Imperial College London

Software Engineering Group Project

Imperial College London

Department of Computing

PathBench3D

Authors (Group #3):
Judicaël Clair (jsc18)
Radina Milenkova (rm4318)
Abel Shields (as17718)
John Yao (jy8518)
Zeena Patel (zbp18)
Danqing Hu (dh4618)

Supervisors:
Prof Paul Kelly
Dr Sajad Saeedi

January 12, 2021

1 Executive Summary

Path planning is a multi-objective optimisation problem, which aims to find the most optimal path between two points, a source and a destination. Typically, the objective is to minimise travel-time by shortening path length, whilst ensuring navigation safety and a smooth trajectory. Path planning algorithms, or path planners, have vital applications in autonomous systems, such as robots and self-driving cars, where human intervention is undesirable [4]. Other applications include search and rescue, large-scale manufacturing, and warehouse management to mention a few. Path planning is a subject undergoing intense study, particularly in the domain of mobile robot navigation [3].

There exists a wide range of path planning algorithms applicable to robot navigation, however, very few attempts have been made to coherently benchmark different algorithms or unify their interface. Therefore, there has been an urgent demand to facilitate the development and benchmarking of classic and machine learning-based planning algorithms, especially with the recent advances in deep learning. To address this challenge, the group has created a unified framework in the hopes of accelerating research in this field.

Throughout this software engineering group project, the team has extended PathBench [1], an open-source project that primarily focuses on benchmarking robot motion planning algorithms. PathBench enables rapid prototyping and evaluation against a suite of path planning algorithms, featuring both classic algorithms and machine-learned models. Moreover, the framework consists of tools for algorithm visualisation and benchmarking according to several customisable metrics, as well as facilities for training machine learning (ML) models. The unification of these functionalities, in addition to a user-friendly and extendable interface, facilitates future development, training, testing, and evaluation of novel motion planning algorithms.

Originally, PathBench enabled development and evaluation of two dimensional (2D) path planning algorithms. The main objective of this project has been to adapt the functionality to three dimensions (3D). Additionally, the new framework, denoted as PathBench3D, nicely integrates with standard software libraries and tools used by robotics researchers. For instance, PathBench3D has a plugin for the Robot Operating System (ROS) framework [7], enabling real-time visualisation of a path planner that is coordinating a physical robot. Overall, PathBench3D allows users to easily compare and thoroughly assess the performance between classic and novel machine learning-based path planners in a multi-dimensional space.

What makes PathBench3D one of a kind is that it is a publicly available, powerful visualisation and planning tool for 2D and 3D with a lot of room for expansion. No other framework exists that accommodates both machine learning and classic algorithms, which is one of the biggest advantage of PathBench3D. The integration of a plethora of additional features ensures the end product is not only highly portable, but also fast and intuitive to use for robotics research, effectively solving the initial problem. PathBench3D is fully developed, with a much more robust and scalable codebase than its predecessor. To further set Pathbench3D apart from the rest of the open source frameworks available, several stretch goals have been achieved, namely the ROS real-time extension.

2 Introduction

2.1 Motivation

As discussed in the Executive Summary, path planning is a very active area of research due to its wide real-world applicability. However, until now, the development life cycle for research-grade algorithms has been rather cumbersome. Researchers in this field have to establish their own custom benchmarking and visualisation framework, including implementing a suite of algorithms for performance comparisons. Even if an open-source algorithm implementation exists (there are many for common algorithms such as A*), time needs to be spent modifying it to make it compatible with the custom framework.

Assessing and comparing algorithms is fundamental to evaluating newly created algorithms in robot motion planning. As a myriad of increasingly unique and complex algorithms are being introduced, there is a pressing need for a standard system of development and evaluation. To address this, the PathBench framework was introduced. PathBench can be used to develop, assess, compare, and visualise the performance and behaviour of 2D path planning algorithms. Although this framework presents a significant advantage for developing and evaluating a wide range of algorithms, it is limited by its inability to simulate a 3D environment. While there are many situations where a problem can be reduced to two dimensions, such as a wheeled robot navigating a room, a vast majority of problems in robotics require three-dimensional path planners. Therefore, to assess the practicality of these planners, a purely 2D framework will not suffice.

2.2 Objective & Main Achievements

The primary aim of this project has been to develop a new framework, PathBench3D, which extends the capabilities of the original software by providing support for assessing algorithms in a three-dimensional space, whilst maintaining backwards compatibility with the existing two-dimensional functionalities. Moreover, the group exceeded the initial expectations in terms of the supported features, with the main achievements detailed below.

When developing a new algorithm, it is desirable to evaluate its performance against a variety of algorithms. For this reason, PathBench3D includes a large collection of algorithms that are typically used in robotics. To this end, all of the classic path planning algorithms that were present in the original framework have been adapted to work with both 2D and 3D maps. These include A*, Dijkstra, Potential Field, Wave-front, Bug1, Bug2, RT, RRT, RRT*, RRT-Connect, SPRM, in addition to numerous state-of-the-art algorithms imported from the Open Motion Planning Library (OMPL) [2]. Note, the OMPL algorithms currently only work with 2D maps. Furthermore, existing machine learning-based algorithms, such as the Waypoint Planning Networks (WPN) algorithm, have also been ported to the new framework. This involved reworking the model trainer and map generator for use in training. Moreover, PathBench3D has significantly more metrics than its predecessor. As a result, roboticists are now able to rapidly and comprehensively compare their novel algorithms against the state-of-the-art.

To further assist users in understanding the behaviour of path planners, PathBench3D provides a user-friendly 3D visualiser capable of displaying a simulation running in a two or three-dimensional space. Through the intuitive Graphical User Interface (GUI), the user is able to effortlessly customise the style of the visualisation. Additionally, PathBench3D has several built-in utilities to visualise the internal state of algorithms. If these are insufficient, the user can instead implement their own custom rendering routines. This high degree of customisability and extensibility allows researchers to understand the behaviour of algorithms of any kind and effectively debug them.

In a typical PathBench3D simulation, the map is entirely visible to the algorithm. However, in the real world, the map may be partially observable with the goal outside of view. Consequently, algorithms that attempt to find the goal without exploration will fail. This is one of many reasons why it is essential to test algorithms on a physical robot or in a realistic simulation. To this end, PathBench3D has a plugin for ROS, which is a framework used by a majority of researchers for robot software development. This extension provides real-time support for visualising the path planner as it controls a robot. As a result, researchers using PathBench3D to develop path planners can rapidly gain insights into the practicality of their algorithms.

Along with the implementation of new features, it was vital to ensure PathBench3D is being extensively tested. Therefore, all key components of the logic and graphics side of our framework are covered by unit and system level tests to provide a stable, immediately deployable software, and prevent unexpected issues in the released product. A key achievement, which was challenging in its implementation, is carrying out automated tests of the visualiser to guarantee a functional user interface and correct rendering of entities, such as the map.

The end product is a highly robust framework for prototyping, benchmarking, and visualising both classic and machine learning-based robot path planners in a two or three-dimensional space. The unification of several components into one platform is beneficial in terms of providing users with the ability to complete a variety of advanced tasks at a very high speed. The platform is now ready for the addition of path planners implemented by Dr Saeedi's Ph.D. students, bringing them closer to their goal of publishing research papers on their novel path planning algorithms.

3 Design and Implementation

3.1 Introduction

PathBench3D has been built upon the existing open-source PathBench codebase, with the code being extensively reworked to include support for 3D. In terms of system design, we tried to preserve the original PathBench code structure as it was already quite intuitive and well-structured. However, there were several areas that were not suitable for 3D. In these cases, we redesigned and integrated alternative methods of implementation. Although many other system design alternatives exist, we believed that preserving most of the original framework's code structure would benefit future work on the Pathbench3D software. The initial PathBench framework provided a solid design foundation, and we concluded that there was no need for us to "reinvent the wheel" in terms of overall system design. If we had tried to do so, we suspect that we would not have had time to achieve all the technical tasks we set out to complete.

3.2 Structure Overview

As illustrated, our framework's structure can be split into four main sections: Machine Learning, Environment, ROS, and Evaluation. The Machine Learning section is where we generate the training dataset and train our models for path planning. The Environment section deals with the map, algorithm and simulation, and how this is visualised graphically. The ROS extension allows real-time interaction with a physical robot. Finally, the Evaluation section provides support for analysis of algorithms through benchmarking methods. Therefore, the main high-level components of our platform are the `Simulator`, `Generator`, `Trainer`, and `Analyzer`, which are linked together by general service libraries and utilities (see Fig. 11).

To launch one or more components (`Simulator`, `Generator`, `Trainer`, `Analyzer`) a `Configuration` object needs to be instantiated, which contains a plethora of settings for each component that users can define appropriately. These settings are then given as input to initialise a `Services` object, which is a collection of services available throughout the entire application. The `Services` object is subsequently used to initialise the desired components, such as the `Simulator`. This entire process is abstracted away by the `MainRunner` together with a command-line interface, which caters to the typical usage of the platform.

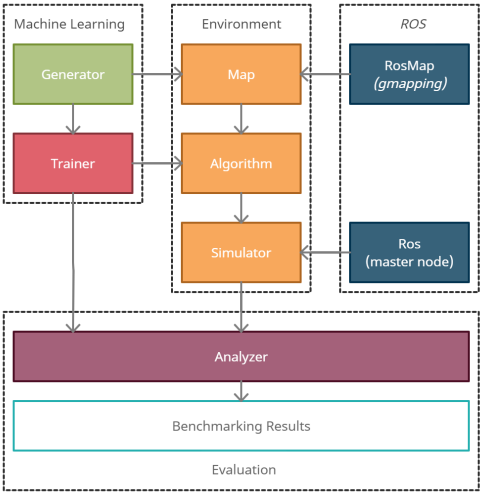


Figure 1: PathBench3D structure high overview [9].

3.3 Environment

3.3.1 Points and Entities

The new design began with the `Point` class, before performing similar modification to the implementation of the entities (obstacle, agent, goal). The fields of the class such as `x: int` and `y: int` were changed, and the `Point`'s inheritance of `NamedTuple` was removed, enabling `Point` to work for an arbitrary number of dimensions. As a result, `Point` can be used for both 2D and 3D removing the need for dedicated code paths, which would require more maintenance and testing. Another benefit of this change was the replacement of the non-intuitive map access in `DenseMap` grids. As the implementation used a list-of-lists, the grid was accessed as `map[y][x]`, and was transposed in a couple of places. With the current implementation, the more intuitive `map.at(Point(x, y, z))` can be used, or if accessing a grid directly (such as in an algorithm), `grid[tuple(point)]`. More details on this implementation can be found in section 3.3.2.

These changes created a couple of difficult bugs to solve, namely an unfortunate `'Point' object has no attribute '_pos'` bug. This bug was triggered when old maps were loaded into the simulator that had been updated with this new three dimensional system. The reason for this was that the PathBench architecture stores its maps with the `.pickle` binary format. Consequently, all the maps generated in the original PathBench implementation were binary-incompatible with the new implementation of `Point`, resulting in undefined behaviour when loading the maps.

In order to solve this, a `smart_load` method was implemented into the `Directory`'s load function, which manually cast all the classes that existed in the original PathBench (namely, `Point`, `Size`, and `DenseMap`) into the new versions that we had created, along with some extensibility in case other classes become incompatible with this or future versions. However, it was later decided to move to the JSON file format as it results in smaller file sizes, in addition to it being easier to maintain backwards/forwards compatibility thanks to its content being human-readable and editable.

3.3.2 Maps

The map component of PathBench3D is the environment in which algorithm simulation is performed. This is partially implemented by the abstract `Map` class. The main purpose of a `Map` instance is to provide information about the simulated world structure, as well as manage the entity (agent, goal, obstacles) interactions within it (see Fig. 2). That is, through the generic `Map` interface, we can determine the properties of a position in the simulated world, such as the cost of moving the agent to it, or whether there is an obstacle at that location. From this information, we are then able to appropriately plan a path.

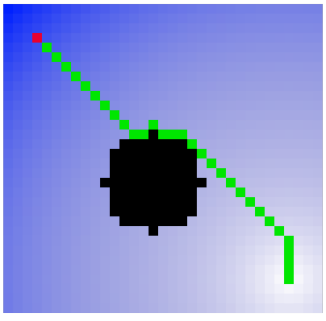


Figure 2: Wave Front algorithm ran on a 2D Small Obstacle map

There are two main types of maps, `DenseMap` and `SparseMap`. The former discretises the map into a grid structure, with the properties of every grid cell being stored in a multi-dimensional static `numpy` array. This design allows us to use the world position as an index into the array, which makes retrieving information about a cell, a constant time operation $\mathcal{O}(1)$; albeit by sacrificing a moderate amount of memory. As mentioned in 3.3.1, in our change of the coordinate system we re-reversed the coordinate positions. To ensure consistency, changes were made across the codebase to ensure that all logical manipulations were carried out in one consistent ordering, and that representations would transpose the coordinates if necessary. Note, the `Map` interface requires that the `DenseMap` also stores a list of all the obstacles in the map. This structure significantly improves performance as we often iterate over all the obstacles with no order requirement. `SparseMap`, unlike `DenseMap`, exclusively stores the aforementioned list of obstacles. Consequently, when determining whether a cell is an obstacle, the entire obstacle list must be iterated over, which has time complexity $\mathcal{O}(n)$, where n is the length of the obstacle list.

Evidently, grid maps with more than a handful of obstacles should use `DenseMap`. Indeed, the class `OccupancyGridMap` (see 3.3.6), which implements Occupancy Grid Maps (OGMs), derives from `DenseMap`. However, `SparseMap` is useful for non-discretised maps, such as point clouds. Note, a point cloud is a set of points where each point is located anywhere in an unbounded 3D space. In the context of robotics, a point represents a point of contact with an obstacle. Point clouds are frequently used in robotics as they are an efficient method of storing detailed information of the world [12], in addition to being a common output format for visual and tactile sensors. There are several algorithms (e.g. MPNet [6]) that exclusively work with point clouds, which is why we have provided elementary support for it (see Fig. 3 below, and Fig. 23 in Appendix).

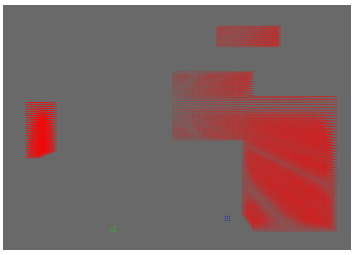


Figure 3: A point cloud visualisation of a block map

3.3.3 Algorithms

All builtin algorithms (and any user-made ones) inherit from the abstract base class `Algorithm`. Implementing a custom algorithm is straightforward: there is one method to override, `_find_path_internal`. The algorithm can move the agent around by executing `self.move_agent(..)`, and PathBench3D automatically detects when the algorithm has found the goal. By default, the visualiser will show the path of the agent that is generated from `self.move_agent(..)` calls. However, the algorithm must manually trigger a graphics update by invoking `self.key_frame()`. This cannot be done automatically for thread safety (see `AlgorithmRunner` in 3.3.4).

To visualise how the algorithm is working, the algorithm can specify a list of `MapDisplay` instances by overriding `set_display_info()`, with each instance rendering an internal part of the algorithm to be shown on the map. For example, `AStar` uses a `SolidIterableMapDisplay` to let the user know where the algorithm has already visited, and a `GradientListMapDisplay` to visualise the priority queue of grid cells that are to be visited next. These are shown in grey and blue respectively in Fig. 4. PathBench3D offers a wide variety of `MapDisplays`, however, if these are insufficient, the user can instead easily implement their own custom `MapDisplay` (see section 3.3.5.1). This design helps with the creation of non-intrusive, generic views, while still allowing for heavy customisation. Furthermore, decoupling the rendering code from the algorithm implementation effectively eliminates any overhead when running without graphics.

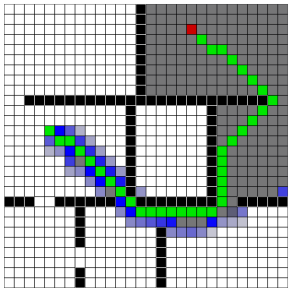


Figure 4: A* algorithm running on a 2D house map

It is essential that the user is able to visualise the running algorithm with minimal overhead. For this reason, a tremendous amount of effort has been devoted to optimising graphics. From an early stage, it was evident that modifications to the internal state of an algorithm must be tracked. This enables the visualiser to immediately determine the parts of the rendered scene that need to be modified. Without this optimisation, at every graphics update, the visualiser would need to compare the algorithm's internal state against a copy of its state from the previous update, and subsequently make a new copy. Alternatively, the visualiser would simply render the scene from scratch every time. Both of these methods are extremely inefficient and are therefore avoided at all costs. To this end, several tracked structures are available, with most of them implementing the appropriate standard interface. For instance, `TrackedList` and `TrackedSet` implement Python's `List` and `Set` interfaces respectively. As a result, no modification to an algorithm implementation is typically required to use a tracked container.

3.3.4 Services

Core functionality of PathBench3D's environment is accessible through a `Services` object. Notable components are the `EventManager`, `AlgorithmRunner`, and `PersistentState`.

The `EventManager` component is a communication service used by the simulator. An event, such as requesting to exit the application, is enqueued and subsequently dispatched when it reaches the head of the queue. During dispatch, all listeners attached to the `EventManager` are notified and appropriately handle the event. Therefore, the event system effectively decouples components, improving both the scalability and maintainability of the framework. Additionally, it enables the use of the Model-View-Controller (MVC) pattern for the simulator implementation, as the Model (logic) is otherwise unable to update the View (graphics).

The `AlgorithmRunner` manages the algorithm session, which contains the `Algorithm`, `BasicTesting` and `Map`. The `AlgorithmRunner` launches a separate thread to run the path planner. This ensures that the visualiser, which is running on the main thread, remains fully responsive during algorithm execution. For instance, the user is still able to interact with the visualised world, such as moving the camera for a better view of the running algorithm. A key technical challenge was synchronising access to the `Map`, as well as any data exposed to the visualiser via `MapDisplay` in the `Algorithm`'s implementation. To this end, a condition variable was used to control access. That is, calls to `Algorithm.key_frame()` in the `Algorithm` implementation will notify the visualiser thread via the condition variable to update the graphics. The algorithm thread is then blocked until the visualiser has completed the update. Note, the visualiser thread is still able to run at the same time as the algorithm thread because most user interactions do not require accessing shared data.

`PersistentState` is a generic interface to a JSON configuration file, which stores data that should survive beyond the lifetime of the application so it can be used in future runs. For instance, the `PersistentStateViews` class, which derives from `PersistentStateObject`, is used to save the visualiser state, such as the colour of entities (e.g. obstacles). `PersistentState` is extensible in that the user can store any serialisable data by implementing the `PersistentStateObject` interface.

Several other services exist, such as the `Resources` service, which provides safe interaction with the file system, as well as the `Debug` service that augments printing messages to console.

3.3.5 Simulator

The simulator is responsible for managing the path planning algorithm and the map it interacts with. Moreover, it provides customisable visualisation of the internal state of the algorithm for both 2D and 3D environments, allowing researchers to understand an algorithm's behaviour and effectively debug it. Additionally, the simulator can be run headlessly, which is useful for automated analysis of algorithms as they are run several times across various maps.

We have tried to maintain a Model-View-Controller (MVC) pattern for the simulator implementation. The model represents the logic code, and the view renders the model. The controllers are used to handle input from the keyboard and the mouse, by either issuing calls to the appropriate functions from the associated model, or dispatching events to notify the view. For instance, the `MapController` manages functionality such as mouse picking in the map to change the agent and goal positions. The MVC pattern effectively decouples logic from visualisation, which makes running headlessly a trivial task. Moreover, although the visualiser is currently the only view and controller, our existing design allows for new ones to be added, such as visualising the simulation remotely (from a different computer).

3.3.5.1 Visualiser

The original framework used PyGame and Tkinter for graphics, however, they only support 2D rendering. Consequently, a different graphics framework had to be used for PathBench3D in order to render 3D simulations. Furthermore, a key objective for PathBench3D was to develop a unified, user-friendly visualiser. Therefore, for convenience, the initialisation and configuration of a simulation, as well customising its view, are all done within the same window, as opposed to the previous version of the software. This implementation also ensures seamless transition between simulations of different space dimensionality. Not only does this streamline design make PathBench3D easy-to-use, but it also significantly improves both the maintainability and scalability of the codebase as this requires the implementation to only use a single graphics framework (unlike PathBench). To this end, the visualiser was written from scratch.

Instead of using low-level application programming interfaces (APIs) for rendering graphics, such as OpenGL or Vulkan, PathBench3D makes use of a graphics engine, as a higher level of abstraction is more suitable for rapidly developing a product of high quality. Even though the aforementioned APIs are capable of very high performance, customised for our use, hardware-accelerated rendering, the team concluded it would be unnecessarily time consuming for our purposes, with too many risks to consider.

When choosing the graphics engine, several metrics were considered essential for the success of the project. These include the degree of customisability, runtime performance, the ability to create a GUI that is suitable for automated testing, etc. There was a number of frameworks available for us to consider, such as QT, Soya3D, and python-ogre. After the initial evaluation of each, it was determined that the most appropriate graphics framework for this project was Panda3D. One of the reasons this framework was chosen is because it provides an easy-to-use API with plenty of documentation and user support available. Therefore, even though the group had no prior experience with this framework, the team was confident that the framework could be used effectively within a short period of time. Furthermore, Panda3D was perfectly suitable for this project as it combines the speed of C++ with the ease of use of Python, to allow for a fast rate of development without sacrificing on performance. Another key consideration when choosing this graphics engine was that if Panda3D lacked necessary functionality (e.g entities that are not straightforward to render), the user can implement their own custom programmable shaders (scripts). Therefore, Panda3D provides the benefit of doing easier tasks quickly, as well as harder ones efficiently. Last but not least, the group determined that with PyAutoGUI — a popular tool to automate interactions with a GUI — it was feasible to perform automated testing of a graphics application that uses Panda3D.

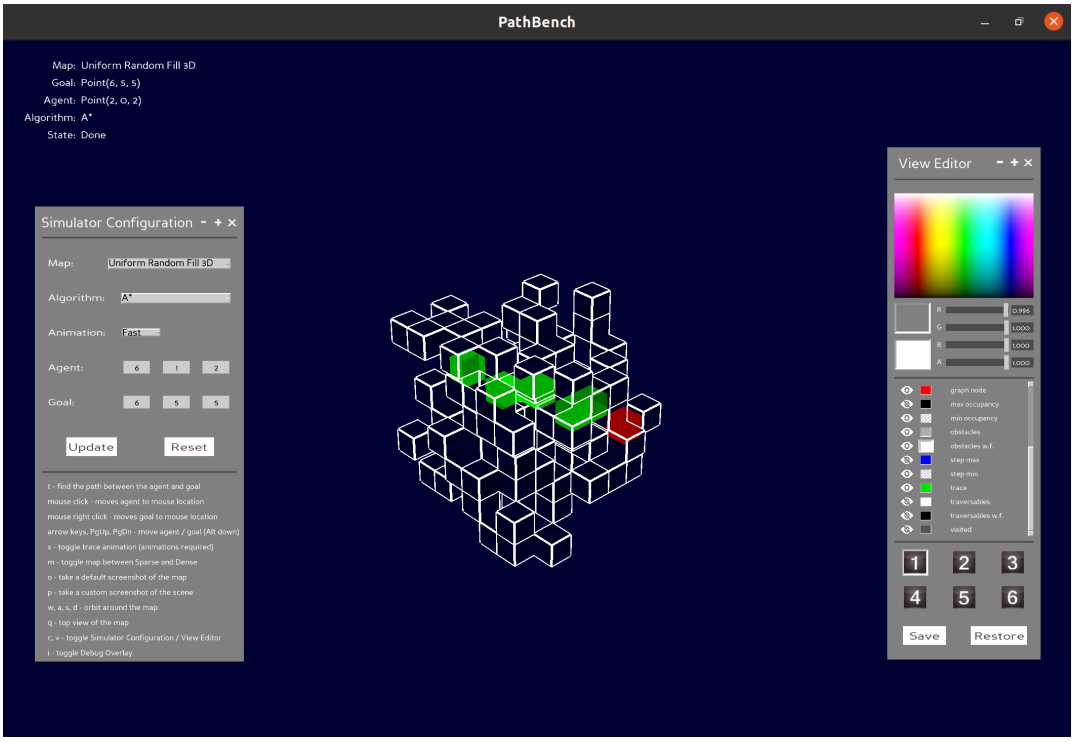


Figure 5: Screenshot of A* being run on an 8x8x8 Uniform Random Fill map. Colours and transparencies have been customised to make it easier to see the path (green) located behind the obstacles (black).

Thanks to several rounds of user testing and subsequent refinement, PathBench3D's GUI is very intuitive and easy to navigate. It consists of several components, notably the Simulator Configuration and View Editor windows.

The Simulator Configuration window allows the user to interactively configure the simulation. That is, a map and algorithm can be selected from appropriate drop-down lists, which can be customised when launching the visualiser through the command-line interface. For instance, specifying `--algorithms A* <path-to-custom-algorithm>` results in the A* algorithm as well as the algorithm implemented in `<path-to-custom-algorithm>` to be present in the algorithm drop-down list. Furthermore, through the Simulator Configuration window it is also possible to precisely control the agent and goal positions, which is typically necessary when repeatedly conducting experiments.

For a suitable view of the simulation, the user can modify its style via the View Editor. A key technical challenge was to allow the user to add custom style information that can be interactively configured through the View Editor, with this configuration persisting beyond the lifetime of the application. To achieve this, the GUI state is saved to persistent storage through the `PersistentState` service (see section 3.3.4), so that it can be used in future runs of the application. This drastically reduces the time needed for configuring the simulation and its visualisation, as once the environment is customised to the user's needs, most settings do not require future modification.

Furthermore, this implementation enables rapid style customisation of the view of an algorithm's internal state, as each algorithm has its own unique structure that requires a custom colour scheme to be viewed nicely. For example, a suitable default colour for visited grid cells in the A* algorithm is grey (see Fig. 4), whereas red is preferable for rendering the internal graph structure of the Rapidly-exploring Random Tree (RRT) algorithm (see Fig. 6). Additionally, each user has their own colour preferences, therefore, colours are not hard-coded into the algorithm's implementation. Instead, they are exposed to the user for customisation through the GUI.

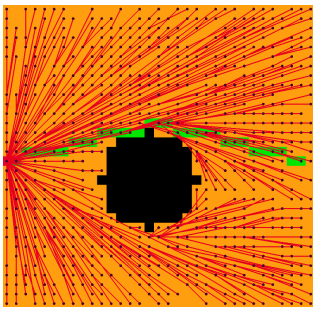


Figure 6: RRT* running on the 2D small obstacle map.

To this end, the `DynamicColour` class was introduced. This class stores modifiable colour and visibility data, in addition to a callback that PathBench3D uses to appropriately take action when the deduced colour (transparent if invisible, actual colour otherwise) is changed. As a result, the colour can be modified from any context, such as through the GUI, as well as by the algorithm implementation itself. `DynamicColour` objects can be created through `PersistentStateViews`'s (see section 3.3.4) member function `add_colour`, which takes as arguments the name of the `DynamicColour` and a default colour. Note, in the context of an `Algorithm` implementation, `self._services.state.views` is a reference to the `PersistentStateViews` global instance. Now, if a `DynamicColour` with the given name already exists, then it is immediately returned, otherwise a new `DynamicColour` object is created with the given default colour. This allows similar algorithms to share colours by using the same names; thereby effectively minimising the number of configurable colours, so as to not overwhelm the user with options. Indeed, RRT, RRT*, and RRT-Connect share the `graph edge` and `graph node` colours to visualise their internal graph structure.

Furthermore, through the View Editor, the user can individually customise six different views that can be promptly switched between, allowing the user to seamlessly inspect various aspects of the algorithm's internal state. Coupled with smooth orbital camera movement, which enables the simulation to be observed from different perspectives, users of PathBench3D can quickly gain unparalleled insights into the behaviour of their algorithms.

We shall now discuss details related to rendering a simulation. The key components are the `Renderer`, `MapView`, and the abstract class `MapDisplay`. The `Renderer` is responsible for rendering primitives, such as lines or spheres, and adopts an immediate mode style of rendering. That is, state such as line thickness can be pushed and popped from appropriates stacks, with the state at the head of the stacks dictating how rendering tasks should be performed. For example, in the code snippet below (List. 1), a white line with a thickness of 5 and a black line with a thickness of 2 will be drawn. As illustrated, this design abstracts away a lot of boiler plate code required for rendering objects in Panda3D. As a result, this empowers PathBench3D users to implement custom rendering routines for visualising the internal state of their own algorithms.

```
renderer = Renderer(...)
renderer.push_line_thickness(2)
renderer.push_line_thickness(5)
renderer.draw_line(WHITE, ...)
renderer.pop_line_thickness()
renderer.draw_line(BLACK, ...)
```

Listing 1: `Renderer` usage

The `MapView`'s purpose is to coordinate all rendering related tasks and exposes numerous rendering functionalities, such as colouring map cells and drawing shapes. Rendering utilities for drawing primitives simply convert the given map coordinates to graphical coordinates, before invoking the appropriate function in the `Renderer`. The `MapView` manages a list of `MapDisplays` provided by the algorithm to render its internal state. For instance, the `GraphMapDisplay`, which implements `MapDisplay`, uses `MapView` to draw lines, circles and spheres in order to visualise the structure of graph-based algorithms (e.g. RRT). Furthermore, the list of `MapDisplays` is ordered based on a so-called `z_index`, leading to user-defined overlaying of these instances. For example, if one `MapDisplay` colours a map cell and a subsequent `MapDisplay` colours the same one, then we perform alpha blending. That is, the colours are mixed together based on their transparency, such that if the colour specified by a `MapDisplay` is opaque, then the current colour of the map cell is ignored and overwritten with the specified colour. See section 3.3.3 for more details related to the usage and design advantages of `MapDisplay`.

It is crucial that the visualiser is as efficient as possible, as the path planning algorithm is blocked whilst its internal state is being rendered. However, this was a challenging task, as we cannot assume anything about how the algorithm will interact with the renderer. For example, the algorithm could use a `MapDisplay` for one key frame, but not the next. Note, a key frame refers to when the graphics is updated to reflect the

current state of the simulation. In the aforementioned scenario, it is expected that all map cells that had been coloured by the now absent `MapDisplay` are refreshed on the second key frame. To achieve this, the initial implementation of the renderer would simply refresh then entire map every time something changed. Evidently, the performance was terrible for any practical simulation since a global refresh, which can easily take several seconds, is essentially occurring at every key frame. Not only would this slow down the visualisation of the algorithm, but any user interaction, such as changing a colour or moving the position of the agent, will cause the visualiser to stall for several seconds. This made the visualiser almost unusable, as even the slightest amount of delay after user input can easily frustrate the most patient of users. Therefore, it is paramount that the renderer only updates what is necessary. The key step towards this goal was to enable the renderer to track any modifications to the algorithm's internal state between key frames. The solution to this problem is detailed in the "Algorithms", section 3.3.3. In addition to tracking any colour changes and their side-effects, the renderer is now able to immediately determine which objects in the scene require modification, and only update those ones. As a result, the visualiser is now very responsive and is able to smoothly display an algorithm running at a high speed.

Although 2D and 3D maps are rendered differently, it is necessary that the public interface of the renderer is dimension-agnostic. Furthermore, it is paramount that code duplication is minimised so that the renderer's implementation is both robust and scalable. To this end, the graphics implementation only deals with 3D coordinates, with necessary conversions to and from 2D being done at the boundary. Note, a 2D point is considered to have a value of 0 for its third dimension (z), which is desirable when creating 3D containers as the size of the last dimension is simply equal to 1. This design strategy effectively allowed sharing of most of the renderer's implementation, with only a few lines of glue code needed to interface with the low-level, dimension-specific components that will now be discussed.

Originally, it was envisioned that the 2D map would be rendered in the same way as 3D maps, as 2D maps are simply 3D maps with the third dimension having size 1. Indeed, during the early stages of development this was the case. However, procedurally generating a mesh for a 3D voxel map, which is simply a collection of cubes, can get computationally expensive due to the large number of vertices needed. This leads to unreasonably long load times for very large maps. Note, an alternative method that was first used is to perform geometry instancing, where a single cube mesh is procedurally generated and subsequently used multiple times in the scene at once. With this method, the map is very fast to load. However, the run-time performance is significantly worse than generating a single large mesh. For this reason, PathBench3D no longer uses geometric instancing. However, a working implementation can still be found in `OldDynamicVoxelMesh`. Furthermore, the single large mesh strategy has been implemented by both `StaticVoxelMesh` and `DynamicVoxelMesh`, with `StaticVoxelMesh` being an implementation optimised for maps that do not change shape, and `DynamicVoxelMesh` when they do. Compromises had to be made for `DynamicVoxelMesh`, which are detailed in "Dynamic Occupancy Grid Maps", section 3.3.6.

The group, however, was determined to minimise the load time for 2D maps. To this end, an entirely new method has been used for rendering 2D maps. As a first attempt, the map was rendered onto a single 2D texture, which was then applied to a flat rectangular mesh consisting of only 4 vertices. Furthermore, the number of pixels allocated to display each cell of a grid map was automatically decreased as the map size increased. For example, a map with each side of length 10 would have a 32x32 square of pixels to render to, but a map with side-length 1000, would have an 8x8 square of pixels to render a single cell.

However, having a texture with a size proportional to the map size was not a good idea. For very large maps this will lead to performance degradation as Graphics Processing Units (GPUs), especially low-end ones, are terrible at using large textures. It is recommended that textures are no larger than 2048x2048 [10]. Therefore, the map was split into blocks that each have their own texture so as to not exceeded the recommended limit. This design is also advantageous for efficiently implementing dynamic maps (see "Dynamic Occupancy Grid Maps", section 3.3.6).

This new method of rendering maps significantly improved loading time, but efficiently updating the texture when the colour of a grid cell changes was a challenging task. Instead of updating a texture pixel-by-pixel, which sends a write request for every pixel modified, it is much faster to simultaneously update a sequence of pixels that are contiguous in memory. However, the wireframe, which is not necessarily opaque, is rendered to the texture. Consequently, it is computationally expensive to determine the sequences of pixels used to render a square. In most cases there are few distinct colour combinations used at any given point in time, making caching suitable. But, in certain cases, such as with the A* algorithm, thousands of unique colour combinations may be used over the course of the simulation. Therefore, a Least-Recently-Used (LRU) dictionary cache was used so that only the most recently used colour combinations remain cached. Through experimentation, an LRU cache with 16 entries was found to work best. Overall, compared to using the same implementation as 3D maps, this rendering method is faster during both initialisation and runtime (see Fig. 10 and 9 respectively).

3.3.6 Dynamic Occupancy Grid Maps

An Occupancy Grid Map (OGM) is a grid map with each cell having a probability of being an obstacle. This map structure is commonly used in robotics. Indeed, the Robot Operating System (ROS) real-time extension requires an OGM representation that must be able to modify itself and grow in size during a simulation (see Fig. 7). To this end, the map class `OccupancyGridMap` was introduced, in addition to modifying several aspects of the visualiser to efficiently render dynamic (modifiable) maps.

`OccupancyGridMap` derives from `DenseMap` and has an additional attribute `weight_grid`, which specifies the probability of each cell being occupied. Above a certain threshold, such as 0.8, the cell is considered to be an obstacle, otherwise it is traversable. Additionally, `weight_grid` is incorporated in movement cost calculations, so that path planners will avoid moving to cells with high occupancy probability even if it would lead to a shorter path. `OccupancyGridMap` is currently the only map that allows cell properties, such as whether it is traversable, to change whilst the path planning algorithm is running. Moreover, when the map is modified, an instance of `MapUpdateEvent`, which contains a list of the cells that have changed, is dispatched. The visualiser listens for this event, and when it is received, it uses this information to update the map visualisation with minimal alterations; it would be unreasonably slow to perform a global map visualisation refresh every time the map changed.

Implementing true dynamic resizing of the map was infeasible as this would require reworking a significant portion of the codebase since the map is often assumed to be static for performance reasons. Performing such a breaking change within a limited time-frame would inevitably lead to subtle bugs and a noticeable performance degradation. For this reason, the existing map structure was instead extended by introducing the `UNMAPPED_ID` property, which signifies that the cell is unmapped. That is, unmapped cells exhibit the same properties as obstacles, but indicates that they do not actually exist. The visualiser, for example, will not render these cells. As a result, the map does have a fixed size, but specifying a very large map size is unlikely to cause issues with performance during run-time. Note, large map sizes will have a higher memory requirement, but it is still relatively low compared to the typical amount of memory available on modern computers. However, during initialisation, the map is iterated over several times, which significantly impacts load-time performance. To mitigate this issue in the future, the existing code would need to be modified to access the map data structures in parallel. Additionally, the map should be sliced into a list of blocks. In the case where all cells in a block are unmapped, the block would not occupy any memory other than a `None` entry in the block list. This would both reduce memory usage and significantly speedup iterating over a largely unmapped grid, as a non-existent block could immediately indicate that hundreds of cells are unmapped.

Although there are still several optimisations to be made on the logic side, the visualiser has been extensively fine-tuned for both 2D and 3D mutable maps to ensure fast rendering of graphics. Key optimisations performed for 3D maps will first be considered. Initially, `StaticVoxelMesh` was used to render mutable maps. This procedurally generates a cube for each and every cell in the 3D map during initialisation, including unmapped ones. Every cube requires 24 vertices (4 per face), therefore, generating a mesh for a large map is very slow, especially since the implementation is done in Python. To combat this issue, `DynamicVoxelMesh` was implemented, which allows cubes to be added during run-time, with only the cubes that are associated to a mapped cell being generated at initialisation. Note, dynamically adding cubes instead of statically creating all of them at initialisation does not cause performance degradation, as we are modifying the existing mesh rather than creating a new mesh for each cube. However, the wireframe rendering performance does not scale very well, since every time we create a wireframe object for a new cube, we are unable to merge it with existing wireframe objects. Consequently, there may eventually be thousands of wireframe objects populating the scene, which significantly impacts the renderer's performance. A promising solution would be to use custom shaders for drawing wireframes. This was attempted but led to numerous unrelated rendering issues that could not be fixed in time.

Now we shall discuss the key optimisation performed for 2D map rendering. 2D maps are split into blocks, with each block having a texture, flat rectangular mesh, and associated cells of the 2D map being rendered to the texture. Each texture is applied to their respective rectangular mesh, with the meshes tiled in the scene forming the flat plane of the entire 2D map. These blocks are created on-demand, meaning that if all the cells in the block are unmapped, the block will not be created. Therefore, lazy block creation significantly improves load-time performance as every block requires a moderately large texture of size 2048x2048 in pixels, which must be cleared to a transparent colour at initialisation. This strategy is particularly effective as mapped cells are typically grouped together, with a majority of blocks containing only unmapped cells. These optimisations have enabled prompt visualisation of modifiable maps.

3.4 Robot Operating System (ROS)

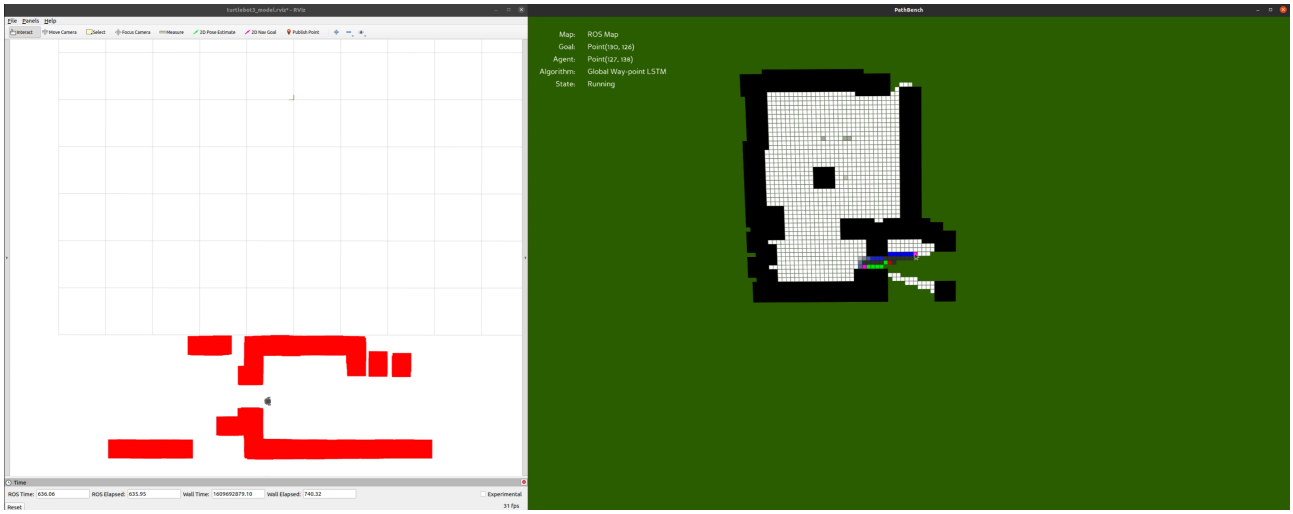


Figure 7: Using PathBench3D (right) for real-time control of a wheeled robot simulated in Gazebo (left).

The original PathBench framework already implemented a real-time extension for the Robot Operating System (ROS). This extension enabled visualisation, coordination and interaction with a physical robot. However, the implementation was inelegant and lacked several key features, which hindered any practical use of the extension. ROS being a framework that is used by the majority of roboticists, makes this extension vital to the success and practicality of PathBench3D in the robotics community. Accordingly, the ROS extension was written from scratch. The result is better usability and integration with 2D maps. Additionally, PathBench3D also supports ROS for 3D maps, however, the thin wrapper around PathBench3D that interacts with external ROS nodes is not implemented. Note, a ROS node is a process that performs computation such as estimating the pose of a robot [13]. Porting the existing wrapper for 2D maps to 3D should be straightforward, only requiring minor tweaks to the conversion of raw map data to a format PathBench3D understands.

The typical usage of the ROS extension is to visualise the path planner as it controls a robot in real-time. The robot is usually equipped with a sensor for simultaneous localisation and mapping (SLAM), which is performed external to PathBench3D. For example, the demonstration included in the PathBench3D repository uses ROS's `gmapping` node, which processes laser scanner sensor data to construct a 2D Occupancy Grid Map (OGM) of the environment, as well as estimate the pose of the mobile robot within it [8]. Furthermore, the map is updated as the robot moves, changing the properties of existing cells, in addition to resizing the map as the robot visits previously unexplored areas. This means that maps within PathBench3D must be modifiable during run-time, which has been efficiently implemented with reasonable compromises (see "Dynamic Occupancy Grid Maps", section 3.3.6).

The main components of the ROS real-time extension are the `OccupancyGridMap`, `RosMap` and `Ros` classes, which have mostly been implemented from scratch; the logic for sending velocity commands to the robot is the only pre-existing code that remains. As previously discussed, all ROS related components support both 2D and 3D maps with the exception of `Ros` that exclusively supports 2D, but little work is required to port it to 3D.

`RosMap` is a simple wrapper around `OccupancyGridMap` (see section 3.3.6) and adds the ability to both request map updates and move the robot from within the path planner. Note, even though map updates are regularly received from external ROS nodes, the path planning algorithm must manually request the simulator to be updated for thread safety. `RosMap` is used by `Ros`, which is a thin wrapper around the PathBench3D simulator that interacts with external ROS nodes, and is itself a ROS node named `path_bench`. The `path_bench` node subscribes to both `map` and `odom`, as well as publishing `cmd_vel`.

When `Ros` is requested to move the robot to a specific grid position, this position is then converted by `Ros` to ROS's world coordinate system, and velocity commands that appropriately move the robot to the desired position are subsequently published via `cmd_vel` using ROS `Twist` messages. To know the current location and orientation of the robot, `path_bench` subscribes to the `odom` node, which publishes `Odometry` messages. This data is used to control the robot, in addition to synchronising in real-time with the agent position in PathBench3D's simulator. Note, agent position modification through the user interface is disabled with `Ros`.

The `path_bench` node also subscribes to the `map` node that publishes `OccupancyGrid` messages, which contain map information. This map data is first pre-processed within `Ros` into a format that PathBench3D understands. To support very large ROS maps, the ROS map is first down-sampled to a suitable size, leading to each cell of

PathBench3D's representation having an associated probability of being occupied (stored in the `weight_grid` attribute of `RosMap`). In the original implementation, the map size was unable to grow larger than the initial map fragment received, which is typically much smaller than the true map size. As a result, the previous implementation would simply keep a small fixed view of the map and ignore any important updates, often leading the agent to leave the map entirely. In contrast, the new implementation allows the user to specify the map size, in addition to the portion of the map the initial map fragment can occupy. This may cause the map fragment to be down-sampled, with the scaling factor being used for all subsequent map updates. Typically, the map is much larger than the space afforded to the initial map fragment, noting that the fragment is positioned at the center of the map. This enables the robot to roam endlessly in any direction, slowly mapping the entire map. Although the view of the large map is fixed, the edge of the map is never reached in practice.

The ROS real-time extension provides all the features that are necessary for analysing novel path planning algorithms in a real-world context. The plethora of features available, namely map down-sampling and dynamic growth, allows the robot to interact with a large and complex environment. As a result, researchers developing path planners can rapidly gain insights into the practicality of their algorithms, such as how they behave when the map is partially visible. The ROS extension has been demonstrated to work with Gazebo — a popular robot simulator — using a `turtlebot3` wheeled robot (see Fig. 7). To this end, the Global Way-point LSTM algorithm was used to guide the robot to a goal outside the initial map view. Although the extension has only been tested with a robot simulator, no changes are necessary for it to be used to control a physical robot. The extension is currently restricted to controlling robots on a 2D plane, however, with few modifications to `Ros`, researchers could conduct experiments in a three-dimensional space to control drones or robot arms for example.

3.5 Generator

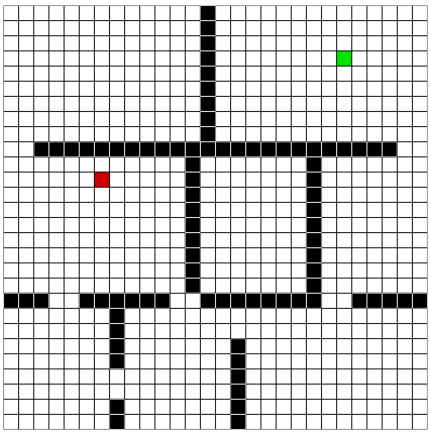
The Generator component is used to generate random maps in varying formats specified by the configuration file passed into it. Beyond the already implemented generation of two dimensional maps in the `DenseMap` format, it has been extended to generate three dimensional maps in both `DenseMap` and `SparseMap` format. Discussed in section 3.3.2, the `SparseMap` format was introduced to the generator for the purpose of allowing the user to create point cloud representation of their maps, allowing for the development of algorithms that train efficiently on point cloud representations such as MPNet. The generator is also responsible for labelling the training data used to train the ML models.

PathBench3D supports three types of map generation algorithms. First, uniform random fill maps, where single obstacles are placed at random in the grid. Second, block maps, where a random number of blocks, each of random size, is placed in the map; and third, house maps, where obstacles form walls to partition the map into sections of empty space. For each of these maps, once the obstacles have been generated with their respective algorithms, the agent and the goal are placed in random empty spaces on the grid.

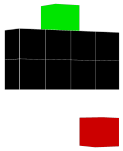
The generator accepts several hyperparameters as input. These include the type of map, number of dimensions, number of generated samples, obstacle fill rate range, minimum room size range, and maximum room size range, as well as the various labels required by the `Trainer`. Default numerical parameters are given as examples for different map sizes in the `generator.py` file. Furthermore, the user also has the ability to set these values manually using the command-line interface. When the user provides an input range (e.g. representing the minimum and maximum room size), a random number is picked from the range and fed into the appropriate generator. The original PathBench implementation made use of many 2D-specific algorithms for generation, such as dual `for` loops iterating over x and y coordinates. This was modified to use an n-dimensional `tuple`, produced by a library function `numpy.ndindex`. Once these changes were made, the rest of the map generation implementation was generalised to work with arbitrary dimensions.

An update made to the house generation algorithm after it had been ported to 3D, was the introduction of doors on every wall. Once all the rooms have been split and the walls have been generated, doors are added to ensure some room connectivity (fig. 15 in Appendix). This modification enabled proper training of machine learning algorithms, as before implementing this door change, 70% of generated 3D house maps had completely separated `Goal` and `Agent` positions, meaning the agent is unable to find the goal, leading to inadequate training data. After implementing this change, only 10% of generated 3D house maps did not contain a clear path from the agent to the goal, which was a vast improvement.

Once all of these maps have been generated, they are saved and stored using the existing `Directory` class. Currently, PathBench3D supports saving and loading maps in both `.pickle` and `.json` formats. The initial framework only supported the saving of 2D maps in the `.pickle` format, which were generated using the python package `dill`. The reason for this was that `dill` supported serialization of lambda functions, which were included in the map code. We then extended this functionality, in addition to supporting JSON file. This implementation decision was made according to four key reasons. First, the pickle files saved by `dill` were very large. A generated 8x8 map, for example, stored in pickle format takes up on average 1kB of storage, compared



(a) A 2D house map



(b) A 3D block map

Figure 8: Both using the default simulator screenshot. An example of the third type, Uniform Random Fill, can be seen in Fig. 5.

to 100 bytes in the JSON format. Second, JSON files are easy to read as they are text-based. Pickle files are binary and cannot be inspected, which leads to more difficulties in the debugging process. Third, JSON file formats are supported by nearly all programming languages, assisting with our goal of increasing system portability. Fourth, throughout the entire project, we were wrestling with issues that arose from using the pickle format. For example, one bug that had to be fixed initially was that in a recent update, `dill` broke backward compatibility between Python 3.7 and Python 3.8. As a result, all of the maps had to be regenerated and stored in the new version before they could be successfully loaded and used. Other issues encountered with the `.pickle` have been outlined in section 3.3.1.

3.6 Trainer

The Trainer class is used to train different Machine Learning-based algorithms. It is a wrapper over the Pytorch python package for machine learning and provides a generic training pipeline for path planning models based on the holdout method and standardized access to the training data generated by the Generator. Pytorch was chosen over Tensorflow for this project as Pytorch uses a dynamic graph model, which is particularly helpful in the implementation of variable length input RNNs (such as those used in LSTMs). All of the learned models currently in PathBench3D are functional only in the original two dimensional space. Our development process on the Trainer took a debugging approach, trying to make as few radical changes as possible while updating all of the interactions with previously altered code such as the `Point` class. We highlight a few key bugs discovered and fixed.

Each learned model in PathBench3D (LSTM, CAE, Combined LSTM+CAE and also VIN) inherits from the `MLModel` class, which trains its models using a standard holdout method. The models are trained on the data labels generated by the Generator class, using data received from the A* algorithm. Each model contains metadata about what labels it needs. For example, LSTM trains on raycast information and angle to the goal, whereas VIN just uses the map and path. Many labels, such as `direction_to_goal_normalized` rely on the `Trace` class information from the A* algorithm. However, a result of one of the changes we had made to the `get_move_along_dir` function (removing the incorrect default return of the function when given a `Trace` that had a leg with distance 0), created a resulting tensor matrix that was improperly sized with fewer steps than expected. Additionally, `Trace` generation, which is based on the Bresham's line algorithm, was incorrectly implemented; exacerbating the previously mentioned issue. However, these bugs were eventually fixed.

The `run_trainer.py` pipeline was also adjusted to include a full train flag. The training of combined algorithms on a combination of maps, such as including all three types of maps in a single model's training or the `CombinedLSTMCAE` algorithm, requires all of the individual components (i.e. the training data for each individual map type, or algorithm type) to be in the training data folder before generation as the combined versions rely on their existence. Therefore, the full train flag was created to ensure that generation carries on in the absence of any prerequisite data, as it generates all of the needed component data before creating the final combined dataset.

3.7 Analyzer

Finally, the analyzer is responsible for assessing and comparing the performance of path planning algorithms. The analyzer must, therefore, manage the statistical metrics used in the practical assessment of the algorithms.

In order to successfully analyse various path planning algorithms, it is useful to have several metrics; the path length is significant, although, sometimes, a fast algorithm with a close-to-optimal path may be preferable, or perhaps an algorithm that consumes less memory. These are all useful in determining the most optimal algorithm for the specific environment, robot, and situation.

The planning metrics can be divided into three categories: cost, reliability, and utility. Cost metrics (for example, distance) are quantities that must be minimized. The reliability metrics demonstrate the consistency of the algorithms and their accuracy (e.g. success rate). Finally, utility metrics quantify certain features of the algorithm performance. The original framework included support for various cost-related metrics, such as session search space, path length, distance traveled, and computation time. It also contained reliability-related metrics: success rate, precision at target, and path deviation. However, no support existed for the analysis of path planning algorithms using utility-related metrics.

From speaking to Dr Saeedi, it was clear that additional metrics were necessary for a more thorough analysis of path planners. The initial objective was to implement the memory consumption metric as Dr Saeedi had confirmed that memory usage was "the top priority".

Table 1: Implemented metrics, each compatible in n dimensions.

Category	Metric	Description
Cost	Memory consumption	Maximum memory usage in MB for each trial
Utility	Obstacle clearance	Mean distance from obstacles during traversal
Utility	Smoothness of trajectory	Average angle change between consecutive segments of paths

The memory consumption metric is extremely useful in distinguishing how path planners would run in a limited or embedded environment. In order to implement this metric, the `tracemalloc` library was used to trace memory blocks allocated by Python. The `tracemalloc.start()` function is called at runtime to start tracing Python memory allocations just before the simulation of the algorithm is run in `sim.start().get_results()`. Once the algorithm has completed running, `tracemalloc.get_traced_memory()` is called to obtain the peak memory usage during the computation in the simulation before completing the tracing of Python memory allocations with `tracemalloc.stop()`. The peak memory was chosen, as it provides a good idea of the maximum demand an algorithm has, rather than using, for example, the average. A reason for this is that the average memory could be manipulated, with a high spike in memory during the time in which an algorithm runs, followed by little to no memory usage for a few seconds to bring down the overall average.

In the first attempt to implement the obstacle clearance measurement, the map was copied and a second "distance map" was computed, with each value being the closest distance to an obstacle. After this, an average was taken of the numbers along the path in this new map. Unfortunately, this approach was slow. Despite the averaging operation being fast, generating this map took up to a second to complete, which considerably slowed down the analysis, since this process was repeated over hundreds of maps. As a result, we implemented a new approach, which involved the manipulation of a list of obstacles purely using `numpy`. Calculating over the whole list at once for efficiency, the distance from each obstacle to the agent is calculated and the minimum over all obstacles is taken. This is repeated for each step of the path, and the average of the minimums is taken. This approach seems more computationally expensive, but it performs much better, presumably due to `numpy`'s C backend.

In order to integrate the measurement of path smoothness, the `get_smoothness()` function was implemented, which takes in the trace and the agent and returns the float value representing the "smoothness". This function returns the average of the differences between angles of consecutive points along the trace.

The Python library, `matplotlib`, was also used to visualise the analyzer results as both violin plots and bar plots illustrating a selection of metrics for each algorithm compared (see Fig. 17 and 18 for an example).

To make the analysis more seamless, a command-line interface was implemented for configuring the algorithm details and dimensions during the analysis phase. Here, the user can run multiple algorithms at once by specifying this in the command line, compare their performance, and draw essential conclusions on the optimality of various path planners as effortlessly as possible. The command line options include:

- `--algorithms` to specify a list of algorithms to include in the analysis.
- `--include-builtin-algorithms` to analyse vs. all included algorithms.

- `--maps` to specify maps to run on.
- `--include-all-builtin-maps`, `--include-default-builtin-maps` specifying builtin maps to run on.
- `--list-algorithms`, `--list-maps` to list the builtins that can be specified.

3.8 Further implementation details

Although PathBench3D's public API is written in Python, a fundamental initial design decision was the programming language to use for the system internals. Initially, the group thought that using Python would not be a good fit due to its lack of speed compared to other languages, such as C++. However, it is easy to integrate a variety of popular, high-performance C++ libraries, as most offer Python wrappers that abstract away the complexities of the C++ implementation with a simple Python API. As a result, Python was used for the entirety of this project as it enables rapid prototyping of ideas without sacrificing on performance. For areas within the PathBench3D codebase that require further optimisation, it is trivial to transfer a small portion of code into C++ to run in a multi-threaded context for example. Another reason why Python was chosen in favour of C++ was that only half of the group had any experience with C++, which is a rather complicated language to use. Therefore, using C++ would have significantly impacted the speed at which new features could be developed, especially during the initial iterations of this project. Since Python is suitable for varying levels of expertise, all team members were comfortable using this programming language.

3.9 Anticipating and mitigating risks

At the start of the project, we faced the challenge of getting to grips with an extensive pre-developed system. We needed to invest a considerable amount of time to familiarise ourselves with the internals of a completely new environment and understand the specific functions of each component before beginning the implementation of our software. In general, we also lacked any background knowledge of classic and learned path planning algorithms. To minimise the uncertainty of being unable to deliver a satisfactory end-product, we dedicated a lot more time to this project during initial iterations compared to later ones. This allowed us to read through the provided documentation in detail, as well as explore the field of robotics to gain a deeper understanding of the task and the structure we would need to adopt for our platform, PathBench3D. Furthermore, splitting our group into a graphics team (two members) and a logic team (four members) helped manage the volume of code we were required to familiarise ourselves with, which enabled us to work much faster in parallel. Each sub-team worked on their corresponding isolated part of the software and, if required, could provide high-level insights on particular parts of the code to members of the other sub-team.

Another risk was the possibility that the original author of the code would be unavailable for discussion, leading to extended difficulty in comprehending the code. On initial inspection, we anticipated that this problem could be compounded by the code not containing sufficient comments explaining vital components. However, this was mitigated by us communicating weekly with the supervisor of the original project, Dr Saeedi, who was able to provide us with valuable insights. Furthermore, towards the second half of the project we were lucky enough to also speak with the students that had previously used the original PathBench framework.

We additionally anticipated that strong teamwork and communication within our group could also be at risk. However, by adopting several software engineering practices, such as Kanban, pair programming, setting weekly meetings, etc, we successfully avoided potential issues. As a result, the six of us have managed to exchange ideas well and combine different skillsets in order to complete tasks fast throughout the project.

Another problem that was encountered early on was that the team members used different operating systems. PathBench required numerous third-party libraries with a few proving difficult to install. For example, the `dataset` package, which has since been removed from PathBench3D, is broken on certain Linux systems. This required us to manually make changes to the `dataset` package in order for PathBench to work. This seemed to be the beginning of some inconsistencies, but, on the other hand, it also forced us to write highly portable code, which was a goal of our project, by avoiding the use of low quality, unstable python packages.

4 Evaluation

4.1 Software Quality

4.1.1 Testing and Styling

Although the provided framework included a few tests for the existing components, it was lacking automated testing and a consistent code style. Throughout this project, we successfully followed several testing practices and used an automatic style guide in order to maintain a high software standard.

An initial focus was put on unit testing, since this is crucial when specifying the expected behaviour of our code. In addition, automated testing has been set up to verify that any changes to the code were non-breaking and that it is safe to merge branches to master. An example of these tests, running on a pull request in GitHub, is shown in Fig. 22. Along with the implementation of new features, it was equally essential that the test suite for the logic code was gradually updated. To keep track of this, an automated code coverage tool, Codecov, was used. This tool generates code coverage reports and allowed monitoring the comprehensiveness of our tests. Fig. 21 illustrates an example of how the coverage shows which parts of the code need more testing. Codecov formed part of our main development pipeline, which was extremely useful in catching any corner cases within our code that hadn't been tested. Additionally, this improved our team's code review workflow and quality, and the practice of thorough user testing helped us maintain the code during new feature development without breaking the existing features.

Initially, we faced a challenge whereby even small changes within the code would introduce subtle bugs in the GUI that could quite easily be left unnoticed. Therefore, carrying out system-level functional testing was fundamental to maintaining a robust version of the code on the master branch, as it would prevent any unexpected issues with displaying the graphics. General difficulties in testing graphics meant much additional research was required. Although we had no prior experience with this, we successfully managed to develop a set of utilities to launch PathBench3D headlessly and subsequently test it using bots (PyAutoGUI) to interact with PathBench3D's user interface. We ensured that every test is run individually in its isolated environment, which mitigated the initial side effects of running the whole test suite at once, leading to unpredictable errors when creating and killing a system window.

An automated graphics test starts by selecting a map and algorithm from the Simulator Configuration window. Next, several modifications are performed using the View Editor. These include choosing the start and end goal positions on the map, testing correct colouring of entities, setting the map transparency, testing movements, such as rotation, toggling between different states, etc. Once sufficient functionality has been tested, a top-level screenshot is taken of the chosen map, which is finally compared to an existing reference. The reference is a test image of the same map that had previously undergone the same transformations, making it a useful baseline for comparison. Finally, Mean Squared Error (MSE) is used as a metric for comparing produced images to their references as it can be applied globally when estimating the perceived image errors. A small error rate (difference) is allowed to pass a test as 3D graphics rendering is not required to be perfectly accurate.

Moreover, our group managed to find and subsequently integrate utilities that enable a user to view the virtual window and interact with it manually. This useful addition enables easy debugging of tests, using a standardised window resolution, ensuring a nice and robust interface. Overall, our graphics tests cover most of the functionality of the View Editor and Simulation Configuration, as users would often interact with these windows. The correct rendering of algorithms is also tested in both 2D and 3D for A* and Potential Field, 3D for Wave-front, 2D for RRT, 3D for RRT-Connect. This essentially covers every aspect of the rendering pipeline (lines, nodes, colouring map cells). Note, some of the algorithms, such as RRT and RRT-Connect, include randomness in their implementation. Therefore, the internal state of the algorithm when the goal is reached can vary, resulting in variable visualisation. To guarantee predictability, we have pre-set random generator seeds and have added deterministic functionality to our pipeline, which is used for the entire test suite. We identified this important edge case at the right time since our tests managed to catch a later bug, where the RRT algorithm could not render the path correctly. Due to the large variety of algorithms and maps offered by PathBench3D, it would be impossible to frequently perform manual testing, which is another benefit of having this process automated. Overall, the provided test suite can guarantee functional and usable code on the master branch.

To help maintain a uniform style within our code, our team also used a project-wide pep8 style guide. However, we found that cleaning up the entire existing codebase was a huge task that would take too long to achieve in our limited time-frame. With the default styling rules in place, there were over 9000 style violations, and even after removing some more minor ones, there were still more than 1000 left. Therefore we decided to only ensure that any code we produced would conform to the style guide. However, there is no automated styling yet, as this would require fixing the whole repository first, which was not a task that was a priority for our stakeholders.

4.1.2 Minimising Duplication

In order to reduce duplication within our codebase, we tried to abstract away any generalisable code into separate classes, making this functionality reusable. On several occasions, we used the Template Method design pattern for our implementation. Overall, the extensive use of this pattern produced a simple code structure that was easy to maintain and keep bug-free. For example, we would occasionally face issues or discover areas of improvement in our code, and having strongly decoupled code helped us recover from these by making the required changes fast and in only one place.

4.1.3 Version Control and CI

Version control and continuous integration also helped improve the quality of our software. For version control, we used GitHub as this is the ideal platform to host an open-source project. Furthermore, GitHub seamlessly integrates several project management tools, such as an issue tracker, which we have used throughout this project. Additionally, we used GitHub's CI/CD tools to automate our software delivery process.

4.1.4 General Git Practices

For this project, we chose to adopt feature branches and rebasing, having the branch names describe the current feature implemented in that branch. To ensure a coherent and feature-specific workflow, we would commit our work often. Once a feature was complete, we would rebase the feature branch on top of the master branch, squashing commits if needed. This allowed for easier debugging, since there was no commit inter-leaving between features. Furthermore, we protected the master branch — no one was allowed to push or make changes within this branch until their code had been reviewed by at least two other members of the team. This also ensured that everyone had a working understanding of the entire codebase. Moreover, it was only possible to push to the master branch when the commit passed the continuous integration tests. These practices were invaluable in maintaining a high standard in our code quality.

4.1.5 Performance Testing

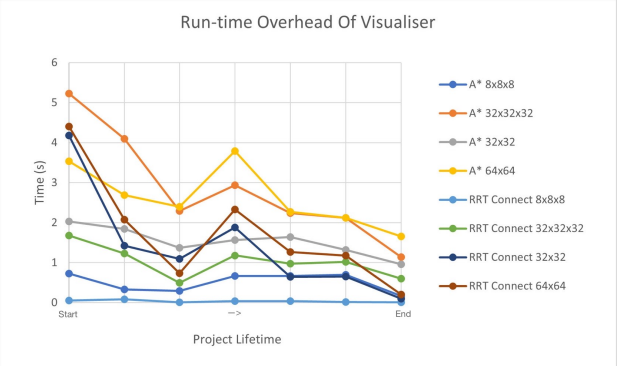


Figure 9: Run-time overhead of visualiser (see Appendix, 7 for system specs)

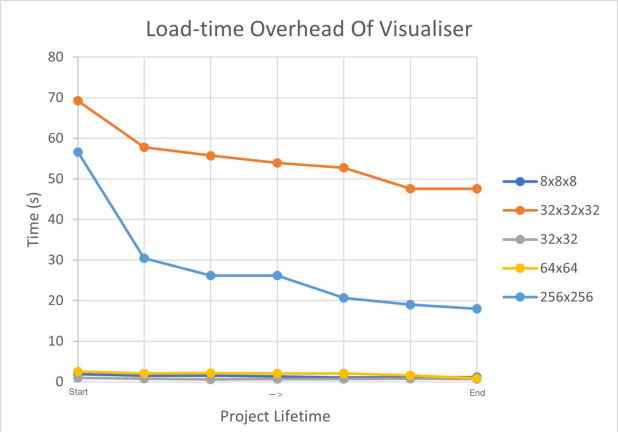


Figure 10: Load-time overhead of visualiser (see Appendix, 7 for system specs)

To understand the scalability of our framework better, we have carried out several performance tests. In order to assess the improvement in performance of our visualiser, we have measured the overhead of graphics when both initialising and running a simulation, recording the results at various points (commits) in the project's history. We have gathered these results for both 2D and 3D maps, each of varying sizes, notably, 32x32x32 for 3D maps, and 256x256 for 2D maps. The aforementioned map sizes provided the greatest insight into how the performance has improved over the lifetime of this project, as even with no optimisation, smaller maps were already fast to initialise and render. Furthermore, these experiments have been performed with A* and RRT-Connect as they greatly differ in terms of how they are rendered, with this combination covering most aspects of the renderer. The results shown above (see Fig. 9 and 10), match the trend we hoped to identify.

4.2 User Evaluation and Testing

Throughout this project, we have frequently communicated with wider members of our project to ensure that we are on track to meet our initial objectives and that our framework aligns with our stakeholders' goals. These include our supervisor - Prof Paul Kelly, external professor - Dr Sajad Saeedi, and students - Riku Murai, Bruce Hsueh and Hussein Jaafar. In order to evaluate the outcome of our work, we have carried out extensive user testing with all three students, and we have spoken to these stakeholders weekly during our project. The feedback that we received during each weeks' session along with the questions we asked allowed us to adjust course frequently throughout the project, and proved very beneficial to ensuring we stayed on the most important development paths. We have outlined some of the final feedback we received that highlights the impact, benefits, and future possibilities of our work to our target users.

"Historically, if you have a new path planning algorithm, then you need to implement your own, and you need to select the key pre-existing algorithms against which you should compare, run your own experiments, set up

your own metrics - what you are offering, is the ability to create a new path planning algorithm - plug into PathBench and then you immediately get evaluation sections for your paper offering a head to head comparison against the state-of-the-art metrics you are interested in." - Prof Kelly

Throughout the development of our software, a key focus has been our end-users, and we have aimed to implement as many useful features as possible: "Overall it's really nice ... Thanks for working so hard, there's way more features than we could have imagined! There's definitely lots of expansion to the original PathBench." - Bruce Hsueh

Our stakeholders have already performed user testing on our platform, noting their positive experience with it: "Regarding multiprocessing, I ran a couple of quick tests and it seems the restructuring of the initial PathBench to PathBench3D has solved some of the pickling issues I was having." - Hussein Jaafar. "With this framework, the overhead time to run other algorithms and benchmark them and compare our custom algorithm against them is very minimal, as this is all executed on the platform so all we need to focus on our algorithm!" - Dr Saeedi.

The ease of use was also highlighted by our users: "I previously hadn't used any 3D planning software but this platform was very intuitive... I clicked one button and it was working right out of the box." - Hussein Jaafar. "Now that everything moved into main it's all easier to run from the command line now so that's a big advantage." - Bruce Hsueh.

We also placed importance in ensuring our 3D visualiser was of a very high quality, providing as smooth as possible user interaction with it, since it is the key environment to assess all kinds of algorithms: "This allows you to see the 3D planning in a very simple and easy way for the user." - Hussein Jaafar.

We are extremely proud to have met all of our initial project objectives, including additional stretch goals, and developed a highly functional, unified framework for our stakeholders, with a huge potential in future research and development: "It will accelerate algorithmic development and benchmarking because if someone starts from scratch or wants to develop a new algorithm usually they'd have to get one from one repository and another from another repository, visiting two places. Here, we bring it all to one place, under the same framework, therefore it's much easier to run them." - Dr Saeedi

Furthermore, in terms of our additional ROS extension that we developed, we also received a positive user feedback: "Regarding ROS, I think the raw functionality especially the dynamic resizing of the map is a very very essential and it sets PathBench apart from other frameworks." - Hussein Jaafar

Overall, we are excited that we have made a key contribution in solving the initial problem roboticists were facing, that will reach a large audience in the robotics research: "The plan to not only use this software, not only to make it available to others, but to even write a paper for a robotic conference proposing and demonstrating its use as a public open benchmarking framework! The paper being proposed is going to enable a lot of people to be using a common and consistent framework for comparing path planning algorithms in a uniform context head to head with uniform metrics and that is going to change the world..." - Prof Kelly

5 Ethical considerations

5.1 Humans

Throughout our project, we have spoken with many wider members of PathBench3D in order to gain any insights and feedback to help drive our development. The participants of user testing and evaluation during our project have provided us with their consent to use this information in our report.

5.2 Protection of personal data

5.2.1 Machine learning

There is much debate about the use of possibly private data in the training of a public model – is it possible to retrieve information about a private dataset? Fortunately, we do not have to consider this, since the datasets used for training are generated by PathBench3D, with all the pre-trained models and code for generation released publicly. Additionally, all the algorithms used are based on papers that are publicly accessible.

5.3 Dual use

Path planning has vital applications in autonomous systems, such as robots, self-driving cars, and even the military, all of which carry a plethora of ethical issues.

5.3.1 Robotics

Historically, robotics was only prominent in traditional industrial environments. Today, this has grown to coworking and coexisting with humans across many areas such as medicine, education, leisure, and the general service domain. Aside from the technical concerns limiting a robot's ability to collaborate with humans, there are still several ethical aspects of robots sharing an environment with humans, and these are crucial to consider. These cover issues regarding the level of autonomy robots should have, to more applied issues, such as the impact of robotic technologies on public safety and personal privacy. [5]

5.3.2 Autonomous vehicles

There are also several ethical issues surrounding autonomous vehicles. When humans navigate the roadways, they must balance key values including safety, legality, and mobility, and it is important for an autonomous vehicle driving on the same roadways to also navigate based on these values. Thus, for engineers of autonomous vehicle technology, it is highly necessary to connect these values to the algorithm design.

5.4 Misuse & Military

Regarding the misuse of our product, there is a chance that our platform could be used for malicious applications of path planning. Military applications present a similar problem. For example, path planners can be used to plan efficient paths for military unmanned vehicles. In this situation, it is vital to consider both the ethics and legal implications behind these vehicles. Most are used in surveillance, however, some vehicles could be used in combat for deploying harmful weapons. This combat capability raises wider ethical concerns, such as, "Is it morally right to kill on a large scale?"

However, there is very little we can do to mitigate this. PathBench3D is open-source, meaning that anyone can download the code, modify it and even release their modified version (see 5.5). Even if PathBench3D itself was unavailable for military use, all of the algorithms it uses, and presumably any created with its help, would be open-source as well. Taking all this into consideration, discussing how to mitigate misuse, either personal or military, seems outside the scope of this project.

5.5 Legal issues

Our project builds upon the existing PathBench software for which there were existing copyright licensing implications. After discussions with our supervisors concerning project licensing, we reached an agreement that the BSD-3 license should be used. This license has allowed us to modify the existing software provided we include the BSD-3 copyright and license notice. Furthermore, we are extending the license to include all the software artifacts produced throughout this project.

6 Conclusion

6.1 Summary

PathBench3D is one of a kind, as it is the only publicly available framework, which unifies several main features into one platform, making it very portable and intuitive to use for its variety of purposes. PathBench3D is a powerful visualisation and planning tool, which supports easy integration and evaluation of both classic and learned motion planning algorithms in a two or three-dimensional space. This platform includes a well-structured development environment, in addition to extensive benchmarking utilities according to standard and customisable metrics. The new highly interactive, user-friendly, feature-rich 3D visualiser effectively assists further analysis of path planners. Lastly, the new real-time extension for ROS presents an essential advantage for researchers using PathBench3D to rapidly gain insights into the practicality of their algorithms.

6.2 Future work

Throughout the development of PathBench3D, we managed to gain as much performance as was possible within the limited time-frame we had. The current bottleneck of the system is that the algorithm is being run on the same processor core as the simulator due to Python's Global Interpreter Lock (GIL), which means that even though these are running on separate threads, they cannot be run in parallel. Consequently, they continuously contend for resources, leading to significant performance degradation. To alleviate this issue, design principles from the Bullet physics engine could be adopted [11]. That is, the simulator would run in a separate system process with shared memory, which should be feasible to implement with little modification to PathBench3D's public interface. Moreover, frequently run code (hot spots) could be implemented in C++ using real multi-threading, in addition to off-loading compute to the GPU or using vector instructions where appropriate.

Furthermore, if the visualiser was fully decoupled from the simulator logic (currently tied to the algorithm) into two separate processes, it would then be possible to develop a low-level interface for memory sharing and communication between the two processes. This would subsequently allow for a variety of languages to implement the purely logic side of the simulator, enabling the usage of algorithms implemented in a variety of languages all at once.

Another future extension to the PathBench3D framework would be to further optimise dynamic occupancy grid maps (OGMs). The group had not foreseen the effects of having traversable grid cells with varying occupancy probabilities, which visually leads to the cells having subtly different colours. This obliterates any cache optimisations that have been performed for ordinary maps. Consequently, the runtime overhead of rendering OGMs becomes unreasonable for very large maps. To combat this, an entirely different rendering strategy is needed. However, integrating OGMs was a last-minute stretch goal, therefore, there was not enough time to properly optimise this functionality.

PathBench3D supports a wide range of path planning algorithms, including all of the classic path planning algorithms present in the original framework in both 2D and 3D, along with several OMPL algorithms in 2D. However, as the diversity of algorithms is rapidly expanding, it is vital for PathBench3D to keep up to date with as many of these as possible. Hence, a future extension would be to include support for even more algorithms, in addition to adding 3D support for the existing state-of-the-art OMPL algorithms. Besides this, Dr Saeedi has verified that "The biggest advantage of this framework is that we have machine learning and classic algorithms all in the same boat under one framework. Currently, no other framework does this." Therefore, it is imperative that this framework is extended with the addition of many novel machine learning-based algorithms.

Furthermore, another interesting extension would be to interface PathBench3D to hardware in order to carry out real-world experiments in three dimensions. Doing so would truly illustrate the extensive capabilities of this framework. Additionally, Dr Saeedi expressed his interest in this and verified that with the existing software, it would not be difficult to achieve. He stated, "Although there was not enough time for hardware experiments, this framework can be easily be interfaced to hardware ... Gazebo is a very good real-world model for this ... When doing real-world experiments in 3D the choice of models is very extensive, and includes robotic arms or even drones."

References

[1] Hsueh B, L D. PathBench. https://github.com/uncobruce/PathBench [Online, accessed 7-October-2020]

[2] OMPL. https://github.com/ompl/ompl [Online, accessed 10-October-2020]

[3] Toma A. Global Way-point LSTM Planner: An Online Machine Learning Solution for Robotic Path Planning [Internet]. Available from: https://github.com/uncobruce/PathBench/blob/master/readme_files/FinalReport.pdf

[4] Path Planning - an overview | ScienceDirect Topics [Internet]. Sciencedirect.com. 2019 [cited 2 January 2021]. Available from: https://www.sciencedirect.com/topics/engineering/path-planning

[5] Van Wynsberghe A, Donhauser J. The Dawning of the Ethics of Environmental Robots [Internet]. 2017. Available from: https://link.springer.com/article/10.1007/s11948-017-9990-3

[6] MPNet [Internet]. Sites.google.com. 2021 [cited 29 December 2020]. Available from: https://sites.google.com/view/mpnet

[7] Robot Operating System [Internet]. En.wikipedia.org. [cited 3 January 2021]. Available from: https://en.wikipedia.org/wiki/Robot_Operating_System

[8] gmapping - ROS Wiki [Internet]. Wiki.ros.org. [cited 3 January 2021]. Available from: http://wiki.ros.org/gmapping

[9] (PathBench V2) PathBench: A Benchmarking Platform for 2D Classic and Learned Path Planning Algorithms.

[10] Choosing a Texture Size — Panda3D Manual [Internet]. Docs.panda3d.org. [cited 29 November 2020]. Available from: https://docs.panda3d.org/1.10/python/programming/texturing/choosing-a-texture-size

[11] Coumans E. Bullet 2.83 Physics SDK Manual [Internet]. 2015 [cited 15 October 2020]. Available from: https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf

[12] Teixeira M, Santos H, de Oliveira A, De Arruda L, Neves Jr F. Robots Perception Through 3D Point Cloud Sensors [Internet]. 2017 p. (pp.525-561). Available from: https://www.researchgate.net/publication/317754814_Robots_Perception_Through_3D_Point_Cloud_Sensors

[13] Nodes - ROS Wiki [Internet]. Wiki.ros.org. [cited 8 January 2021]. Available from: http://wiki.ros.org/Nodes

7 Appendix

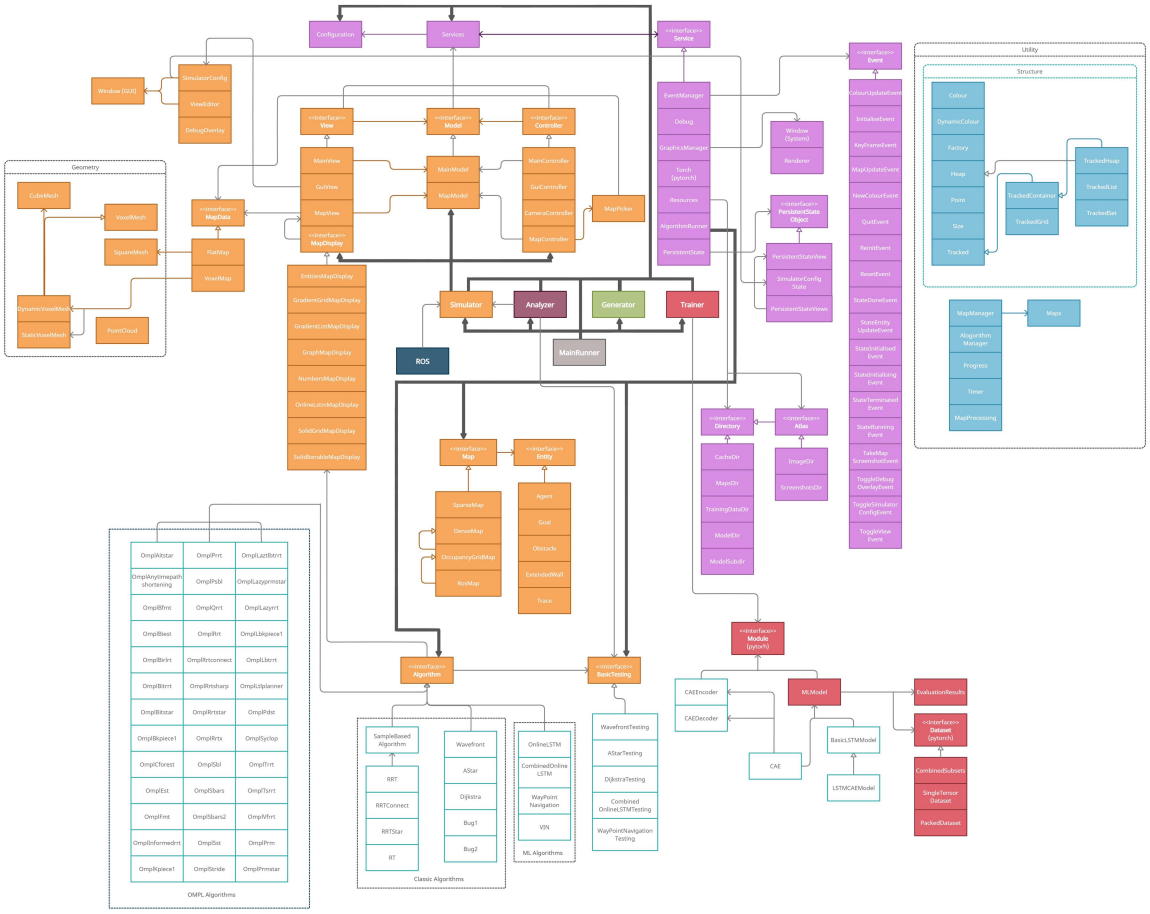


Figure 11: PathBench3D platform architecture.

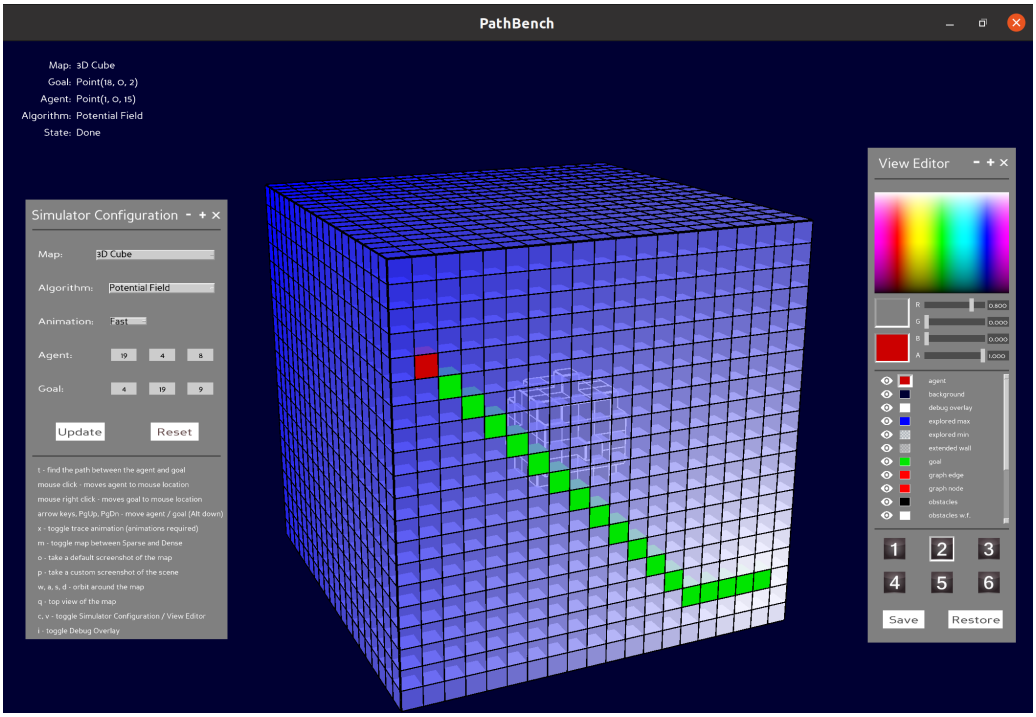


Figure 12: PathBench3D simulation of Wave-Front algorithm ran on a 3D Cube map

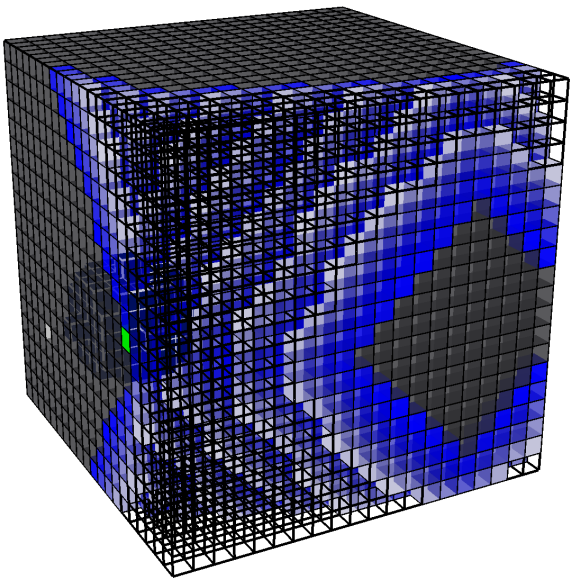


Figure 13: Dijkstra algorithm ran on a 3D cube map

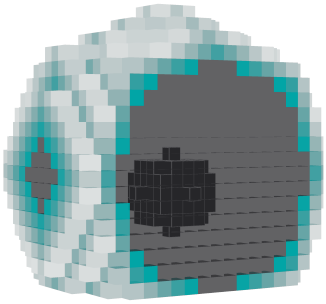


Figure 14: Dijkstra algorithm ran on a 3D cube map without a wireframe for better visualisation

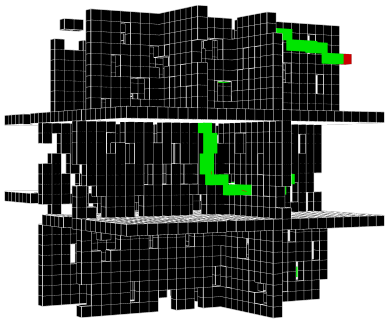


Figure 15: A* algorithm ran on a 3D house map

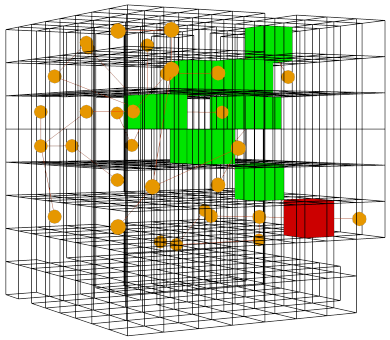


Figure 16: RRT algorithm ran on a 3D Uniform Random Fill map

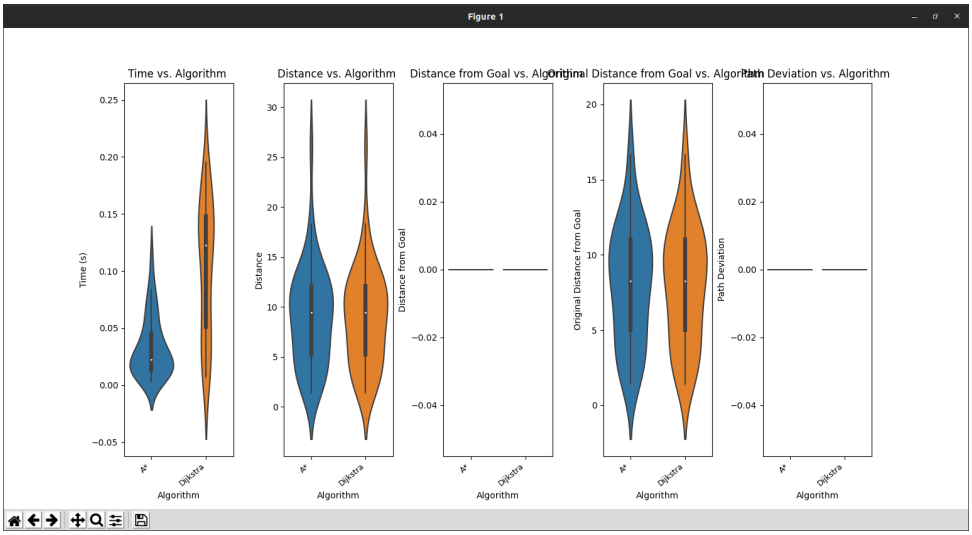


Figure 17: A* vs Dijkstra in the analyzer

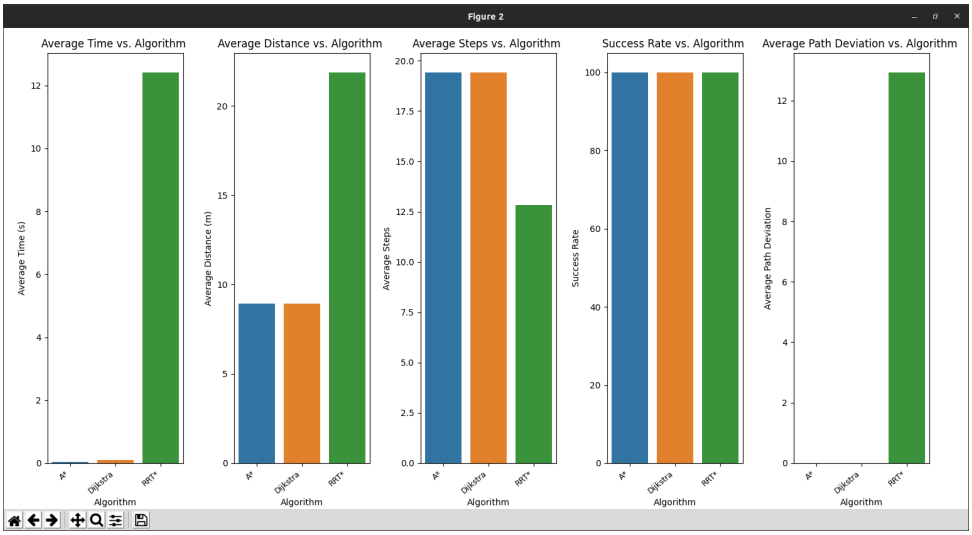


Figure 18: A*, Dijkstra and RRT* in the second plot of the analyzer

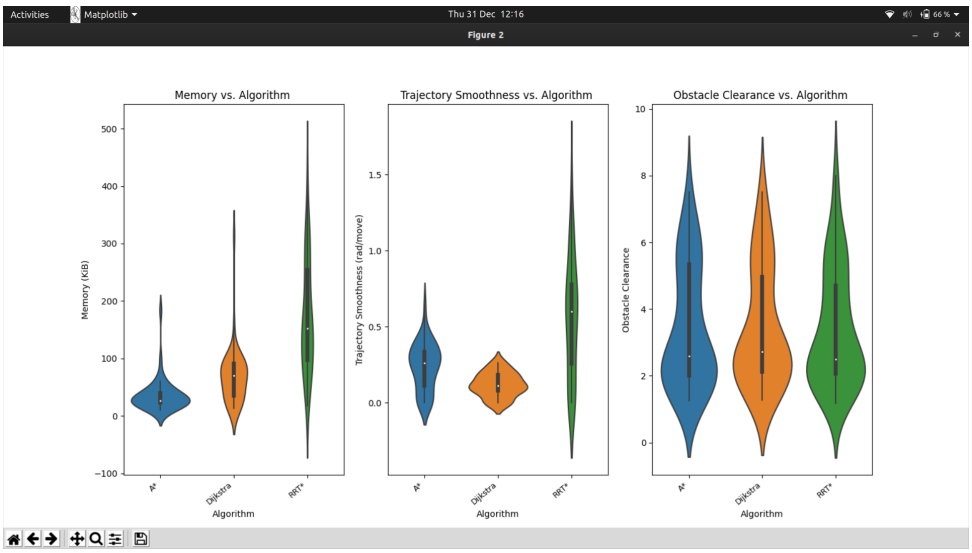


Figure 19: A plot showing the new metrics that we added.

```
  Algorithm  Average Path Deviation  Success Rate  Average Time  ...  Average Trajectory Smoothness  Average Obstacle Clearance  Average Search Space  Average Memory
0        A*                    0.00         100.0        0.0331  ...                           0.10                        3.56                 17.21           37.33
1  Dijkstra                    0.00         100.0        0.1032  ...                           0.05                        3.58                 49.65           71.09
2        A*                    0.00         100.0        0.0325  ...                           0.10                        3.56                 17.21           37.33
3  Dijkstra                    0.00         100.0        0.0963  ...                           0.05                        3.58                 49.65           71.09
4        A*                    0.00         100.0        0.0314  ...                           0.10                        3.56                 17.21           37.33
5  Dijkstra                    0.00         100.0        0.1011  ...                           0.05                        3.58                 49.65           71.09
6      RRT*                   12.93         100.0       12.4181  ...                           0.43                        3.63                   NaN          180.06

[7 rows x 12 columns]
********************************
    Algorithm        Time   Distance  Distance from Goal  Path Deviation  Original Distance from Goal  Trajectory Smoothness  Obstacle Clearance  Search Space    Memory
0          A*    0.020161   8.656854                 0.0        0.000000                     8.062258               0.078540            2.965028     17.968750   110.850
1          A*    0.009308   3.414214                 0.0        0.000000                     3.162278               0.130900            7.000000      6.640625    28.686
2          A*    0.114463  18.313708                 0.0        0.000000                    16.401220               0.087266            2.552158     46.875000   172.366
3          A*    0.007496   2.828427                 0.0        0.000000                     2.828427               0.000000            5.600000      5.468750    16.542
4          A*    0.014031   3.828427                 0.0        0.000000                     3.605551               0.130900            2.441714      8.203125    16.118
..        ...         ...        ...                 ...             ...                          ...                    ...                 ...           ...         ...
310      RRT*  22.805385   4.236068                 0.0       -1.821854                     2.236068               0.057956            4.671686           NaN   303.291
311      RRT*  31.019788  26.515075                 0.0      -15.615581                    10.630146               0.330440            6.354351           NaN   319.075
312      RRT*   0.497169  36.660491                 0.0      -20.003637                    12.206555               0.593981            1.978514           NaN    46.391
313      RRT*   2.626821  27.273477                 0.0      -17.030836                     9.486833               0.359527            3.200000           NaN    87.119
314      RRT*   0.031654  26.294183                 0.0      -16.465755                     9.219544               0.240349            2.534031           NaN    14.839

[315 rows x 10 columns]
```

Figure 20: A section of the raw data generated by the analyzer, in a pandas dataframe.

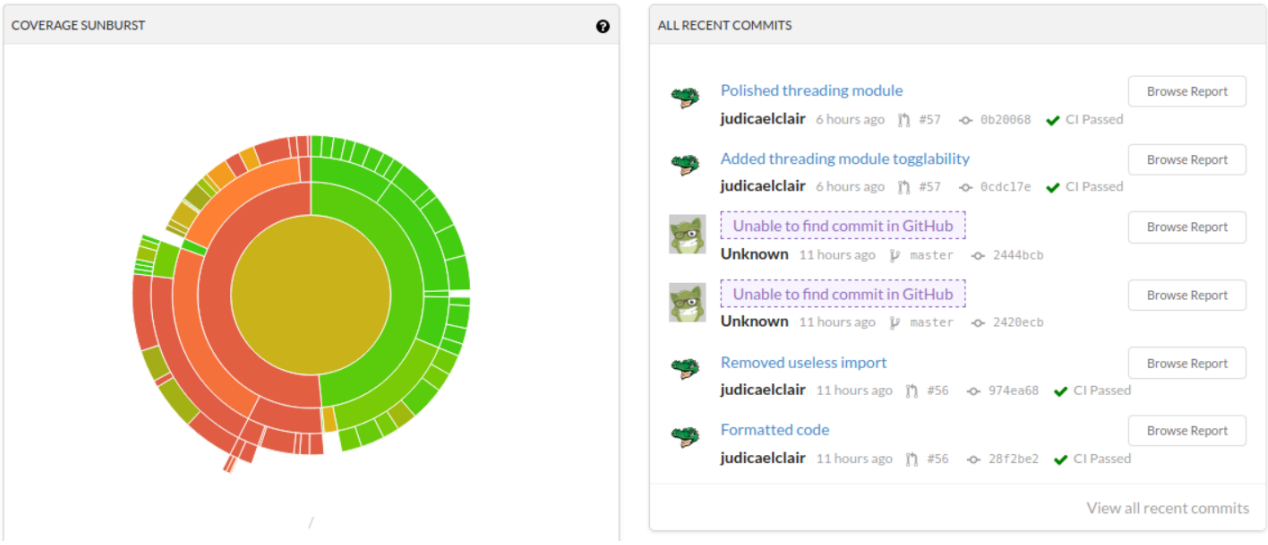


Figure 21: A screenshot of the site providing code coverage.

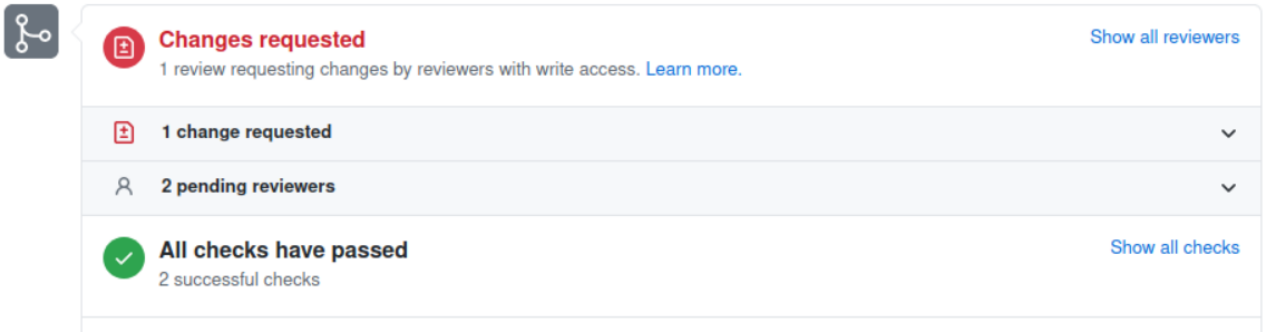


Figure 22: A screenshot of GitHub tests running on a PR

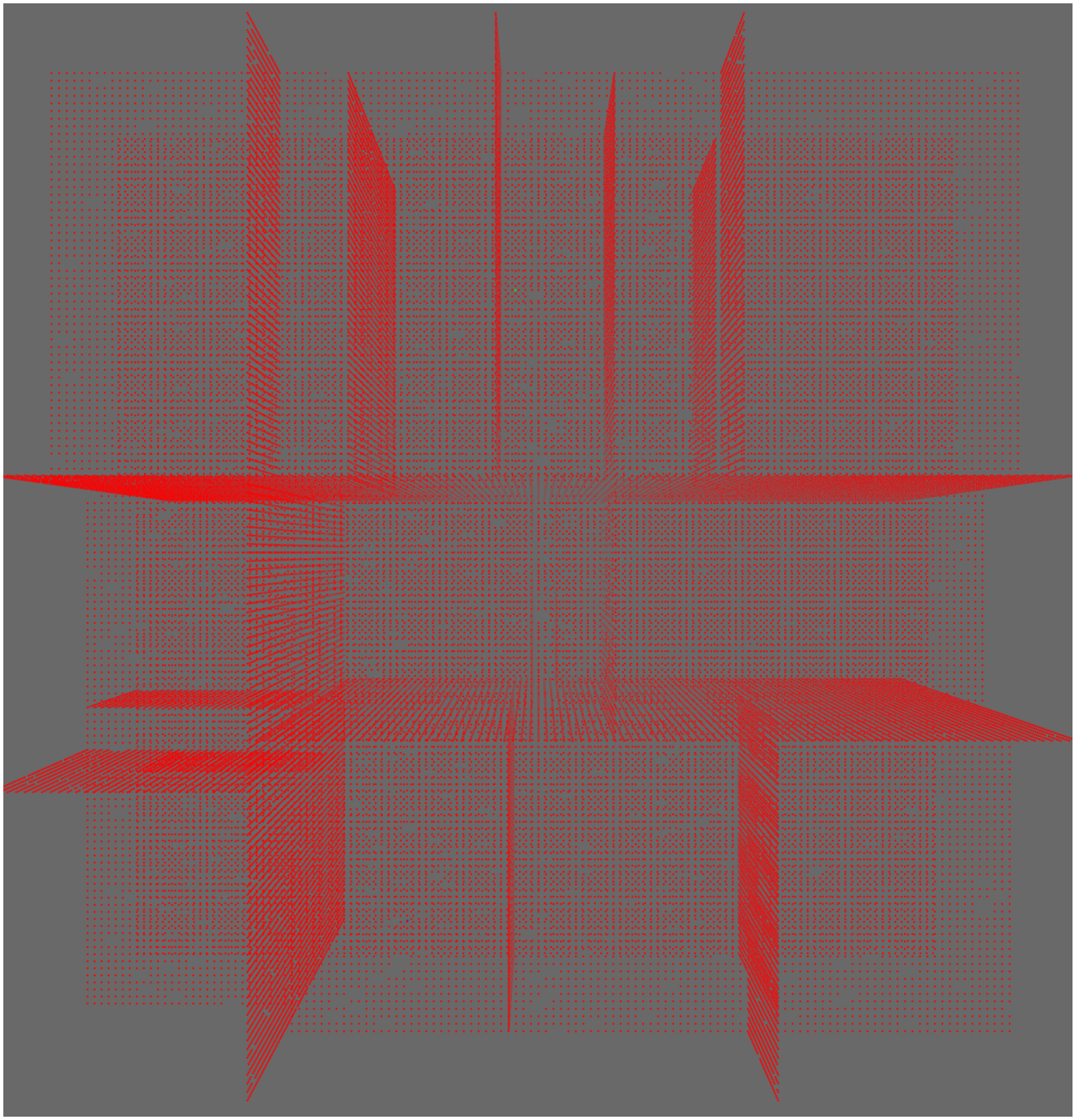


Figure 23: A screenshot of a pointcloud representation of a house map.

Specification of the system used for performance tests

- Tests were performed in a Virtual Machine (VM).
- Ubuntu 18.04.5 LTS
- Memory: 4.8 GiB
- Processor: Intel® Core™ i7-8500Y CPU @ 1.50GHz
- Graphics: llvmpipe (LLVM 10.0.0, 256 bits)
- Gnome: 3.28.2
- OS type: 64-bit
- Virtualisation: Oracle
- Disk: 210.3 GB