Pair<>(obs.position(), obs.orientationForward());
      }
      obs = moveToward(state, nextNodePos, 20);
      return Pair<>(obs.position(), obs.orientationForward());
    })
    .on((AgentState state) -> {
      if (state.wom == null) return null;
      var agentNode = state.getGridPos(state.pos());
      var destNode = state.getGridPos(destination);
      var destNodePos = state.getBlockCenter(destNode);
      if ((destNodePos - state.pos()).lengthSq() <= DIST_THRESHOLD)
        return Pair<>(state.currentPathToFollow, true);

      int pathLength = state.currentPathToFollow.size();
      if (pathLength == 0 ||
          destNode != state.currentPathToFollow.get(pathLength-1))
      {
        var path = state.pathfinder.findPath(state.getGrid(),
                                             agentNode, destNode);
        if (path == null) return null;
        path = smoothenPath(path);
        return Pair<>(path, false);
      }
      else {
        return Pair<>(state.currentPathToFollow,false);
      }
    })
    .lift();
}
```

The *guard* is defined in lines 28-47 and the *effect* part in lines 3-26. The Tactic has also been slightly altered from the original iv4XR implementation (see Section 2.3.2). Instead of using the NavGrid (2D version) to get the nodes (lines 15 and 30-32), we use a generic reference to the generic *grid* class. If, for example, this Tactic is ran with the Octree, then the `getGridPos()` method of the Octree will be called. Also, the `yTurnTowardsACT()` and `moveTowards()` method from lines 21 and 25 were changed to work regardless of orientation. This change was explained in Section 5.4.

The goal `closeTo()` can also be done with a specified block type as the target instead of a location. This is the `closeToBlock()` GoalStructure. Instead of taking a `Vec3` as a target, it takes a selector function and block side as parameters. Using the selector function, it then tries to find a block inside the *WOM* that satisfies the selector condition and calculates the block's position. From that point on, the GoalStructure works the same as a regular `closeTo()` GoalStructure.

In addition to the normal `closeToBlock()` GoalStructure, we also made a more intelligent one, capable of exploring and pressing a (single) button to potentially find a way to a goal. This `smartCloseToBlock()` goal is defined as follows:

```
1  public Function<AgentState,GoalStructure> smartCloseToBlock(
2    Agent agent,
3    Function<AgentState, Predicate<WorldEntity>> selector,
4    BlockSide side)
5  {
6    return (AgentState state) -> {
7      Function<Void, GoalStructure> G = (unused) -> goal()
8        .toSolve((Pair<Vec3,Vec3> posAndOrientation) -> {
9          var block = findClosestBlock(state.wom,
10                                        selector.apply(state));
11         if (block == null) return false;
12
13         Vec3 targetPos = getSideCenterPoint(block, side);
14         Vec3 targetSq = state.getBlockCenter(targetPos);
15
16         var agentPos = posAndOrientation.fst ;
17         return (targetSq - agentPos).lengthSq() <= DIST_THRESHOLD;
18       })
19       .withTactic(
20         FIRSTof(navigateToBlockTAC(selector, side), ABORT()) )
21       .lift() ;
22
23     return FIRSTof(
24       SEQ(G.apply(null),
25         DEPLOYonce(agent, faceTowardBlock(selector))
26       ),
27       SEQ(explore(selector, side).apply(state),
28         G.apply(null),
29         DEPLOYonce(agent, faceTowardBlock(selector))
30       ),
31       SEQ(DEPLOYonce(agent, closeToButton(0)),
32         pressButton(0),
33         explore(selector, side).apply(state),
34         G.apply(null),
35         DEPLOYonce(agent, faceTowardBlock(selector))
36       )
37     );
38   };
39 }
```

This GoalStructure is similar to the `closeToBlock()` GoalStructure. The goal `G` it creates in lines 7-21 is the same `closeToBlock()` goal, but it takes the current state to create itself and it itself to the SEQ GoalStructure combinator. The `smartCloseToBlock()` GoalStructure is a combinator of the goal G and some others. It uses a FIRSTof to first try to find a path to the goal in the current *WOM* (lines 23-24). If it succeeds the goal G, then it follows it up with a `faceTowardsBlock()` goal (line 25). If it fails, because it cannot find the target block or if no path to the target block is possible, then it goes to the `explore()` goal (line 27). The `explore()` goal explores the world until there are no more reachable unknown nodes. If it finishes, then it tries goal G (go to goal block) once again (line 28). The reason we made G a function that creates a GoalStructure is because we want to create it to use the newly updated state. So we apply the function to create a fresh instance of goal G. If the goal G fails again, then we try to go to a button (line 31). If there is a button, then after getting to it, we switch to the `pressButton()` goal (line 32). The `pressButton()` goal, presses the button and sets all nodes that collide with a door to be UNKNOWN again. Thus, after succeeding in pressing a button, we can explore again. Finally, after exploring for the last time, we try to go to the goal block one final time (line 34). If at any point in the final SEQ (lines 31-35) any goal fails, then the whole `smartCloseToBlock()` GoalStructure fails.

The `exploreTo()` GoalStructure does pathfinding to some target location, but once it finds a node neighbouring an unknown node, it returns a path to that node. The GoalStructure repeats this until it finds a path to the node containing the target location. The `explore()` GoalStructure repeatedly goes to the closest node neighbouring an unknown node until the entire navigable world is known. The `exploreToFind()` GoalStructure does the same as `explore()`, except it early outs whenever it finds a path to some goal block. Lastly, the `closeToButton()` GoalStructure simply goes to a button.

5.8 Voxel Grid

With the Octree fully working, we will explain how we made the simpler data-structure, which we will use as a baseline to compare the Octree with. This data-structure is the VoxelGrid, which as the name implies is simply a 3D grid of voxels. The 3D grid is implemented as follows:

```
ArrayList<ArrayList<ArrayList<Voxel>>> grid;
```

With a `Voxel` being defines as follows:

```
public class Voxel {
  public byte label;

  public Voxel() {
    this.label = Label.UNKNOWN;
  }
  public Voxel(byte label) {
    this.label = label;
  }
}
```

A voxel only contains a single piece of data, namely its `label`. Just like the Octree, the `label` can be one of the same four possible types: BLOCKED, OPEN, MIXED or UNKNOWN. Voxels have two constructor methods. One with a provided label and one without. With a label provided, the voxel will simply start with that label. With no label provided, the voxel will get the `UNKNOWN` label. This is used whenever the VoxelGrid needs to be expanded in order to contain the whole viewing range of the agent.

Whenever the agent observes an object, the VoxelGrid will update the labels of the voxels that overlap with the object. This is done using the following `addObstacle()` method:

```
1  public void addObstacle(WorldEntity block) {
2    List<DPos3> obstructedVoxels = getObstructedVoxels(block);
3    for (var voxel : obstructedVoxels) {
4      get(voxel).label = Label.BLOCKED;
5    }
6  }
```

First, in line 2, `getObstructedVoxels()` will calculate which (potential) voxels overlap with the block. This is in the form of a list of 3d integer coordinates. The x,y,z parts of the coordinates can then be used as indices in the 3D grid to get the corresponding voxel. Then it loops over all the voxel coordinates and sets their labels to be `BLOCKED`. Adding an obstacle to the *grid* cannot be done if the obstacle is outside of it. To ensure there are no errors, we use the `expand` method to expand the *grid* so that the viewing range is always inside the *grid*. That way, whenever we observe and add an obstacle to the *grid*, we can know for sure that the obstructed voxel coordinates are all inside the *grid*. The method `getObstructedVoxels()` works as follows:

```
1  List<DPos3> getObstructedVoxels(WorldEntity block) {
2    List<DPos3> obstructed = new LinkedList<>() ;
3
4    Vec3 maxCorner = SEBlockFunctions.getBaseMaxCorner(block);
5    Vec3 minCorner = SEBlockFunctions.getBaseMinCorner(block);
6
7    float vpadding = new Vec3((AGENT_HEIGHT - voxelSize) / 2);
8    minCorner = minCorner - padding;
9    maxCorner = maxCorner + padding;
10
11   var corner1 = gridProjectedLocation(minCorner) ;
12   var corner2 = gridProjectedLocation(maxCorner) ;
13   for(int x = corner1.x; x <= corner2.x; x++) {
14     for (int y = corner1.y; y <= corner2.y; y++) {
15       for (int z = corner1.z; z <= corner2.z; z++) {
16         var voxel = new DPos3(x,y,z) ;
17         obstructed.add(voxel) ;
18       }
19     }
20   }
21   return obstructed ;
22 }
```

Given a block, it gets the minimum and maximum corners of the block (lines 4-5). Then it pads them with some extra distance to make it so that the agent can consider itself a single point during pathfinding. Padding is explained in Section 5.5. After that, it projects the minimum and maximum corners to VoxelGrid coordinates. This projection subtracts the VoxelGrid position from it, to make the position relative to the VoxelGrid (with (0,0,0) being the bottom-left-front corner). Then it divides the relative position by the voxel size and rounds the value down to the nearest integer, to get the x,y,z coordinates in the 3D grid. After projection, lines 13-17 add all voxels between the minimum and maximum corners to the list of obstructed voxels. Finally, it returns this list.

The method `checkAndExpand()`, which is responsible for making sure no observed blocks are outside the VoxelGrid, is as follows:

```
public void checkAndExpand(Boundary range, AgentState state) {
  if (boundary.contains(range))
    return;

  if (range.pos.x < boundary.lowerBounds.x) {
    DPos3 gridSize = size();
    int diff = ⌈boundary.lowerBounds.x - range.pos.x⌉;
    for (int x = 0; x < diff; x++) {
      ArrayList<ArrayList<Voxel>> newX =
          Stream.generate(() -> Stream.generate(Voxel::new)
              .limit(gridSize.z)
              .collect(Collectors.toCollection(ArrayList::new)))
          .limit(gridSize.y)
          .collect(Collectors.toCollection(ArrayList::new));
      grid.add(0, newX);
    }
    boundary.lowerBounds.x -= voxelSize * diff;
    state.currentPathToFollow.forEach(DPos3 -> DPos3.x += diff);
  }
  else if (range.upperBounds().x > boundary.upperBounds.x) {
    DPos3 gridSize = size();
    int diff = ⌈range.upperBounds.x - boundary.upperBounds.x⌉;
    for (int x = 0; x < diff; x++) {
      ArrayList<ArrayList<Voxel>> newX =
          Stream.generate(() -> Stream.generate(Voxel::new)
              .limit(gridSize.z)
              .collect(Collectors.toCollection(ArrayList::new)))
          .limit(gridSize.y)
          .collect(Collectors.toCollection(ArrayList::new));
      grid.add(newX);
    }
    boundary.upperBounds.x += voxelSize * diff;
  }
  if (range.pos.y < boundary.lowerBounds.y) {
    DPos3 gridSize = size();
    int diff = ⌈boundary.lowerBounds.y - range.pos.y⌉;
    for (int x = 0; x < gridSize.x; x++) {
      for (int y = 0; y < diff; y++) {
        ArrayList<Voxel> newY = Stream.generate(Voxel::new)
            .limit(gridSize.z)
            .collect(Collectors.toCollection(ArrayList::new));
        grid.get(x).add(0, newY);
      }
    }
    boundary.lowerBounds.y -= voxelSize * diff;
    state.currentPathToFollow.forEach(DPos3 -> DPos3.y += diff);
  }
  else if (range.upperBounds.y > boundary.upperBounds.y) {
    DPos3 gridSize = size();
    int diff = ⌈range.upperBounds.y - boundary.upperBounds.y⌉;
    for (int x = 0; x < gridSize.x; x++) {
      for (int y = 0; y < diff; y++) {
        ArrayList<Voxel> newY = Stream.generate(Voxel::new)
            .limit(gridSize.z)
            .collect(Collectors.toCollection(ArrayList::new));
        grid.get(x).add(newY);
      }
    }
    boundary.upperBounds.y += voxelSize * diff;
  }
  if (range.pos.z < boundary.lowerBounds.z) {
    DPos3 gridSize = size();
    int diff = ⌈boundary.lowerBounds.z - range.pos.z⌉;
    for (int x = 0; x < gridSize.x; x++) {
      for (int y = 0; y < gridSize.y; y++) {
        for (int z = 0; z < diff; z++) {
          grid.get(x).get(y).add(0, new Voxel());
        }
      }
    }
    boundary.lowerBounds.z -= voxelSize * diff;
    state.currentPathToFollow.forEach(DPos3 -> DPos3.z += diff);
  }
  else if (range.upperBounds.z > boundary.upperBounds.z) {
    DPos3 gridSize = size();
    int diff = ⌈range.upperBounds.z - boundary.upperBounds.z⌉;
    for (int x = 0; x < gridSize.x; x++) {
      for (int y = 0; y < gridSize.y; y++) {
        for (int z = 0; z < diff; z++) {
          grid.get(x).get(y).add(new Voxel());
        }
      }
    }
    boundary.upperBounds.z += voxelSize * diff;
  }
}
```

Although the code is optimized Java code, the idea is simple and works in any language. Lines 5-18 are for expanding to the right (positive x), 20-32 for expanding to the left (negative x), 34-46 for expanding downwards (negative y), 48-59 for expanding upwards (positive y), 61-72 for expanding to the front (negative z) and 74-84 for expanding to the back (positive z). All the negative directions update the VoxelGrid's minimum position (lower boundary) and update the `currentPathToFollow` too (lines 17-18, 45-46 and 71-72). Because we store the state's current path it is following as a list of node coordinates, we have to update the coordinates if the minimum position has changed. For example, position `DPos3(2,0,0)` would be two `voxelSize` units to the right of the lower boundary. So if we expanded to the left, the lower boundary has moved `diff` units to the left, so the path's coordinates all need to be moved `diff` units to the right. Positions are calculated using the lower boundary, so if we expanded in a positive direction, we do not need to update the `currentPathToFollow`. At the start of every expansion direction is calculating how much needs to be expanded (`diff` in lines 7, 22, 36, 50, 63 and 76). Then for the x-axis, we generate an `ArrayList` of `ArrayLists` of `Voxels` with lengths `size.y` and `size.z`. Lines 9-14 (lines 24-29 are the same) is an efficient method to do this in Java. Then we add this at x-index `0` of the grid if we expand to the left (line 15) or at the end if we expand to the right (line 30). For the y-axis, we basically do the same, except we loop over the x-array of the 3d array and generate an `ArrayList` of `Voxels` instead. Then for every x in the 3D array, we add the new y-array at the end or at index `0`. For the z-axis, we loop over the x-array, then loop over the y-array and simply add a new voxel at the end or at index `0`. All of these together make `addObstacle()` work. `removeObstacle()` and `setUnknown()` work the same way, but instead of making voxel labels `BLOCKED`, they make them OPEN and UNKNOWN respectively.

Given a VoxelGrid node, which you can easily get by projecting the agent's position to the VoxelGrid, you have the coordinates of its voxel, so you can easily get its neighbours by checking the voxels neighbouring the coordinates (+1 or -1 for x,y,z).

5.9 3D NavGrid

The original iv4XR implementation used a 2D NavGrid as its data partitioning data-structure. To be able to compare our Octree implementation with the original implementation, we need to upgrade the 2D NavGrid to 3D. We decided to keep the changes as minimal as possible and follow the same design structure as the original. This however, meant that it would not be possible to do exploration with it, as the original 2D implementation could also not do it. Adding exploration functionality the same way as done for the Octree and VoxelGrid was also not possible, because they do so with a label. The NavGrid, by nature, does not store a label, as it only stores BLOCKED nodes. Changing the 2D NavGrid to 3D was simple, requiring just an additional axis to be stored and used for querying.

6 RESULTS

With the Octree, VoxelGrid and 3D NavGrid implemented, we need to test how efficient the Octree is compared to the baseline VoxelGrid and 3D NavGrid. In this section, we will test the performance

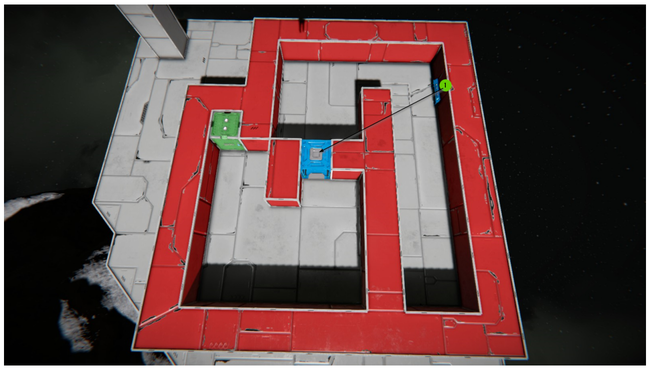


Figure 6: Test world 1

of our Octree data structure for both speed and memory efficiency. We will compare the performance of the method in four test scenarios with two baseline methods.

6.1 Tests

The test worlds in which we will run the program are as follows:

Test world 1. A simple small world with a little room for moving around, one closed door, a button which opens said door and a goal past the door. This test world is the same as one from the original iv4XR test worlds, but fully enclosed, so the 3D pathfinder cannot reach the goal without going through the door, and with a button to open the closed door. This test shows our method can be used for basic level tests. Figure 6 shows the layout of the test world, with the roof removed for visibility. The (red) walls also have been reduced to just 1 block high. The door (blue block in the center) is initially closed and can be opened by pressing the button on the (blue) button-panel at the top-right. The green block on the left is the goal to reach. The agent starts at the bottom right and has to get in front of the green block. To solve this test goal, we use the `smartCloseTo()` GoalStructure. It will first fail to find a path to the goal. Then it will explore all nodes it can reach. For a viewing range of about 20 units, this will have the agent end up just around the corner, between the door and button panel. After that, it will go to the button panel and press it. Then, once more, it will fail to find a path to the goal, because the area behind the door was outside the viewing range. This, the agent will explore again and end up just behind the door. From there, it will find a path to the goal, so it stops exploring and directly go towards it.

Test world 2. A complicated small world with three doors and four buttons. The layout can be seen in Figure 7. Just like the previous figure, we once again removed the roof and shortened the wall heights. All three (blue) doors start closed. Button 1 (at the top-left) toggles door 1, button 2 (in the middle) also toggles door 1 and button 3 (at the bottom-left) toggles all three doors. The order the agent has to take to reach the goal is: first, press button 1 to open door 1, then press button 3 to open the door to the goal, but also closes door 1, then press button 2 to once again open door 1, finally go to the goal. This test is the exact same test as CR3_3_3_M from the lab recruits game testing contest 2021 [22], but with a roof. This test shows our method can be used for complicated test scenarios. It is possible to create a GoalStructure smart enough to

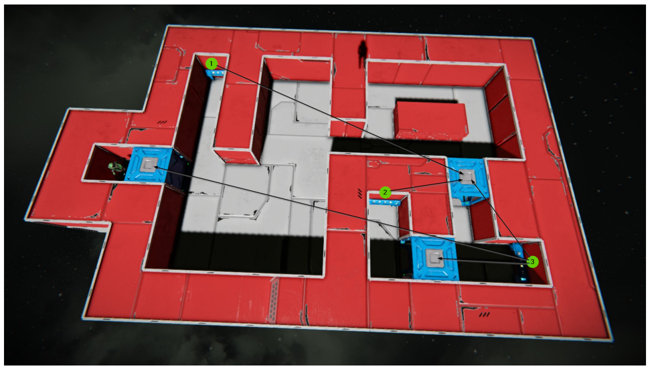


Figure 7: Test world 2

solve the test world without providing information on what the buttons do or in what order to press them. However, we decided to just provide the button order and test if the agent can complete the test with knowledge of the puzzle solution. The GoalStructure we used is defined as follows:

```
GoalStructure G = SEQ(
  DEPLOYonce(agent, closeToButton(0)),
  pressButton(0),
  DEPLOYonce(agent, closeToButton(2)),
  pressButton(2),
  DEPLOYonce(agent, closeToButton(1)),
  pressButton(1),
  DEPLOYonce(agent, close3DTo(
      agent, "TargetDummy",
      SEBlockFunctions.BlockSides.BACK,
      50f, 0.5f))
);
```

Just like in test world 1, the viewing range was also set to 20 units, but this time, the world is small enough for the agent to never need to explore.

3. Glass box. A very large and very empty world. The world is a giant enclosed glass cube to prevent the agent from going too far away from the important locations. Inside the glass cube is a building where the agent starts, another unconnected building floats some distance away with the goal inside, but the door is closed. The button to open the door much higher, some distance away. Just like in test 1, the agent has to press the button to open the door, to reach the goal. This test, shows the method works for large open 3D spaces and allows us to measure how the different techniques work in such environments. We originally planned to test it using the `smartCloseTo()` method. However, we found that for such large open spaces, that the exploration method for the VoxelGrid and NavGrid was very time-consuming. So we split the test for this test world up into two separate tests. Test 3a is almost the same as the original, however we skip any exploration steps and make sure the entire world is explored by setting the observation radius to encompass the entire test world. Test 3b simply measures the time it takes to complete a single explore call. We position the agent at the center of the test world. We measure the time it takes for exploration to the closest unknown node for the viewing ranges 10 to 70 with every multiple of 10 in between. Figure 8 shows the test world from multiple angles. The first (top-left) image shows the zoomed out layout. The second image shows the building in which the agent starts the test. The third image shows the building in which the goal is. The closed door is marked in blue. The button to open the door is shown in the fourth image (bottom-left). The button is also marked blue, and you can see the blue door in the background. The final image shows the goal (marked green) behind the glass inside the building in the third image.

The final test world, test world 4 is the same as test world 3, but with a lot of added extra obstacles to fill the empty space. This test allows us to measure the difference empty space makes on the memory and time efficiency of the Octree compared to the baseline VoxelGrid and NavGrid data structures. Figure 9 shows this test world from the same angles as those in Figure 8. The original buildings, doors, and buttons have not changed, so the test plays out the same, except the agent has to plan around a lot of additional obstacles.

6.2 Metrics

To show how effective our Octree data-structure is for testing, we will compare our method in multiple ways. We will measure the total time it takes the tests from start to finish, as well as the time spent on some sub-tasks. These sub-tasks include: the travel time, pathfinding time, exploration time, expanding the 'grid' (Octree/VoxelGrid/NavGrid), adding to or removing obstacles from the 'grid', retrieving the neighbouring nodes from a node and initializing the 'grid'. Initializing the grid is only done once, so the time it takes has no significant impact on the performance, as long as it doesn't take too long. Every update, the agent observes the world and all new obstacles are added to the grid. We measure the time it takes to add all new obstacles as a group in a single observation and take the average over the observations for the `addToGrid()` time. When the grid has to be expanded, we measure the time it takes to do so and take the average of all expanding times. Every time pathfinding to some location is done, we measure the time it took and take the average. Pathfinding is done using the A* search algorithm, with as heuristic the Euclidean distance to the target. Although more expensive, we chose the Euclidean distance over the Manhattan distance to reduce the amount of nodes the pathfinder has to go through, as this is quite a large difference in 3D. Once a path has been found, pathfinding will not be done again until the agent has reached the end of the path. Every time exploration is done, which is just pathfinding, but going to no specific location and thus expanding out in every direction all at once, we measure the time and record the average. Exploration is done using A*, but without a heuristic, making it effectively equivalent to Dijkstra's algorithm. Pathfinding using A* uses the `GetNeighbour()` method to get the neighbours, which is the second biggest difference in time efficiency between the different methods. Just like for the pathfinding time, the exploration time does not include travel time, only the time it takes to find a path. The biggest difference being the number of nodes that need to be traversed. So we also measure the average time it takes for one `GetNeighbour()` call. Finally, we also measure the total time spent on moving around in the game world. This allows us to see if the difference in total time is due to an overall more time efficient data structure, or because the test took longer because the agent just moved slower.

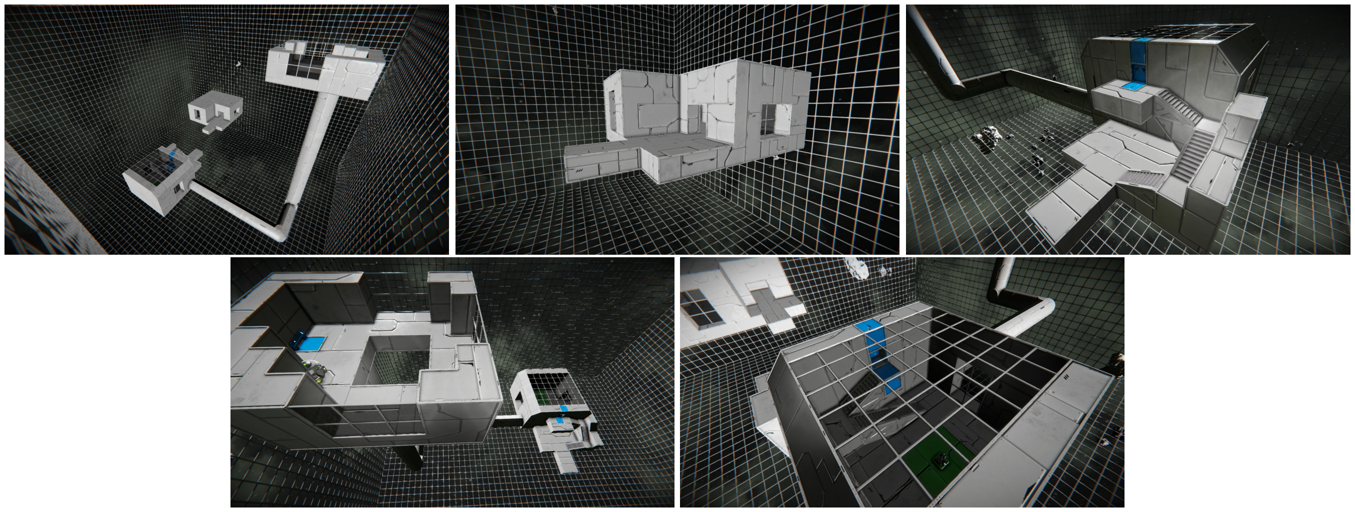


Figure 8: Test world 3

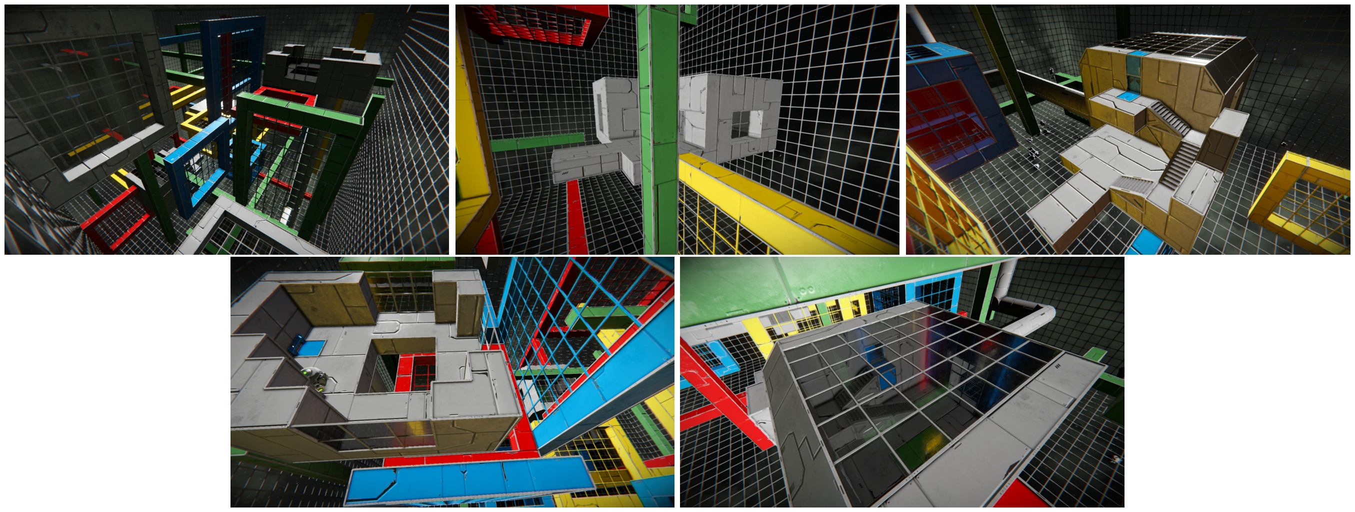


Figure 9: Test world 4

We also measure the memory usage of the data structures in all four test worlds. We used a viewing range of 20 for the two smaller test worlds: 1 and 2. For test worlds 3 and 4 we used a viewing range of 50 to reduce the time it takes to measure it. We measure the memory usage by exploring the world until all reachable locations are explored. For the Octree, this means we have to explore the whole world. For the VoxelGrid, we have to go to the two furthest corners of the map. And for the NavGrid, we just need to have observed all blocks, which we do by having the viewing range encompass the entire test world. To measure memory usage, we use *IntelliJ Ultimate*'s profiler to take a memory snapshot after exploring the world or filling the 'grid'. To measure the time for tests, we simply start a timer at the start of the code we want to measure and end it at the end. For code that generally takes longer, we measure the time in milliseconds, the rest we measure in nanoseconds.

6.3 System Specifications

The desktop PC we used for our performance measurements has the following hardware and software specifications.

(1) Processor (CPU): Intel(R) Xeon(R) W-2125, 4 cores, 4.00GHz, 4.01 GHz
(2) Memory (RAM): 32.0 GB (31.7 GB usable) DIMM, 2666 MHz
(3) Caches: L1: 265 KB, L2: 4.0 MB, L3: 8.2 MB
(4) Graphics Processing Unit (GPU): NVIDIA Quadro P4000, 8 GB GDDR5
(5) Operating System: Windows 10 Pro, version 22H2
(6) Java Version: Jetbrains Runtime 17.0.9 (jbr-17)

6.4 Results

In this section, we will show the resulting statistics that came out of the tests. We split the results into memory usage, speed and testing capabilities and will go over them one by one.

Table 1: Memory usage of data-structures per test

Memory	Test 1	Test 2	Test 3	Test 4
Octree	4.97 MB	2.90 MB	88.90 MB	107.15 MB
VoxelGrid	19.02 MB	16.81 MB	309.23 MB	310.48 MB
NavGrid	2.26 MB	1.37 MB	77.26 MB	102.43 MB

6.4.1 Memory.

In Table 1 you can see the amount of memory is used for tests 1 to 4. In all tests, the NavGrid uses the least amount of memory, with the Octree a close second place. The VoxelGrid uses significantly more memory than the other two. Test 3 and 4 have the same dimensions, but test 4 has significantly less empty space in it. For the VoxelGrid, this just means that some open nodes will be blocked. The memory usage will stay the same. The small difference in the table is because of semi-random exploration, expanding the grid slightly differently. The 3D grid in test 3 had size (253x272x217), whilst the grid in test 4 had size (254x273x216). For the Octree, a more filled world means that a lot of large fully empty nodes have to be split up, going from around 1.27 million nodes to 1.53 million. So for more full worlds, the Octree will have to store more nodes, which takes more memory space. The NavGrid only stores BLOCKED nodes, so more obstacles in the world results in more blocked nodes, which takes more memory. All in all, the NavGrid is the most memory efficient data-structure of the three based on the test worlds, with the Octree a close second place. However, the test worlds all use walls of a single block wide, never having large areas of BLOCKED nodes grouped together. So the Octree will mostly combine OPEN nodes or UNKNOWN nodes together at large scales. In worlds with more obstacles close together than open areas, then the Octree will stay equally memory efficient, but the NavGrid will be far worse. Thus even though the NavGrid is slightly more memory efficient for most scenarios, the Octree, being slightly slower in such scenarios, is in our opinion still the better data-structure memory wise, because it is more consistently memory efficient across all scenarios. It is still theoretically possible to have a world where the VoxelGrid is more memory efficient than both the Octree and NavGrid. This is the case for a world with alternating obstacles and open space. Every Octree node uses 32 bytes, while a VoxelGrid node only uses 16 bytes. The VoxelGrid also has a lot of memory overhead because of storing the voxels in a 3D `arrayList`. This overhead is about 8 bytes per voxel. A single NavGrid node uses 56 bytes, excluding the overhead. So for a world where the Octree can minimally optimize nodes together into a single parent-node and where there are more BLOCKED nodes than OPEN nodes, the VoxelGrid should be more memory efficient than the Octree and NavGrid. This however is very unlikely in any 3D world to happen, so we can safely say the VoxelGrid method is still the worst data-structure for testing in 3D worlds of unknown size.

A short explanation on the memory usage values is as follows. For Octrees the memory usage of a single Octree node is 32 bytes shallow size in Java. Shallow size is just the memory used by the Object, including the memory used by references to other Java Objects. The memory used of the referenced Objects is not included. The retained memory size is an Object's shallow size + all the memory used by Objects it is referencing to. An Octree node holds 1 byte for the code, 1 byte for the label, 3*4 = 12 bytes for the references to the parent, children array and the boundary. Lastly, the Octree node Object header is another 16 bytes. In total, it uses 2+12+16 = 30 bytes, which is padded to 32 bytes to be a multiple of 8. The Boundary holds a position and a size for a total of 4 floats, which is 16 bytes. Add the Object header of another 16 bytes for exactly 32 bytes. The parent is not counted as part of the retained size of a node, because it is part of another node. The same counts for the children references in the array. Although the actual Octree nodes aren't counted towards the retained memory size, the array that holds them is. The array's overhead is 16 bytes and then another 4 bytes for the 8 references to the children, for a total of 48 bytes. If the node is a leaf-node, then the memory usage for the array is 0 bytes, because it will be null. The ration parent-nodes to leaf-nodes will start at 1:1 with a single leaf-node and with increasing tree size approach 1:8 The total (retained) memory usage m for n Octree nodes is then approaching $m = n * 64 + n * 48$.

For the VoxelGrid, the memory usage of a single Voxel is 16 bytes. This is because although it holds only a single byte of data, the Voxel is a Java Object and is thus padded to be its minimum size of 16 bytes. We need it to be an Object to be able to use it as a node in our 3D grid, which is a 3D `ArrayList`. It is possible to only use a single byte with a 3D array, but expanding an array during exploration would be too time-consuming. The shallow memory usage is the number of nodes $n * 16$ bytes. The retained memory includes the `ArrayLists` extra memory usage is the overhead of 24 bytes and the backing array. An `ArrayList` whenever it is expanded, expands its size with more than 1, so that we do not need to change the size too often. The extra unused size is the backing array and only uses the reference memory size. The reference size is 8 bytes, but for references to Objects with small memory consumption, compressed OOP (ordinary object pointer) is done to reduce it to 4 bytes per reference. For the VoxelGrid, this compression is done for the z-axis `arrayList`, because the Objects it references are just 16 bytes. The y-axis and z-axis do not use compression, because the y-axis `arrayList` references a whole z-axis `arrayList` and the x-axis `arrayList` references a whole y-axis `arrayList` of z-axis `arrayLists`.

For the NavGrid, the memory usage of a single node is 56 bytes. A node contains a key, a value, hash of a key and a pointer to the next node. The key is a pointer to a `DPos3`, which takes 24 bytes, the pointer takes 4 bytes. The value, although unused, still takes 16 bytes. The hash takes 4 bytes, and the pointer to the next node also takes 4 bytes. The node itself is padded to 32 bytes (4+16+4+4=28). With the `DPos3` added, a single node takes 56 bytes. Unoccupied `HashSet` entries reserve 4 bytes for the pointer to a node whenever it gets added. So for a `HashSet` of size n with a capacity of c, the retained memory usage could be calculated using the formula $n * 56 + c * 4$. This would be correct for a `HashMap`, but a `HashSet` uses a bucket structure and replaces some nodes with tree-nodes. This is done to improve the time it takes to do lookups. Tree-nodes contain a key, a value, hash of a key and pointer to the next node just like the normal node. However, it also contains a pointer to the previous node, parent-node, right and left nodes, before and after nodes, and a boolean. These add another 6*4+1=25 bytes to the original 28 bytes, bringing it to 53 bytes. Then it is padded to 56 bytes, which together with the `DPos3` makes a single tree-node take 80 bytes.

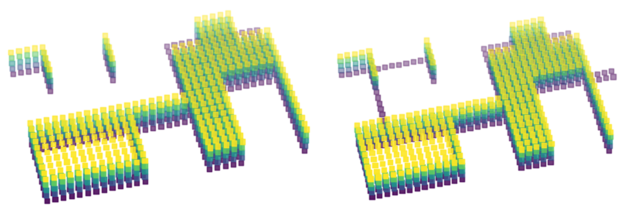


Figure 10: Visualisation of all accessible OPEN VoxelGrid nodes, with on the left all doors closed and on the right all doors open.

For only tree-nodes, the formula would then be $n * 80 + c * 4$. From our observations, the ratio of tree-nodes to normal nodes seems to be around 3:2.

6.4.2 *Speed.*

In Table 2 you can see the time it took each test in total and for some important parts individually. Table 3 shows the number of agent turns each test took. For tests 1 and 2, we used a viewing range of 20 units. For test world 2, this means exploration is never done, which is why it is missing in the table. For test world 1, there is exploration, which the NavGrid cannot do. So to circumvent this, we took the resulting location from the VoxelGrid explore call and made it so that the NavGrid does pathfinding towards this location instead of using the explore method. This does however skew the average pathfinding result of the NavGrid, making it less comparable with the other two. For tests 3a and 4a, we used a viewing range of 120 units, which is just enough for the agent to always have their next goal in their viewing range. The total time for each test is similar for the Octree, VoxelGrid and NavGrid in tests 1 and 2. In tests 3a and 4a, the total time of the Octree is significantly lower than both the VoxelGrid and NavGrid. The difference between the VoxelGrid and NavGrid is insignificant. The average pathfinding time in test 1 is a bit longer for the VoxelGrid than the Octree. The NavGrid is a bit slower than the VoxelGrid, however, as mentioned before, this could be skewed. Theoretically it should be very similar to the VoxelGrid, and based on a lower `GetNeighbour()` time, it should be a little faster. Removing the pathfinding time that replaced the exploration call from the average gets us an average time of 19.9 ms, which is, just like theorized, a little faster than the VoxelGrid is, but still slower than the Octree. This is because the Octree has combined multiple open nodes together, reducing the number of nodes that need to be traversed. In test 2, the reverse is happening, with the Octree taking a bit longer on average. This is because in test 2, as visualised in Figure 10 every space is exactly 1 node wide, so no open nodes in the Octree could be combined, making the amount of nodes to traverse the same for both the Octree and VoxelGrid. The NavGrid is a bit slower than the VoxelGrid, because the time it takes to get a node's neighbours is a bit longer. The VoxelGrid has the lowest `GetNeighbour()` time in all tests. This also explains the faster pathfinding speed for the VoxelGrid in test 2. The `GetNeighbour()` function is significantly slower for the Octree compared to the other two, which is no surprise, because it has to calculate the neighbours with a complicated recursive method instead of simply retrieving them. Initializing the grid for

Table 2: Time spent on different sections for tests 1-4. Tests 1 and 2 have a viewing range (VR) of 20 units, tests 3a and 4a have a VR of 120. Except for the total tests time, total move time and initialize grid time, every other time is an average.

Test 1	Octree	VoxelGrid	NavGrid
Total test time	50.8 s	48.6 s	52.5 s
Total move time	26.3 s	24.1 s	22.7 s
Pathfinding	16.5 ms	22.7 ms	26.9 ms**
Exploration	21.4 ms	24.9 ms	
GetNeighbour	6.5 µs	1.1 µs	0.6 µs
Initializing grid	1.0 ms	44.9 ms	
Adding to grid	312.3 µs	141.0 µs	253.1 µs*
Expanding grid	1.0 ms	2.5 ms	

Test 2	Octree	VoxelGrid	NavGrid
Total test time	89.2 s	77.3 s	79.1 s
Total move time	47.8 s	45.0 s	44.1 s
Pathfinding	16.0 ms	11.2 ms	14.5 ms
GetNeighbour	18.9 µs	1.1 µs	1.6 µs
Initializing grid	1.0 ms	23.9 ms	
Adding to grid	234.7 µs	344.7 µs	270.0 µs*
Expanding grid	1.0 ms	3.2 ms	

Test 3a	Octree	VoxelGrid	NavGrid
Total test time	777.1 s	1384.4 s	1136.2 s
Total move time	70.2 s	63.2 s	63.7 s
Pathfinding	0.4 s	410.7 s	293.6 s
GetNeighbour	24.3 µs	5.7 µs	12.0 µs
Initializing grid	4.0 ms	1911.4 ms	
Adding to grid	417.7 µs	492.1 µs	2498.5 µs*
Expanding grid	1.0 ms	63.8 ms	

Test 3b	Octree	VoxelGrid
VR 10	2.0 ms	0.9 s
VR 20	4.0 ms	35.6 s
VR 30	15.0 ms	543.6 s
VR 40	8.0 ms	1889.8 s
VR 50	66.0 ms	9561.4 s
VR 60	18.9 ms	15919.2 s
VR 70	3974.2 ms	28395.2 s

Test 4a	Octree	VoxelGrid	NavGrid
Total test time	1313.3 s	2598.8 s	2394.8 s
Total move time	74.8 s	68.7 s	68.3 s
Pathfinding	1.3 s	691.2 s	564.3 s
GetNeighbour	34.6 µs	7.8 µs	19.8 µs
Initializing grid	2.0 ms	2044.1 ms	
Adding to grid	2.9 ms	4.4 ms	32.8 ms*
Expanding grid	1.3 ms	76.0 ms	

Test 4b	Octree	VoxelGrid
VR 10	13.0 ms	0.5 s
VR 20	81.8 ms	12.8 s
VR 30	698.1 ms	222.3 s
VR 40	2074.3 ms	1075.0 s
VR 50	2005.6 ms	3620.2 s
VR 60	10234.8 ms	14457.4 s
VR 70	56182.5 ms	30144.5 s

* includes expanding time.

** includes fake exploration, 19.9 ms without it.

Table 3: Number of agent turns per test.

Test world	1	2	3	4
Octree	283	461	661	868
VoxelGrid	291	367	455	728
NavGrid	342	390	460	747

the Octree always takes around the same amount of time, regardless of viewing range. The differences between tests is purely because of PC performance fluctuations. The VoxelGrid's initialize time is significantly higher than the Octree and increases even further with larger viewing ranges. The NavGrid doesn't need to initialize itself, neither does it need to expand. It stores all data in a single `HashSet`. This does however mean that whenever a node is added to a full `HashSet`, that it has to expand it there. So the expand time is incorporated in the adding to grid time. This makes the average Adding to grid time significantly higher for the NavGrid. Comparing the adding to grid time for the NavGrid with the combined time of Adding to grid and Expanding grid of the Octree and the VoxelGrid better shows the difference. Doing this, we can see that the Octree is still the fastest, taking 1.2–4.2 milliseconds. The NavGrid is the next best, with between 0.3–32.8 milliseconds. The VoxelGrid takes between 3.5–80.4 milliseconds. Adding a new obstacle to the grid takes longer for the Octree than for the VoxelGrid. Expanding the Octree, however, is much faster than the VoxelGrid, taking the same amount of time, regardless of test or viewing range, whereas the VoxelGrid's increases with the size of the world. Continually expanding will take longer and longer. For test 3b, we can see that even for a viewing range as low as 10, the VoxelGrid still takes about 500 times as long for a single explore call. For test 5b, for a viewing range of 10, the VoxelGrid takes about 38 times as long for a single explore call. This huge difference increases even more for larger viewing ranges, and decreases slightly with more added obstacles in the world. Where the VoxelGrid's time consistently increases with the viewing range, the Octree has a general increase with some fluctuations. Especially, the drop-down from 15 milliseconds for test 3b's viewing range 30 to 8 ms for viewing range 40 seems weird. However, this can be explained with the amount of steps it takes to find a path to a node neighbouring an unexplored node. In an Octree, similar nodes are combined into a single larger node. The path found for VR 40 simply was able to travel along a gigantic open node and quickly find an unexplored node. This giant open node could not be combined into one yet for VR 30, because it was partially unexplored, so it was still split into multiple open nodes. The giant node however neighbours a lot of smaller nodes of different labels. If even one of those many neighbours is unexplored, then we can already stop the exploration, making it faster than the one with a smaller viewing range. The same explanation can be given for VR 60.

6.4.3 *Testing.*

In test world 1, we prove that our testing framework can do reachability tests in a simple scene where a door is blocking the way, but a button can open the door. Test world 2, proves reachability tests can be done in more complicated test worlds with a puzzle on how to open the final door. In test worlds 3 and 4, we prove that our method can do testing in large 3D environments. This includes button-door logic. All our tests showcase reachability tests, however the program can do a lot more. We adapted iv4XR to work in 3D, but almost all testing functionality was left intact. This means the following things still can be tested, but now in 3D.

(1) Reachability.
(2) Removing or placing blocks.
(3) Using toolbar items (like the grinder to destroy blocks).
(4) Activating blocks (like a button panel).

To do this, you can simply add a corresponding GoalStructure after a `closeTo()` Goal. For example, in test 1, the test currently tests if the battery block is reachable by going there. At the end of the goal, the agent will be in front of and facing towards the battery block. So you can simply add a new goal to the SEQ using the `grinded()` GoalStructure. You can also add a check before the grinding goal to check if the block is intact with the `targetBlockOK()` GoalStructure. Everything iv4XR was able to do in SE before in 2D, it can now do in 3D. We also made it possible to do exploration in the game world to find objectives outside the viewing range. This makes it possible to test larger worlds, where observing everything at once with a very large observation radius is not recommended. Setting the observation radius in a large world with many blocks makes the program much slower and take more memory. This is because every time we update the state, we observe the world inside the viewing range and check for differences with the previous observation. This is stored in the *WOM*, and with a larger viewing range, the amount of memory used by the *WOM* is also larger. With more observed blocks, merging the previous and current *WOM* and checking the differences also takes longer. Not to mention the time it takes to simply observe all the blocks in SE inside the viewing range. Thus setting the observation radius smaller not only makes testing in large worlds more human-like, because the agent has to explore to find the objective(s), it also reduces the time spent on observing the world and merging the observations. For testing purposes, both the Octree and VoxelGrid can do all the above. The NavGrid can do all of it, except for exploration.

6.5 Conclusion

All in all, we can see from the results that as far as memory is concerned, the VoxelGrid is by far the worst way to go about storing the world. Both the NavGrid and Octree use significantly less memory, with the Octree taking slightly more memory than the NavGrid for all test worlds. The test worlds however only show realistic world scenarios for a 3D open world game. Worst case scenario for the Octree is a big world with alternating empty and open space. This would mean the Octree has almost no nodes to combine. This presumably would make the amount of memory the Octree uses similar to the VoxelGrid. It would still be able to combine some nodes though, because the padding makes it so that with a good node size (see Section 5.5) obstacles are always multiple nodes wide. Thus, even in such a scenario, we presume the Octree would still use a little less memory than the VoxelGrid. The worst case for the NavGrid is if the entire world is filled with obstacles. This would make it even worse than the VoxelGrid. So as long as the world has a lot more empty space than obstacles, then the NavGrid uses the least amount of memory space, with the Octree a close second. In all other scenarios, the Octree uses the least amount of memory.

As far as speed is concerned. The Octree is slower in returning the neighbours of a node than both the VoxelGrid and the NavGrid. For almost all test scenarios, the Octree is faster than both the VoxelGrid and NavGrid for pathfinding and exploration. However, for worlds with only 1 block wide spaces, where the Octree cannot combine open nodes, the Octree is actually slower than the VoxelGrid and NavGrid. Initializing the Grid for all three data structures takes an insignificant amount of time. But of them, the VoxelGrid is the slowest. Adding blocks to the grid is a little slower for the Octree than for the VoxelGrid or NavGrid. The reverse is true for expanding the grid, where the Octree takes a consistent amount of time regardless of grid size, but the VoxelGrid takes longer for larger worlds and shorter for small worlds. For very small worlds like test 1 and 2, expanding is thus better for the VoxelGrid, but for medium or large worlds, the Octree is significantly faster.

For testing purposes, both the Octree and VoxelGrid can complete the same amount of testing tasks, making them equally good if we disregard time and memory efficiency. The NavGrid, with how we implemented it, cannot do exploration, restricting its testing capabilities around exploring unknown worlds.

Overall, the Octree is a significantly better data structure both in terms of memory and speed than the VoxelGrid baseline, with the only exception being small worlds with little room for moving around. It is also significantly faster and has more testing capabilities compared to the NavGrid.

7 DISCUSSION AND CONCLUSIONS

7.1 Conclusion

In this thesis, we created an efficient automated testing framework for 3D games with omnidirectional movement. To do this, we extended the iv4XR project to work in 3D. The iv4XR project already had the automated testing capabilities necessary for 2D use cases, using a NavGrid to store a plane of the game world into memory. We created three different spatial partitioning data-structures (*grids*) to store the full 3D game world in memory, and a way to navigate this game world by planning in the stored *grid*. We created an Octree data-structure that hierarchically stores OPEN, BLOCKED or UNKNOWN nodes and can be extended and updated at runtime. We created two baseline methods to compare against. A simple 3D grid of Voxels (VoxelGrid) and a 3D version of the original NavGrid, adapted to work the same way as the Octree and VoxelGrid. To be able to do testing based on the data stored in the data-structures, all *grids* have a way to add or remove blocks from the game world to the *grid*. They also have a way to extend the bounds of the data-structure and manage which nodes are known or still unknown. To be able to do pathfinding, all *grids* have their own method of finding the corresponding nodes of in-game locations and a function to retrieve or calculate the neighbouring nodes. We adapted some goal-structures and tactics from iv4XR to be usable in 3D, and created some new goal-structures and tactics that can be used for automated testing tasks. We created 4 test worlds for the game Space Engineers to measure the testing capabilities, memory efficiency and speed of the Octree method and compared the results with the two baselines. From the results, we conclude that the Octree is significantly faster in almost all test scenarios compared to the baselines, VoxelGrid and NavGrid. memory wise, it is significantly more memory efficient than the VoxelGrid, but slightly less memory efficient than the NavGrid. For worlds with larger fully filled areas, the Octree becomes more memory efficient than the NavGrid. The 3D implementation is able to do any testing task the original 2D implementation of the iv4XR project could do, but now it can additionally do so in 3D and it can do exploration.

7.2 Discussion

From our results, we conclude that our Octree implementation is superior to the baseline VoxelGrid and NavGrid implementations. However, there remain some negatives in how we approached the problem and compared the different methods.

First, we only ran every test once, taking the average over parts of the code that ran multiple times. This made the results more prone to CPU fluctuations, making it harder to draw conclusions from small differences in speed values. This is especially a problem for the total test time and initializing *grid* time, which are only measured once per test.

Second, Space Engineers' movement in 3D is done by flying using a jetpack. Whenever the program instructs the agent to move forward, it gives it a few short bursts of speed in the corresponding direction. This is equivalent to shortly holding the forward key in-game. Holding forward, slowly increases the speed until it reaches its maximum velocity and when no velocity is added in a direction, the agent will automatically slow down until it stops. However, the precise calculations for the speed are calculated by SE and the program cannot access this data. This makes it hard to control the speed, which is a problem, when we want to efficiently move. If we add too much velocity, then the agent may overshoot its target, and it would have to move back to the target location. If we add too little velocity, then the agent takes a very long to reach its target location. This is made worse with continuously adding velocity. If we add too much, the agent will continue to gather speed. For a short distance, this may be fine, but then at a longer distance, it will have too much speed. If we add too little, it will lose its speed and not accumulate any. Then there is the problem how long a state update cycle takes. Update cycles take around the same amount of time, with the exceptions being whenever a new path has to be planned or the Goal has to be changed. There is however one exception: the observation. Given a viewing range, everything within that range to the agent in the environment is observed. This observation is done every cycle and takes a large amount of time for larger observations. With a larger viewing range, more blocks will be observed, taking more time. With more blocks in the world, more blocks will be observed, taking more time. And with more passed time for a cycle, because the automated testing framework and the game run asynchronously, the movement velocity is being lost. So, for larger observations, the agent will fly slower, more frequently undershooting target positions. For smaller observations, the agent will fly faster, more frequently overshooting target positions. In our tests, we saw both happen. In test 1 and 2, with a viewing range of 20, it would pick up speed very fast, overshooting quite often. In tests 3 and 4, with a viewing range of 120, it would take too long to re-add velocity, losing its speed before adding more, undershooting quite often. In test 4, where a lot more blocks are, this would happen even more often than in test 3. When update cycles are short, velocity is added a fast intervals, picking up speed. Travelling in a straight line means the agent will gain speed for longer, increasing the chance to overshoot. For the VoxelGrid and NavGrid, all nodes are aligned in a simple grid structure, so often a lot of points on the path can be removed as they are lie on the line between two other nodes in the path. This creates many long straight line, resulting in higher chances of overshooting. Octree nodes can have different sizes and because the point on the path are the centers of nodes, where certain nodes were removed from the path for the VoxelGrid or NavGrid, for the Octree they sometimes are not, because their center does not lie on the line between them. This makes the Octree paths often consist of more midway points, resulting in less long straight lines and more shorter ones, reducing the chances of overshooting. Whenever we saw undershooting happen (so very slow movement), we manually added a little extra velocity (in the same direction) to speed up the test. We tried to do this as consistently as possible for all tests, but it does influence the time results. Thankfully, the only time it influences is the total test time. As the paths in a test are almost the same for all methods, the movement time should also be almost the same. So by looking at the difference in total movement time between the Octree, VoxelGrid and NavGrid, we can estimate the difference the manually added speed made on the total test time.

We decided to limit rotation to be only around the agent y-axis. Everything works regardless of agent orientation, but we decided to only do yaw rotations. This decision is because of two reasons. First, the plugin used to send commands to SE does have a parameter for all rotation axis's, however, the roll parameter did not work. Second, although changing it to rotate around all axis's would make the movement feel more human, this is not as relevant to our thesis, so we decided against it.

Finally, the test worlds we created show the differences between techniques for small worlds, with test 1 being a bit more open than the very tightly filled test 2. Test 3 was extremely open, which for the game Space Engineers is a quite realistic scenario. Test 4 was supposed to show how the Octree takes more memory and time if the same world was more filled with obstacles. It is, however, still very open, which makes it skewed towards better results for the Octree and NavGrid than the VoxelGrid. It would have been better to add another test world, with it being so full, the Octree can hardly combine any nodes. This would allow us to see the statistics of a world which is worst case for the Octree. In such a test world, the VoxelGrid method may very well be the best choice. Even if such a test world would be the outlier of test worlds you actually want to use for automated testing. Additionally, we used the Euclidean distance for A*, because it would reduce the amount of node to go through for pathfinding. For exploration, this does not reduce the amount of node to traverse. This causes the pathfinding to do an unnecessary more expensive calculation for every node to traverse, causing all results of test 3b and 4b to be significantly slower than could be. Changing this distance calculation to Manhattan distance would make the explore method for all *grids* much faster, but also decrease the difference between the Octree and VoxelGrid times. This is because the VoxelGrid has to traverse many more nodes than the Octree, hence the large difference between times. If the calculation is faster for all nodes, then the VoxelGrid will benefit more from this speedup. This would make the slower `getNeighbour()` speed play a bigger role in the comparison. Still, with how large the difference is, this would not change the result that the Octree is significantly faster at exploration.

7.3 Future Work

There is still some room for improvements on the current work as well as possible future works. In this section, we will go over a few of them.

First of all, the NavGrid only stores blocked nodes, so it cannot differentiate between OPEN and UNKNOWN nodes. This means exploration cannot be implemented the same way it was done as for the Octree and VoxelGrid. Keeping track of frontier nodes makes it possible to do exploration with a NavGrid, but would also make it less memory efficient and a little slower. Implementing this and comparing it to the Octree and VoxelGrid methods is future work.

Next, the Octree currently calculates the neighbours by recursively checking the parent for the neighbour, essentially going up before going back down. An alternative way to get the neighbours is taking a position just a small distance from the edge of the current node in some direction, and then checking in what leaf-node that position is. This starts from the root-node and recursively checks in which child it is until it finds the neighbour. It remains to be seen if this is faster or slower.

Every Octree node stores its position, whilst the VoxelGrid nodes do not do this. They only store a label and the position can be calculated because the node size and the corners of the grid are known. Doing the same for Octree nodes would reduce the memory needed, but could make it a little slower. The VoxelGrid can also be improved memory wise, because it currently uses an `arrayList`, which uses 16 bytes minimum per node, whilst only 1 byte is used for the label. Finding out a better way to store the labels without significantly complicating the expanding process can still be researched. These optimizations can change the pros and cons of the methods. Just like Sparse Voxel Octrees (SVO) [3], implementing our Octree implementation with a similar sparse layout could potentially improve the method.

For the testing parts of the project. A lot of extra logic, goal-structures and tactics can be added and researched. For example, figuring out a good way to go about testing test world 2, without explicitly stating the order the buttons need to be pressed, but making it possible for the agent to figure out the puzzle itself. Also, we only tested the method on the game Space Engineers. The method is created for SE and tested on it. So, testing its effectiveness on other similar omnidirectional 3D games is still future work.

Space Engineers has gravity shoes, allowing you to walk on any surface at any orientation. An interesting thing to research would be how to make navigation in 3D space more human, by setting it so that the agent, when close to a surface, prefers to rotate and land on the surface to walk short distances. Figuring out how to implement the switch between landing an taking off would encompass detecting close surfaces, rotating so the agent so the up orientation is on the surface normal, flying down (agent orientation) until the feet touch the surface and then turning off the jetpack. Then a decision has to be made on when the agent should decide to land. Being grounded, means having more precise control over the movement, but also losing free movement up and down or gaining very fast speed for longer distances. If there is gravity, then landing is much easier and the agent can jump once the jetpack is off. All in all, this would make for some interesting research for more human-like space engineer or astronaut behaviour.

REFERENCES

[1] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software* 86, 8 (2013), 1978–2001.

[2] aplib: Action.java 2021. https://github.com/iv4xr-project/aplib/blob/master/src/main/java/nl/uu/cs/aplib/mainConcepts/Action.java

[3] Daniel Brewer. 2019. 3d flight navigation using sparse voxel octrees. In *Game AI Pro 360: Guide to Movement and Pathfinding*. CRC Press, 273–282.

[4] Douglas Cline. 2017. 19.5: Appendix - Coordinate transformations. https://phys.libretexts.org/Bookshelves/Classical_Mechanics/Variational_Principles_in_Classical_Mechanics_(Cline)/19%3A_Mathematical_Methods_for_Classical_Mechanics/19.05%3A_Appendix_-_Coordinate_transformations

[5] Jian S. Dai. 2015. Euler–Rodrigues formula variations, quaternion conjugation and intrinsic connections. *Mechanism and Machine Theory* 92 (2015), 144–152. https://doi.org/10.1016/j.mechmachtheory.2015.03.004

[6] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. 2007. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. 31–36.

[7] Edsger W Dijkstra. 2022. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*. 287–290.

[8] Christian Dornhege and Alexander Kleiner. 2013. A frontier-void-based approach for autonomous exploration in 3d. *Advanced Robotics* 27, 6 (2013), 459–468.

[9] Nathaniel Fairfield, George Kantor, and David Wettergreen. 2007. Real-Time SLAM with Octree Evidence Grids for Exploration in Underwater Tunnels. *Journal of Field Robotics* 24, 1-2 (2007), 03–21. https://doi.org/10.1002/rob.20165 arXiv:https://onlinelibrary-wiley-com.proxy.library.uu.nl/doi/pdf/10.1002/rob.20165

[10] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[11] Irene Gargantini. 1982. Linear octtrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing* 20, 4 (Dec. 1982), 365–374. https://doi.org/10.1016/0146-664X(82)90058-2

[12] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[13] Andreas Herzig, Emiliano Lorini, Laurent Perrussel, and Zhanhao Xiao. 2017. BDI logics for BDI architectures: old problems, new perspectives. *KI-Künstliche Intelligenz* 31 (2017), 73–83.

[14] Robert C Holte, Maria B Perez, Robert M Zimmer, and Alan J MacDonald. 1996. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI, Vol. 1*. 530–535.

[15] Keen Software House. 2019. Space Engineers.

[16] John Funge Ian Millington. 2019. *Artificial intelligence for games* (3 ed.). CRC Press.

[17] iv4xr-se-plugin: UUTacticLib.java 2021. https://github.com/iv4xr-project/iv4xr-se-plugin/blob/uubranch3D/JvmClient/src/jvmMain/java/uuspaceagent/UUTacticLib.java

[18] Chris L Jackins and Steven L Tanimoto. 1980. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing* 14, 3 (1980), 249–270.

[19] L. Kavraki and J.-C. Latombe. 1994. Randomized preprocessing of configuration space for path planning: articulated robots. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'94)*, Vol. 3. 1764–1771 vol.3. https://doi.org/10.1109/IROS.1994.407619

[20] L.E. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation* 12, 4 (Aug. 1996), 566–580. https://doi.org/10.1109/70.508439

[21] Chinmay S. Kulkarni and Pierre F.J. Lermusiaux. 2020. Three-dimensional time-optimal path planning in the ocean. *Ocean Modelling* 152 (2020), 101644. https://doi.org/10.1016/j.ocemod.2020.101644

[22] Lab Recruits Game Testing Contest 2021 2021. https://github.com/iv4xr-project/JLabGym/blob/master/docs/contest/contest2021.md

[23] Marino Mangeruga, Alessandro Casavola, Francesco Pupo, and Fabio Bruno. 2020. An Underwater Pathfinding Algorithm for Optimised Planning of Survey Dives. *Remote Sensing* 12, 23 (2020). https://doi.org/10.3390/rs12233974

[24] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[25] Donald Meagher. 1982. Geometric modeling using octree encoding. *Computer Graphics and Image Processing* 19, 2 (1982), 129–147. https://doi.org/10.1016/0146-664X(82)90104-6

[26] Guy M Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. (1966).

[27] Dennis Nieuwenhuisen, Arno Kamphuis, Marlies Mooijekind, and Mark H Overmars. 2004. Automatic construction of roadmaps for path planning in games. In *International Conference on Computer Games: Artificial Intelligence, Design and Education*. Citeseer, 285–292.

[28] Mark H Overmars. 2005. Path planning for games. In *Proc. 3rd Int. Game Design and Technology Workshop*. 29–33.

[29] Johannes Pfau, Jan David Smeddinck, and Rainer Malaka. 2017. Automated game testing with icarus: Intelligent completion of adventure riddles via unsupervised solving. In *Extended abstracts publication of the annual symposium on computer-human interaction in play*. 153–164.

[30] Cristiano Politowski, Yann-Gaël Guéhéneuc, and Fabio Petrillo. 2022. Towards automated video game testing: still a long way to go. In *Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation (GAS '22)*. Association for Computing Machinery, New York, NY, USA, 37–43. https://doi.org/10.1145/3524494.3527627

[31] Cristiano Politowski, Fabio Petrillo, and Yann-Gaël Guéhéneuc. 2021. A Survey of Video Game Testing. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. 90–99. https://doi.org/10.1109/AST52587.2021.00018

[32] Rui Prada, I. S. W. B. Prasetya, Fitsum Kifetew, Frank Dignum, Tanja E. J. Vos, Jason Lander, Jean-yves Donnart, Alexandre Kazmierowski, Joseph Davidson, and Pedro M. Fernandes. 2020. Agent-based Testing of Extended Reality Systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 414–417. https://doi.org/10.1109/ICST46399.2020.00051

[33] ISWB Prasetya, Mehdi Dastani, Rui Prada, Tanja EJ Vos, Frank Dignum, and Fitsum Kifetew. 2020. Aplib: Tactical agents for testing computer games. In *International Workshop on Engineering Multi-Agent Systems*. Springer, 21–41.

[34] I. S. W. B. Prasetya, Maurin Voshol, Tom Tanis, Adam Smits, Bram Smit, Jacco van Mourik, Menno Klunder, Frank Hoogmoed, Stijn Hinlopen, August van Casteren, Jesse van de Berg, Naraenda G.W.Y. Prasetya, Samira Shirzadehhajimahmood, and Saba Gholizadeh Ansari. 2020. Navigation and exploration in 3D-game automated play testing. In *Proceedings of the 11th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Virtual, USA) *(A-TEST 2020)*. Association for Computing Machinery, New York, NY, USA, 3–9. https://doi.org/10.1145/3412452.3423570

[35] Hanan Samet. 1988. An Overview of Quadtrees, Octrees, and Related Hierarchical Data Structures. In *Theoretical Foundations of Computer Graphics and CAD (NATO ASI Series)*, Rae A. Earnshaw (Ed.). Springer, Berlin, Heidelberg, 51–68. https://doi.org/10.1007/978-3-642-83539-1_2

[36] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 419–423.

[37] Shaojie Shen, Nathan Michael, and Vijay Kumar. 2012. Autonomous indoor 3D exploration with a micro-aerial vehicle. In *2012 IEEE International Conference on Robotics and Automation*. 9–15. https://doi.org/10.1109/ICRA.2012.6225146

[38] A. Stentz. 1994. Optimal and efficient path planning for partially-known environments. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*. 3310–3317 vol.4. https://doi.org/10.1109/ROBOT.1994.351061

[39] J Tuovenen, Mourad Oussalah, and Panos Kostakos. 2019. MAuto: Automatic mobile game testing tool using image-matching based approach. *The Computer Games Journal* 8 (2019), 215–239.

[40] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software testing, verification and reliability* 22, 5 (2012), 297–312.

[41] Tanja Vos, Paolo Tonella, Wishnu Prasetya, Peter M Kruse, Alessandra Bagnato, Mark Harman, and Onn Shehory. 2014. FITTEST: A new continuous and automated testing process for future internet applications. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 407–410.

[42] Kai M Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. 2010. OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems. In *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, Vol. 2. 3.

[43] Zhenjie Zheng, Mi Pan, and Wei Pan. 2020. Virtual Prototyping-Based Path Planning of Unmanned Aerial Vehicles for Building Exterior Inspection. Kitakyushu, Japan. https://doi.org/10.22260/ISARC2020/0004