Optimisation of pathfinding in a dynamic 3D space

Master Thesis

for attainment of the academic degree of M.A.

Carina Krafft

2269579

University of Applied Sciences Hamburg
Faculty of Design, Media and Information
Time-Dependent Media / Sound – Vision – Games

First Examiner: Prof. Dr.-Ing. Sabine Schumann

Second Examiner: Prof. Dr. Larissa Putzar

Hamburg, 08.12.2021

Thema der Arbeit

Optimierung von Pfadfindung im dynamischen 3D-Raum

Schlüsselwörter

Pfadfinde Algorithmen, dynamischer 3D-Raum, any-angle Pfadfinde Algorithmen, dynamische Pfadfinde Algorithmen, Any-angle Moving Target D* Lite, Suchgraphen, Job System, Burst Compiler

Kurzzusammenfassung

Diese Masterarbeit konzentriert sich auf die Implementierung eines optimierten Pfadfinde-Systems, welches einen KI Agenten sicher durch eine sehr dynamische 3-dimensionale Umgebung navigieren kann. Hierfür werden verschiedene Pfadfinde Algorithmen implementiert und, unter anderem, mithilfe von Unity's C# Job System und Burst Compiler optimiert. Darüber hinaus wird ein neuer hybrider Suchgraph vorgestellt und ein Pfadfinde Algorithmus vorgeschlagen, welcher Theta* und MT-D* Lite kombiniert.

Topic of the master thesis

Optimisation of pathfinding in a dynamic 3D space

Keywords

Pathfinding, dynamic 3D-space, any-angle pathfinding algorithms, dynamic pathfinding algorithms, Any-angle Moving Target D* Lite, search graphs, job system, Burst compiler

Abstract

This master thesis focuses on implementing an optimised pathfinding system to navigate an AI agent through a highly dynamic three-dimensional environment. For this purpose different pathfinding algorithms are implemented and optimised, for example by using Unity's C# Job System and Burst compiler. Additionally, a new hybrid search graph is introduced, and a pathfinding algorithm combining Theta* with MT-D* Lite is proposed.

Contents

List of Figures

Listings

1 Introduction

1.1 Motivation

When using a pathfinding system in a highly dynamic three-dimensional environment, for example in a game, one is faced with a number of challenges. For example the AI agent has to move through the environment from a start to an end position smoothly on a short and natural looking path without colliding with any static or moving obstacles. To master these challenges the used pathfinding system has to meet certain requirements. The main requirement certainly is to produce short paths from the given start to goal point. Additionally, to avoid any collisions in the highly dynamic environment, the pathfinding system has to update its found path very frequently and fast to account for the changes in the environment and provide alternative free paths for the AI agent to use.

In 2019 the author of this thesis started working with pathfinding systems and the question of how to deal with the special requirements of highly dynamic 3D environments in her bachelor thesis with the title "Implementation and comparison of pathfinding algorithms in a dynamic 3D space" [Kra19]. The aim of this bachelor thesis was to implement and compare a selection of different pathfinding algorithms to identify which one is best used in a dynamic three-dimensional environment, especially in games. For this, not only different pathfinding algorithms were implemented and their performances analysed, but also two different search graphs were evaluated. The results of the bachelor thesis were informative, but the implementation of the pathfinding algorithm and the used search graphs were not performant enough to realistically be used in a game. Even though short paths were found, the long computation times of the pathfinding algorithms and the even longer update time of the search graphs caused the travelling AI agent to collide with moving obstacles too often.

Therefore, the motivation for this master thesis is to extend and improve upon the results of the previous bachelor thesis, to develop and implement a pathfinding system which can meet all the requirements to run smoothly and successfully in a dynamic 3D environment, such as a game.

1.2 Related work

As the previous bachelor thesis was already focused on pathfinding in dynamic 3D space especially in games, it presented some at that point used applications, practices, and tools for implementing pathfinding in games in chapter 2.3 [Kra19]. Since then not much has changed in this area. For the Unity game engine the Navigation and Pathfinding system using a NavMesh,

a NavMesh Agent, etc. is still the standard and its features have not changed significantly [Tec21h]. Also, the "Pathfinder 3D" package presented in chapter 2.3.3 in [Kra19] has not changed much. Even though it was updated to version 0.4.0 in 2019, it still does not support dynamic obstacles [Gra19].

Also in the context of games, but not related to the movement of players or NPCs, the paper "G-SpAR: GPU-Based Voxel Graph Pathfinding for Spatial Audio Rendering in Games and VR" by Mirza Beig, Bill Kapralos, Karen Collins, and Pejman Mirza-Babaei uses pathfinding for spacial audio in a 3D space in the game engine Unity [BK+19]. This work is interesting for two main reasons. First, they use pathfinding for a very different use-case than most other applications, namely for audio instead of movement. Secondly, unlike other pathfinding systems, their implementation is GPU-based instead of CPU-based.

As well as using the GPU for faster computation, other pathfinding implementations utilize the Data Oriented Technology Stack (DOTS) to achieve very fast runtimes of their algorithms in Unity. For example in a blog post the developer Jannik Staub explains his usage of the Entity Component System (ECS) and the Burst compiler in a pathfinding system [Sta21]. Another example was published on GitHub by the user Dave Anderson and shows the use of ECS Jobs and the Burst compiler for a 2-dimensional pathfinding application [And19].

Apart from optimising the runtime of well known algorithms by using the GPU or an improved compiler, another approach is to use or develop algorithms, which reduce or eliminate time consuming aspects of common algorithms. One good example for this is the pathfinding algorithm "Anya" developed by Daniel Harabor, Alban Grastien, Dindar Öz, and Vural Aksakalli. The algorithm is described in their paper "Optimal Any-Angle Pathfinding In Practice" [HG+16]. "Anya" is an any-angle pathfinding algorithm, which eliminates the use of the expensive line-of-sight checks required by other any-angle algorithms such as Theta*.

Lastly, it should be mentioned that pathfinding, and its optimisation for realistic use-cases, is currently more often found in the field of robotics than games. One interesting work of pathfinding in 3D for quatrotors is the paper "Planning Dynamically Feasible Trajectories for Quadrotors using Safe Flight Corridors in 3-D Complex Environments" by Sikang Liu, Michael Watterson, Kartik Mohta, Ke Sun, Subhrajit Bhattacharya, Camillo J. Taylor, and Vijay Kumar [LW+17]. Even though not all aspects and approaches of this paper can be transferred from robotics to games, especially the use of Jump Point Search [HG11] for path planning and the construction and usage of the Safe Flight Corridors could potentially be used as inspiration for pathfinding systems in 3D games.

1.3 Thesis goal

The goal of this thesis is to implement a pathfinding system which produces good results and therefore is realistically usable in highly dynamic three-dimensional environments, for example in a game. The developed and implemented system has to enable an AI agent to traverse the dynamic environment from a start to a goal position safely, without colliding with any obstacles.

A pathfinding system consists of a pathfinding algorithm along with a suitable search graph. The search graph has to be able to update and adapt to the changing environment quickly, and the combined system of pathfinding algorithm and search graph has to produce short paths and allow the algorithm to compute new paths fast. To be more precise, the pathfinding system should be able to react to changes in the environment by fully updating its found path within 16 ms. This equals the frame duration in a game with 60 frames per second (FPS). Hereby it is important for the pathfinding system to update asynchronously, for example on a background thread, and not block the main thread.

1.4 Structure

Before implementing a performant pathfinding system, chapter 2 first provides an overview over relevant theoretical knowledge required for different steps of the development process.

Afterwards, chapter 3 examines the current performance of the pathfinding algorithms and search graphs based on their implementation in the previous bachelor thesis [Kra19] (3.1) and identifies their main performance issues (3.2).

Based on the results of this performance analysis a concept for optimising the pathfinding system is introduced in chapter 4, which is then implemented in chapter 5. Here at first the pathfinding algorithms JPS, Theta*, and (Basic) MT-D* Lite are implemented and optimised (5.1), then a new search graph is introduced and implemented in chapter 5.2. Next, a new pathfinding algorithm AAMT-D* Lite combining Theta* and MT-D* Lite is suggested, implemented, and tested in chapter 5.3. Lastly, chapter 5.4 details the final pathfinding update loop and the success rate of the developed pathfinding system.

Having implemented and optimised all pathfinding algorithms and search graphs, chapter 6 concludes this thesis. After a brief summary (6.1), conclusions are drawn from the developed pathfinding system (6.2), and possible further optimisations and extensions to the pathfinding system are discussed (6.3).

The thesis' development is done using the Unity game engine version 2020.3.3f1 (LTS) and it is run on a Predator Helios 300 Laptop by Acer, with a Nvidia GeForce GTX 1060 graphics card and an intel core i7 processor.

2 Foundations

The following chapter will lay a foundation of theoretical knowledge necessary for implementing the optimisations of the pathfinding algorithms and the search graph. It will start by briefly explaining the basics of Unity's C# Job System and Burst compiler. Next the pathfinding algorithms Jump Point Search and (Basic) Moving Target D* Lite are presented. Lastly, the concept of the Separating Axis Theorem is explained, which is required to check for collision between obstacles and nodes as part of the search graph's updating process.

As this thesis is based on the previous bachelor thesis "Implementation and comparison of pathfinding algorithms in a dynamic 3D space" [Kra19], it is assumed that all foundations and concepts of the pathfinding algorithms and search graphs used in that bachelor thesis are known to the reader. Additionally, a working knowledge of the game engine Unity is presupposed.

2.1 Unity's C# Job System and Burst compiler

Unity's C# Job System and Burst compiler are both packages, which are part of the fairly new Data-Oriented Technology Stack (DOTS) [Tec21a]. Please note that the following chapter is only a brief overview of the basics of these two packages with relevant information for this thesis. There are many more aspects to both systems, which are not covered in this chapter.

The **C# Job System** was first released in Unity version 2018.1 as a Preview package. In Unity version 2020.3.3 LTS, which is used for this thesis, the newest job system package is available in the version 0.11.0-preview.6 (as of October 3rd 2021). According to the Unity documentation, the job system provides a way to write "simple and safe multithreaded code" [Tec21c]. Hereby an important aspect is that the C# Job System integrates with Unity's internal native job system, so that Unity and user-written code share worker threads [Tec21d]. Mostly the job system has one worker thread per logical CPU core. Jobs (small units of work, specified for one task) are scheduled in a job queue from which the worker threads take items and execute them. Unless dependencies between jobs are defined, the order of execution of jobs in the queue is not defined [Tec21l]. One main issue in regard to writing multi-threaded code are race conditions. These can, for example, occur when multiple threads read and write to the same reference in an indeterministic order. To solve this issue Unity's C# Job System does not send references to data into jobs, but passes each job a copy of the required data. Due to the way data is copied and send into the job, only blittable data types can be accessed within jobs [Tec21k]. This copying of data and, thereby, isolation of the job was put in place by Unity as a safety system to make working with threads safer. As a result, it avoids race conditions, but on the

other hand prevents the return of results from the job back to the main thread. The solution to this issue is the use of managed value type called "NativeContainer" [Tec21g]. A NativeContainer is a wrapper for native memory containing a pointer to an unmanaged allocation, which allows accessing of shared data between the main thread and the job. NativeContainers have to be created with a specific memory allocation type and manually disposed after usage. As the use of NativeContainers bypasses the above mentioned safety systems of only allowing the use of copied data within jobs, Unity also built a safety system into all NativeContainers, which can be used in the Editor and Play mode. Part of this is the detection of memory leaks and race conditions caused by NativeContainers. Examples of commonly used NativeContainers in this thesis are the NativeArray and NativeList, which can be used in a similar fashion to a normal array or list, as well as the NativeHashSet and the NativeMultiHashMap. [Tec21g]

To create a job, one has to create a struct inheriting from an appropriate "job" interface such as *IJob* [Tec21f] or *IJobParallelFor* [Tec21i]. There are more job types available, but as only these two types are used in this thesis, this chapter will only focus on these two. Each job has to implement an *Execute* function, which is called when the job is executed by the worker thread. For structs inheriting from IJob, referred to as "jobs", the *Execute* function is called once on one core [Tec21f]. Structs inheriting the IParallelFor interface, also called "ParallelFor jobs", execute the *Execute* function multiple times running across multiple cores. Hereby a NativeArray acts as its data source and *Execute* is invoked once for each element in the NativeArray. To the *Execute* function in the ParallelFor job an integer parameter is passed, which is used as an index to access entries of the data source [Tec21i]. To show the difference between these two job types more clearly, figure 1 shows the profiler analysing the part of a frame, which updates the hybrid search graph (see chapter 5.2.2). The *UpdateOverlapJob* is an IParallelFor job and its execution is framed by a red box in the image. It can be seen that the job's execution is spread across multiple worker threads running in parallel. The *UpdatePathCostsJob* job on the other hand is only executed on one thread. It is framed in purple in figure 1.

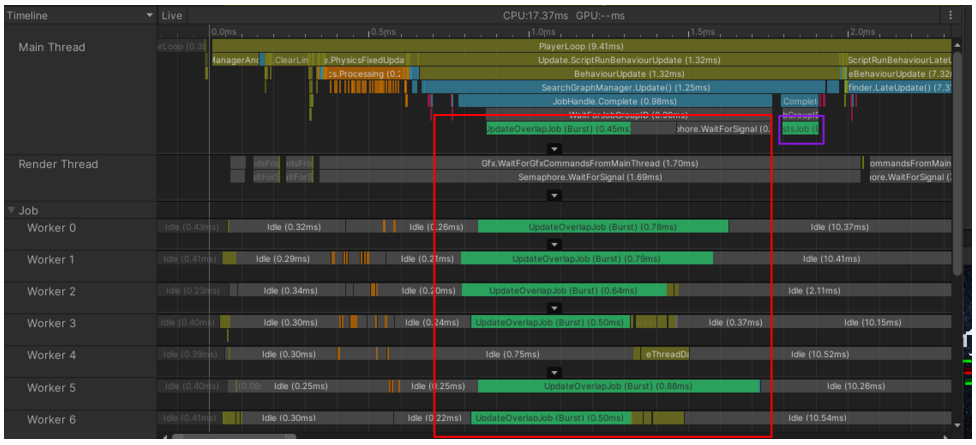


Figure 1: Profiler of updating the hybrid search graph. *UpdateOverlapJob* ParallelFor job framed in red, *UpdatePathCostsJob* job framed in purple.

After its creation a job can be added to the job queue using the *Schedule* method [Tec21j]. Scheduling a job does not instantly cause its execution. Calling *JobHandle.Complete* [Tec21y] starts the execution of a job and the *JobHandle.ScheduleBatchedJobs* function [Tec20] can be used to flush the batch of scheduled jobs. These functions can only be called from the main thread. It has to be ensured that a job is completed before it is allowed to access the NativeContainers used by the job on the main thread. [Tec21e]

To further improve the performance of jobs, they can be compiled using the **Burst compiler** [Tec21a]. The Burst compiler uses LLVM[1] to translate from IL/.NET bytecode to very optimised native machine code [Tec21b]. Explaining the exact workings of the Burst compiler would go into too much detail for this thesis. Therefore, this chapter will focus on giving a brief overview of relevant Burst specific information required for the implementation of the pathfinding algorithms and search graph in this thesis. To use the Burst compiler a job struct has to be decorated with the [BurstCompile] attribute. The safety restrictions of the job system already ensure that jobs can only contain blittable and native container data types. Burst limits the use of data types further to unmanaged types only. This includes, for example, the types bool, int, double, and float, but not char or string. For vectors one should use the Unity.Mathematics [Tec21z] package, which is optimised to be used with Burst. Burst only supports managed arrays if they are readonly and loaded from a static readonly field. Therefore, most arrays need to be replaced by NativeContainers like NativeArray. Aside from adapting the code to only use supported data types, most of the other code in the job is supported by Burst. The most significant exception to this is the use of try/finally, catch, and foreach-loops. These are not supported. [Tec21b]

Similar to the job system and NativeContainers, the Burst compiler also offers the option to enable safety checks in the Editor using the Job menu under Jobs->Burst->Safety Checks. Even though this helps preventing fatal errors during the development process, it also causes a noticeable performance overhead [Tec21b]. Therefore, this safety feature is disabled for all performance analyses and time measurements in this thesis.

To get started using the Burst compiler, the talk "Getting started with Burst" presented by Lee Hammerton at the Unity Copenhagen 2019 provides a good overview [Ham19]. For a full user guide of the Burst compiler see [Tec21b]. In that user guide a lot more details of the Burst compiler are described which are very specific and were not investigated further during this thesis. Next to considering the data type and language support limitations of Burst, one should also adjust the general approach of writing code for this compiler. For example in his talk "Intrinsics: Low-level engine development with Burst" at the Unity Copenhagen 2019 Andreas Fredriksson talks about not thinking of vectors as horizontal values, but rather use them vertically [Fre19a, Fre19b]. What this means exactly for the implementation of functions can be

[1] a compiler infrastructure [LLV21]

seen in chapter 5.1 at the example of the optimisation of pathfinding algorithms, especially the computation of rhs- and f-values. Generally speaking, one should keep in mind that working with the job system and Burst compiler the approach to writing code should be data oriented.

2.2 Jump Point Search

Jump Point Search (JPS) was first introduced by Daniel Harabor and Alban Grastien in 2011 in their paper "Online Graph Pruning for Pathfinding on Grid Maps" [HG11]. JPS is an approach that can be used to optimise the runtime of the A* pathfinding algorithm on a grid map by applying a set of pruning rules, which can reduce the number of expanded nodes. In their paper Harabor and Grastien use an undirected grid map with uniform path costs where each node has up to eight neighbours (on the horizontal and vertical axis, as well as diagonal). [HG11]

JPS applies two sets of rules to eliminate as many nodes as possible. The first set of rules are pruning rules applied to the neighbours of a node. Where A* simply takes all traversable neighbours and adds them to the open list with their according f-value, JPS prunes as many neighbours as possible and only adds the remaining ones to the open list. When moving straight forward on the horizontal or vertical axis all neighbours n' of a node n are pruned for which a path exists, which goes from $parent(n)$ to n' without passing through n and is shorter or has an equal length to a path that goes from $parent(n)$ to n' and passes through n. [HG11]

$$length(\langle parent(n), \dots, n' \rangle \backslash n) \leq length(\langle parent(n), n, n' \rangle)$$

An example of this rule can be seen in figure 2i. Here the parent of the currently expanded node n is the node with the number 4. Due to the thereby caused straight movement on the horizontal axis, all nodes except number 5 get pruned.

When moving in a diagonal direction the above rule needs to be adjusted only slightly, so that the path which excludes n is strictly dominant [HG11].

$$length(\langle parent(n), \dots, n' \rangle \backslash n) < length(\langle parent(n), n, n' \rangle)$$

The result of this rule can be seen in figure 2iii, where the parent of the currently expanded node n is node number 6. After pruning only the neighbours number 2, 3, and 5 remain.

All remaining neighbours after applying the above pruning rules are called *natural* neighbours [HG11] and are marked white in figure 2.

In addition to these *natural* neighbours JPS defines so-called *forced* neighbours of the currently expanded node n if any neighbour of n is blocked [HG11]. Hereby a *forced* neighbour is defined as a neighbour, which is not a *natural* neighbour and the following rule applies [HG11]:

$$length(\langle parent(n), n, n' \rangle) < length(\langle parent(n), \ldots, n' \rangle \backslash n)$$

Figure 2ii and 2iv show blocked nodes in black and therefrom resulting *forced* neighbours in red.

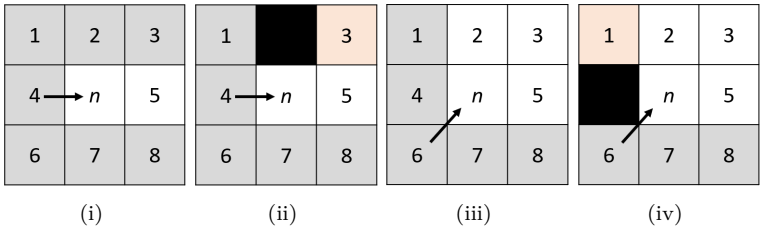


Figure 2: Neighbour pruning rules. The array indicates the direction from the parent of n to n. Grey nodes get pruned, white nodes represent *natural* neighbours of n, and red nodes show *forced* neighbours of n [HG11, 3].

The only exception to these pruning rules is the start node. The parent node of the start node is null and therefore none of its neighbours are pruned [HG11].

After pruning as many neighbours as possible, JPS applies its second set of rules, so-called jumping rules, to each of the remaining *natural* and *forced* neighbours of a node. Instead of evaluating the direct neighbour n' of a node n and adding it to the open list, an alternative successor n_{succ} (also called jump point) of n is searched for, which is further away [HG14]. To do so, JPS steps from n in the relative direction $\vec{d}$ from n to n' until either an obstacle is reached or a jump point is found. When an obstacle is reached, nothing is returned and no node is added to the open list. A jump point is found when a node is reached which is the goal node, has at least one forced neighbour, or if $\vec{d}$ is a diagonal move and on either isolated horizontal or vertical direction of $\vec{d}$ a jump point is found [HG11]. This node is then returned, replaces n' as a successor of n and is therefore added to the open list. Hereby the jump point's g-value is calculated as $g(n_{succ}) = g(n) + distance(n, n_{succ})$. [HG11]

Figure 3 shows two examples of finding jump points. In figure 3i JPS steps straight to the right from n until a jump point is found. n_{succ} is a jump point of n because the node z is a forced neighbour of n_{succ}. Therefore, n_{succ} is returned and added to the open list. Figure 3ii shows the more complex search for a jump point on a diagonal axis. When moving diagonally, not only is the next node checked for being blocked, being the goal node, or having forced neighbours, but also the horizontal and vertical axis have to be checked for possible jump points. In 3ii these additional checks along the horizontal and vertical axis are represented by dashed lines. n_{succ} is a jump point of n because checking along the horizontal axis the node z is found, which has a forced neighbour and is therefore a jump point. [HG11]

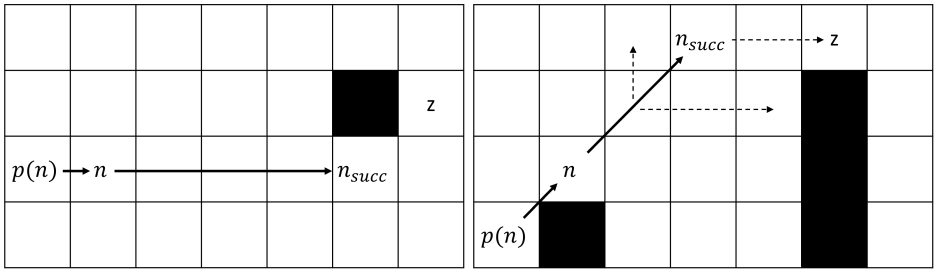


Figure 3: Finding jump points with horizontal and diagonal movement [HG11, 3]

In their paper Harabor and Grastien only discuss the usage of Jump Point Search on a two-dimensional search grid. It can also be used in three-dimensional cell-grid based search graphs, though, as described in the paper "Planning Dynamically Feasible Trajectories for Quadrotors using Safe Flight Corridors in 3-D Complex Environments" published in 2017 by Sikang Liu, Michael Watterson, Kartik Mohta, Ke Sun, Subhrajit Bhattacharya, Camillo J. Taylor, and Vijay Kumar [LW$^+$17]. To do so, only the pruning rules for natural and forced neighbours need to be adjusted slightly.

Figure 4 shows these adjustments for straight movement along the horizontal axis in a 3x3x3 cell-grid. Hereby each image shows one slice of the grid in the xz-plane. In the scenario in the top row all nodes are traversable. Here all neighbours except the one in the current moving direction are pruned. In the scenario in the second row two nodes are blocked. Here additional nodes are marked as forced neighbours. For simplicity only the case of straight movement is shown in this chapter, as this is the most relevant case used in this thesis. For details on neighbour pruning in 3D with diagonal movement see [LW$^+$17] page 2.

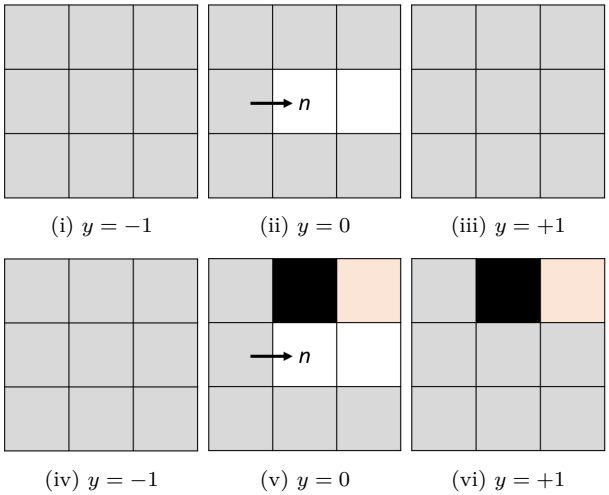


Figure 4: Neighbour pruning rules in three dimensions. Grey nodes get pruned, white nodes represent *natural* neighbours of n, and red nodes show *forced* neighbours of n. In the top row a scenario where all nodes are traversable; in the bottom row a scenario with blocked nodes. [LW$^+$17, 2]

2.3 (Basic) Moving Target D* Lite

Basic Moving Target D* Lite (Basic MT-D* Lite) and Moving Target D* Lite (MT-D* Lite) are two incremental heuristic search algorithms introduced in 2010 by Xiaoxun Sun, William Yeoh, and Sven Koenig [SYK10]. Henceforth the wording "(Basic) Moving Target D* Lite" or "(Basic) MT-D* Lite" is used when referring to both of the algorithms.

As the name suggests (Basic) Moving Target D* Lite is based on the dynamic pathfinding algorithm D* Lite [KL02] and addresses the issue of the algorithm having to reset each time the goal node changes, for example due to the destination moving. (Basic) MT-D* Lite calculates its shortest path and handles a changed environment following almost the same rules and patterns as D* Lite. One difference between the implementation of (Basic) MT-D* Lite as described in [SYK10] and the implementation of D* Lite introduced in [KL02] is the direction of the algorithm. In its classic implementation D* Lite is backwards oriented, meaning it starts at the goal node and propagates its information until the start node is reached. (Basic) MT-D* Lite in contrast starts at the start node and propagates its information towards the goal node. This does not change the underlying logic of the algorithm, but small changes have to be made to certain calculations. For example, the nodes' rhs-value is now based on the g-value of its predecessors instead of its successors and the heuristic of a node is relative to the goal node instead of the start node. The main issue to solve for the forward directed dynamic algorithm is how to account for a changing start node. The reason for reversing the algorithm's direction of (Basic) MT-D* Lite in comparison to D* Lite is discussed in chapter 5.1.4. [SYK10]

Apart from these minor adjustments regarding the algorithm's direction, the main difference between D* Lite and (Basic) MT-D* Lite lies in an additional step handling a changed goal node or start node, respectively. The implementation of this additional step is the only difference between Basic MT-D* Lite and MT-D* Lite.

2.3.1 Basic Moving Target D* Lite

The simpler variation of handling changing start nodes is Basic Moving Target D* Lite. Having found a first shortest path following the basic logic of D* Lite, Basic MT-D* Lite waits for path costs in the environment to change or for the start node to deviate from the shortest path from $n(start)$ to $n(goal)$. When one of these changes is detected, it updates its km-value to account for possible movement of the goal node, updates all nodes with changed path costs, just like D* Lite does, sets the parent node of the new start node to null, updates the rhs-value of the old start node, and inserts or removes it into/from the open list dependent on whether it is consistent or inconsistent (see figure 5). [SYK10]

Due to the fact that a node's rhs-value can be defined as

$$rhs(n) = \begin{cases} c, & \textbf{if } n = n_{\text{start}} \\ \min_{n' \in \text{Pred}(n)}(g(n') + c(n, n')), & \textbf{else} \end{cases}$$

with c being an arbitrary finite value, it is not necessary to update the old start node's rhs-value [SYK10].

Having updated all affected nodes and km, Basic MT-D* Lite runs its next iteration of calculating a shortest path based on the current open list without resetting.

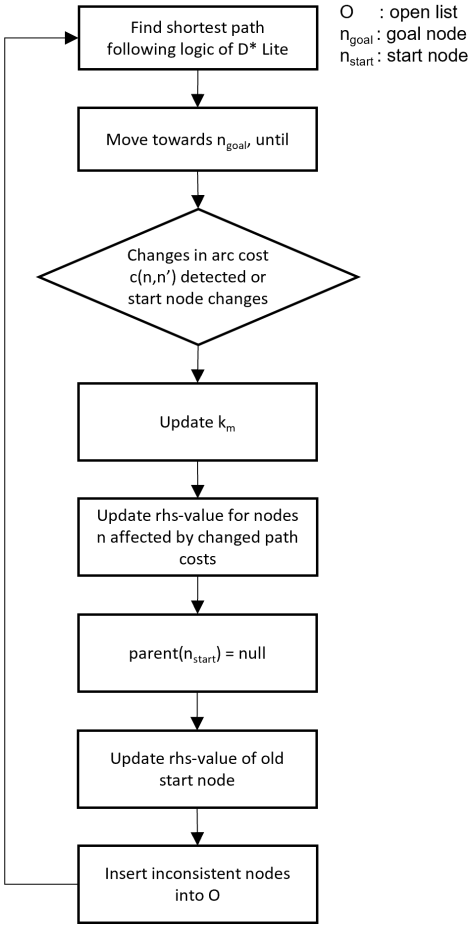


Figure 5: Structure of Basic MT-D* Lite [SYK10]

2.3.2 Moving Target D* Lite

The incremental search algorithm Moving Target D* Lite is an optimised version of Basic MT-D* Lite. It only differs from Basic MT-D* Lite in its handling of changed start nodes. Instead of simply updating the last start node, it uses a more sophisticated approach.

MT-D* Lite uses a so-called deleted list, which contains all nodes which are in the previous search tree, but are not included in the subtree rooted at the new start node. When detecting changed path costs or a new start node, which is not on the path from $n(start)$ to $n(goal)$, MT-D* Lite updates km and all nodes with changed path costs the same way that Basic MT-D* Lite does. It then sets the parent of the new start node to null and creates the current deleted list. In a first step the algorithm iterates over all nodes in the deleted list and sets their parents to null and their rhs- and g-values to infinite, and removes them from the open list. In a second step the algorithm iterates over the deleted list a second time and updates the rhs-values of all its nodes. Hereby all nodes that are inconsistent after having an updated rhs-value are added to the open list. [SYK10]

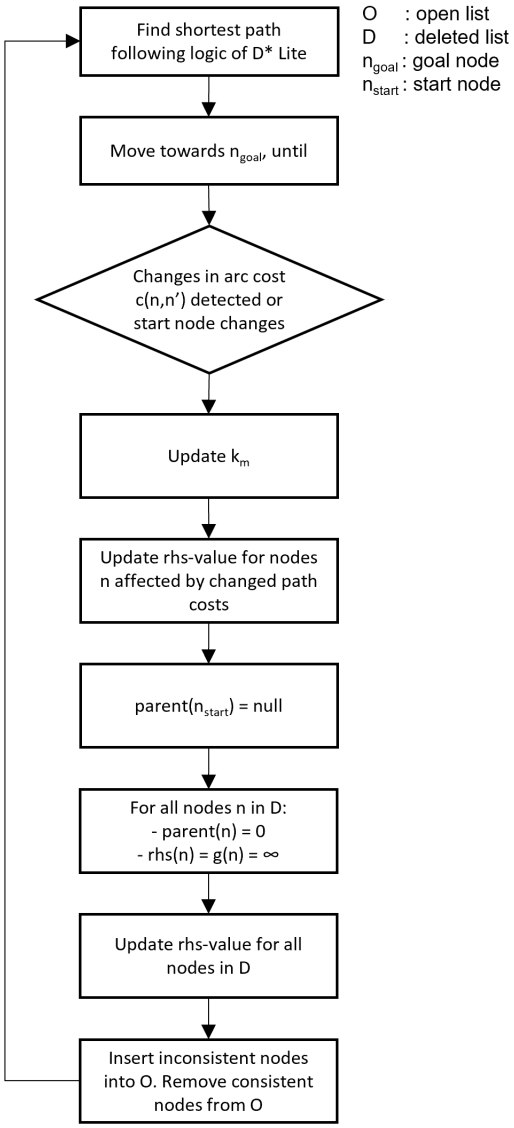


Figure 6: Structure of MT-D* Lite [SYK10]

2.4 Separating Axis Theorem

One part of updating any search graph is to check which of its nodes are blocked by obstacles and which are free, in order to set each node's path costs correctly. To do so, collision detection between two three-dimensional objects is required. One option to do this is to use the "Separating Axis Theorem", short SAT.

In a blog post about this matter William Bittle explains the core of the theorem as:

> "If two convex objects are not penetrating, there exists an axis for which the projection of the objects will not overlap." [Bit10]

This axis is called the "separating axis" and is shown as the dotted red arrow in figure 7i [Huy08, p. 3]. It can be seen that the projections of the objects A and B marked in purple and green on the axis do not overlap. Therefore, the shown axis is a separating axis. For each two convex objects only a limited number of possible separating axis exist, which have to be checked for overlapping projections. Taking a look at figure 7ii it can be seen that in this two-dimensional example a so-called "separating line" exists that separates the two objects A and B, and that is orthogonal to the separating axis. [Huy08]

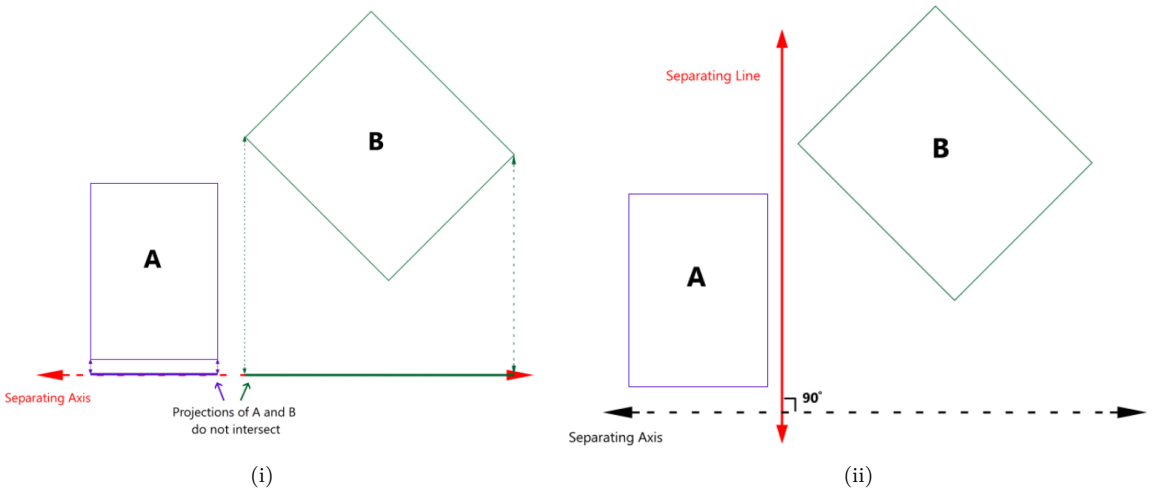


Figure 7: Separating Axis Theorem at a two dimensional example [Huy08, p. 3,4]

In 2D every possible separating line is parallel to at least one edge of one of the polygons [Huy08]. Therefore, all possible separating axis, are axis which are orthogonal to at least one of the objects edges. In other words, to check all separating axis, one has to check all normals of the polygon [Bit10]. Figure 8 shows all possible separating axis of the two triangles based on their normals.

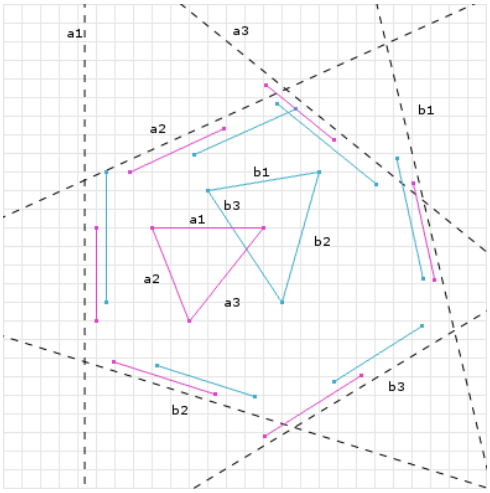
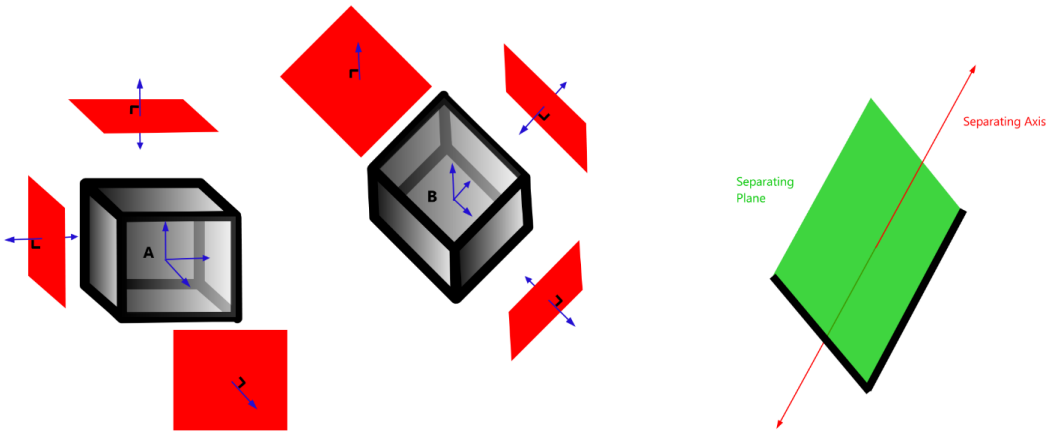


Figure 8: All possible separating axis based on the polygons' normals [Bit10]

Transferring this concept to three dimensions, two disjointed convex polyhedra are no longer separated by a separating line, but by a separating plane. This plane is either parallel to at least one face of one of the polyhedra or it contains an edge from each of the objects [Huy08, Ebe99]. All possible separating axis are orthogonal to one of these separating planes [Huy08]. To determine the axis orthogonal to the planes parallel to the faces of the polyhedra, one can use the planes' normals as shown in figure 9i. Here the blue arrows indicate the possible separating axis. To determine the axis orthogonal to planes which contain an edge from each polyhedra, the cross product of these two edges can be computed. This cross product returns a vector orthogonal to the plane spanned by the two edges. An example of this can be seen in figure 9ii where the thick black lines show two edges spanning up the green separating plane and therefore determining the red separating axis orthogonal to the plane. [Huy08]



(i) Separating planes parallel to the objects' faces and the corresponding separation axis [Huy08, p. 20].

(ii) Separation plane spanned by two edges and its corresponding separation axis [Huy08, p. 24].

Figure 9: Separating axis orthogonal to separating planes

This concept is not restricted to three dimensions, but can be defined more generic. Van den Bergen defines a separating axis as follows in his book "Collision Detection in Interactive 3D Environments":

> "For a pair of nonintersection polytopes, there exists a separating axis that is orthogonal to a facet of either polytope, or orthogonal to an edge from each polytope." [dv03, p. 78]

Hereby he defines a "polytope" to be a convex object [dv03, p. 23] and "edges" and "facets" to be features (also known as faces) of a polytope of one and two dimensions respectively [dv03, p. 23]. Moreover, in his book he uses the abbreviation "SAT" for "separating-axis test" [dv03, p. 78].

Once all possible separating axis are defined, both objects are projected onto all axis and it is determined whether their projections overlap. To project an object onto an axis the dot product of each of its vertices and the axis is computed and the minimum and maximum value determined [Bit10]. This approach can be used regardless of whether the objects are two- or three-dimensional. A coding example of this is shown in listing 1. Once one axis is found for which the objects' projections do not overlap, the algorithm can be interrupted and return false [Bit10]. In this case the objects do not intersect. A coding solution for determining the overlap of two objects' projections can be seen in listing 2.

```
1  private static void Project(float3 axis, NativeList<float3> vertices, out float
       projectionMin, out float projectionMax)
2  {
3      projectionMin = float.MaxValue;
4      projectionMax = float.MinValue;
5      for (int j = 0; j < vertices.Length; j++)
6      {
7          float projection = math.dot(axis, vertices[j]);
8          projectionMin = math.min(projectionMin, projection);
9          projectionMax = math.max(projectionMax, projection);
10     }
11 }
```

Listing 1: A possible *Project* function determining the projection of an obstacle onto an axis

```
1  private static bool ProjectionsDisjoint(float minA, float maxA, float minB,
       float maxB)
2  {
3      return (minA < minB && maxA < minB) || (minB < minA && maxB < minA);
4  }
```

Listing 2: Pseudocode for checking if two projections overlap. The function returns true if the projections do not overlap.

3 Analysis of the current performance

As already mentioned in the introduction, this master thesis is based on the work of the bachelor thesis "Implementation and comparison of pathfinding algorithms in a dynamic 3D space" [Kra19]. Therefore, the search graphs and a selection of pathfinding algorithms described and implemented in that bachelor thesis will be the foundation for optimising pathfinding in dynamic 3D space in this master thesis. Before starting to optimise the pathfinding algorithms and search graphs, in this chapter their current status and performance is analysed and performance critical issues identified. If not stated otherwise, all distance measurements in this thesis are indicated in Unity units (Uu) and all time is indicated in milliseconds (ms).

3.1 Current performance

In a first step a new project was created in the game engine Unity using version 2020.3.3f1 (LTS). All relevant data including code, scenes, prefabs, etc. were taken from the above mentioned bachelor thesis [Kra19] to replicate the same test environment and implementation of pathfinding algorithms and search graphs as in the previous thesis. Both, the waypoint based and the cell grid search graph were implemented, as well as the algorithms A*, Theta*, and D* Lite. These pathfinding algorithms were chosen because the bachelor thesis proved these to be the most promising ones. During this first implementation step only minor changes were made to the code such as adapting to naming conventions and restructuring namespaces. Thesviele changes did not have an impact on the algorithms' and search graphs' performance.

Figure 10: Test environment with dynamic object (coloured), static objects (white), dynamic destination (orange sphere bottom left), and AI agent (green sphere top right)

3.1.1 Pathfinding algorithms

Having set up the project, the current performance of the algorithms is measured. Hereby, the most relevant performance factors are the algorithms' path length, number of expanded nodes, and computation time. The shorter the path length, the fewer nodes get expanded, and the lower the computation time, the better is the algorithm's performance. Additionally, the relation between the number of expanded nodes and the computation time can be an indicator of whether the time to expand one node has to be optimised, or whether reducing the number of expanded nodes should be focussed upon.

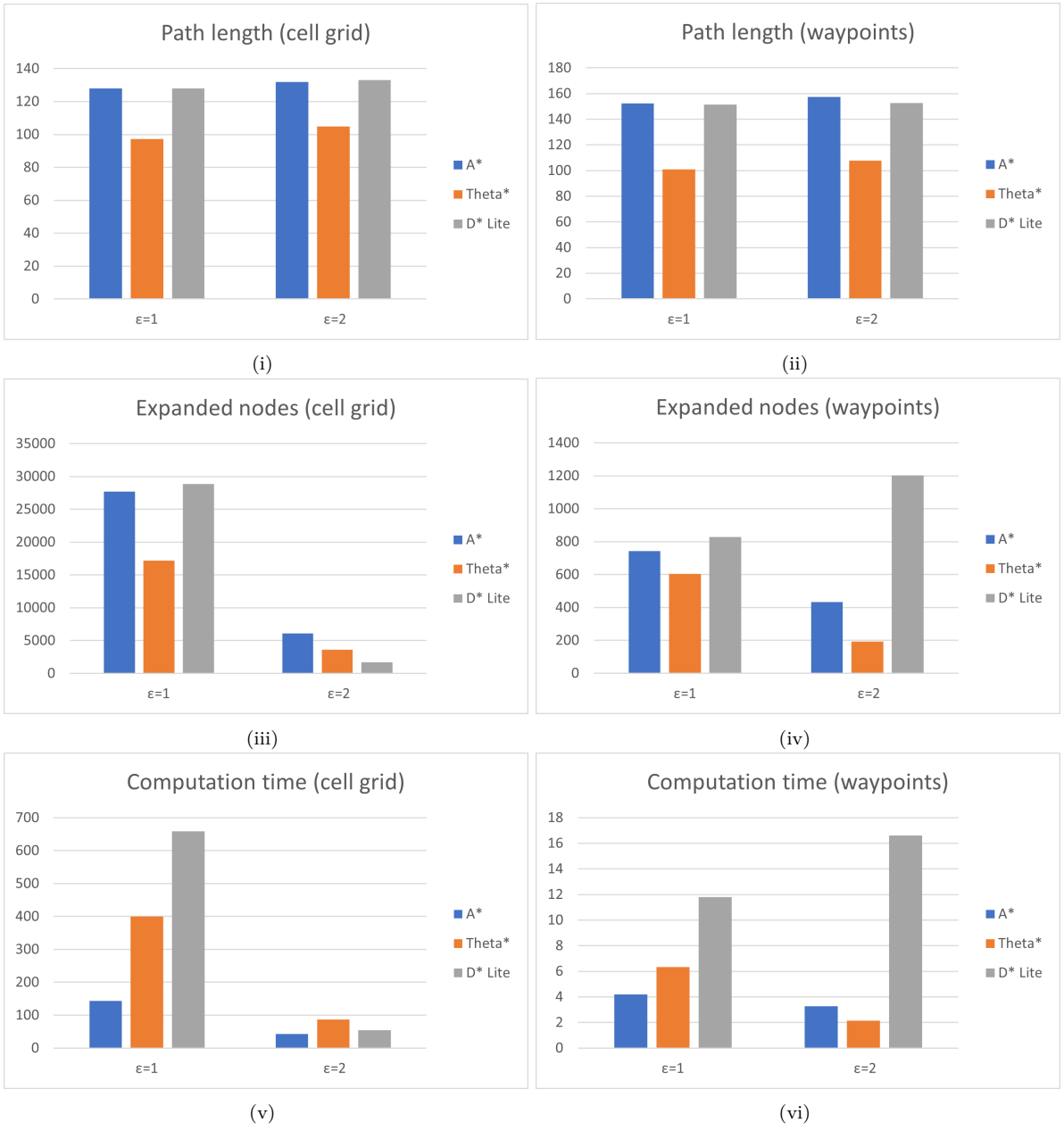


Figure 11: Comparison of path length (in Uu), expanded nodes, and computation time (in ms) in a cell-grid and waypoint-based search graph with different inflation factors ε

Figure 11 shows the average path length, number of expanded nodes, and computation time of A*, Theta*, and D* Lite in the cell grid and the waypoint based search graph. For these measurements the AI agent was made static, so the distance between the start and end point of the path is not influenced by the agent's traversal on the path.

The results of these measurements are as to be expected and comparable to the results of the bachelor thesis. The average path lengths are slightly longer in the waypoint based search graph than in the cell grid graph. Moreover, in both graphs Theta* results in significantly shorter paths than A* and D* Lite. The number of expanded nodes and the average computation time is overall significantly lower in the waypoint based graph than the cell grid graph. In both graphs Theta* expands fewer nodes than A*, whereas D* Lite in most cases expands more nodes than A*. The only exception to this is in the cell grid search graph when using an inflation factor of $\varepsilon = 2$. One reason for the high number of expanded nodes of D* Lite is the dynamic destination in this scenario. Each time the goal node changes, the dynamic algorithm has to be reset and can not reuse previously computed data to update its found path. Additionally, the algorithm has to reset each time any node changes its neighbours, as is the case with indirect neighbours in the waypoint based search graph. Looking at the average computation time of the pathfinding algorithms it can be said that D* Lite has the highest runtime. Even though it has a very similar number of expanded nodes to A* when using $\varepsilon = 1$, it has a significantly higher computation time. Also, in most cases Theta* has a higher computation time than A*, even though it expands fewer nodes than A*. The relation between number of expanded nodes and computation time of the algorithms is further investigated in figure 12 using a cell grid search graph and an inflation factor of $\varepsilon = 1$. Not considering some inaccuracies in the measured data, it can be seen quite clearly that with the same number of expanded nodes, A* is more performant than Theta* and D* Lite, which is conform with the results seen in figure 11. Additionally, it can be said that Theta* and D* Lite are similarly performant given the same number of expanded nodes. The difference in computation time in figure 11 can be explained by the fact that D* Lite more often expands more nodes than Theta* and, therefore, has a higher average number of expanded nodes resulting in a higher average computation time.

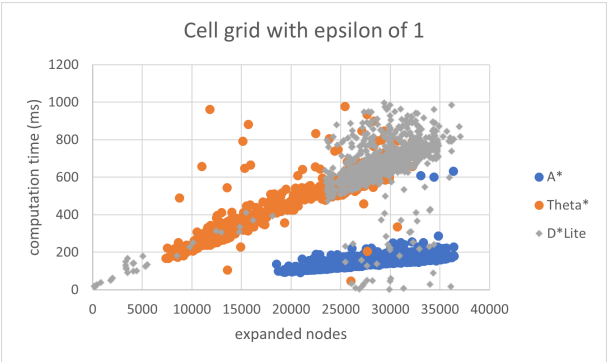


Figure 12: Expanded nodes relative to the computation time

3.1.2 Search graphs

Next the current performance of the search graphs is measured. Figure 13 shows the average update duration of the waypoint based and cell grid search graph. These results are also comparable to the data measured during the previous bachelor thesis. Even though the cell grid search graph has about 27 times more nodes than the waypoint based search graph (57344 vs. 2062) it has a significantly lower average update duration.

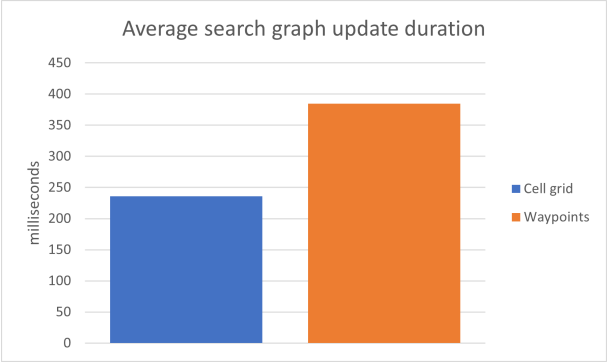


Figure 13: Average search graph update duration

3.2 Identification of performance issues

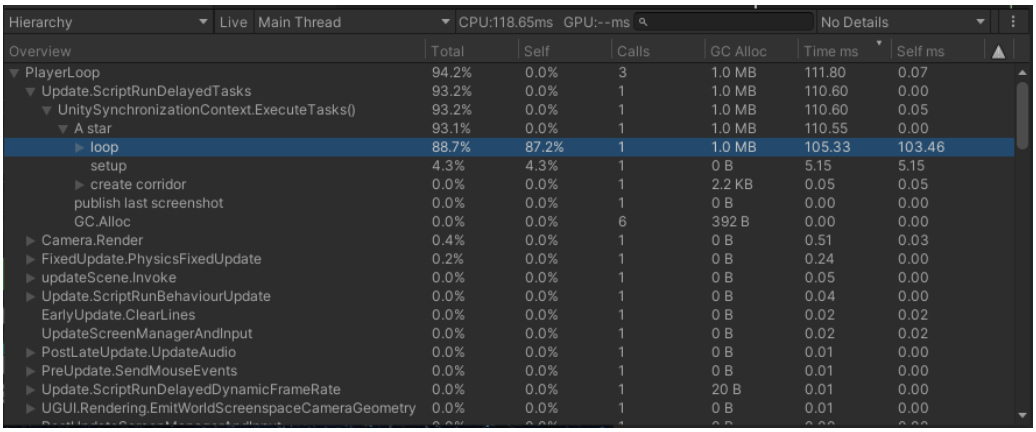
After getting a broad overview of the current performance of the search graphs and algorithms, in the following chapter a closer look is taken at each search graph and algorithm individually to identify performance critical aspects of their implementations. These will be the starting point for the then following optimisations.

3.2.1 Pathfinding algorithms

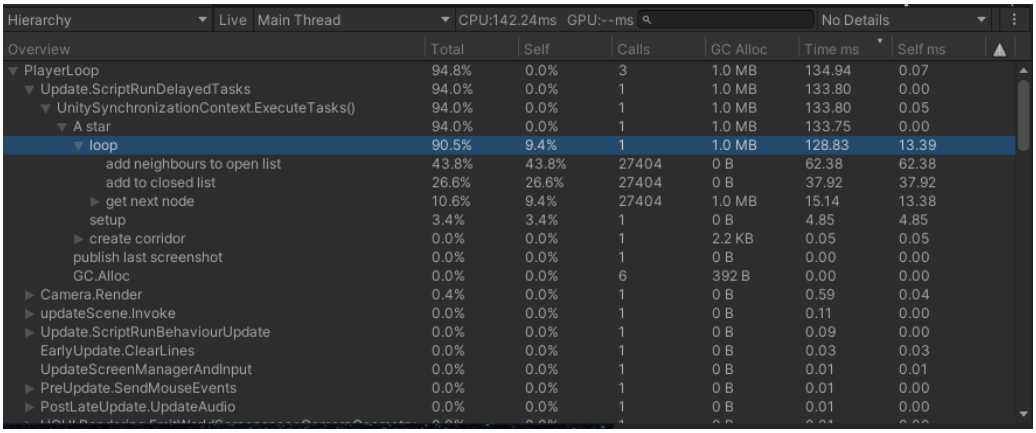
All pathfinding algorithms are analysed using an inflation factor of $\varepsilon = 1$ in the cell grid search graph. In the environment all dynamic objects, as well as the destination, and AI agent are made static. This ensures that the same path is found each time and results of different runs are comparable. Additionally, the algorithms are put on the main thread instead of a background one so Unity's profiler can be used.

Using Unity's profiler it has to be considered that adding profiler commands [Tec21w] impacts the code's performance. This can be seen clearly in figure 14. In 14i the implementation of A* is split into three parts: "setup", "loop", and "create corridor". "Loop" contains the main work of the algorithm and thereby takes the longest to compute. To investigate the inner working of the algorithm further and to pin-point the performance issues, additional profiler commands

were added within “loop” as can be seen in 14ii. Doing so the time of “loop” increased from 105.33 ms to 128.83 ms. Therefore, it is important to first get a general overview of where the most cost intensive operations are before going into more detail step by step.



(i)



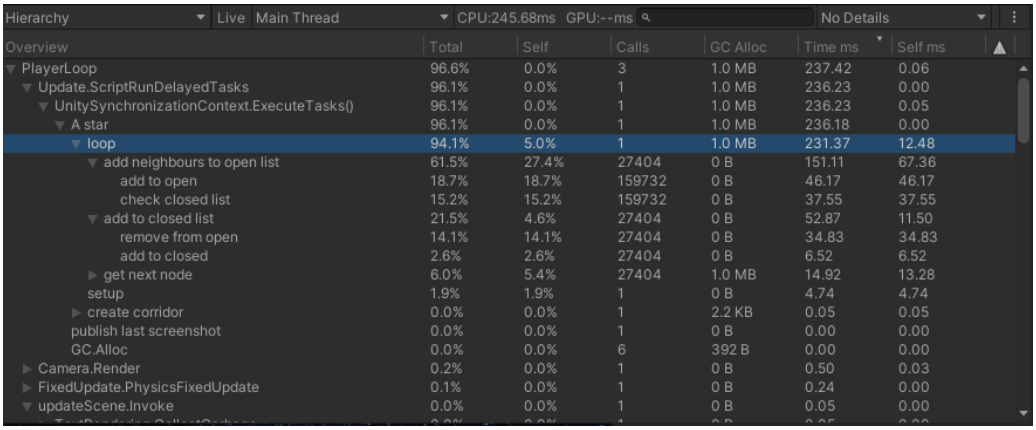
(ii)

Figure 14: Profiling performance of A*

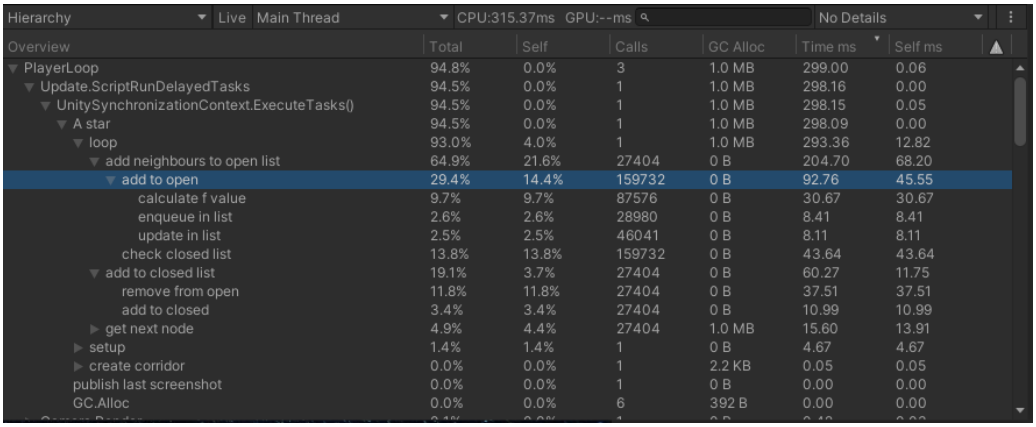
3.2.1.1 A*

Figure 14 shows the first two steps in analysing the most critical performance aspects of A*. As seen in figure 14ii getting the next node to expand from the open list takes the least time, whereas adding the neighbours of the current node to the open list takes the most time.

Figure 15i inspects the tasks “add to closed list” and “add neighbours to open list” further, with figure 15ii expanding the subitem “add to open”. It can be seen that checking if a list contains a specific item and removing an item from a list are similarly cost intensive, whereas enqueueing an item or updating its priority is comparatively cheap. Another important cost factor of the algorithm is calculating the nodes’ f-values, which also contains the calculation of the g-value.



(i)



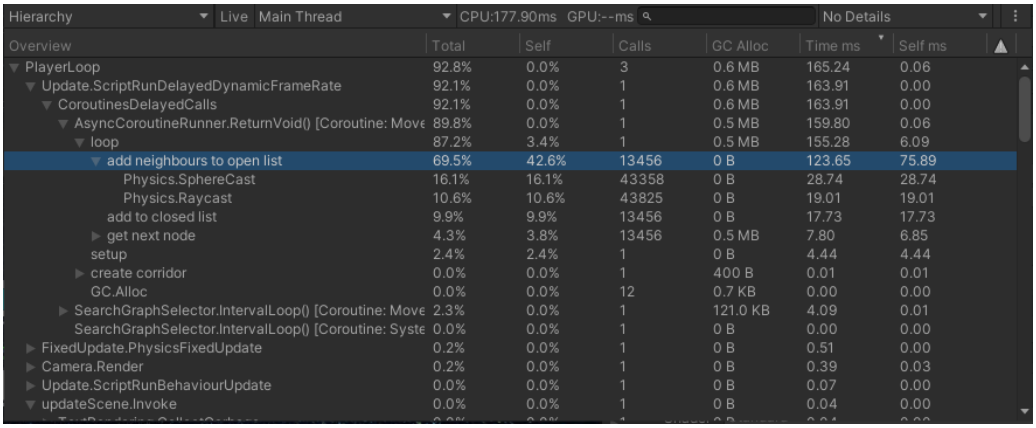
(ii)

Figure 15: Detailed profiling performance of A*

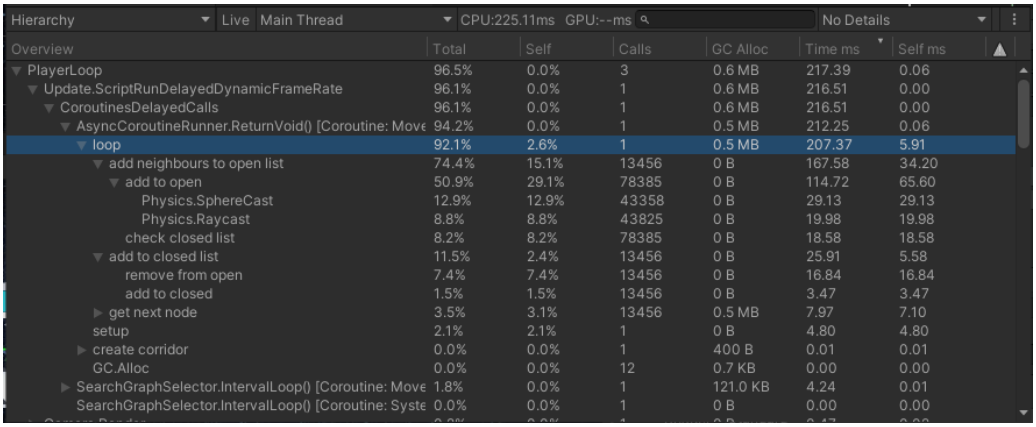
Overall the analysis showed that next to reducing the number of expanded nodes and, thereby, reducing the number of function calls, a focus should be put on optimising the lookup and deletion of items from lists, as well as the calculation of the nodes' f-values.

3.2.1.2 Theta*

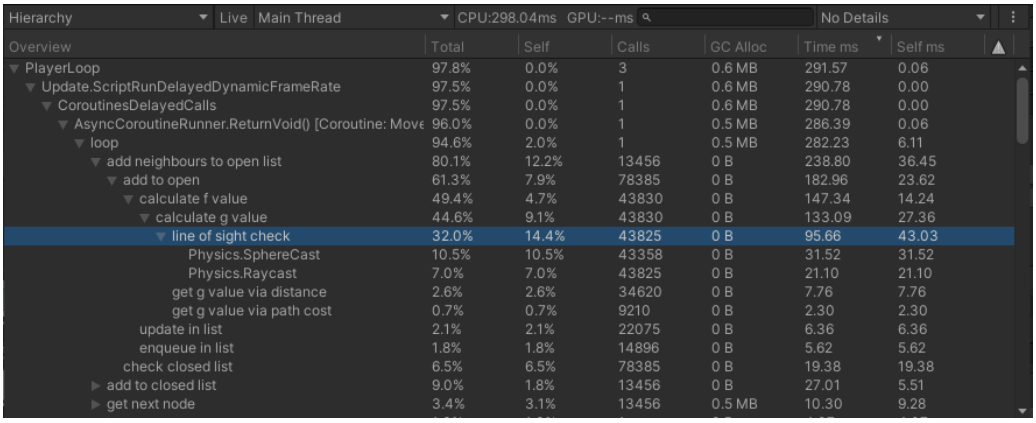
Next the implementation of the pathfinding algorithm Theta* is analysed to identify performance critical aspects. Originally, this algorithm was implemented using coroutines instead of running on the background thread. This is due to the fact that Theta* requires line-of-sight checks, which were implemented using Unity's physics system. Due to the feature used to access the background thread in the bachelor thesis, it was not possible to use any of the Unity API on that thread [Kra19, p. 51]. This included Unity's physics system. Therefore, all line-of-sight checks had to be performed on the main thread. For the performance analysis in this chapter, the algorithm is executed completely in one frame instead of splitting it up into multiple chunks and spreading it over multiple frames as was the case in the original implementation.



(i)



(ii)



(iii)

Figure 16: Profiling performance of Theta*

Figure 16 shows the performance profiling of Theta*. 16ii shows that in this algorithm the lookup of nodes in the closed list and removing items from the open list, is not as performance critical as in A*. The most performance relevant factor in Theta* is the calculation of the nodes' f-values, which includes the computation of the g-value, which in turn requires a line-of-sight check as can be seen in 16iii. Optimising these line-of-sight checks is the central point

of optimising Theta*. As well as reducing the number of required line-of-sight checks, another approach should be to implement these checks in such a way that they do not require Unity's physics.

3.2.1.3 D* Lite

For analysing the performance of D* Lite the algorithm was put onto the main thread and reset at every run. The strength of the dynamic algorithm is that it reuses previous results to update the algorithm upon detecting changed path costs in the search graph. The algorithm has to reset in two instances, though: when the goal node changes, or nodes in the search graph get assigned new neighbours. In the given test environment, the first is the case, due to having a dynamic destination, which changes the goal node regularly. The second is given when using a waypoint based search graph because that search graph can update its indirect neighbours when objects move. Therefore, to have comparable results over multiple profiling passes, it was decided to reset the algorithm at every run as if the goal node would have changed since the last run.

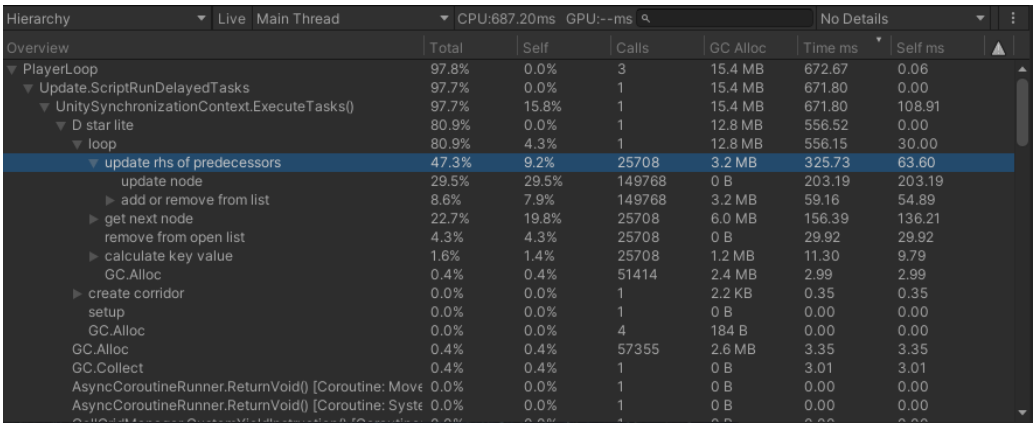
Profiling the performance of D* Lite figure 17 clearly shows that getting the next node (see task "get next node") and updating the next node's predecessors' *rhs*-values (see task "update rhs of predecessors") are the most time consuming processes.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	97.4%	0.0%	3	15.4 MB	550.79	0.06
Update.ScriptRunDelayedTasks	97.0%	0.0%	1	15.4 MB	548.68	0.00
UnitySynchronizationContext.ExecuteTasks()	97.0%	19.1%	1	15.4 MB	548.68	108.42
D star lite	77.3%	0.0%	1	12.8 MB	436.91	0.01
loop	77.2%	4.6%	1	12.8 MB	436.54	26.04
update rhs of predecessors	42.1%	41.3%	25708	3.2 MB	238.05	233.51
get next node	23.0%	21.5%	25708	6.0 MB	130.47	122.02
remove from open list	5.3%	5.3%	25708	0 B	29.97	29.97
calculate key value	1.6%	1.3%	25708	1.2 MB	9.05	7.51
GC.Alloc	0.5%	0.5%	51414	2.4 MB	3.02	3.02
create corridor	0.0%	0.0%	1	2.2 KB	0.35	0.35
setup	0.0%	0.0%	1	0 B	0.00	0.00
GC.Alloc	0.0%	0.0%	4	184 B	0.00	0.00
GC.Alloc	0.5%	0.5%	57355	2.6 MB	3.33	3.33
AsyncCoroutineRunner.ReturnVoid() [Coroutine: Move	0.0%	0.0%	1	0 B	0.00	0.00
AsyncCoroutineRunner.ReturnVoid() [Coroutine: Syste	0.0%	0.0%	1	0 B	0.00	0.00
CellGridManager.CustomYieldInstruction() [Coroutine:	0.0%	0.0%	1	0 B	0.00	0.00
FixedUpdate.PhysicsFixedUpdate	0.1%	0.0%	1	0 B	1.11	0.00
Camera.Render	0.0%	0.0%	1	0 B	0.53	0.03

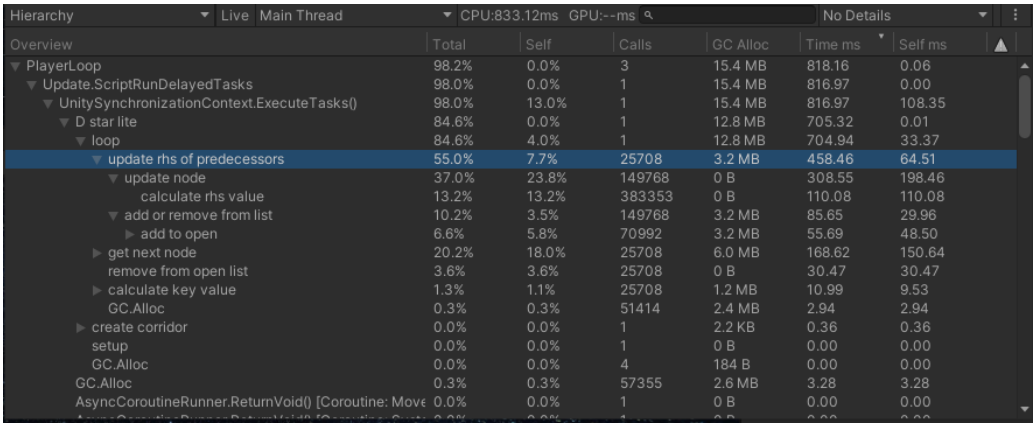
Figure 17: Profiling performance of D* Lite

First the process of updating the *rhs*-values of the next node's predecessors is profiled in more detail in figure 18. The exact cause of the performance issues in the task "update node'" is not clear. Neither calculating the *rhs*-value (see "calculate *rhs* value" in 18ii) nor other parts of the "update node" function include any obvious expensive calls, like lookups in lists or similar. Most likely the sheer amount of calls leads to even seemingly cheap operations to add up and impact the performance significantly. The task "add or remove from list" contains the comparison of the *rhs*- and *g*-value of the node. Additionally, it then either removes the node from the open list

or the new key value is calculated and the node added to the open list or its list entry updated. Even though when looking at the numbers this operation seems unperformant, it has relatively little impact on the overall performance of the algorithm.



(i)



(ii)

Figure 18: Profiling performance of updating *rhs*-value of predecessors in D* Lite

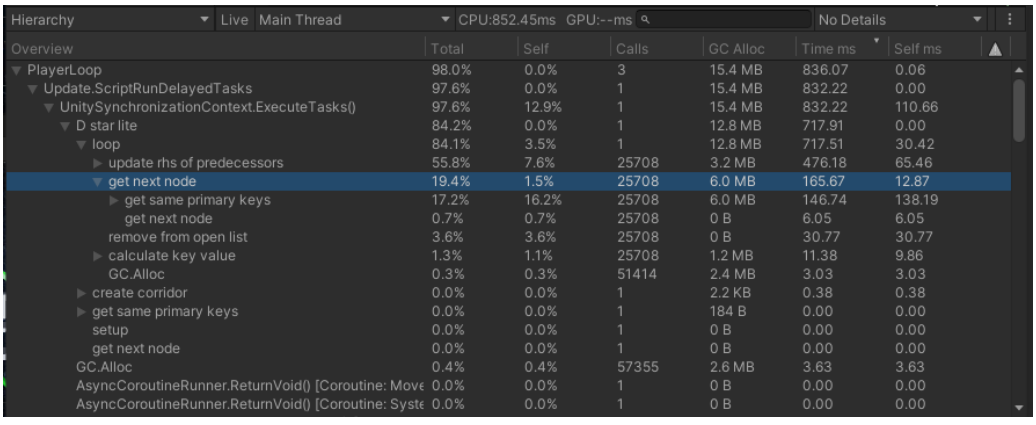
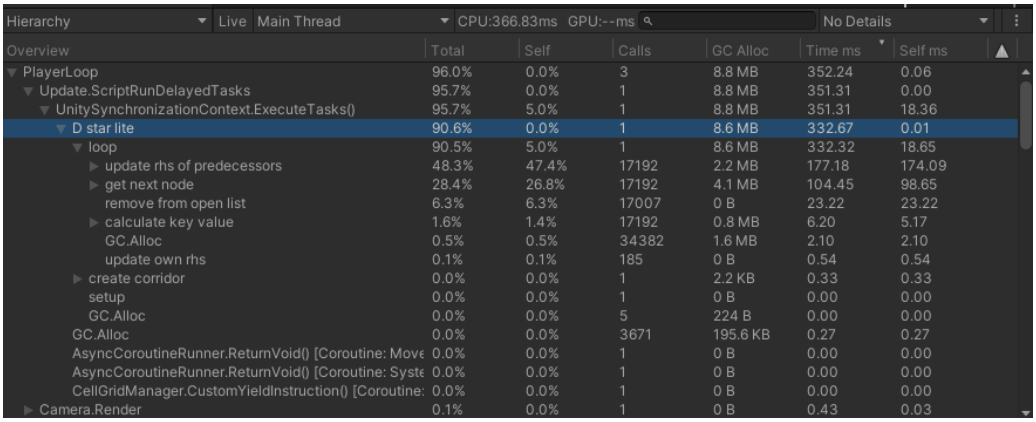


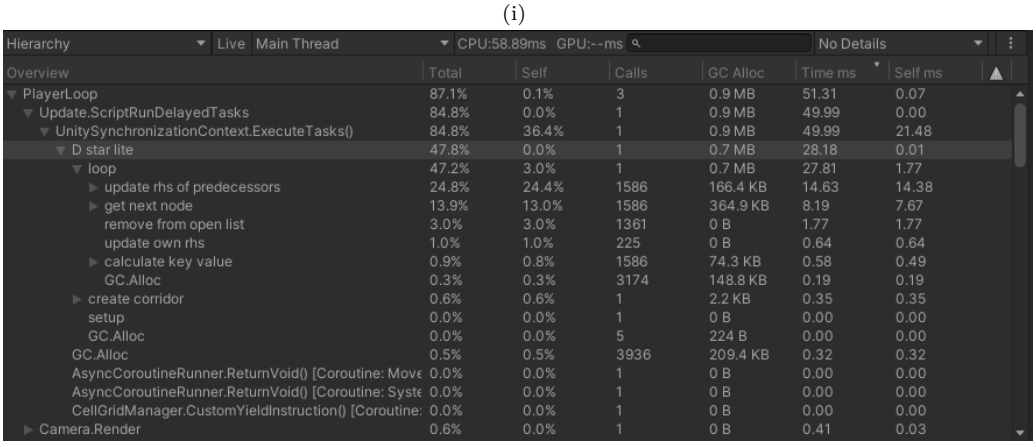
Figure 19: Profiling performance of getting next node in D* Lite

Next the task “getting next node” is investigated further in figure 19. Here the most expensive operation is getting a list of all nodes with the same primary priority key from the open list in order to compare their secondary keys. As already seen with A* handling large lists is a performance issue that needs to be optimised.

As already mentioned at the beginning of this chapter the strength of D* Lite is that it normally does not need to reset when path costs in the search graph change. Therefore, a few additional measurements were made in the cell grid search graph with dynamic obstacles, a static AI agent and a static destination. This way the strengths of the algorithm could be utilized in its full potential. Figure 20 shows the result of profiling two different runs of the algorithm in this dynamic environment. As expected, the runtime of the algorithm varies significantly, but nevertheless the same two aspects “update rhs of predecessors” and “get next node” have the highest runtime.



(i)



(ii)

Figure 20: Profiling performance of D* Lite in dynamic environment with static destination

When optimising the computation time required per expanded node, a focus should be put on implementing a faster solution to finding all nodes in the open list with the same primary priority keys, as well as optimising the updating of rhs-values of each expanded node’s predecessors.

Overall, optimising D* Lite will heavily rely on reducing the number of expanded nodes, though. The frequent resetting of the algorithm due to changed neighbours or an updated goal node needs to be eliminated in order to efficiently use the strength of the dynamic algorithm, which is to reuse previous calculations to compute new paths in a dynamic environment.

3.2.2 Search graphs

After analysing and identifying performance critical aspects in the implementation of the pathfinding algorithms, in the following chapters a similar analysis is done for the search graphs. The goal is to identify the most performance critical aspects of updating the search graphs when they adapt to the changes in the dynamic environment. To analyse this, all dynamic objects in the test environment were moving throughout the following tests.

Similar to Theta* the search graphs were originally implemented using coroutines and their update method was spread over multiple frames, as they require the use of Unity's physics system, which could not be used on the background thread. For the following analysis, though, their update functions were put entirely into a single frame.

3.2.2.1 Cell grid

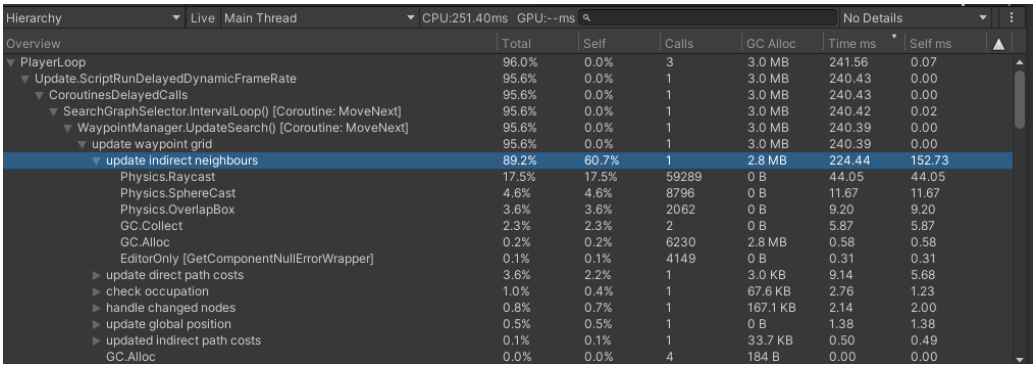
First the performance of updating the cell grid search graph is profiled in figure 21. The most expensive point hereby is "check occupation". This operation checks each cell of the search graph for if it is free or blocked by an obstacle using Unity's *Physics.OverlapBox* [Tec21s] function. As well as optimising the overall performance of this operation, the implementation of an occupation check which does not use Unity's physics system should be investigated.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
EditorLoop	56.1%	56.1%	3	0 B	120.43	120.43
PlayerLoop	43.7%	0.0%	3	2.3 MB	93.72	0.10
Update.ScriptRunDelayedDynamicFrameRate	42.4%	0.0%	1	2.3 MB	90.98	0.00
CoroutinesDelayedCalls	42.4%	0.0%	1	2.3 MB	90.98	0.00
SearchGraphSelector.IntervalLoop() [Coroutine: M	42.4%	0.0%	1	2.3 MB	90.97	0.03
CellGridManager.UpdateSearch() [Coroutine: M	42.4%	0.0%	1	2.3 MB	90.93	0.00
check occupation	36.8%	18.5%	1	1.9 MB	78.97	39.80
Physics.OverlapBox	16.5%	16.5%	57344	0 B	35.38	35.38
GC.Alloc	1.7%	1.7%	57344	1.9 MB	3.78	3.78
handle changed nodes	2.4%	2.4%	1	180.1 KB	5.25	5.25
update path costs	2.0%	2.0%	1	215.0 KB	4.29	4.29
check free blocked changed	1.1%	1.1%	1	16.3 KB	2.40	2.40
GC.Alloc	0.0%	0.0%	2	80 B	0.00	0.00
DelayedCallManager.CancelCallDelayed	0.0%	0.0%	1	0 B	0.00	0.00
CellGridManager.UpdateSearch() [Coroutine: S	0.0%	0.0%	1	0 B	0.00	0.00
CellGridManager.UpdateSearch()	0.0%	0.0%	1	136 B	0.00	0.00
GC.Alloc	0.0%	0.0%	1	24 B	0.00	0.00
SearchGraphSelector.IntervalLoop() [Coroutine: S	0.0%	0.0%	1	0 B	0.00	0.00
FixedUpdate.PhysicsFixedUpdate	0.6%	0.0%	5	0 B	1.28	0.01

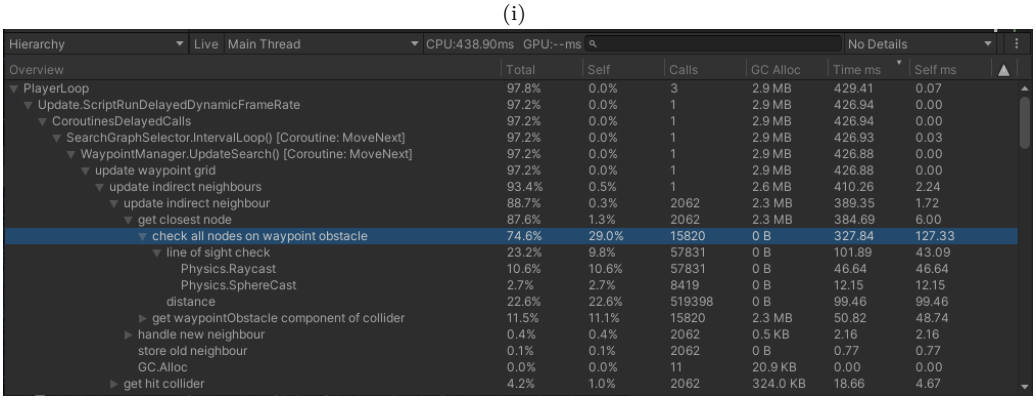
Figure 21: Profiling performance of cell grid search graph

3.2.2.2 Waypoints

Next the performance of updating the waypoint based search graph is profiled as seen in figure 22. In this search graph updating all indirect neighbours of all nodes is the most time consuming operation (see "update indirect neighbours" in figure 22i). Investigating the performance of this function further, it can be seen that here, too, a physics based line-of-sight check is used, which impacts the search graph's performance negatively. Therefore, minimising or eliminating the use of Unity's physics, especially for line-of-sight checks, should be focused upon to optimise the runtime of the waypoint based search graph.



(i)



(ii)

Figure 22: Profiling performance of waypoint based search graph

3.2.3 Conclusion

After analysing the performance of the pathfinding algorithms and the updating of the search graphs, a few common factors can be identified that impact their performances significantly.

Number of updated and expanded nodes

First, the probably most important factor to optimise is the number of nodes that have to be updated by the search graph each time the dynamic environment moves and that get expanded by the pathfinding algorithms each run. Reducing this number is a crucial part in optimising the search graphs and pathfinding algorithms.

Physics

Another important factor impacting the algorithms' and search graphs' performances is the use of Unity's physics system. As mentioned in chapter 3.2.1.2 and 3.2.2, the use of physics prevents Theta* and the updating of the search graphs to be run on the background thread due to the feature used to access that thread in the current implementation. Additionally, the analysis has shown that the use of physics functions is the most cost intensive aspect of Theta* and the updating of the search graphs. Therefore, using the physics system more sparsely needs to be part of the optimisation process. Especially implementing a non-physics based line-of-sight check is expected to be beneficial for Theta* and the waypoint based search graph.

Lists

Lastly, handling large lists is quite expensive as it was shown for A* and D* Lite. Checking if an item is contained in a list, removing items from a list, and getting all items with a certain condition impacts the algorithm's performance greatly and needs to be limited and optimised.

4 Concept

Having analysed the performance of the current implementation of the pathfinding algorithms and search graphs as well as having identified the most performance critical aspects in the previous chapter, in this chapter a concept is evolved on how to optimise the pathfinding algorithms and the used search graph, especially its updating.

4.1 Algorithm characteristics

4.1.1 Multi-threading

In the current implementations of A* and D* Lite multi-threading was achieved by running most of the algorithms' computations asynchronously using the so-called "async-await" [Ver17] feature in combination with the "Async Await Support" asset [Ver18]. In the following optimisation process this will be replaced and multi-threading will be instead implemented using Unity's C# Job System in combination with the Burst compiler.

4.1.2 Number of expanded nodes

One critical aspect in optimising the performance of the pathfinding algorithms is the reduction of the number of expanded nodes. For A* this can be achieved by using Jump Point Search in a cell grid search graph. To reduce the number of expanded nodes of D* Lite, the main focus will be on preventing the algorithm from resetting each time the goal node changes. For this D* Lite will be extended by implementing (Basic) Moving Target D* Lite.

4.1.3 Line-of-sight checks

The main performance issue of Theta* is its physics-based line-of-sight check. Using Unity's physics system is expensive on one hand, as can be seen in the performance analysis in chapter 3.2.1.2, and on the other hand it makes it difficult to implement line-of-sight checks asynchronously. Unity's C# Job System, which is used in this thesis to implement multi-threading, does in theory support the use of physics in jobs, for example by utilizing the *SherecastCommand* [Tec21x]. Using physics in jobs is restricted by a set of rules, though, and requires a very specific setup, which makes it only implementable in certain scenarios. For the use-case of line-of-sight checks for Theta*, it is hardly a viable option and is therefore excluded in this thesis.

Therefore, part of optimising Theta* is the implementation of line-of-sight checks, which can be performed asynchronously within the C# Job System and do not require physics.

4.2 Search graph

Next to a performant pathfinding algorithm, a performant search graph is a crucial component of a successful pathfinding system in a dynamic 3D space. Hereby, it has to be kept in mind that the setup of the used search graph can have a great impact on the performance of the pathfinding algorithms. The ideal search graph should have as few nodes as possible, produce short paths, and be fast to update. Neither the cell grid search graph, nor the waypoint based search graph produce a satisfying result on its own. The waypoint based search graph has fewer nodes than the cell grid graph, but is very unperformant when updating, mainly because it has to check for new indirect neighbours at each update. This changing of indirect neighbours also causes D* Lite to reset. In the cell grid search graph all nodes have consistent neighbours and each node by itself is fast to update. In total the search graph has a lot of nodes, though, which not only impacts the update duration of the graph, but also the computation time of the pathfinding algorithms.

Therefore, in this thesis a third search graph is suggested: A combination of the cell grid search graph and the waypoint based one, featuring the positive aspects of both graphs. Around static objects nodes are placed in a similar pattern as nodes in the waypoint based search graph. Around areas of dynamic objects nodes are lay-outed in a cell grid. In contrast to the old waypoint based search graph, in this new approach each node is assigned a consistent set of neighbours when the search graph is first created and does not change its neighbour when the dynamic environment changes. This technique reduces the total number of nodes in the search graph but still ensures that all nodes have consistent neighbours. Furthermore, to improve the path lengths of non any-angle algorithms, nodes will also know their diagonal neighbours, instead of only their horizontal and vertical ones.

5 Implementation

After having formulated a concept for optimising the performance of the pathfinding algorithms and search graph in the previous chapter, this chapter will detail the implementation of these optimisations. The implementation uses Unity version 2020.3.3f1 (LTS) and is run on a Predator Helios 300 Laptop by Acer, with a Nvidia GeForce GTX 1060 graphics card and an intel core i7 processor. If not stated otherwise all algorithms use an inflation factor of $\varepsilon = 1$ and for all performance measurements the safety system of the job system and Burst compiler is disabled. Additionally, all listed code should be considered pseudocode that only shows relevant aspects of the respective class or function. The full Unity project and source code for the final implementation of the pathfinding system can be downloaded from GitHub using the following link: https://github.com/CarinaKr/Pathfinding-in-dynmic-3D-space [Kra21].

5.1 Optimised pathfinding algorithms

First the optimisations of the pathfinding algorithms will be described in the following chapters. Starting, 5.1.1 gives a general overview of required adaptations to the nodes' data type and other changes relevant for all pathfinding algorithms due to the use of Unity's C# Job System and Burst compiler. Afterwards the optimisations of each individual pathfinding algorithm is explained in more detail.

5.1.1 General setup for the C# Job System and Burst compiler

To start using Unity's C# Job System and Burst compiler the required packages need to be downloaded, for example via Unity's package manager. In this implementation the following packages are used:

- Jobs Version 0.8.0-preview.23
- Burst Version 1.4.1
- Mathematics Version 1.2.1

As described in 2.1 in Unity's C# Job System each job is a struct, which inherits from the interface *IJob* and implements the *Execute* method. In this paper each pathfinding algorithm is implemented in its own struct. Even though the individual pathfinding algorithms share some common functionality it is not possible to derive them from a common parent struct or class, as structs only support inheritance of interfaces. Instead, several static helper classes are used,

which can be easily accessed from all structs and enable the sharing of functions between multiple pathfinding algorithms. Additionally, all pathfinding jobs inherit from the *IPathfinderJob* interface, which allows for an easier handling of setting up the different pathfinding jobs in a common *Pathfinder* class.

When using the Burst compiler with the job system only certain data types can be used within the job structs. This includes primitive data types such as int, float, double, bool, etc., as well as NativeContainer types such as NativeArray, NativeList, etc. Structs can also be used in a job, but only if they only contain fields of those supported types. For the implementation of the pathfinding algorithms this means that the previously implemented *Nodes* class can no longer be used. Instead, nodes in the cell grid are now represented by the struct *CellNode*, which contains all necessary data, such as the node's current rhs-value, g-value, priority key, local and global position, its neighbours (successors) and predecessors, its parent node, whether it is currently blocked, and if it is currently in the open list. Within the pathfinding jobs, cell nodes are stored in a NativeArray<CellNode> called *nodes*. Each specific node can be accessed by its index in this NativeArray. For example, the fields *startNode* and *goalNode* are of data type int and point towards the correct index in the *nodes* array. The index of a node in the NativeArray is based on its position in the search graph and is consistent throughout the pathfinding process.

Upon starting a new pathfinding job a copy of the list with the currently updated nodes is passed from the search graph to the pathfinding job and stored in this NativeArray<CellNode> *nodes*. Passing a copy of the nodes to the pathfinding algorithm has the advantage that it allows the search graph to update its nodes without having to account for the pathfinding algorithm using their values simultaneously for computations. The downside of this approach is that no values can be stored within specific nodes over multiple runs of the algorithm. This is absolutely necessary for D*-based algorithms, though, such as (Basic) MT-D* Lite, as they require the nodes' g- and rhs-values to be saved for future runs. Therefore, in (Basic) MT-D* Lite the NativeArray *nodes* only gets a copy of the search graph's current nodes on the first call of the job and whenever the algorithm has to reset due to the AI reaching its destination. In all other instances the updated nodes from the search graph are passed into the job as a separate NativeArray called *globalNodes*. Before starting a new run (Basic) MT-D* Lite updates the path costs of all nodes in *nodes* which were changed by the search graph since the last run of the algorithm using the values of *globalNodes*. Another beneficial value to save over multiple runs is each node's heuristic. As all pathfinding algorithms can benefit from saving this value and only having to update the nodes' heuristic when the goal node changes, it was decided to extract the storage of the nodes' heuristic out of the *CellNode* struct and into a separate NativeArray<double> called *heuristic* for all non-dynamic algorithms. Hereby the length of the *heuristic* array has the same length as the *nodes* array and the indices of entries in the two arrays match. In other words, the node stored at index 0 in the *nodes* array, has the heuristic

which is stored in *heuristic* at index 0. The content of the *heuristic* array is filled the first time the pathfinding algorithm is executed and then saved over multiple runs of the algorithm. Only when the goal node changes, the content of the array gets updated. As D*-based algorithms store their nodes over multiple runs as described above, it is not necessary for them to extract the nodes' heuristic in a separate NativeArray.

Using a NativeArray<CellNode> to store the list of current nodes *nodes* in the pathfinding jobs, result in the restriction that the *CellNode* struct can not contain any fields which have a NativeContainer data type. In the previous implementation a node's neighbours and predecessors were stored in lists and the path costs to all neighbours in a dictionary. As mentioned in the beginning of this section, a struct used in a job with Burst compiler can only contain fields with data types which are supported by Burst. But even changing these managed data types of lists and dictionaries to NativeContainer data types which are normally supported by Burst, is not possible when using the struct implementing fields with these types in a NativeContainer itself. In other words, it is not possible to use a NativeContainer<NativeContainer> type and therefore also no NativeContainer<CellNode> type, if *CellNode* implements any NativeContainer fields. Therefore, neighbours, predecessors, and path costs are stored in 3x2 matrices of data types int3x2 and double3x2. Hereby, the int3x2 matrices for the neighbours and predecessors store the index of the according neighbour or predecessor node. If a node has less than six neighbours or predecessors the empty spaces are set to an index of -1. After storing all neighbours of a node in an int3x2 matrix the path costs from the node to these neighbours are stored in the same positions in the path cost's double3x2 matrix as the index for the neighbours is in the *neighbours* int3x2 matrix. In other words, the path cost stored in *pathCost[0][0]* of a node is the cost from the node to its neighbour with the index stored at *neighbours[0][0]*.

Next to changing the data type of the nodes' neighbours and path costs, also the data type of the open list has to be changed when running the pathfinding algorithms using Unity's C# Job System and Burst compiler. In the optimised implementation, the open list is stored as a NativeHeap provided by the user Amarcolina on GitHub [Kry20]. This implementation of a heap provides all common functionalities, such as *Insert*, *First*, and *Peek*, in addition to the option to remove items from within the list. By default the NativeHeap provides the sorting functions minimum and maximum for all standard number data types such as int, float, and double. Additionally, it allows the implementation of custom comparison functions to be used when sorting the heap. In the NativeHeap *openList* nodes are stored as a copy of the *CellNode* struct with the node's current values.

Further adjustments to the implementation of pathfinding algorithms when using the Burst compiler affect the use of foreach-loops and Vector data types, such as Vector3. When using the Burst compiler foreach-loops are not possible and have to be replaced with for-loops.

Additionally, all Vector3 usages were replaced by float3, which is provided by the Mathematics-package and provides similar functionality as Vector3. Mostly this did not require any additional code changes. The only exception to this is the calculation of the distance between two points, which is especially needed in the computation of a node's heuristic. Instead of using the Vector3.Distance method, the distance between two float3 positions is now calculated as:

$$distance(a, b) = \sqrt{(b.x - a.x)^2 + (b.y - a.y)^2 + (b.z - a.z)^2}$$

Following the example of [Fre19a] on slides 14–17 and 38–47, updating the heuristic of all nodes in the search graph is optimised by dividing the nodes in clusters of four. Each cluster stores its nodes' x-, y-, and z-positions in float4 vectors each and calculates the heuristic of all four nodes in one calculation (see listing 25 in the Appendix).

In a last step to adapt the implementation of the pathfinding algorithms to the job system and Burst compiler, the way of determining a corridor for the AI to follow and the method of storing this corridor information needs to be changed. Instead of passing a list of nodes from the pathfinding algorithm to the AI, now a NativeList<float3> is passed back from the pathfinding job. This NativeList contains the global positions of all nodes in the corridor along the shortest path from the start node to the goal node in the correct order starting at the start node and ending at the goal node. The AI's movement script is adapted to follow this list of float3 positions.

For the implementation and time measurements of the pathfinding algorithms in the following chapters, the pathfinding jobs are started from the main thread and using the job's *Complete* method it is waited for the job to complete before continuing the main thread. This is due to the fact that the updating of the search graphs is not optimised, yet, and therefore might impact the performance of the algorithms. For the final implementation, the main thread will not be paused whilst the pathfinding algorithms perform their calculations.

5.1.2 A* with Jump Point Search

To optimise the computation time of the A* pathfinding algorithm in this chapter the implementation of Jump Point Search (JPS) is detailed. As described in chapter 2.2 JPS can improve the runtime of pathfinding algorithms by decreasing the number of expanded nodes.

Before starting to implement JPS, A* is first adapted to run in Unity's C# Job System in combination with the Burst compiler following the changes described in the previous chapter. As a part of this the calculation of each node's neighbours' g- and f-values was implemented to run in parallel, instead of in a for-loop for one neighbour after the other, as seen in the

AddNeighboursToOpen function shown in listing 3. Concrete this means that the fields *gValues* and *fValues* are of type double3x2 and allow for the simultaneous calculation of values for all neighbours of the currently expanded node.

```
int AddNeighboursToOpen(int nodeIndex)
{
    node = nodes[nodeIndex];
    ...
    gValues = node.gValue + node.pathCosts;
    fValues = gValues + heuristics;
    ...
}
```

Listing 3: *AddNeighboursToOpen* function in A*

Additionally, one of the main performance issues of A* in its previous implementation was addressed; the lookup of nodes in the closed list. This was solved by removing the closed list. Instead, each node stores a boolean value to remember whether it is in the closed list or not.

Building upon this Burst-optimised implementation of A*, JPS was added. For this the core loop of the algorithm had to be adjusted, so that instead of adding all neighbours of a node to the open list jump points are found first, which are then added to the open list. This difference can be seen in the listings 4 and 5. Line 3 in listing 4 gets replaced by the lines 3–5 in listing 5.

```
1 while (nextNode != goalNode && nextNode != -1)
2 {
3     AddNeighboursToOpen(nextNode);
4     nextNode = GetNextNode();
5 }
```

Listing 4: Core loop of A*

```
1 while (nextNode != goalNode && nextNode != -1)
2 {
3     GetRemainingNeighbours(nextNode);
4     GetSuccessors(nextNode);
5     AddJumpPointsToOpen(nextNode, successor);
6     nextNode = GetNextNode();
7 }
```

Listing 5: Core loop of JPS

The most important function in this algorithm is finding a node's jump points to use them as successors instead of the node's direct neighbours. This is done as part of the *GetSuccessors* method called in line 4 in listing 5. As Harabor and Grastien only detail an implementation

of finding jump points in an 8-way connected two-dimensional search grid in [HG11], the two GitHub repositories "jps" by the user kevinsheehan [She16] and "EpPathFinding.cs" by the user juhgiyo [Kry20] were partly taken as reference for the implementation of JPS in the three-dimensional search grid with no diagonal movement. Both repositories include functions of finding jump points in a search grid without diagonal movement and [Kry20] even uses a 3D grid. As described in 2.2 JPS steps in the relative direction from the explored node to its neighbour until a blocked node is hit or a jump point is returned. A node is returned as a jump point if it is the goal node, has any forced neighbour, or, in case of diagonal movement, if there is a jump point returned on the individual horizontal or vertical axis. As there is no diagonal movement in the search grid used in this thesis, the last condition changes. Instead of only having to check for jump points on additional axis in the case of diagonal movement, additional axis have to be checked in any case.

When stepping along one axis to check for possible jump points all perpendicular directions $\vec{d_p}$ to the current stepping direction $\vec{d}$ need to be checked for possible jump points, too, as well as all perpendicular directions to each $\vec{d_p}$. A similar approach can be seen in [She16] in the *JPSDiagNever.java*-script line 70–72, where the positive and negative horizontal axis are checked for jump points in case of a jump point search on the vertical axis. A similar approach can be seen in [Kry20] in the *JumpPointFinder.cs*-script line 1859–1879. To avoid walking in circles and endlessly calling the *GetJumpPoint* function on all perpendicular axis, in the implementation in this thesis the last search axis is always excluded from the perpendicular axis search. Listing 6 shows pseudo-code of the relevant part of the *GetJumpPoint* method for this.

For a better understanding of this process the following paragraph goes through the function step by step with the aid of an example. This example assumes that after pruning a node n, it has one remaining neighbour n', which is located to the right (in positive x-direction) of n. A *GetSuccessors* function now calls *GetJumpPoint* with the node index of n, the direction index 0, and *checkDirections* set to true. The *directionIndex* is the index of a direction in the *normalizedDirections* array. This array defines all straight directions in the search graph as float3 values in a specific order. *normalizedDirections[0]* equals the float3 (1, 0, 0). The matrix *checkDirections* indicates what directions need to be considered when checking for jump points on perpendicular axis. Hereby each matrix-index represents one of the possible six directions in the search grid. The row-index stands for the axis and the column-index for whether the direction is positive or negative. *checkDirections[0][0]* represents the direction (1, 0, 0) and thereby *normalizedDirections[0]*, whereas *checkDirections[0][1]* represents the direction (-1, 0, 0) and thereby *normalizedDirections[1]*. A similar approach is taken to fill each nodes' *orderedNeighbours* matrix, which is used to determine the next possible jump point node in line 3 in listing 6. Here also the matrix-index determines in which direction the neighbour lies relative to a node, whereby the order is the same as in the *checkDirections* matrix.

Before checking for jump points along perpendicular axis, the *checkDirections* matrix is updated in consideration of the current direction in line 8 and 9. For this example this means that the indices [0][0] and [1][0] are set to false. In line 11–19 all remaining directions are checked for possible jump points, starting in this example with the direction index 2, which represents the upwards direction float3 (0, 1, 0). Calling *GetJumpPoint* in line 15 the updated *checkDirections* matrix is passed, which already has the values in the first row set to false. This is important in this second call of *GetJumpPoint*, as here now also the indices [0][1] and [1][1] of this matrix are set to false. Therefore, in the for-loop and if-clause in line 11 and 13 the x-axis as well as the y-axis are ignored and only the z-axis is checked in positive and negative direction for possible jump points. When both directions on the z-axis are checked for jump points the method returns to line 16 of its first call. If a jump point was found in the positive y-direction it gets stored in the according position in the *jumpPointsOnLines* matrix (line 16). If on any checked axis a jump point is returned, the *jumpPointNode* is returned (line 24) and replaces n' as a successor of n.

```
1  int GetJumpPoint(int nodeIndex, int directionIndex, bool3x2 checkDirections)
2  {
3      int jumpPointNode = nodes[nodeIndex].orderedNeighbours[directionIndex % 2][
       directionIndex / 2];
4      ...
5
6      int3x2 jumpPointsOnLines = -1;
7
8      checkDirections.c0 &= normalizedDirections[directionIndex] == 0;
9      checkDirections.c1 = checkDirections.c0;
10
11     for(int i = 0; i < 6; i++)
12     {
13         if(checkDirections[i % 2][i / 2])
14         {
15             int jumpPoint = GetJumpPoint(jumpPointNode, i, checkDirections);
16             jumpPointsOnLines[i % 2][i / 2] = jumpPoint;
17             returnJumpPoint |= jumpPoint != -1;
18         }
19     }
20
21     if(returnJumpPoint)
22     {
23         AddOrUpdateNodeJumpPointsKey(jumpPointNode, jumpPointsOnLines);
24         return jumpPointNode;
25     }
26 }
```

Listing 6: Checking for a jump point on perpendicular axis as part of the *GetJumpPoint* function

As this process of determining all jump points of a node is very time consuming it should be called as seldom as possible. In the pseudo-code of JPS detailed in [HG11] as well as the repositories [She16] and [Kry20] the respective code passages which handle the determination of jump points by checking additional axis for jump points, only return whether a jump point was found and do not store the result any further. This way these jump points have to be found again later when this node gets expanded. To avoid having to find the same jump points for a node twice, all returned jump points of a node are already stored in a dictionary in line 23 of listing 6.

A similar approach is taken when returning a jump node because it is having forced neighbours. Listing 7 shows a different part of the *GetJumpPoint* function which handles the return of a jump point node if it has any forced neighbours. Instead of simply registering that forced neighbours were found and then later getting these forced neighbours when the node gets expanded, all found forced neighbours are stored in a dictionary.

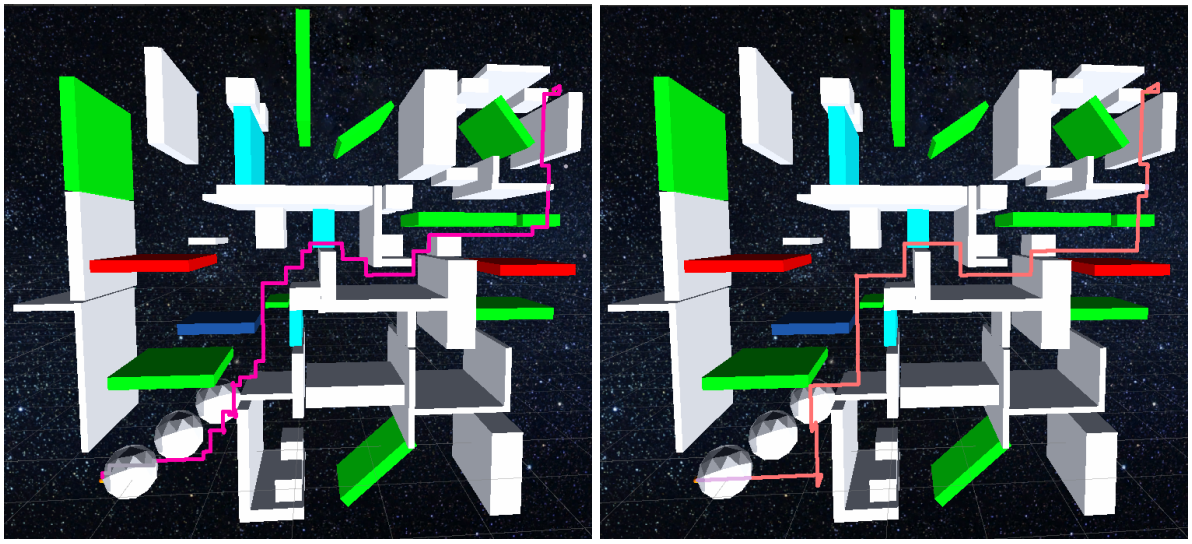
```
1  int GetJumpPoint(int nodeIndex, int directionIndex, bool3x2 checkDirections)
2  {
3      int jumpPointNode = nodes[nodeIndex].orderedNeighbours[directionIndex % 2][
       directionIndex / 2];
4      ...
5
6      int3x2 jumpPointForcedNeighbours = -1;
7      if(GetForcedNeighbours(ref forcedNeighbours, jumpPointNode, directionIndex))
8      {
9          AddOrUpdateForcedNeighboursKey(jumpPointNode, forcedNeighbours);
10         return jumpPointNode;
11     }
12
13     int3x2 jumpPointsOnLines = -1;
14     ...
15
16 }
```

Listing 7: Checking for forced neighbours as part of the *GetJumpPoint* function

Using these two dictionaries speeds up the neighbour pruning in the *GetRemainingNeighbours* function (see listing 5) and requires *GetJumpPoints* to be called significantly fewer times in the *GetSuccessors* function.

The influence of the use of jump points in JPS can be seen clearly when comparing the shortest paths found by JPS to those of the traditional A* pathfinding algorithm. Figure 23 shows the shortest path found by A* on the left and JPS on the right. Even though both found paths have the same length, the path found by A* has a lot more directional changes than the one

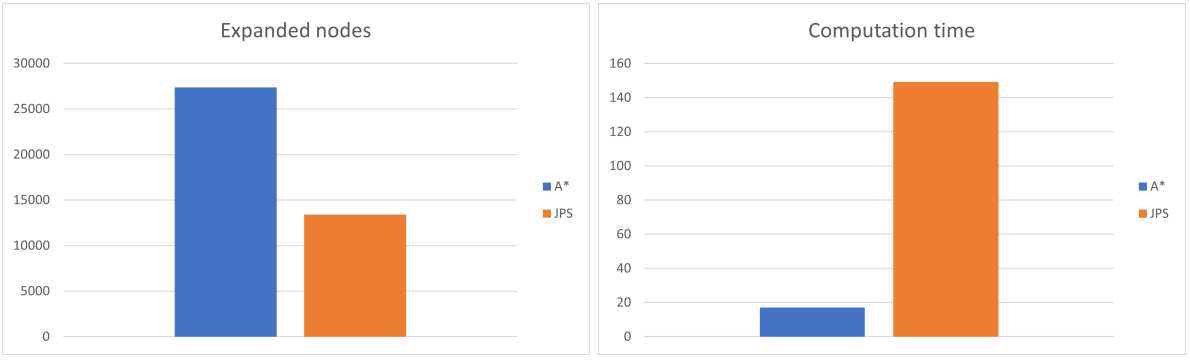
found by JPS. By using jump points the path moves longer in one direction and only changes directions when this is absolutely necessary.



(i) A* (ii) JPS

Figure 23: Shortest paths found by A* (left) and JPS (right) using a cell grid and $\varepsilon = 1$

In a next step the performance of JPS is evaluated. Running the algorithm in the test environment with a moving destination and dynamic obstacles, but a static AI agent, has shown that even though JPS on average expands less than half the nodes of A*, it has an almost eight times higher computation time (see figure 24).



(i) (ii)

Figure 24: Comparison of the average number of expanded nodes and computation time (in ms) using a cell grid and $\varepsilon = 1$

The main reason for this is the massive overhead caused by not only checking each node along one axis for possible jump points, but each time also having to check all perpendicular axis for possible jump points. Even though the storage of forced neighbours and jump points in dictionaries already cut down the computation time significantly by reducing the number of

times *GetJumpPoint* has to be called, the recursive call of the method in line 15 in listing 6 still slows down the determination of successor so greatly that JPS can not be used in this implementation. After several optimisation iterations on this function, it was decided to stop the optimisation of JPS. Even though some other aspects of the algorithm, such as the storage and reading of the dictionaries, might still be further optimisable the overall computation time of the algorithm would most likely be reduced only slightly by these optimisations. As long as the core element of finding jump points is the bottleneck in the performance, JPS is not optimisable to a point where it can compete with A*.

5.1.3 Theta*

As a basis for Theta*, the implementation of A* using Unity's C# Job System and Burst compiler from the previous chapter 5.1.2 is used. Theta* uses the same same core loop as A* (see listing 4). The main difference between A* and Theta* is the determination of the nodes' parent and the therefrom resulting calculation of the g- and f-value.

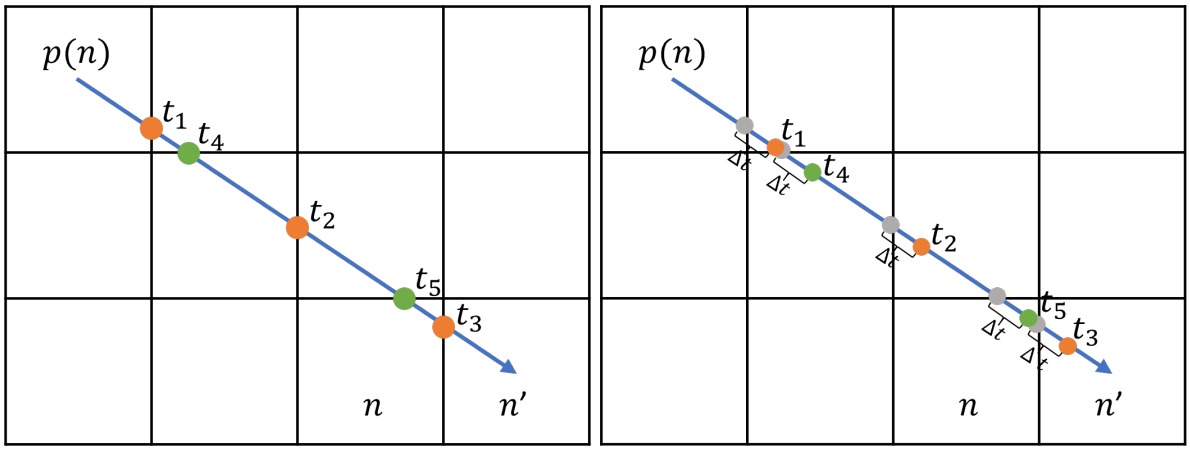
In the implementation of A* the f-values of each expanded node's neighbour is computed as part of the *AddNeighboursToOpen* function, which is called from the core loop (see listing 4 line 3). The relevant part of this function is shown in listing 3. The *AddNeighboursToOpen* function of Theta* looks very similar to the one of A*. Instead of using the expanded node's g-value and path costs for calculations, though, updated path costs and g-values are computed based on possible new parent nodes in the function *FillPathCostsAndParents* in line 8 of listing 8. The resulting parameter *baseGValues* represents the g-values of the correct parent node of each neighbour n' and *pathCosts* is the distance from $parent(n')$ to n'.

```
1  int AddNeighboursToOpen(int nodeIndex)
2  {
3      node = nodes[nodeIndex];
4      ...
5      double3x2 baseGValues = node.gValue;
6      pathCosts = nodes[nodeIndex].pathCosts;
7      newParents = nodeIndex;
8      FillPathCostsAndParents(nodeIndex,node.neighbours,node.parentNode,
9                              ref baseGValues, ref newParents, ref pathCosts);
10
11     gValues = baseGValues + pathCosts;
12     fValues = gValues + heuristics;
13     ...
14 }
```

Listing 8: *AddNeighboursToOpen* function in Theta*

The full pseudocode for the *FillPathCostsAndParent* function can be found in listing 26 in the Appendix. The main functionality of this function is to determine for each neighbour n' of the currently expanded node n if there is an unobstructed direct path between n' and the parent node of n. This is done in two steps. First, it is checked if n, $parent(n)$, and n' are on a straight line. Given that the used search graph is grid-based and does not allow diagonal movement, this can be done by comparing the x-, y-, and z-position values of n, $parent(n)$, and n' and storing the result in a boolean-vector (see line 23–26 in listing 26). If the vector contains two *true*-values and the path cost from n to n' is less than infinity, there is an unobstructed direct path between the nodes and $parent(n)$ can be set as the parent of n'.

In cases where the three nodes are not on the same line, a more complex line-of-sight check needs to be run between n' and $parent(n)$. The basis of this line-of-sight check is to determine which nodes have to be free to have a clear line-of-sight and thereby an unobstructed path between n' and $parent(n)$. To do this, first a line is defined from $parent(n)$ to n'. Hereby the origin of the used coordinate system is defined as the center of the node $parent(n)$ and one unit in the coordinate system equals the edge length of one cell grid node. This line can therefore be defined as $\boldsymbol{l(t) = t * \vec{n'}}$, where $\vec{n'}$ is the local position of the node n' relative to $parent(n)$. Next all values t_i, which result in an intersection of l with the walls, edges, or corners of cell nodes are computed one axis after the other. Figure 25i shows this in a simplified version by using the example of a two dimensional cell-grid search graph. The orange markers t_1, t_2, and t_3 show the intersection points of l with the vertical lines of the cell grid. The green markers t_4 and t_5 the intersection points with the horizontal lines. To reconstruct the cell node index which needs to be checked based on t_i, all t_i-values are offset by a small value Δt as seen in figure 25ii. Now each value t_i can clearly point towards one node in the cell grid.



(i) (ii)

Figure 25: Determining intersection of the search graph with the line from $parent(n)$ to n'

Based on these offset t_i-values all nodes can be identified which lie on the line from $parent(n)$ to n' and therefore need to be checked whether they are free or not. These nodes are marked grey in figure 26. Taking a closer look at this figure it can be seen that l only travels very briefly through the nodes identified by t_1 and t_5 and is very close to the corner of an adjacent, not-checked node at this time. If the AI agent moving along the calculated path is very small in comparison to the cell nodes, this distance is enough to prevent collision, even if the node marked with an "x" would be blocked. If this is not the case, though, the agent might collide with obstacles, even if the line-of-sight check returned true.

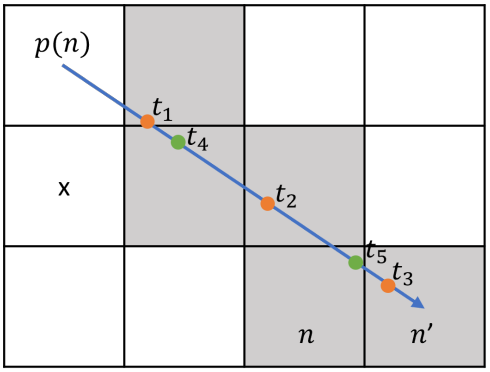
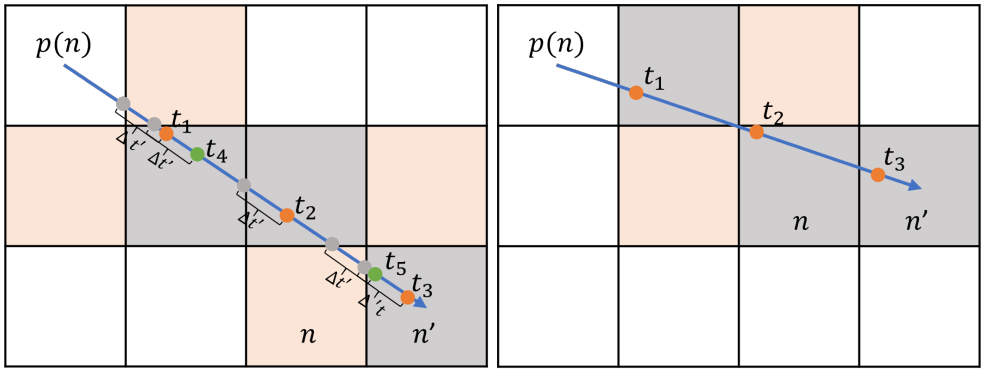


Figure 26: Checked nodes in the line-of-sight

To avoid this issue, Δt can be increased in order to offset points like t_1 and t_5 so far they are already in the next node. This way nodes with very short travel times are skipped. Instead a pattern appears which features nodes that are diagonal to each other as can be seen in figure 27i. Each time two consecutively checked nodes are diagonal to each other additional nodes need to be checked. These nodes are marked red in figure 27. Diagonal checked nodes also appear when l passes exactly through an edge or corner of a thee dimensional node as can be seen in figure 27ii. Here, too, the size of the AI agent has to be considered and additional adjacent nodes are checked.



(i) (ii)

Figure 27: Checked nodes in the line-of-sight with consideration to the AI agent's size

Given this additional work needed for diagonal nodes, the case of all diagonal nodes is handled separately. If the vector from $parent(n)$ to n' passes through all nodes diagonally, the step of calculating t_i-values is skipped. Instead the center point of all nodes on the line-of-sight are calculated directly by stepping into the signed direction l until n' is reached. Additionally adjacent nodes are checked using the same approach as in the examples described above.

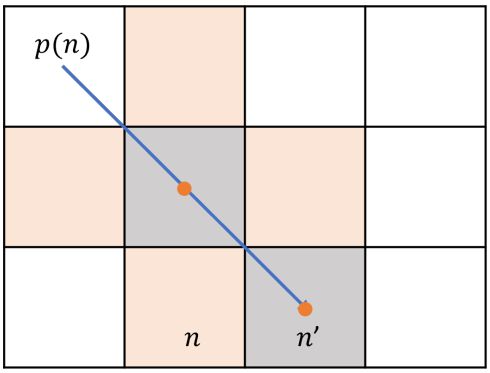
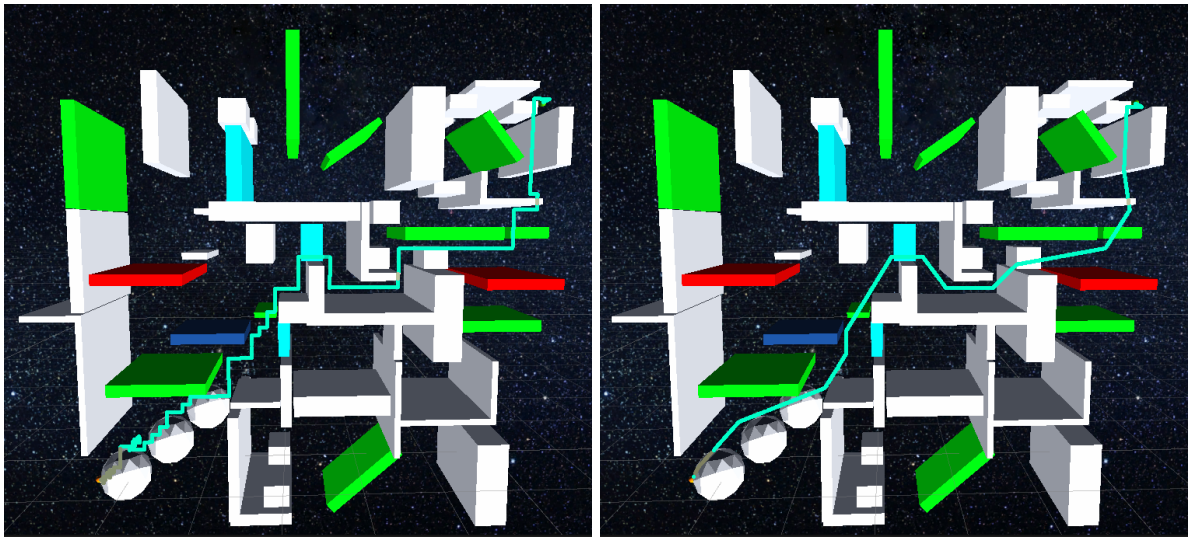


Figure 28: Checked nodes in fully diagonal line-of-sight check

If none of the checked nodes is blocked, the line-of-sight check returns true and $parent(n)$ can be set as the parent of n'. Then the correct path cost as well as g-value are determined for calculating the f-value of n' (see line 33–37 in listing 26).

Figure 29 shows the different shortest paths found by A* on the left and Theta* on the right. As expected the path found by Theta* features longer stretches of straight lines and fewer sharp turns, than the one of A*. This is due to the fact that Theta* is not restricted to movement along the axis of the search graph but can bend around all angles. This gives the paths of this algorithm a more natural and organic look than the ones of A*, which is preferable in games.

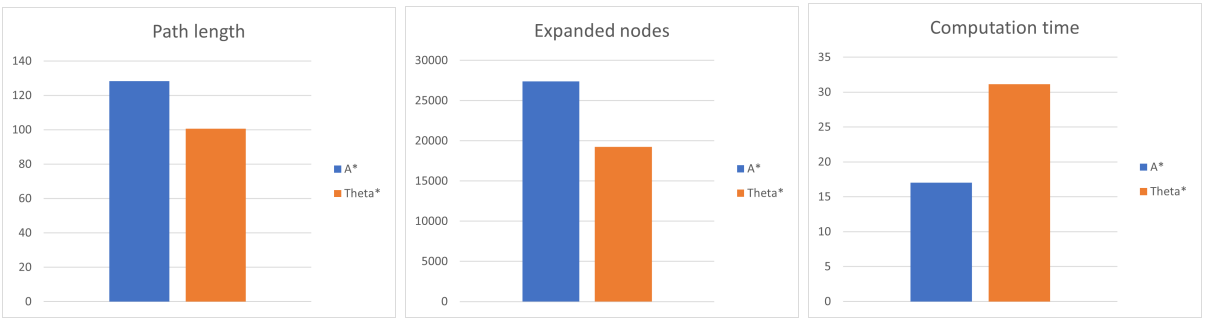


(i) A* (ii) Theta*

Figure 29: Shortest paths found by A* (left) and Theta* (right) using a cell grid and $\varepsilon = 1$

To conclude this chapter the performance of this optimised version of Theta* is analysed. The main issue with the previous implementation of Theta* was the fact that Unity physics were required for the line-of-sight checks. This not only made them quite expensive, but also prevented the algorithm to be implemented using the background thread. This issue was successfully solved in this optimised version of Theta*. Using the approach of conducting line-of-sight checks described in this chapter it is possible to run Theta* in a job parallel to the main thread. In combination with the Burst compiler the performance of Theta* was improved massively. Using a cell grid search graph with an inflation of $\varepsilon = 1$ and the setup of the test environment described in 3.1.1 featuring a static AI agent and a dynamic destination and obstacles, the algorithm used to take on average about 400 ms to compute (see figure 11v). In the optimised implementation using the same setup Theta* on average takes about 31.13 ms to compute (see figure 30iii).

To put these results of the optimised implementation of Theta* into context, the algorithm is compared to an optimised implementation of A*. Using the same test setup as before figure 30 shows the average path length, number of expanded nodes, and computation time of A* and Theta*. Because the calculation of the path length requires extra computation time, the measuring of the path length is done in a separate run than the measuring of the computation time and number of expanded nodes. As expected Theta* produces shorter paths than A* due to its ability to bend around all angles (see figure 30i). It also expands fewer nodes than A* (see figure 30ii), but due to the rather expensive line-of-sight checks on average requires a longer computation time than A* (see figure 30iii). As the difference in computation time is not too immense, though, Theta* is still competitive to A*, especially because it produces shorter and more natural looking paths.



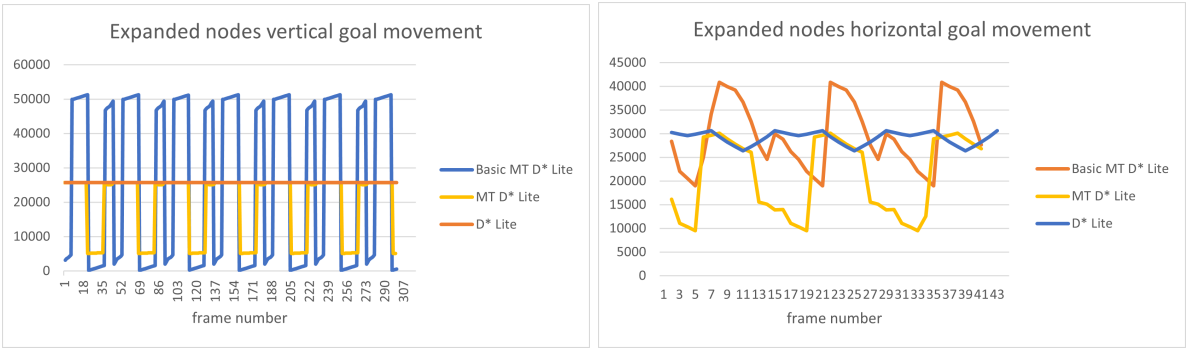
(i) (ii) (iii)

Figure 30: Comparison of the average path length (in Uu), number of expanded Nodes (in ms), and computation time using a cell grid and $\varepsilon = 1$

5.1.4 (Basic) Moving Target D* Lite

Optimising D* Lite by using (Basic) MT-D* Lite instead is implemented in two steps. First the new algorithm is implemented to reduce the number of expanded nodes and in a second step it is implemented using Unity's C# Job System and Burst compiler to reduce the runtime of the algorithm.

Reusing a lot of the implemented functionality of D* Lite from the previous bachelor thesis, (Basic) MT-D* Lite is first implemented using a backwards direction. Figure 31 compares the number of expanded nodes of D* Lite, Basic MT-D* Lite, and MT-D* Lite for a static environment and AI agent over multiple frames, with a vertically moving destination in 31i and a horizontally moving destination in 31ii. All algorithms are run in a cell grid based search graph with an inflation factor of $\varepsilon = 1$.



(i) (ii)

Figure 31: Expanded nodes in backwards directed dynamic algorithms over multiple frames using a cell grid and $\varepsilon = 1$

As D* Lite has to fully reset each time the goal node changes, its number of expanded nodes can be seen as the number of nodes any of the algorithms would need to expand if they were to reset, instead of reusing data from their previous run. It can be seen that MT-D* Lite in a worst case expands as many nodes as D* Lite, whereas Basic MT-D* Lite expands about twice as many nodes as D* Lite in a worst case. Comparing the high- and low-points in the graphs with the destination's movement, it can be observed that (Basic) MT-D* Lite performs very well when the destination is moving towards the AI agent along, or in proximity to, the calculated shortest path. When the destination is moving away from the AI agent, though, Basic MT-D* Lite perform worse than if it would fully reset, and MT-D* Lite expands as many nodes as if it would reset. This is easily explainable on the example of MT-D* Lite. When the destination moves away from the start node, the subtree of nodes which originates at the new goal node is rather small. This way the deleted list, containing all nodes in the search graph except the ones in the subtree, will be almost the complete search graph and thereby almost all nodes will be reset.

Based on this knowledge it makes sense to revert the algorithm's direction, as it is also implemented in the paper of Xiaoxun Sun, William Yeoh, and Sven Koenig [SYK10]. Since the AI agent will always move along the computed shortest path, the subtree of the new start node will be a considerably larger part of the search tree and the deleted list smaller.

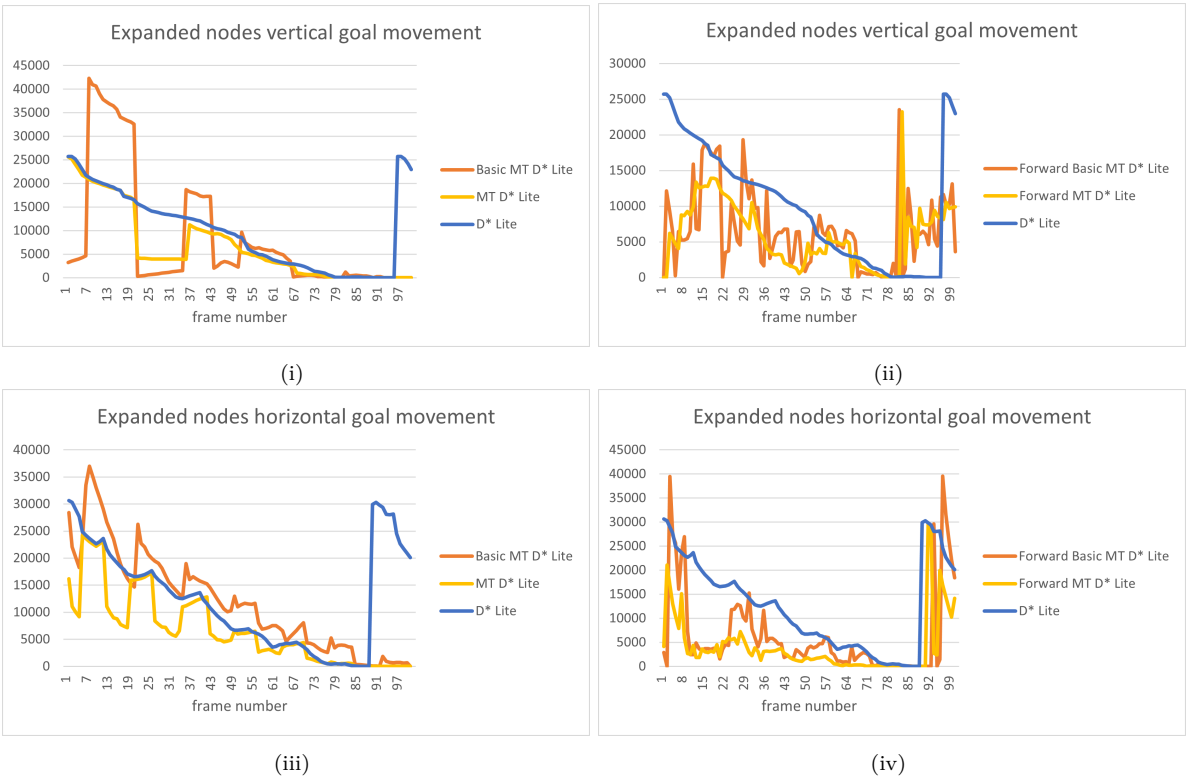


Figure 32: Comparison of expanded nodes in forwards (right column) and backwards (left column) directed dynamic algorithms over a number of frames using a cell grid and $\varepsilon = 1$

Figure 32 shows the number of expanded nodes over the course of the AI agent moving from its start position to the destination using vertical goal movement in the top row and horizontal goal movement in the bottom row, as well as either the backwards directed algorithms in the left column and forwards directed algorithms in the right column. The only exception is D* Lite, which is always backwards directed. The very high peaks towards the end of the x-Axis is the AI resetting to its start position after having reached the destination. It can be seen that the forward directed versions of (Basic) MT-D* Lite generally expand fewer nodes than their backwards directed versions and D* Lite. Looking at the forward directed (Basic) MT-D* Lite versions figure 32iv shows that in the environment with a horizontally moving destination MT-D* Lite mostly expands fewer nodes than (Basic) MT-D* Lite, whereas figure 32ii does not conclusively show whether Basic MT-D* Lite or MT-D* Lite is more suited to minimise the number of expanded nodes.

Before starting to transfer the implementation to use Unity's C# Job System and Burst compiler, one additional optimisation is added to (Basic) MT-D* Lite. Keeping the basic previous implemented logic of D* Lite, (Basic) MT-D* Lite expands an over-consistent node by setting the node's g-value equal to its rhs-value, removing the now consistent node from the open list, and updating the rhs-values of all its successors. Hereby, each neighbour's rhs-value is calculated as:

$$rhs(n) = \min_{n' \in \text{Pred}(n)}(g(n') + c(n, n'))$$

This implementation requires two foreach-loops. The first iterating over all neighbours of the over-consistent node and the second iterating over all predecessors of each neighbour node (see pseudocode in listing 22 in the Appendix). In an optimised version the rhs-values of an over-consistent node's (n) neighbour (n') can be calculated as:

$$rhs(n') = \min(rhs(n'), g(n) + c(n, n'))$$

This optimised implementation only requires one foreach-loop over the over-consistent node's neighbours and is therefore a lot more performant. For pseudocode see listing 23 in the Appendix. This implementation of updating an over-consistent node's neighbours can also be seen in the code shown in "Moving Target D* Lite" [SYK10] in figure 2 line 30–33. Figure 33 shows the improvement this optimisation has on the average computation time of MT-D* Lite. The initial implementation of MT-D* Lite has an average computation time of about 640 ms in a static environment where the algorithm fully resets each run. With the described optimised updating of over-consistent node, MT-D* Lite only has an average runtime of about 291 ms given the same scenario, which is less than half of the un-optimised runtime.

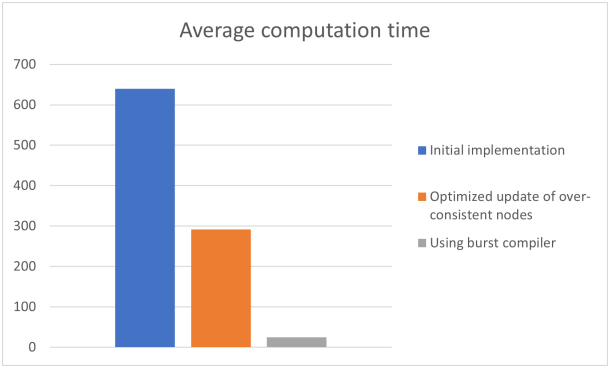


Figure 33: Average computation time (in ms) of MT-D* Lite in a static environment with resetting the algorithm every run

In the next step the implementation of (Basic) MT-D* Lite is optimised using Unity's C# Job System and Burst compiler. To be able to use these systems, (Basic) MT-D* Lite is adapted to the changes described in 5.1.1, such as referencing and storing nodes only by their index in the passed *nodes* NativeArray<CellNode> and getting a node's neighbour-indices at their position in the 3x2 matrix. Additionally, a custom comparison function is implemented to be used when sorting the NativeHeap of the open list. This comparison function considers the primary keys of nodes as well as their secondary keys. This way the expensive creation of a list of all nodes with the same primary key and the sorting of that list based on the nodes' secondary key is eliminated.

As an additional adaptation to the Burst compiler an optimisation of the computation of a node's *rhs*-value is implemented. In its previous implementation (Basic) MT-D* Lite had to iterate over all predecessors of a node to determine the minimum *rhs*-value, similar to the code shown in listing 22 line 5–19. Assuming the path cost from a node n to its neighbour n' is the same as from n' to n, all possible *rhs*-values can be calculated simultaneously as seen in the pseudocode in listing 9 in line 11. This eliminates one foreach-loop and one *if*-condition and makes use of the math-library, which works extremely performant with the Burst compiler. A similar approach was taken to optimise the updating of over-consistent node's neighbours' *rhs*-values further (see pseudocode in listing 24 in the Appendix).

```
1  private void UpdateNodeParallel(int nodeIndex)
2  {
3      CellNode tempNode = nodes[nodeIndex];
4      int3x2 predecessors = tempNode.predecessors;
5      double3x2 gValueOfPred = new double3x2(nodes[pred[0][0]].gValue,
6                                             nodes[pred[1][0]].gValue,
7                                             nodes[pred[0][1]].gValue,
8                                             nodes[pred[1][1]].gValue,
9                                             nodes[pred[0][2]].gValue,
10                                            nodes[pred[1][2]].gValue);
11     double3x2 rhsValues = gValueOfPredecessors + tempNode.pathCosts;
12     double rhs = math.min(math.cmin(rhsValues.c0), math.cmin(rhsValues.c1));
13     bool3x2 isParentNode = rhsValues == rhsValue;
14     isParentNode &= rhsValue < math.INFINITY_DBL;
15     int3 parentsColumn0 = math.select(new int3(-1), pred.c0, isParentNode.c0);
16     int3 parentsColumn1 = math.select(new int3(-1), pred.c1, isParentNode.c1);
17     int parent = math.max(math.cmax(parentsColumn0), math.cmax(parentsColumn1));
18
19     tempNode.parentNode = parent;
20     tempNode.rhsValue = rhsValue;
21     nodes[nodeIndex] = tempNode;
22 }
```

Listing 9: Optimised update of nodes' *rhs*-value

Having implemented (Basic) MT-D* Lite with all its optimisations using Unity's C# Job System and Burst compiler, the computation time and number of expanded nodes were measured in a fully dynamic environment. Figure 34 shows the results of an exemplary run of the AI from its start to the destination in a scenario with a vertically moving destination on the left and a horizontally moving destination on the right.

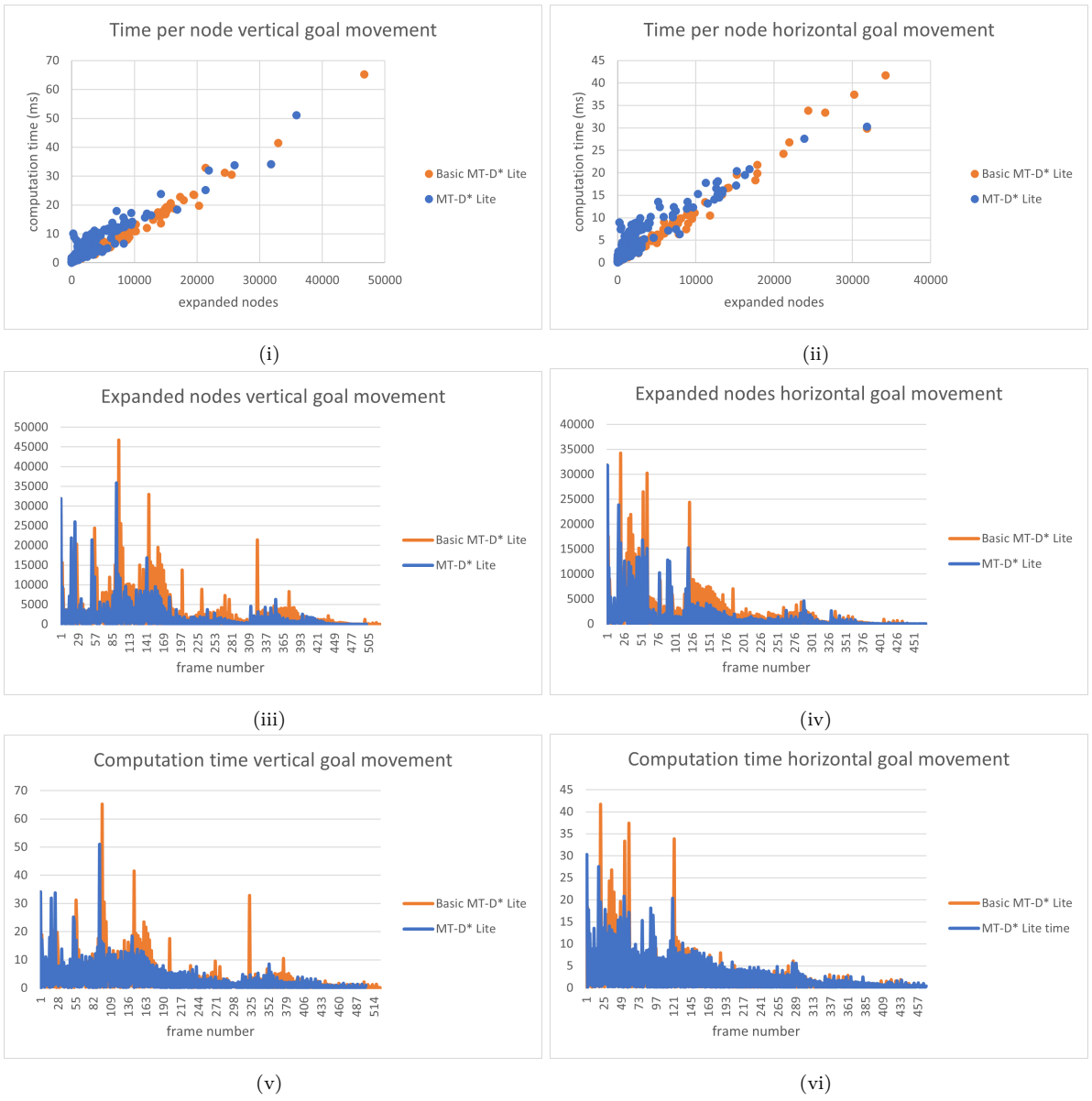
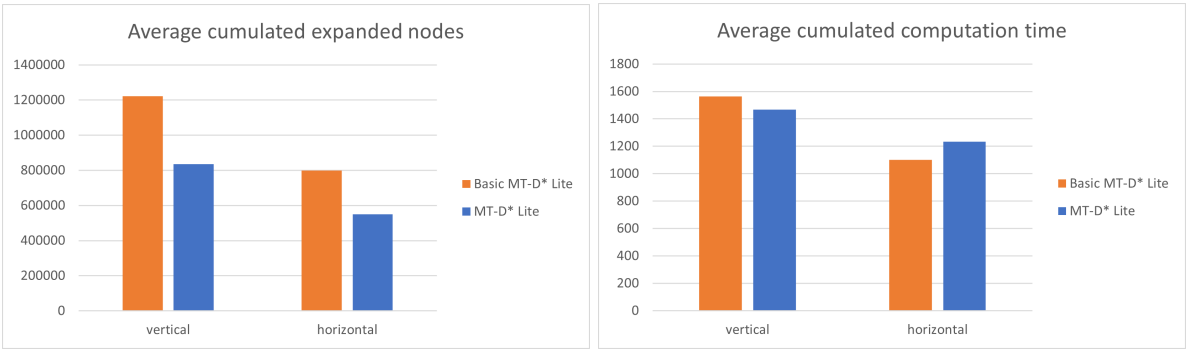


Figure 34: Comparison of expanded nodes and computation time (in ms) in a fully dynamic environment for a full run of the AI agent from start to destination using a cell grid and $\varepsilon = 1$

Figure 34i and 34ii show that Basic MT-D* Lite generally is slightly faster than MT-D* Lite relative to the number of expanded nodes. As expected figure 34iii and 34iv show that Basic MT-D* Lite expands more nodes than MT-D* Lite.

Lastly figure 34v and 34vi show that MT-D* Lite generally has a lower computation time than Basic MT-D* Lite when running the algorithm with the vertically moving destination. When testing the algorithm with the horizontally moving destination, though, MT-D* Lite is only faster at the beginning, but gets slightly slower than Basic MT-D* Lite towards the end of the run.

This trend is confirmed when averaging the total number of expanded nodes and total computation time over multiple runs of the AI agent from its starting position to the destination as seen in figure 35.

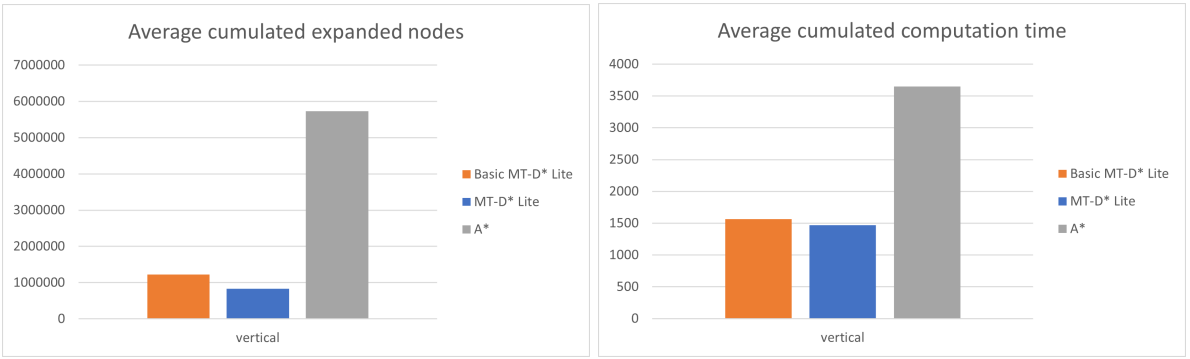


(i) (ii)

Figure 35: Average cumulated number of expanded nodes and computation time (in ms) in multiple runs of the AI agent from start to destination using a cell grid and $\varepsilon = 1$

In conclusion it can be said that implementing (Basic) MT-D* Lite as a replacement of D* Lite and utilizing Unity's C# Job System in combination with the Burst compiler results in a great improvement. Due to the reverse direction of MT-D* Lite to D* Lite it does not make sense to compare the two algorithms' values in the same test setup used in chapter 3.1 as the difference in direction impacts the number of expanded nodes greatly when using a static AI agent. Instead a look is taken at the end results found in the bachelor thesis [Kra19, p. 76] on page 76 in figure 52ii. Here the average total number of expanded nodes and computation time of D* Lite in a fully dynamic environment is shown. It can be seen that about 1,444,702 nodes got expanded in total and about 23,978.5 ms of computation time was required in total. Figure 35 shows that these numbers are significantly lower for the optimised versions of (Basic) MT-D* Lite. For example in an environment with a vertically moving destination (which is the same movement the destination had in the measured environment in the bachelor thesis), MT-D* Lite expands about 835,078 nodes, which are 609,627 nodes less. Looking at the required average total computation time the difference is even more significant. In the same test, MT-D* Lite requires with 1468 ms only about 16.3 times less computation time than D* Lite in the previous implementation. To additionally highlight the strength of the dynamic algorithms in comparison to a non-dynamic algorithm, figure 36 shows the average total number of expanded nodes and

computation time for (Basic) MT-D* Lite and A*, using the same test setup as for figure 35 and a vertically moving destination. The non-dynamic pathfinding algorithm A* expands more than four times as many nodes as the dynamic algorithm Basic MT-D* Lite and has more than twice its computation time. Here, again, it is worth taking a look at figure 52ii on page 76 of the bachelor thesis [Kra19, p. 76]. Where D* Lite had the downside of resetting regularly and thereby having a higher computation time than A*, (Basic) MT-D* Lite does not have to reset, allowing it to have a significantly lower computation than A*.



(i) (ii)

Figure 36: Average cumulated number of expanded nodes and computation time (in ms) in multiple runs of the AI agent from start to destination with a vertically moving destination using a cell grid and $\varepsilon = 1$

5.2 Hybrid search graph

When working with pathfinding algorithms not only the used algorithm and its performance impacts the performance and efficiency of the pathfinding system, but also the used search graph plays a vital role. Therefore, after having optimised the performance of the pathfinding algorithm in the last chapters, the next chapters will focus on optimising the search graph.

The search graph impacts the overall pathfinding system mainly in two aspects. First, the search graph impacts the runtime of the pathfinding algorithms dependent on its number of nodes. The fewer nodes a search graph has, the fewer nodes the pathfinding algorithm has to expand, the lower its computation time is. Hereby, it has to be kept in mind, though, that too few nodes can impact the pathfinding algorithm negatively by producing longer paths.

The second aspect is the time the search graph needs to update all its nodes to the current state of the environment. This is especially significant when performing pathfinding in a dynamic environment. In a dynamic environment not only the pathfinding algorithm has to re-run or update each time the environment changes, but first the search graph has to be updated and adjusted to the dynamic obstacles. Only when the search graph represents the current state of the environment correctly the pathfinding algorithm can produce a correct and usable result.

The work done in the previous bachelor thesis [Kra19] has shown that updating the search graph often took considerably longer than running the pathfinding algorithm and was therefore the limiting factor in the update rate of computing a shortest path (see figure 50 in [Kra19, p. 75]). A result of this was a reduced success rate of the AI agent reaching its destination without colliding with moving obstacles. By the time the search graph was updated, some of the obstacles had already moved, thereby blocking the path computed based on an out-dated search graph. To solve this issue it is important to not only optimise the runtime of the pathfinding algorithms, but also to optimise the updating time of the search graph.

In the following chapters both of these aspects will be addressed and a possible optimisation solution implemented and tested.

5.2.1 Construction

The aim when constructing an optimised search graph is to reduce the number of nodes to a necessary minimum, while keeping enough nodes to produce short and natural looking paths, even without having to use any-angle algorithms. In the previous work [Kra19] the two approaches chosen were a cell grid search graph and a waypoint based search graph. Both had severe shortcomings. The cell grid search graph produced short paths and had a relatively high success rate for the AI to reach its destination. It did however have more than 57000 nodes, which increased the runtime of the algorithms and the time required for updating the search graph. The waypoint based search graph featured considerably fewer nodes, which decreased the runtime of the pathfinding algorithms significantly, but showed a very low success rates for the AI to reach the destination without colliding with one of the dynamic obstacles. This was caused by multiple design flaws in the setup of the nodes in this search graph. Waypoints were placed around all dynamic and static objects with a little distance to the colliders to account for the movement of the obstacles and the radius of the AI agent. Whenever dynamic obstacles moved, their waypoints would move along with them. This caused the nodes to re-determine their so-called "indirect" neighbours, which were always the closest node assigned to a different object than the current node. This re-assigning of neighbours did not only take a considerable amount of time in the updating process of the search graph, but also forced the dynamic algorithms to reset each time the environment had changed, reducing their overall effectiveness. In consequence, this change of neighbours was the first big issue which impacted the success rate negatively. The second issue was the positioning of the waypoints in too close proximity to the dynamic obstacles without any additional nodes where the AI could move towards to. If the AI was on the way between two node adjacent to a dynamic obstacle and that obstacle moved towards it, the AI agent could only adjust its trajectory towards another node, which was still close to the obstacle. There was no option to "take a step away" from the obstacle to gain some distance to it.

Observing the AI agent in this scenario it became clear that missing the option to "take a step away" caused a lot of collisions. Additionally, the design of placing nodes only along obstacles was flawed because it caused the AI agent to primarily move along walls instead of through the middle of large free spaces. This did not only result in longer paths, but also made the paths look less smooth and natural.

Learning from the results of the previously used search graphs, it was decided to design a new search graph, which combines the best features of both previous search graphs whilst avoiding their respective downsides. This "Hybrid Search Graph" should feature few nodes, constant neighbours, and enough nodes around dynamic obstacles to give the AI space to avoid obstacles that move towards it.

The hybrid search graph is based on a cell grid layout of nodes, where all unnecessary nodes are eliminated. In other words, each node does not necessarily get its directly adjacent nodes assigned as neighbours, but only "used" nodes are considered when assigning neighbours. This way all the mathematical upsides of a grid based approach are kept, for example that based on each node's local position its index can be determined and vice versa. Also the method of line-of-sight checks described in chapter 5.1.3 can be reused. The search graph is baked in editor time once before running the pathfinding system and then loaded each time the application using the pathfinding system starts. This approach saves performance when starting the pathfinding process and allows for more freedom in the computation during the baking process of the search graph, as it is not performance relevant. The whole process of baking the hybrid search graph is fully automated, so the developer only has to press one button to trigger and one to complete the re-computation of the search graph after the environment was changed during the development process.

Constructing the hybrid search graph is done in four steps as described in the following chapters. Hereby it is assumed that either only one AI agent is travelling through the environment at any given time, or that multiple agents ignore collisions between each other.

5.2.1.1 Used nodes

The first step in creating the hybrid search graph is to bake a consistent set of nodes of the type *HybridNode*. Once generated this set does not change over the course of the pathfinding process. Starting with all the nodes generated by the cell grid search graph, the nodes are then divided into different categories and flagged accordingly.

Static

Static nodes are constant, never changing nodes which are positioned around static obstacles in the environment. Because the shortest path will only have to change its direction at edges of obstacles, only nodes at the edges of a static obstacle are flagged as *static*. Using colliders and layers it is analysed which nodes are blocked by static obstacles and using this information edge nodes are determined and flagged accordingly. This approach to reduce the number of nodes to only the necessary ones, by only using the nodes around edges is similar to the line of thought used in the previously used waypoint based search graph. *Static* nodes are always free and do not need to be checked for blockage when updating the search graph. Figure 37 shows *static* nodes in yellow.

Dynamic

Dynamic nodes are nodes that can be blocked or free during the pathfinding process. They fill the areas in which dynamic obstacles can move. *Dynamic* nodes are the only nodes which need to be checked for blockage and updated during the updating process of the search graph. As the obstacles in the environment used in this thesis move on predefined and repetitive paths and patterns, *dynamic* nodes are identified by observing the dynamic environment over a period of time. Any node which is blocked by a dynamic obstacle at any time during this process is flagged as *dynamic*. Other environments with less repetitive movement patterns might require different methods of determining *dynamic* nodes. This could for example be done by placing colliders in the scene around areas where dynamic obstacles can move. In figure 37 *dynamic* nodes are marked in magenta.

Dynamic neighbour

To determine *dynamic neighbour* nodes, all surrounding nodes of a *dynamic* node including neighbours on all six sides as well as nodes adjacent to all edges and corners, are marked as a *dynamic neighbour* during the baking process. This is done to create a buffer of nodes around the *dynamic* nodes to later give the AI space to move around dynamic obstacles and away from approaching obstacles. *Dynamic neighbour* nodes are similar to *static* nodes in the sense that they are always free and do not need to be checked during the updating of the search graph. In figure 37 *dynamic neighbour* nodes are marked in cyan.

All nodes marked as either *static*, *dynamic*, or *dynamic neighbour* are herefrom referred to as “used” nodes, whereas all other nodes are “unused” and ignored in the following setup of the hybrid search graph. The hybrid search graph seen in figure 37 features 21259 used nodes.

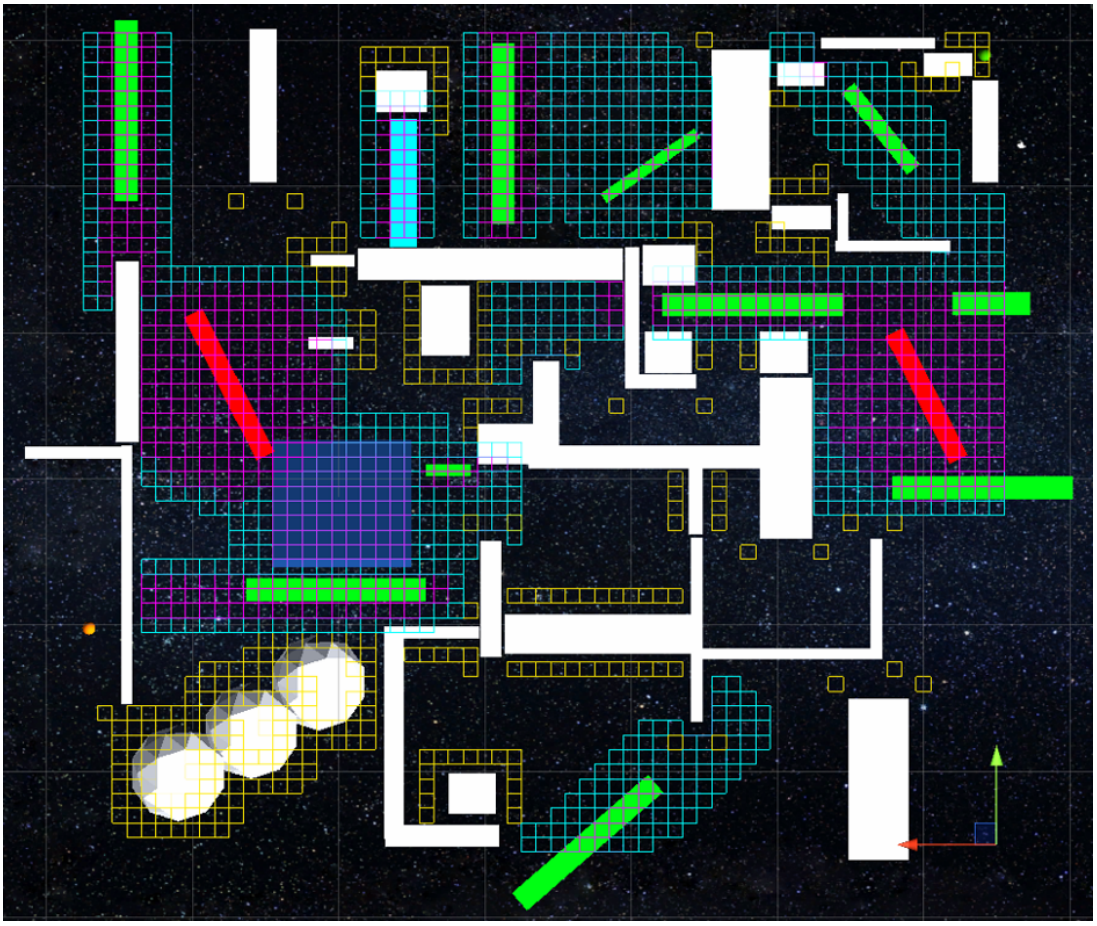


Figure 37: Orthographic front view of the final baked hybrid search graph. *Static* nodes in yellow, *dynamic* nodes in magenta, *dynamic neighbour* nodes in cyan.

5.2.1.2 Neighbours

When all used nodes are flagged correctly and thereby the set of nodes has been defined, all neighbours and their path costs are determined for these used nodes. As with the cell grid search graph all neighbours are stored by their indices in intNxM matrices. Each node is defined to have six "side neighbours" around its six sides stored in an int3x2 matrix, 12 "edge neighbours" at its edges stored in an int4x3 matrix, and eight "vertex neighbours" around its vertices stored in an int4x2 matrix. Hereby each node gets one set of ordered and one of unordered neighbour nodes for side, edge, and vertex neighbours. Neighbours of *dynamic* nodes are determined differently to those of *static* and *dynamic neighbour* nodes, as will be described in the following paragraphs.

Dynamic

Per definition *dynamic* nodes are enclosed by either other *dynamic* nodes or *dynamic neighbour* nodes. Therefore, determining the side, edge, and vertex neighbours of *dynamic* nodes is fairly easy. Each adjacent node on all sides, edges, and vertices is evaluated and the results entered

into the according neighbour matrix. For each node not blocked by a static obstacle the node's index is entered into the matrix. For each node blocked by a static obstacle -1 is written into the matrix at the according position. As *dynamic* nodes can change their free/blocked status during the updating of the search graph, the path costs to their neighbours also change frequently. Therefore, it is not necessary to pre-calculate the path costs during the creation of the search graph.

Static and dynamic neighbour

Determining the neighbours of *static* and *dynamic neighbour* nodes is slightly more complex. Both types of nodes potentially do not have adjacent used nodes in all directions and their neighbours may lay several grid positions away.

First each node's **side neighbours** are determined. The connections between nodes and their side neighbours are shown in grey in figure 38.

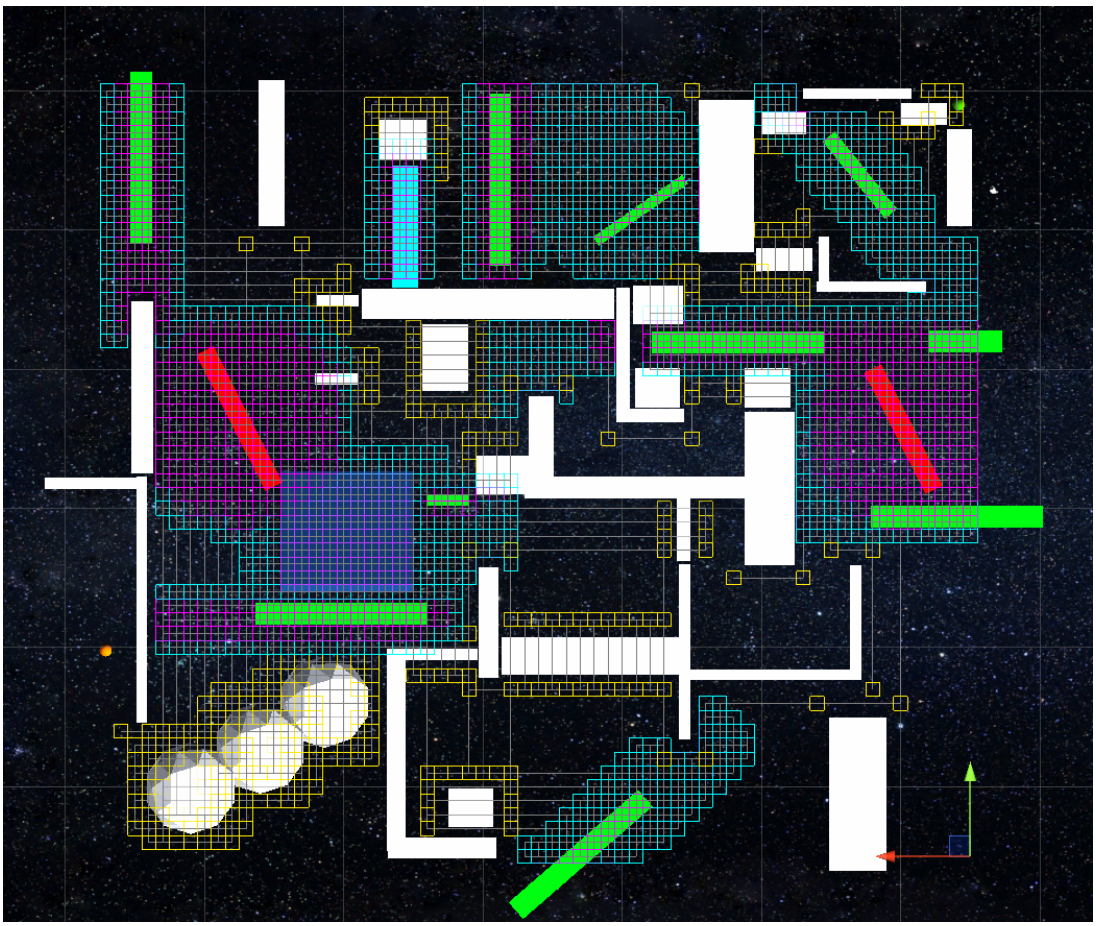


Figure 38: Orthographic front view of the final baked hybrid search graph with connections to side neighbours shown in grey

Listing 10 shows pseudocode for setting a node's side neighbours based on the first used node found in each side direction. The *normalizedDirections* array used for this (see line 3 in listing 10) contains all normalized directions along the positive and negative x-, y-, and z-axis and is the same as used for Jump Point Search described in chapter 5.1.2. The *GetFirstUsedNode* function called in line 6 steps through the cell grid in the passed direction *normalizedDirections[i]* until a used node or a blocked node is reached. It then either returns the index of the used node or -1 if a blocked node is encountered. When talking about free and blocked nodes in this paragraph this only relates to whether they are blocked by static obstacles or not. Dynamic obstacles are not taken into account at this point to determine the free/blocked status of a node.

```
1  private void UpdateSideNeighbours(int index)
2  {
3      int[] orderedNeighbours = new int[normalizedDirections.Length];
4      for (int i = 0; i < normalizedDirections.Length; i++)
5      {
6          orderedNeighbours[i] = GetFirstUsedNode(index, normalizedDirections[i]);
7      }
8      ...
9      //fill unordered neighbours matrix based on ordered neighbours array
10     //determine path costs based on distance between the node and its neighbours
11 }
```

Listing 10: *UpdateSideNeighbours* function

Defining **edge neighbours** of *static* and *dynamic neighbour* nodes is similar to determining side neighbours as seen in listing 11. The *normalizedEdgeDirections* array used in this function is similar to the *normalizedDirections* array. It contains all possible relative directions from a node to all its edges. Iterating over this array each direction is first split into its individual direction components (line 9 in listing 11) providing a primary and a secondary direction. For example, the *normalizedDirections* vector $(1, 1, 0)$ is split in its primary direction $(1, 0, 0)$ and its secondary direction $(0, 1, 0)$. Next it is checked whether any adjacent node in the primary or secondary direction is blocked (line 12–14). If this is the case -1 is written into the *orderedEdgesArray* and this *normalizedEdgeDirection* is skipped (line 15–19). Only if adjacent nodes in the primary and secondary direction are free, the *GetFirstUsedNode* function is called in line 21, which returns the index of the identified neighbour for this edge direction or -1 if no neighbour could be found. Figure 39 shows the connections of each node to its edge neighbours in green. As this is an orthographic front view of the environment, edge neighbour connections sometimes look like side neighbours connections, due to the fact that they connect two nodes which are offset only on the z-Axis, which is not visible in the orthographic view.

```
1  private void UpdateEdgeNeighbours(int index)
2  {
3      int3 primaryDirection, secondaryDirection;
4      int[] orderedEdgesArray = new int[normalizedEdgeDirections.Length];
5      int stepCount = 0;
6
7      for (int i = 0; i < normalizedEdgeDirections.Length; i++)
8      {
9          GetSplitDirections(normalizedEdgeDirections[i],
10                             out primaryDirection,
11                             out secondaryDirection);
12         int3 splitEdgeNeighbours = GetFreeSplitEdgeNeighbours(index,
13                                         primaryDirection,
14                                         secondaryDirection);
15         if (math.any(splitEdgeNeighbours == -1))
16         {
17             orderedEdgesArray[i] = -1;
18             continue;
19         }
20
21         orderedEdgesArray[i] = GetFirstUsedNode(index,
22                                                 primaryDirection,
23                                                 secondaryDirection,
24                                                 ref stepCount);
25     }
26
27     ...
28     //fill unordered edge neighbours matrix based on ordered array
29     //determine path costs based on distance between the node and its edge
       neighbours
30 }
```

Listing 11: *UpdateEdgeNeighbours* function

Listing 12 shows the *GetFirstUsedNode* function with the overload for two direction vectors in more detail. The function recursively steps along the primary direction (line 5 and 6) until the end of the search space or a blocked node is reached (line 7–10). At every step the function checks the nodes along the secondary direction using the same *GetFirstUsedNode* function with an overload for only one direction which is also used in the determination of nodes side neighbours (see line 6 in listing 10) and is detailed in listing 27 in the Appendix. If a used node is found in the secondary direction this node's index is returned as the edge neighbour (line 16). Whilst checking the secondary direction for used or blocked nodes, the *GetFirstUsedNode* function counts its iteration steps until it returns. This count is used in line 19–22 to catch an edge case. If the first node stepped into in the secondary direction is blocked then no edge neighbour can

be determined, because the line-of-sight between the original node and any later determined used node in the secondary direction would most likely be blocked by this blocked node. An exception to this might occur in an immensely large search grid, but this is not the case in the environment used in this thesis and, therefore, this possibility is omitted at this point.

```
private int GetFirstUsedNode(int index, int3 primDir, int3 secDir,
                             ref int secondaryStepCount)
{
    secondaryStepCount++;
    index = (allNodes[index].localPosition + primDir).PositionToIndex(
            searchSpace);
    if (index == -1 || !allNodes[index].isFree)
    {
        return -1;
    }

    int stepCount = 0;
    int usedNodeInSecondaryDir = GetFirstUsedNode(index, secDir, ref stepCount);
    if (usedNodeInSecondaryDir != -1)
    {
        return usedNodeInSecondaryDir;
    }

    if (stepCount <= 1)
    {
        return -1;
    }

    return GetFirstUsedNode(index, primDir, secDir, ref secondaryStepCount);
}
```

Listing 12: *GetFirstUsedNode* function with overload for a primary and secondary direction

Lastly **vertex neighbours** are determined. The approach for this is very similar to the one taken to determine edge neighbours. The main difference is that the *normalizedEdgeDirection* was only split in a primary and secondary direction, whereas the *normalizedVertexDirection* is split into a tertiary direction as well. The *GetFirstUsedNode* function used for this contains overloads for three directional vectors. The function steps along its primary direction until reaching the end of the search graph or a blocked node. At each step the *GetFirstUsedNode* function seen in listing 12 is called passing the secondary and tertiary direction. In other words, each time a step is taken in the primary direction, the secondary direction has to be checked, and each time a step in then taken into the secondary direction, the tertiary direction gets checked. When a used node is found in the tertiary direction its index is returned and written into the according matrix. Otherwise -1 is returned and stored. Vertex neighbour connections are marked in white in figure 39.

As already seen in the comments in line 10 in listing 10 and line 28 in listing 11 the path costs to all neighbours are always calculated as the distance between the original node's position to the neighbour's node position. If no neighbour is identified (the matrix has an entry of -1) then the path cost is set to infinite.

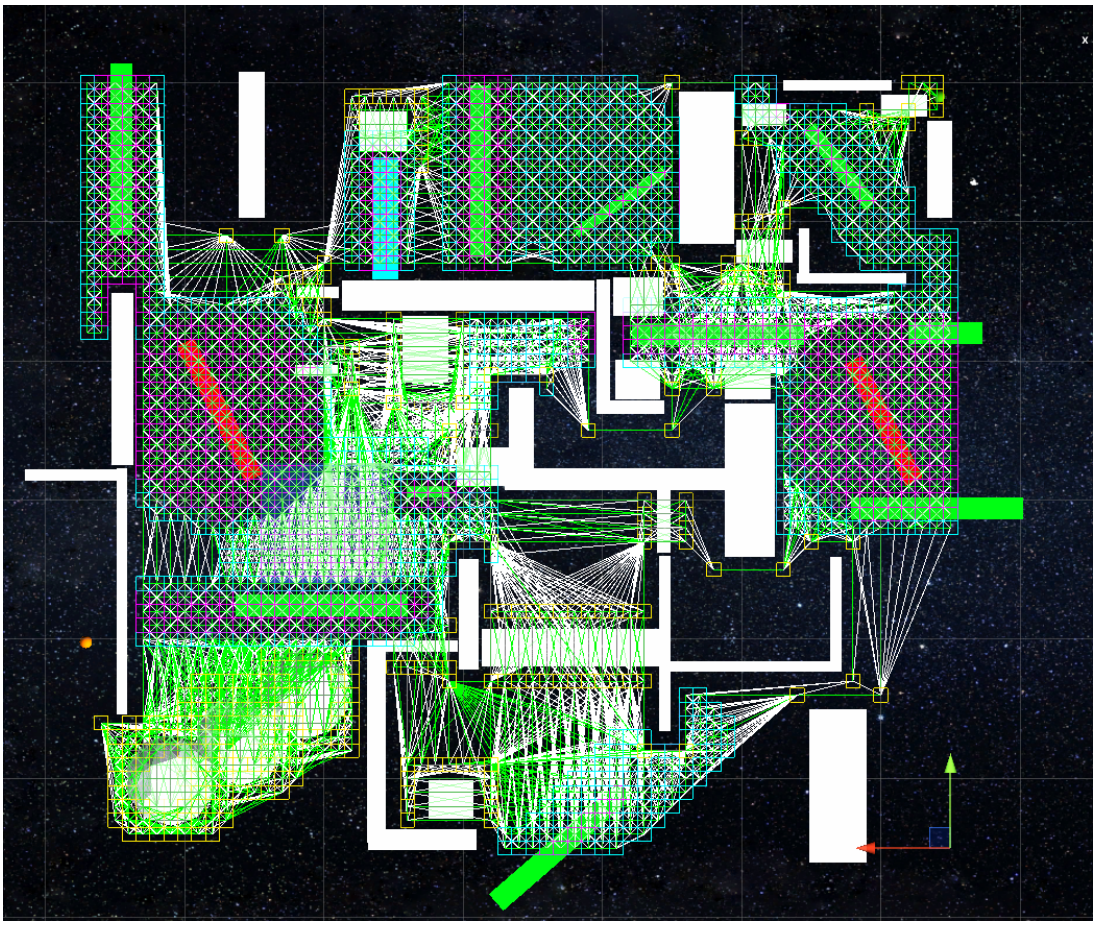


Figure 39: Orthographic front view of the final baked hybrid search graph. Side neighbours in grey, edge neighbours in green, and vertex neighbours in white.

5.2.1.3 Predecessors

After having determined the neighbours of all nodes in the hybrid search graph the next step is to determine the predecessors of all nodes. This is necessary, because the dynamic pathfinding algorithms (Basic) MT-D* Lite require predecessors to calculate the *rhs*-values of nodes. In the cell grid search graph previously used in this thesis each node's predecessors were equal to its neighbours. In the hybrid search graph this is not the case. Especially *static* nodes can bundle a large amount of paths coming from different *dynamic neighbour* nodes. In consequence they can have a large amount of predecessors. An example of this can be seen in figure 40 where the two *static* nodes shown in yellow in the center of the image show a lot of edge neighbour connections.

Some of these connections are the neighbours of the *static* node to their edge neighbours. Other lines show the connections of surrounding *dynamic neighbour* nodes to the *static* nodes as their neighbours. Therefore, these *static* nodes have more edge predecessor than edge neighbours.

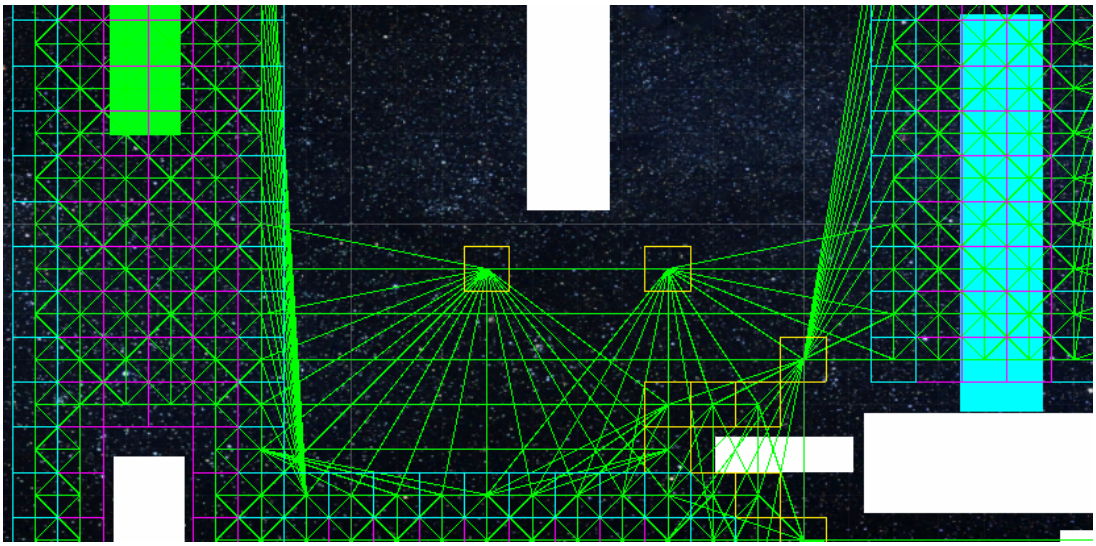


Figure 40: Zoomed in orthographic front view of the hybrid search graph with edge neighbours in green

If the job system and Burst compiler were not used, the easiest option would be to populate a list or array of predecessors for each node each time it is assigned as a neighbour to another node. As discussed previously in chapter 5.1.1 it is not possible to use any arrays, lists, dictionaries or similar data types in the node structs, though, as the nodes will be passed as a NativeArray into the pathfinding job later. Using matrices to store the data instead, as is done for the neighbours of nodes, is not a viable option for predecessors, as their number changes for each node and can be rather large. In the baked hybrid search graph seen in 39 certain *static* nodes have more than 190 predecessors. To solve this issue it was decided not to store the information of predecessors for each node separately, but to construct a *predecessors* fields of data type SerializedDictionary<int, List<int>> containing the predecessor information for all nodes. Whilst determining the neighbours of each node, the *predecessors* dictionary is populated with indices of used nodes as keys and lists of their preprocessors as values. To pass the predecessor information into the pathfinding jobs later, the dictionary then has to be cast to a variable of type NativeMultipleHashSet<int, int>. A NativeMultiHashSet is somewhat similar to a dictionary in the sense that it stores values in relation to keys. Unlike a dictionary, though, it can store multiple values to the same key. Therefore, it can store the indices of all used nodes as keys and all the indices of their predecessors as values. Listing 13 shows pseudocode for this. The reason for first collecting all predecessor information in a SerializedDictionary<int, List<int>> instead of a NativeMultipleHashSet<int, int> right away, is that it is easier to save and load a serialized dictionary, than a NativeContainer type.

```
NativeMultiHashMap<int, int> pred =
                new NativeMultiHashMap<int, int>(0, Allocator.Persistent);
foreach (int key in predecessors.Keys)
{
    foreach (int predecessor in predecessors[key])
    {
        pred.Add(key, predecessor);
    }
}
```

Listing 13: Parsing a SerializedDictionary<int, List<int>> to a NativeMultipleHashSet<int, int>

5.2.1.4 Saving the search graph

The last step in baking the hybrid search graph is to save the resulting search graph, so it can be loaded later. It was decided to save the search graph information in a Json format as a text file. To save the information of used nodes they were first copied into *HybridNodeSaveState* structs. One reason for doing so is to reduce the amount of saved data. The original *HybridNode* struct in which all nodes are represented contains a lot of data only necessary for the running of the pathfinding algorithms, but not for the construction of the search graph. The *HybridNode-SaveState* on the other hand only stores relevant data of each node necessary to reconstruct the search graph upon loading, including the node's index, neighbours, path costs, and its usage flag (*static*, *dynamic*, or *dynamic neighbour*). Another reason for using a second struct for saving used nodes is that the *HybridNode* struct consists mostly of properties. As properties are not serialized they can not be handled by Unity's JsonUtility [Tec21n] tool. The JsonUtility tool can also not handle arrays. Therefore, to transfer the *HybridNodeSaveState* array of used nodes into a Json format a simple wrapper class was implemented based on a discussion in an expert forum [Ans16]. The dictionary with the predecessor information is saved separately to the used nodes array. To transfer the *predecessors* dictionary into a Json format the MiniJSON class developed by Calvin Rien is used [Rie13].

5.2.2 Updating

As discussed at the beginning of this chapter, constructing a search graph with a balanced number of nodes is only the first step in developing an optimised search graph, which is suitable to be used in a highly dynamic three-dimensional environment. The second step is to ensure the search graph is updated fast enough to deliver an accurate representation of the current status of the changing environment. The following sections will detail the updating process of the hybrid search graph presented in the previous section. For optimal performance the aim

hereby will be to implement the updating process in such a way that as much of its functionality as possible can be performed in a job using the Burst compiler.

The goal of updating the search graph is to determine the correct path costs of every node to all its neighbours. In order to do so, the first step is to establish for each node whether it is blocked by an obstacle or not. In the previous implementation of the updating process this was achieved by using colliders and physics to determine collisions between objects and nodes. The approach of using physics to determine the free/blocked status of a node is completely discarded in this thesis for two main reasons. Firstly, using Unity's physics operations is rather expensive. The performance analysis conducted in chapter 3.2.2 shows that their use is the main performance issue when updating the search graphs. Secondly, using physics asynchronously is quite complex and restricting. As mentioned in chapter 4.1.3, Unity's C# Job System, as used in this thesis for multi-threading, does support the asynchronous use of physics in jobs. Its usage is restricted by a set of rules, though, and only applicable in certain use-cases. Therefore, it was decided to take a different approach on implementing collision detection between objects in this thesis, which can run in a job using the Burst compiler and does not require physics. For this the technique of the "Separating Axis Theorem" (SAT) as described in chapter 2.4 is used to determine whether a node is blocked by an obstacle or not.

5.2.2.1 Preparing vertices, edges, and normals for SAT

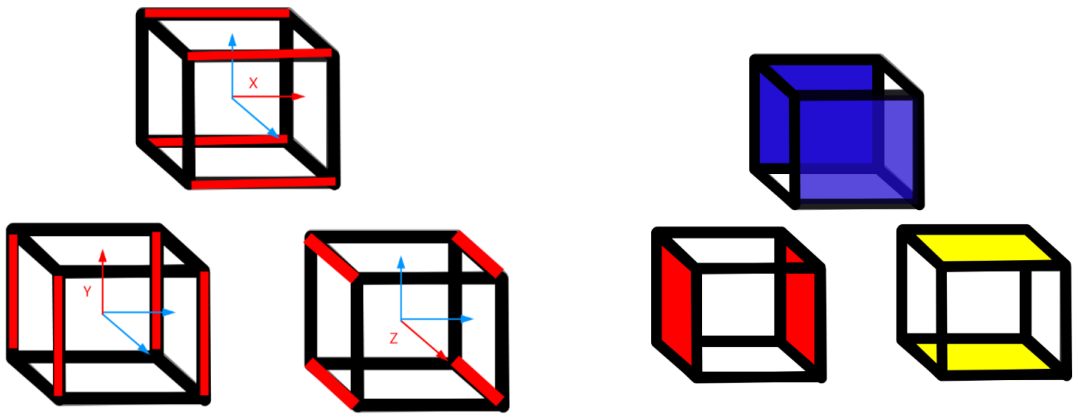
To check whether two objects (an obstacle and a search graph node) overlap the SAT requires information about each object's vertices, edges, and the normals of its faces.

Nodes

Determining the edges and normals of nodes is rather simple. As all nodes are cubes with the same orientation, the directions of their edges are identical, as are the directions of their normals. Due to the cube geometry the nodes only have three unique edges and three unique faces resulting in three unique normals. Additionally, the directions of the normals is identical to the ones of the edges and the edges are oriented along the local axis of the node (see figure 41) [Huy08]. Given that the nodes' orientation is static in the hybrid search graph the relevant axis for the SAT can be calculated once when starting the program and stored to be used each time the search graph updates. To do so the local axis of the nodes are transformed into global space and stored in a NativeArray<float3> *globalBasicAxis* field.

Next to the nodes' orientation, also their positions are static in the hybrid search graph. Therefore, each node's global vertex positions can be calculated once at the beginning when loading the search graph and stored to be used by SAT later. Hereby, only nodes with a *dynamic* flag have to be considered as they are the only ones having to be updated, as discussed in chapter 5.2.1.1.

In order to pass and use these nodes' vertex positions in the *UpdateOverlapJob* struct, the information is stored in a *globalNodeVerticesMap* field of type NativeMultiHashMap<int, float3>. Hereby, the used key is the node's index in the *dynamicNodes* list and the values are the global positions of all its vertices.



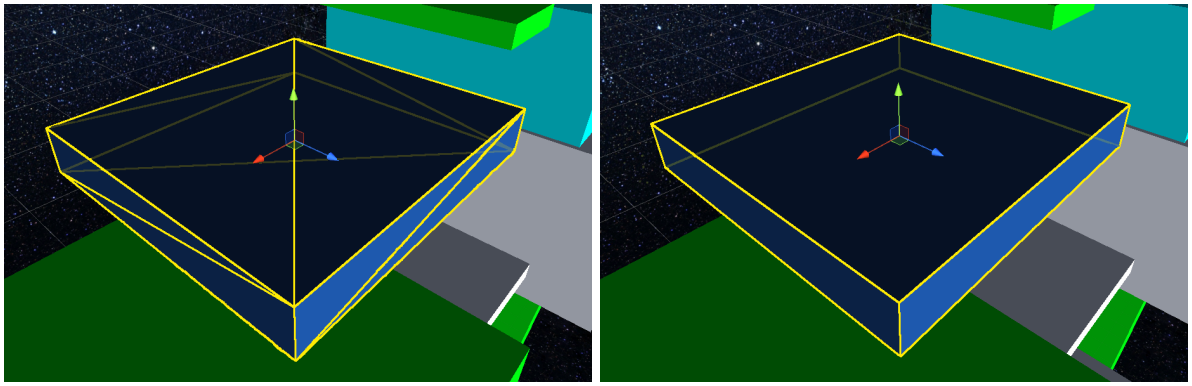
(i) Unique edges of a cube [Huy08, p.23] (ii) Unique faces of a cube [Huy08, p.19]

Figure 41: Axis of unique edges of a cube are identical to its unique normals and correspond to the cube's local axis

Dynamic obstacles

For the Separating Axis Theorem to detect overlap between nodes and obstacles, the next step is to determine the required information about vertices, edges, and normals of all dynamic obstacles. The aim hereby is to pre-calculate as much data as possible only once when starting the program, whilst still accounting for the obstacles' changing positions and rotations. For this it was decided to attach a *DynamicObstacle* class to each dynamic obstacle which stores information about the obstacles' vertices, edges, and normals in local space relative to the obstacle. Even when the obstacles moves or rotates these local values do not change. The *DynamicObstacle* class provides options to generate vertices, edges, and normals based on the objects' mesh or based on a box- or mesh-collider. When using a box collider the local vertices can be easily calculated using the collider's center and size, and the local edge and normal directions are the vectors (1,0,0), (0,1,0), and (0,0,1). When basing the obstacle collision on a mesh, the local vertices can be accessed by calling *Mesh.vertices* [Tec21q], and normals via *Mesh.normals* [Tec21o]. To only store unique normals of a mesh, all duplicate and inverse vectors are removed from the returned list. Determining all unique edges of a mesh is a little more complicated. Based on the list returned by *Mesh.triangles* [Tec21p] all edges of the triangles the mesh is constructed of can be determined. This does not only include the edges along the rim of the mesh, though, but also edges across faces of the mesh as can be seen in figure 42i. To solve this, all edges are checked against all normals to see whether they are orthogonal to each other.

Any edge along the rim of the mesh will be orthogonal to two normals, whereas edges across surfaces of the mesh are only orthogonal to one normal. After removing all edges orthogonal to less than two normals, the edges marked in yellow in figure 42ii remain. In a last step again all duplicates and inverted vectors are removed from the list of edges.



(i) All triangle edges determined using *Mesh.triangles* shown in yellow

(ii) All edges along the rim of the mesh shown in yellow

Figure 42: Edges of a mesh

When the *UpdateOverlapJob* job is first initialized it is passed all local vertices, edges, and normals of all dynamic obstacles. Each time the search graph is updated a transformation matrix is passed into the job for each dynamic obstacle, which can be used to transform the local values into world space within the job.

5.2.2.2 Determining overlap using SAT

Having determined and passed all necessary information for SAT into the *UpdateOverlapJob* job, collision between all dynamic obstacles with all nodes is checked. Listing 14 shows the *Execute* function of the *UpdateOverlapJob* struct, which is called each time the search graph is updated. The *UpdateOverlapJob* struct inherits from *IJobParallelFor*, which allows it to run multiple *Execute* functions on multiple worker threads in parallel. One *Execute* function is executed for each dynamic obstacle. Hereby the *jobIndex* represents the indices of the dynamic obstacles. To start the overlap check for a dynamic obstacle its local vertex, normal, and edge values are transformed from local to global space using the according local-to-world matrix (line 5–10). The resulting native lists are stored in the variables *globalVertices*, *globalNormals*, and *globalEdges* (line 8–10). In this step all global normal and edges are excluded that are parallel to any of the *globalBasicAxis*. The *globalBasicAxis* NativeArray contains all axis representing the nodes' edges and normals. Next the global normal and edge vectors of the dynamic obstacle along with the *globalBasicAxis* NativeArray are used to generate a list of all possible separating axis called *allAxis* (see line 12–14). This *allAxis* list contains all *globalBasicAxis* values, all global normals of the dynamic obstacle, and the cross products of all combinations of global

edges of the dynamic obstacle with all *globalBasicAxis* values. Having determined all possible separating axis, the dynamic obstacle's global vertices are projected onto all of the axis and the results stored in the *obstacleProjections* NativeList<Projection> (line 19 and 20). To project the three-dimensional object onto an axis the dot products of each vertex and the axis are calculated and the minimum and maximum values stored in a *Projection* struct. Using the hereby computed list of projections for the obstacle, next a for-loop is called iterating over all *dynamic* nodes and checking whether the dynamic obstacle and that node overlap (line 22–29). If this is the case the according boolean in a *nodeIsBlocked* NativeArray is set to true.

```
1  public void Execute(int jobIndex)
2  {
3      //fill lists with global vertices, normals, and edges based on the local
4      //variables and the localToGlobal matrices of the dynamic obstacles
5      NativeList<float3> globalVertices = new NativeList<float3>(Allocator.Temp);
6      NativeList<float3> globalNormals = new NativeList<float3>(Allocator.Temp);
7      NativeList<float3> globalEdges = new NativeList<float3>(Allocator.Temp);
8      LocalPointMapToGlobalList(j, localVertices, ref globalVertices);
9      LocalVectorMapToGlobalList(j, localNormals, ref globalNormals);
10     LocalVectorMapToGlobalList(j, localEdges, ref globalEdges);
11
12     NativeList<float3> allAxis = new NativeList<float3>(Allocator.Temp);
13     SeparatingAxisTheorem.CollectAxis(globalBasicAxis,globalNormals,
14                                       globalEdges, ref allAxis);
15
16     //project the dynamic obstacle onto all possible separating axis
17     NativeList<Projection> obstacleProjections =
18                                    new NativeList<Projection>(Allocator.Temp);
19     SeparatingAxisTheorem.ProjectOntoAxis(allAxis,globalVertices,
20                                           ref obstacleProjections);
21
22     for (int i = 0; i < dynamicNodes.Length; i++)
23     {
24         if (SeparatingAxisTheorem.IsOverlapping(obstacleProjections, allAxis,
25                 allNodes[dynamicNodes[i]], i, globalNodeVerticesMap))
26         {
27             nodeIsBlocked[i] = true;
28         }
29     }
30 }
```

Listing 14: Checking for overlap between dynamic obstacles and nodes in the *UpdateOverlapJob* job

The *SeparatingAxisTheorem.IsOverlapping* function called in line 24 in listing 14 is worth taking a closer look at and is therefore shown in listing 15. First the projections of the node and the obstacle onto the three basic axis storing the node's normals and edges are compared (line 8–17).

As the node is static its projection onto these *globalBasicAxis* axis' can be pre-calculated. This is done once when loading the search graph and the result is stored in the *HybridNode* struct as a float3x2 matrix called *basicProjection*. Hereby, the first column contains the minimum and the second column the maximum value of the projections on the basic axis, whilst the row number indicates the axis index in the *globalBasicAxis* array.

```
1  public static bool IsOverlapping(NativeList<Projection> obstacleProjections,
2                                   NativeList<float3> allAxis, HybridNode node,
3                                   int verticesMapNodeIndex,
4                                   NativeMultiHashMap<int,float3> nodesVerticesMap)
5  {
6      //the first three axis in allAxis are always the basic axis. For these axis
7      //each hybridNode already has projections stored.
8      //basicProjections[min/max][x/y/z]
9      for (int i = 0; i < 3; i++)
10     {
11         if (ProjectionsDisjoint(node.basicProjections[0][i],
12             node.basicProjections[1][i],obstacleProjections[i].min,
13             obstacleProjections[i].max))
14         {
15             return false;
16         }
17     }
18
19     //for all other axis project the node onto the axis and check if it overlaps
20     //with the pre-computed obstacle projection
21     for (int i = 3; i < allAxis.Length; i++)
22     {
23         Project(allAxis[i], verticesMapNodeIndex, nodesVerticesMap,
24                 out float nodeProjectionMin, out float nodeProjectionMax);
25         if (ProjectionsDisjoint(nodeProjectionMin, nodeProjectionMax,
26             obstacleProjections[i].min, obstacleProjections[i].max))
27         {
28             return false;
29         }
30     }
31
32     return true;
33 }
```

Listing 15: *IsOverlapping* function called to check whether a dynamic obstacle and a node overlap

If the projections of the node and the dynamic obstacle do not overlap on any one of these basic axis, the function returns false (line 16). If this is not the case the *IsOverlapping* function continues by projecting the node's global vertices onto all other possible separating axis (line 23 and 24) and checking whether these projections overlap with the according pre-calculated

projections of the dynamic obstacle onto that axis (line 25 and 26). If the projections of the node and the dynamic obstacle do not overlap on any of the possible separating axis the objects do not overlap and the function returns false (line 28). Otherwise the node and the dynamic obstacle overlap and the *IsOverlapping* function returns true (line 32).

5.2.2.3 Updating path costs

After updating each *dynamic* node's free/blocked status using the Separating Axis Theorem, the path costs of all nodes with a changed status have to be adjusted.

This is done in the *UpdatePathCostsJob* struct, which inherits from the *IJob* interface. Iterating over all *dynamic* nodes it is first determined which nodes have changed their free/blocked status and which stayed the same. For this the results of the *nodeIsBlocked* NativeArray are used, as computed in the previously run *UpdateOverlapJob* job. All changed nodes are stored in a *changedNodes* NativeList<int>. In the next step a for-loop iterates over this list and calls an *UpdatePathCosts* function for all its entries. Within this function each node's path costs to its neighbours is set to either infinite, if the node or any relevant neighbour is blocked, or to the distance between the nodes if all relevant nodes are free. For side neighbours, only the node and the neighbour have to be free. For edge neighbours also relevant side neighbour have to be free, which are marked red in figure 43. If for example the direction from the node n to its edge neighbour n' is the vector (1,-1,0), the side neighbours in the directions (1,0,0) and (0,-1,0) have to be free. Similarly, for vertex neighbours, relevant side and edge neighbours have to be free, otherwise the path cost is set to infinite. For example for the vertex neighbour in direction (1,1,1) the side neighbours in the directions (1,0,0), (0,1,0), and (0,0,1) have to be free as well as the edge neighbours in the direction (1,1,0), (1,0,1), and (0,1,1).

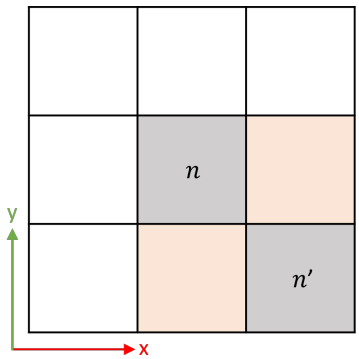


Figure 43: Checked adjacent neighbours when updating edge path costs

Each time the path costs from a node to any of its neighbours is updated, that neighbour is added to a *changedNeighbours* list. In a second step the path costs of all nodes in the *changeNeighbours* list are updated and the list is united with the *changedNodes* list. As only *dynamic* nodes are

updated and by the design of the search graph all their neighbours are directly adjacent to them, it is assured that all their predecessors are also their neighbours. Therefore, no additional predecessors need to be considered when updating path costs of changed nodes.

5.2.2.4 Performance analysis

Having discussed all the steps necessary to update the hybrid search graph, the following section will take a look at the time required for the update.

Figure 44 shows the Unity profiler whilst updating the hybrid search graph as shown in figure 37, which contains 13,696 *dynamic* nodes. As the 16 dynamic obstacles in the used test environment can change the free/blocked status of the *dynamic* nodes in each frame, the search graph is updated once a frame accordingly.

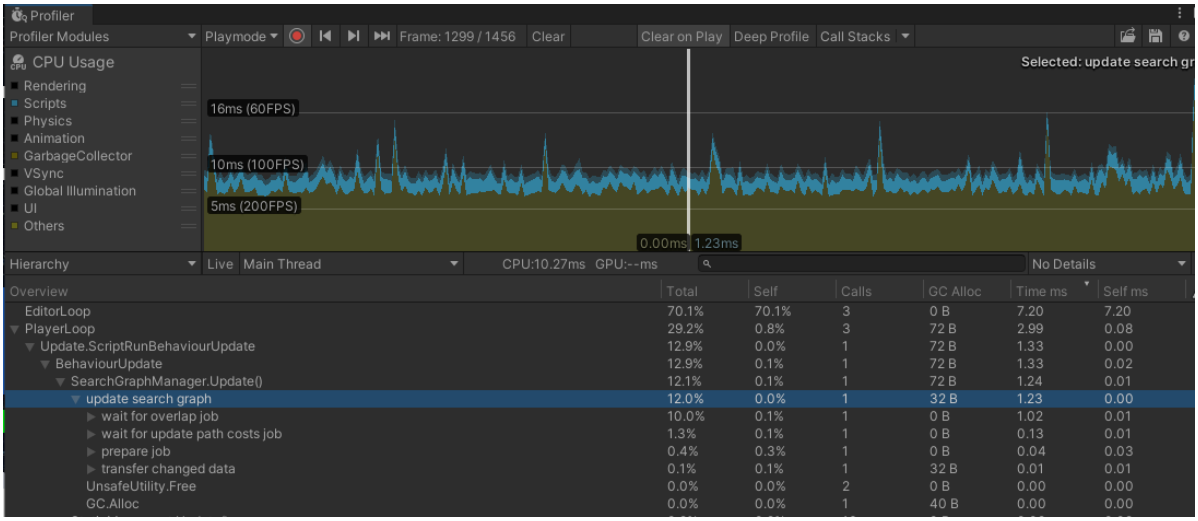


Figure 44: Profiler showing the performance of updating the hybrid search graph with disable Burst safety checks

As can be seen in the profiler, updating the search graph is split into four main steps (note that the profiler is sorted by time and not by order of execution). The step labelled as “prepare job” in the profiler is called first and handles the final setup of the used jobs before they can be scheduled. This includes getting the local-to-world transformation matrices from all dynamic obstacles and passing them into the correct job. Next the *UpdateOverlapJob* job handling the updating of the nodes’ free/blocked state is scheduled and the search graph waits for it to complete (label “wait for overlap job”). After this job has completed, its results are passed to the *UpdatePathCostsJob* struct. This next job is scheduled and its completion waited for (label “wait for update path costs job”). Once this second job is completed, its results are transferred back to the hybrid search graph, which is labelled “transfer changed data” in the profiler. This mostly involves transferring the path costs of all changed nodes output by the job to the array

of hybrid nodes used in the search graph. As expected, executing the *UpdateOverlapJob* job to check the overlap of obstacles and nodes requires the most time out of these steps. In the frame selected in figure 44 that job requires 1.02 ms to complete, updating the changed nodes' path costs takes about 0.13 ms, preparing the job 0.04 ms, and transferring the data back to the search graph requires about 0.01 ms. Even though these number can change slightly based on the number of determined changed nodes, they are pretty stable and allow the program to run with an average of more than 60 FPS. The spikes in the performance seen in the profiler are due to the Unity Editor and are not related to the search graph. Taking a look back at the previous implementation of updating the search graphs in chapter 3.1.2 it can be said that the updating of the hybrid search graph is with an average of 1.35 ms a huge improvement in performance. The comparison of average update durations of the different search graphs can be seen in figure 45.

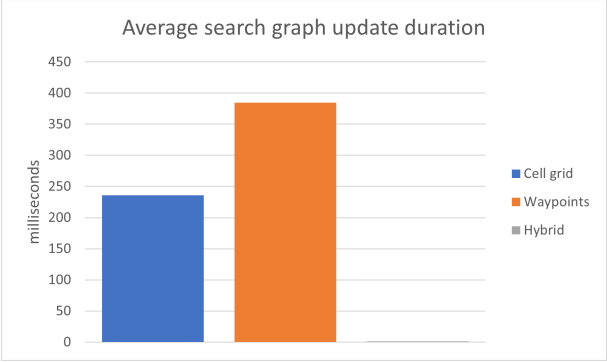


Figure 45: Average search graph update durations

5.2.3 Pathfinding algorithms

Having constructed and implemented the hybrid search graph, the next step is to adapt the pathfinding algorithms from their implementation in the cell grid search graph to the new search graph. This is done for A*, Theta*, and (Basic) MT-D* Lite. As Jump Point Search did show poor performance before and is also designed to be used in a cell grid search graph, it is not adapted to the hybrid search graph. In a next step the performance of the algorithms in the new search graph is analysed and compared to their performance in the cell grid search graph.

5.2.3.1 Adapting the pathfinding algorithms

The overall implementation of the pathfinding algorithms stays the same for the hybrid search graph as it was for the cell grid search graph. The main difference is that for the cell grid search graph only the side neighbours of nodes had to be considered, whereas for the hybrid search graph

also edge and vertex neighbours are relevant. Listing 16 shows pseudocode of the core loop of A* in the hybrid search graph. The only change to the previous implementation of the core loop, seen in listing 4, are the lines 4 and 5. The function *AddEdgesToOpen* and *AddVerticesToOpen* are implemented following the same pattern as *AddNeighboursToOpen* described in listing 3 in chapter 5.1.3.

```
1 while (nextNode != goalNode && nextNode != -1)
2 {
3     AddNeighboursToOpen(nextNode);
4     AddEdgesToOpen(nextNode);
5     AddVerticesToOpen(nextNode);
6     nextNode = GetNextNode();
7 }
```

Listing 16: Core loop of A* in the hybrid search graph

Similarly, the implementation of Theta* only had to be adjusted slightly to take edge and vertex neighbours into consideration by adding according *AddEdgesToOpen* and *AddVerticesToOpen* functions. To reduce the number of expensive line-of-sight checks they are only performed if the direction from the node n to its parent $p(n)$ is not the same as from n to its neighbour n'. If the directions are the same and the path cost from n to n' is less than infinite, there is a line-of-sight between n' and $p(n)$ and $p(n)$ can be set as the parent of n'.

When adapting (Basic) MT-D* Lite to the hybrid search graph the main change was the determining of the nodes' *rhs*-value as this is based on the nodes' predecessors. In the cell grid search graph each node had six predecessors at the most, which were always equal to its neighbours. In the hybrid search graph only *dynamic* nodes' predecessors are equal to their neighbours, whereas for *dynamic neighbour* and *static* nodes the amount of predecessors is variable. To update a *dynamic* node's *rhs*-value a similar approach is taken as described in listing 9 in chapter 5.1.4, expect that not only the minimum *rhs*-value based on the side neighbours, but also based on the edge and vertex neighbours is determined. Then, the minimum of these three *rhs*-values is determined along with its according parent. The resulting combination of *rhs*-value and parent is then assigned to the node. To determine the correct *rhs*-value for a *dynamic neighbour* or *static* node, a NativeMultiHashMap<int,int> containing information about the nodes' predecessor is used as described in chapter 5.2.1.3. When updating a node's *rhs*-value an enumerator for the current node index is returned from the NativeMultiHasMap (see line 5 in listing 17), which is used to iterate over all predecessors of that node in a while-loop (line 7 in listing 17) to determine the node's parent node and *rhs*-value (line 10–15 in listing 17).

```
private void UpdateNode(int nodeIndex)
{
    ...

    NativeMultiHashMap<int, int>.Enumerator enumerator =
                                predecessorsMap.GetValuesForKey(nodeIndex);
    while (enumerator.MoveNext())
    {
        int pred = enumerator.Current;
        if (nodes[pred].gValue < minRHS)
        {
            double rhs = nodes[pred].gValue + GetPathCost(pred, nodeIndex);
            if (rhs < minRHS)
            {
                minRHS = rhs;
                bestParentNode = pred;
            }
        }
    }
    enumerator.Dispose();

    ...
}
```

Listing 17: Determining a *dynamic neighbour* or *static* node's *rhs*-value

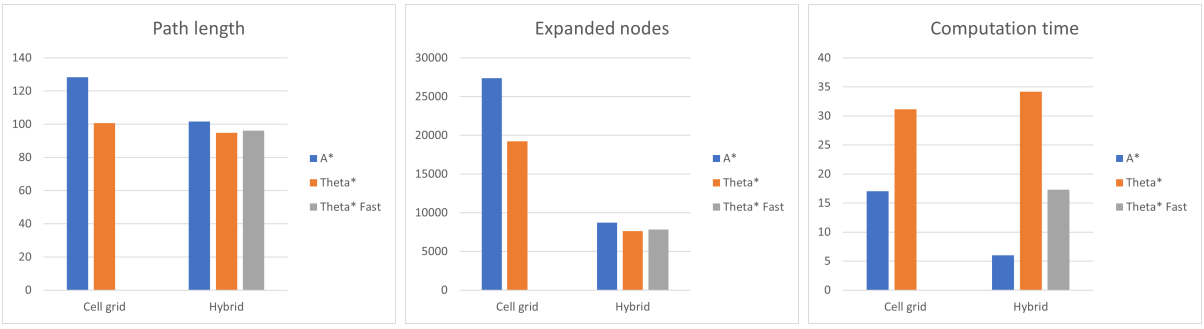
After adjusting the implementation of the algorithms for the hybrid search graph, first performance analyses showed that even though considerably fewer nodes are expanded, the required computation time did not improve as much as expected. One cause for this is the fact that each node can have more neighbours and predecessors in the hybrid search graph than in the cell grid search graph. Not only does this increase the number of mathematical operations to compute g-, f- and rhs-values, but also more nodes have to be added to or updated in the open list. In combination with the increased size of the *HybridNode* struct, due to the storage of more information for neighbour indices and path costs, this adding and updating of nodes in the open list became a performance issue. To solve this, not the whole *HybridNode* struct is stored in the open list, but a smaller struct only containing the node's index and priority is stored in the open list. This struct is called *HeapNode* or *DStarHeapNode* depending on the pathfinding algorithm. Another performance issue caused by the size of the *HybridNode* struct occurred when copying the array containing all nodes of type *HybridNode* from the search graph to the pathfinding job. As only the free/blocked status and the path costs of a node can change during the updating of the search graph, it was decided to separate this information from the rest of the *HybridNode* struct into a *HybridNodePathCosts* struct. This way the complete array of nodes only has to

be copied to the pathfinding job once when instantiating the jobs. Whenever the search graph updates only the array of *HybridNodePathCosts* has to be copied to the pathfinding algorithm. Hereby it is important to ensure that the order in the two arrays is the same, so all information of a certain node can be accessed using the same index in both arrays.

5.2.3.2 Performance analysis

Following the adjustments of the pathfinding algorithms, in the next chapter the performance of the pathfinding algorithms A*, Theta*, and (Basic) MT-D* Lite in the hybrid search graph is analysed and compared to their performance in the cell grid search graph.

First, the performance of A* and Theta* is analysed and the results presented in figure 46. The algorithms' performance in the cell grid search graph is presented on the left sides of the graphs and the performance in the hybrid search graph on the right. For the measurements the environment was setup the same way as for the analysis in chapter 5.1.2 and 5.1.3 where the destination and the obstacles are dynamic, and the AI agent is static. Also, as the measuring of the path length requires additional computation time, the tracking of time is done in a separate run than the measuring of the path length.

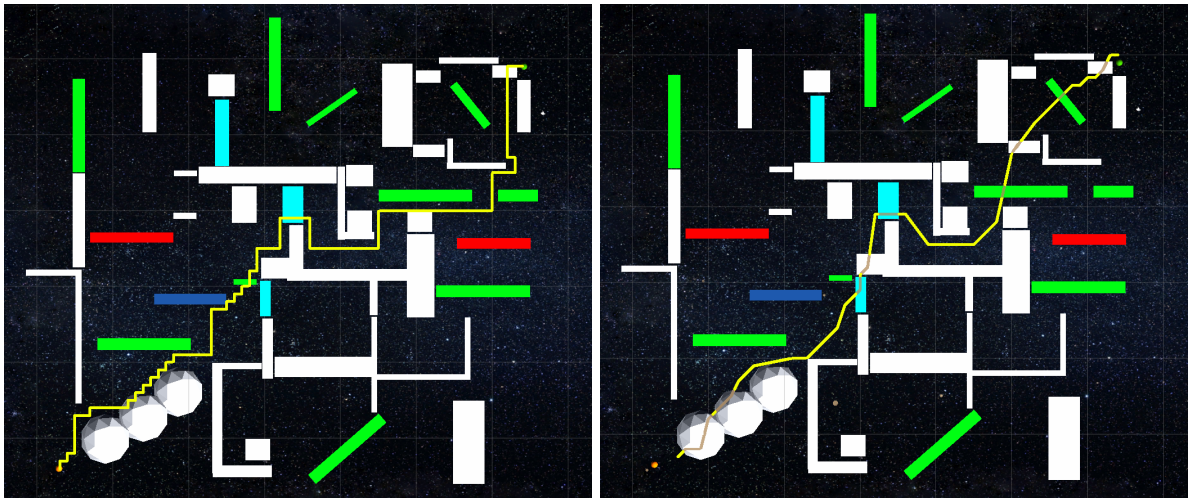


(i) (ii) (iii)

Figure 46: Comparison of the average path length (in Uu), number of expanded nodes, and computation time (in ms) using the cell grid and hybrid search graph with $\varepsilon = 1$

Starting with the path length in figure 46i it can be seen that both A* and Theta* find shorter paths in the hybrid search graph than in the cell grid search graph. Hereby the difference of A* between the two search graphs is more significant than the one of Theta*. This was expected as the any-angle algorithm Theta* was not bound to the cell grid layout in the previous search graph and therefore the difference to the new search graph is not as significant. A* on the other hand could only move along horizontal and vertical lines in the cell grid search graph, whereas in the hybrid search graph connections between *static* nodes and from *static* to *dynamic neighbour* nodes are more direct and not necessarily oriented along the horizontal and vertical axis. Figure 47 shows the different paths found by A* in the cell grid search graph on the left and the hybrid

search graph on the right. Noteworthy is also that the difference in path length between A* and Theta* is considerably smaller in the hybrid search graph than in the cell grid search graph. This, too, can be explained by the more direct connections between nodes in the hybrid search graph.

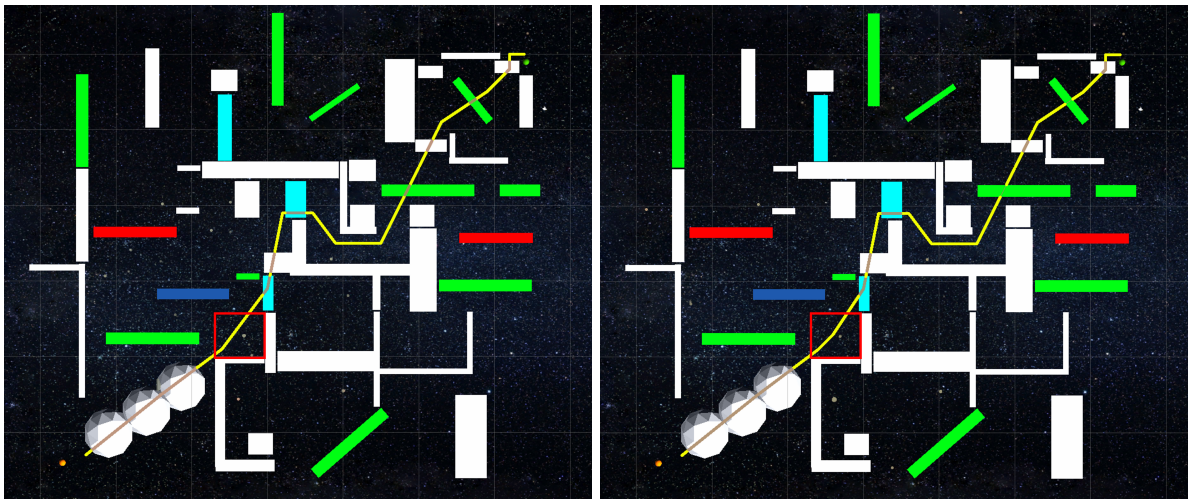


(i) Cell grid (ii) Hybrid

Figure 47: The path of A* in the cell grid and hybrid search graph with $\varepsilon = 1$

Next the number of expanded nodes and the computation time of the algorithms is analysed in figure 46ii and 46iii. As expected, the algorithms expand considerably fewer nodes in the hybrid search graph than in the the cell grid search graph. This is mainly due to the immense difference in the number of nodes between the cell grid and hybrid search graph (57344 vs. 21259). Given this difference in expanded nodes it would be expected that the algorithms are significantly faster in the hybrid search graph. As can be seen in figure 46iii this expectation is fulfilled by A*. The average computation time of this algorithm drops from around 17 ms in the cell grid to about 6 ms in the hybrid search graph. Against this expectation, though, the computation time of Theta* rises from 31.13 ms to 34.17 ms in the hybrid search graph. The main reason for this is probably the increased number of line-of-sight checks caused by the increased number of neighbours of each node. To reduce this effect one option could be to only run line-of-sight checks when expanding *static* and *dynamic neighbour* nodes. When expanding *dynamic* nodes the algorithm would act like A*. This version of Theta* was tested and added to the analysis in figure 46 as "Theta* Fast". It can be seen that the path length of this Theta* variation is slightly longer than the one of the normal Theta* version, but shorter than the one of A*. The same is true for the number of expanded nodes. When it comes to the computation time, though, it can be seen that even though Theta* Fast is slower than A* it is considerably faster than Theta*. Given the only marginal loss in path length this performance improvement might make the Theta* Fast variant a usable alternative to Theta* in a highly dynamic environment, such as a game.

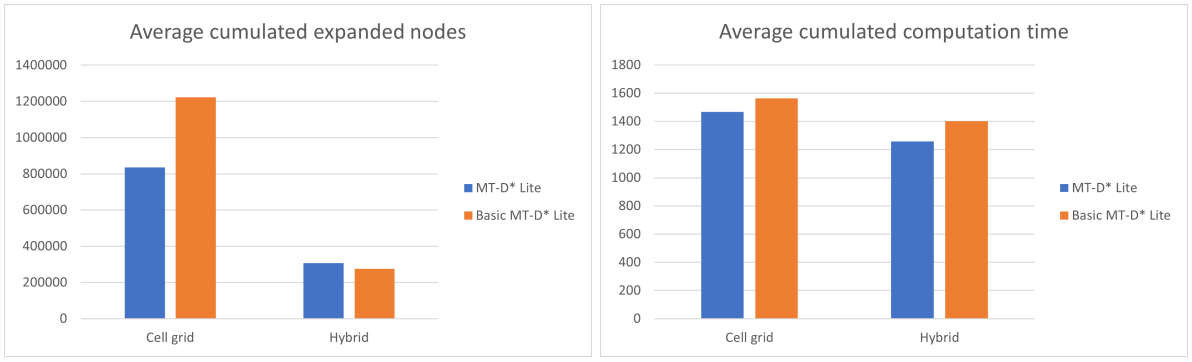
Figure 48 shows the paths found by the two Theta* variants. The difference in length between them is only about 0.04 Unity units and thereby hardly noticeable. For clarification the difference of the two paths is marked by a red frame in the images. Moreover, when comparing figure 48ii to the path found by A* in figure 47ii, the Theta* Fast path looks a lot smoother, i.e. it features fewer changes in direction and these changes are less extreme.



(i) Theta* (ii) Theta* Fast

Figure 48: The path of Theta* (left) and "Theta* Fast" (right) in the hybrid search graph with $\varepsilon = 1$. The difference between the two paths is marked by the red frame.

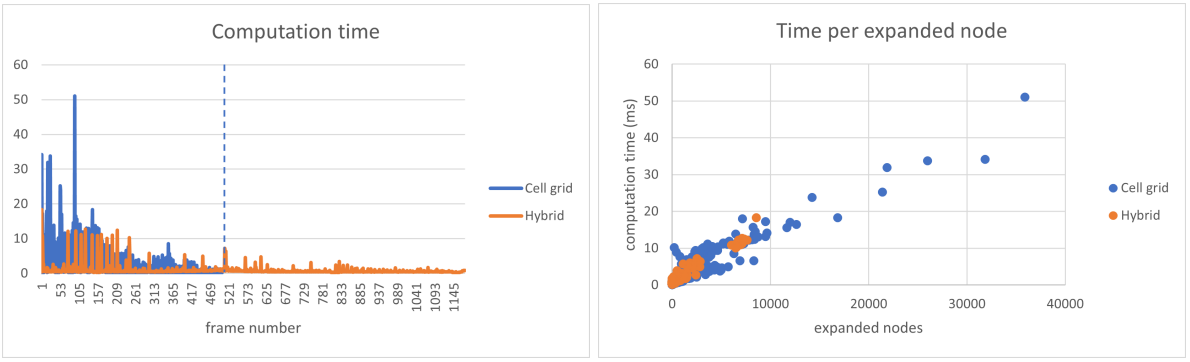
Lastly the performance of the dynamic pathfinding algorithms (Basic) MT-D* Lite is analysed. To do so the destination, obstacles, and the AI agent were set to dynamic and the cumulated computation time and number of expanded nodes was tracked over multiple runs of the AI agent from its start position to the destination. Figure 49 shows the comparison of the results form the cell grid search graph and the hybrid search graph for Basic MT-D* Lite and MT-D* Lite.



(i) (ii)

Figure 49: Average cumulated number of expanded nodes and computation time (in ms) in multiple runs of the AI agent from start to destination using the hybrid search graph and $\varepsilon = 1$ with disabled Burst safety checks

As was already the case with Theta* the dynamic algorithms expand a lot less nodes in the hybrid search graph than the cell grid one (see figure 49i), but the improvement of the computation time is not equally significant. One reason for this might be the large number of possible predecessors, which have to be considered when calculating a node's *rhs*-value. Especially *static* nodes can have an extensive number of predecessors, as discussed in chapter 5.2.1.3. Another reason for this only moderate improvement can be seen in figure 50. Here the performance of MT-D* Lite is shown for one single run of the AI agent from the start position to the destination. In the left image the computation time for each iteration is listed. The dotted blue line indicates the iteration at which the AI agent reached the destination in the cell grid search graph. It can be seen that in the cell grid search graph the algorithm has an overall higher computation time and also shows much higher peaks than in the hybrid search graph. Due to this higher computation time the algorithm is executed less often in this graph than in the hybrid search graph, causing the AI to reach the destination after fewer algorithm iterations. Therefore, much more iterations count into the cumulated computation time of the hybrid search graph than into the one of the cell grid, contributing to the unexpected high cumulated computation times. Overall, it can be said that given a similar amount of expanded nodes, the dynamic algorithm MT-D* Lite is neither significantly faster nor slower in the hybrid search graph than in the cell grid search graph, as can be seen in figure 50ii where the computation time is shown relative to the number of expanded nodes. Given the lower number of expanded nodes, though, the average computation time per iteration is lower making (Basic) MT-D* Lite more performant in the hybrid search graph than the cell grid one.



(i) (ii)

Figure 50: Computation time (in ms) and time per expanded node in a fully dynamic environment for full run of MT-D* Lite from start to destination using the hybrid search graph and $\varepsilon = 1$

In conclusion it can be said that most of the tested pathfinding algorithm are more performant in the hybrid search graph than in the cell grid search graph. They show shorter path lengths, expand fewer nodes, and overall have a lower computation time. The only exception to this is Theta*. Even though this algorithm, too, has a lower path length and expands fewer nodes in the hybrid search it has a higher computation time due to its expensive line-of-sight checks.

An alternative might be to use the Theta* Fast variation of the algorithm, which only uses line-of-sight checks for *static* and *dynamic neighbour* nodes and otherwise behaves like A*.

5.3 Any-angle Moving Target D* Lite

So far this thesis has focused on optimising pathfinding algorithms in dynamic 3D space by optimising the runtime of known pathfinding algorithms such as A*, Theta*, and (Basic) MT-D* Lite in chapter 5.1, and by optimising the used search graph in chapter 5.2 to reduce the number of expanded nodes. Hereby it can be seen that the dynamic pathfinding algorithm (Basic) MT-D* Lite overall expands the fewest nodes and has the best computation time. This is to be expected as this algorithm is specially designed to work in a dynamic environment and is able to reuse computations from previous iterations to update its paths. The second very promising pathfinding algorithm is Theta*. Even though it has a high computation time due to its line-of-sight checks, it produces the shortest and most natural looking paths. Therefore, in this chapter the use of a new pathfinding algorithm is proposed, which combines the dynamic aspects of (Basic) MT-D* Lite with the any-angle aspects of Theta*: Any-angle Moving Target D* Lite, short AAMT-D* Lite.

After giving an overview of the concept of this dynamic any-angle algorithm in 5.3.1, more details about the implementation and optimisation of the algorithm in the hybrid search graph are given in chapter 5.3.2.

5.3.1 Concept

The concept of AAMT-D* Lite is heavily based on MT-D* Lite. Therefore, figure 51 shows a basic construct of this dynamic any-angle pathfinding algorithm with the main changes to MT-D* Lite, as described in the following chapter, marked by red frames.

The main difference between MT-D* Lite and AAMT-D* Lite is that whenever the rhs-value of a node n is computed in AAMT-D* Lite it is not necessarily based on the g-value of a predecessor n' as is the case with MT-D* Lite, but can also be based on the parent of that predecessor $p(n')$ as is done in Theta*. As a result the rhs-value of a node in AAMT-D* Lite is calculated as followed:

$$rhs(n) = \begin{cases} c, & \textbf{if } n = n_{\text{goal}} \\ \min_{n' \in \text{Pred}(n)}(g(p(n')) + c(n, p(n')), & \textbf{if line-of-sight from n to p(n')} \\ \min_{n' \in \text{Pred}(n)}(g(n') + c(n, n'), & \textbf{else} \end{cases}$$

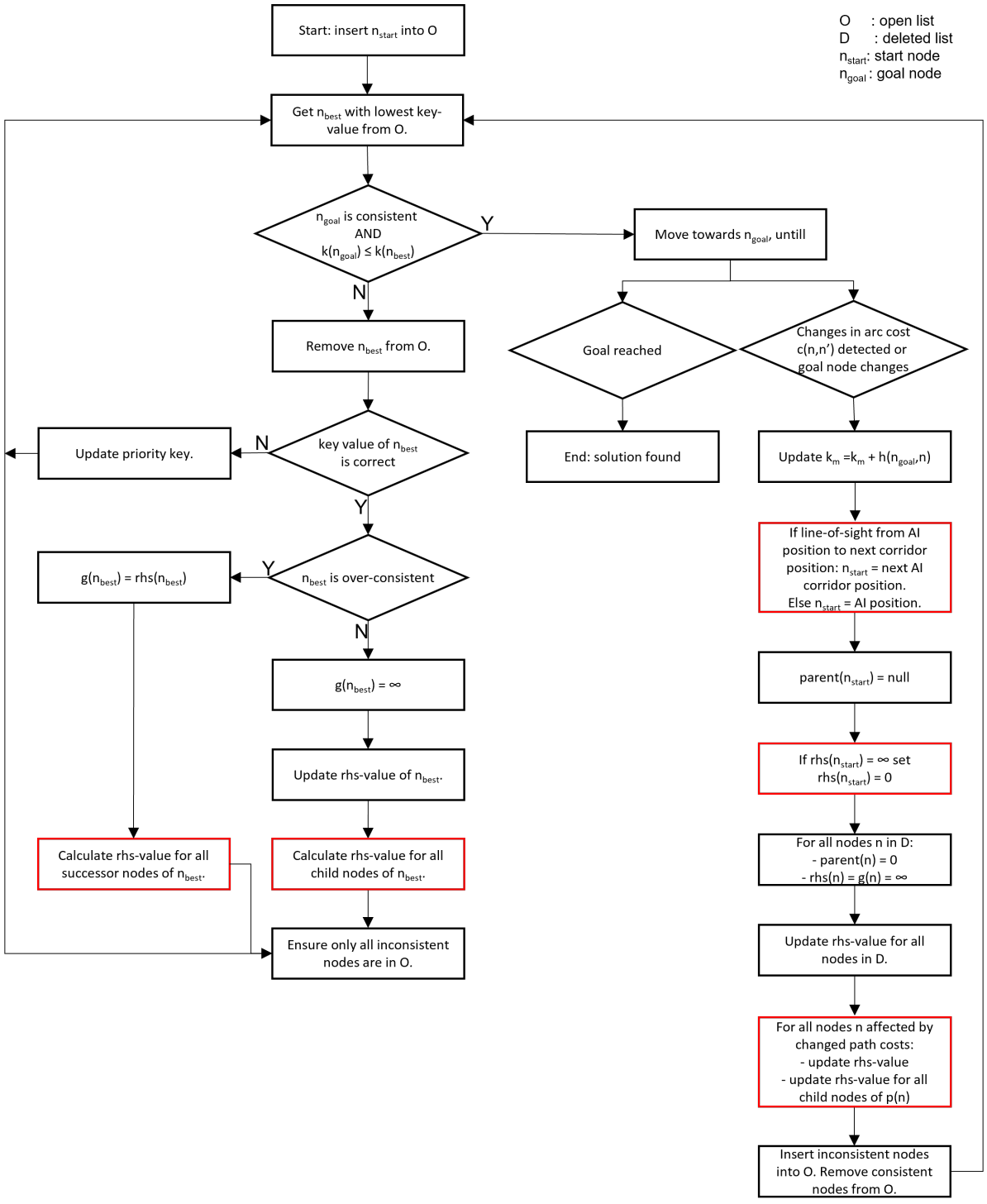


Figure 51: Structure of AAMT-D* Lite with changes to MT-D* Lite marked in red

Due to the any-angle aspect of AAMT-D* Lite, a node's parent is not necessarily one of its predecessors, but can be a node multiple neighbours away. This requires two main adjustments to be made to the concept of MT-D* Lite for AAMT-D* Lite.

The first adjustment has to be done in the expanding of under-consistent nodes. In the implementation of MT-D* Lite in this thesis, an under-consistent node n is expanded by setting its g-value to infinite, updating its own rhs-value, and then updating its successors' rhs-values. For performance reasons hereby only successors are considered who are child nodes of n. A "child node" of n is any node n' for which $p(n') = n$ is true. In MT-D* Lite any node's parent can only be one of its predecessor, therefore any node's child nodes are part of its successors. This is not the case for AAMT-D* Lite. As a result it is not sufficient to only update the rhs-values of successors who are child nodes of the under-consistent node n, but it has to be ensured that all child nodes of n update their rhs-values.

The second adjustment affects the handling of changed path costs in the search graph when the dynamic environment changes. In MT-D* Lite all nodes with changed path costs update their rhs-value and thereby also their parent. In other words, if the path costs of a node to its parent is increased, e.g. by the parent node being blocked, the child computes a new rhs-value and parent. In AAMT-D* Lite it is possible, though, for a dynamic obstacle to block the line-of-sight, and thereby the path, between a parent node and its child without changing the path costs of either of the nodes. Therefore, a child node might not be informed about the changed path costs in the search graph and not update its rhs-value correctly. To solve this, it has to be ensured that all nodes whose line-of-sight to their parent gets disturbed by the changed environment, update their rhs-value in addition to the nodes whose path costs changed. This can be done by iterating over the list of all changed nodes' parents and updating their child nodes. The idea behind this is that all nodes on the line-of-sight between a parent and child node will have the same parent node. For example if the line-of-sight between a node n and its parent $p(n)$ is interrupted by the changed node $n_{changed}$, all child nodes of the parent of $n_{changed}$, which equals $p(n)$, are updated, including n.

Another characteristic of AAMT-D* Lite has to be pointed out, which is the determination and updating of the start node. As with MT-D* Lite and Theta* the new start node is usually defined to be the next node in the current corridor of the AI agent. Due to the any-angle feature of AAMT-D* Lite it is possible for the AI's current position and the next corridor position to be quite far apart. Therefore, after each update of the search graph it is checked whether there is a free line-of-sight between the current AI position and the next corridor position. If this is the case then the next corridor position is defined as the new start node. If there is no free line-of-sight the new start node is defined to be the closest node to the current AI position. Once a new start node is determined AAMT-D* Lite handles this change in the same way as MT-D* Lite would, with the addition that the new start node's rhs-value is set to zero should it be infinite.

Lastly a multiplication factor φ is introduced to be used when calculating the key value of under-consistent nodes. As AAMT-D* Lite is based on Theta* it is not optimal, just as Theta* is also not optimal. This means the path found is not necessarily the shortest one. Due to this fact it is possible that under-consistent nodes are not expanded before the algorithm terminates, because their key value is too high and therefore their priority in the open list too low. A similar issue was discussed in the bachelor thesis [Kra19] during the implementation of D* Lite with an inflation factor $\varepsilon > 1$ [Kra19, p. 60]. For D* Lite it was decided to exclude the use of the inflation factor for under-consistent nodes. For AAMT-D* Lite this concept is expanded by multiplying the heuristic with a decreasing factor $\varphi <= 1$ for under-consistent nodes.

$$k(n) = \begin{cases} [rhs(n) + \varepsilon * h(n_{\text{start}}, n); rhs(n)], & \textbf{if } n \textbf{ is over-consistent} \\ [g(n) + \varphi * h(n_{\text{start}}, n); g(n)], & \textbf{if } n \textbf{ is under-consisten} \end{cases}$$

5.3.2 Implementation

Having explained the concept of AAMT-D* Lite in the previous chapter, this chapter details some of the implementation aspects of this algorithm, focussing on the computation of the nodes' *rhs*-values as this is the most performance-critical aspect. Overall, the implementation of AAMT-D* Lite is based on the implementation of MT-D* Lite in the hybrid search graph as described in chapter 5.1.4 and 5.2.3.1. The characteristics of AAMT-D* Lite are then implemented to expand on this basis.

Listing 18 shows pseudocode for updating the *rhs*-values of an over-consistent node's side neighbours. Hereby lines 3–8 are very similar to the lines 3–11 in listing 8 in chapter 5.1.3, which shows the function *AddNeighboursToOpen* used in Theta* to calculate the f-value of a node's neighbours and insert them into the open list. The rest of the *UpdateNeighboursOfNode* function in listing 18 is very similar to the *UpdateNeighboursOfNode* function of MT-D* Lite shown in listing 24, with the exception that the *newRhsValues* variable in line 10 is not computed based on the node's g-value and path cost, but if possible on the g-value and therefrom resulting path costs of the node's parent. The functions to update the *rhs*-values of an over-consistent node's edge and vertex neighbours follow the same logic of the one updating the side neighbours.

```
1  private void UpdateNeighboursOfNode(int nodeIndex)
2  {
3      HybridNode node = nodes[nodeIndex];
4      double3x2 baseGValues = node.gValue;
5      double3x2 pathCosts = nodePathCosts[nodeIndex].pathCosts;
6      int3x2 newParents = nodeIndex;
7      FillPathCostsAndParents(nodeIndex,node.neighbours,node.parentNode,
8                              ref baseGValues, ref newParents, ref pathCosts);
9
10     double3x2 newRhsValues = baseGValues + pathCosts;
11     double3x2 oldRhsValues = GetOldEdgeRhsValues(node.neighbours);
12     bool3x2 useNewRhsValue = newRhsValues < oldRhsValues;
13
14     for (int i = 0; i < node.neighboursCount; i++)
15     {
16         CellGridManagerHelper.IndexToPosition(i, neighboursCountSize, ref pos);
17         if (useNewRhsValue[pos.x][pos.y])
18         {
19             HybridNode neighbour = nodes[node.neighbours[pos.x][pos.y]];
20             neighbour.rhsValue = newRhsValues[pos.x][pos.y];
21             UpdateParent(node.index, newParents[pos.x][pos.y]);
22             neighbour.parentNode = newParents[pos.x][pos.y];
23             nodes[node.neighbours[pos.x][pos.y]] = neighbour;
24             UpdateNodeInOpenList(nodes[nodeIndex].neighbours[pos.x][pos.y]);
25         }
26     }
27 }
```

Listing 18: Updating the *rhs*-value of an over-consistent node's neighbour

As described in the previous chapter it is important for AAMT-D* Lite to update the *rhs*-value of all child nodes of an under-consistent node during its expansion. For this a MultiNativeHashSet<int,int> field called *childNodes* stores all the necessary information about child-parent-node pairs. Hereby the parent nodes are the keys in the set and their children are the values. The *childNodes* field is updated in an *UpdateParent* function, which is called each time a node is assigned a new parent node, as can for example be seen in listing 18 in line 21. When expanding an under-consistent node an enumerator iterates over all child nodes of the currently expanded node and calls a function to update their *rhs*-values.

As discussed in chapter 5.2.3.1 it is necessary in MT-D* Lite to differentiate between the updating of *rhs*-value of *dynamic* nodes versus *static* or *dynamic neighbour* nodes. This is the same for AAMT-D* Lite. Listing 28 in the Appendix shows pseudocode for computing the *rhs*-value of a *dynamic* node in AAMT-D* Lite. A large part of this *UpdateDynamicNodeRhs* function is very

similar to the one for computing the *rhs*-value of a *dynamic* node in MT-D* Lite. Listing 19 shows pseudocode for the part of this function which differs from MT-D* Lite and is responsible for computing the *rhs*-value based on the predecessors' parents.

```
1  collectedParents.Clear();
2  //get minimumRhsValue based on the node's predecessors
3  ...
4
5  double minRHS = minimumRhsValue;
6  int bestParent = -1;
7
8  CollectParents(predecessors);
9  CollectParents(edgePredecessors);
10 CollectParents(vertexPredecessors);
11 collectedParents.Remove(-1);
12 collectedParents.Remove(nodeIndex);
13
14 var enumerator = collectedParents.GetEnumerator();
15 while (enumerator.MoveNext())
16 {
17     int predParent = enumerator.Current;
18     double rhs = nodes[predParent].gValue +
19             Distance(node.localPosition, nodes[predParent].localPosition);
20     if (rhs <= minRHS
21         && LineOfSight(node.localPosition,nodes[predParent].localPosition))
22     {
23         minRHS = rhs;
24         bestParent = predParent;
25     }
26 }
27 enumerator.Dispose();
28
29 ...
```

Listing 19: Determining a *dynamic* node's *rhs*-value and parent based on its predecessors' parents

As many predecessors might have the same parent node, all unique parents of the predecessors are collected in a NativeHashSet *collectedParents* in line 8–10. Afterwards the indices -1 and *nodeIndex* are removed from the NativeHashSet (line 11 and 12). -1 means the predecessor does not have a parent, therefore no parent has to be considered when calculating the new *rhs*-value. *nodeIndex* is the index of the node for which the *rhs*-value needs to be calculated. If the parent of a predecessor is this node, then it needs to be excluded to avoid setting a node's parent to itself. From line 15–26 it is iterated over all remaining *collectedParents* and for each entry a *rhs*-value based on that parent is computed (line 18 and 19). If this *rhs*-value is lower than the currently lowest *rhs*-value, a line-of-sight check is run from the node to the predecessor's

parent to determine whether the computed *rhs*-value is valid or the line-of-sight between the two nodes is obstructed (line 21). If there is a free line-of-sight between the nodes, the *minRHS* variable is set to the computed *rhs*-value and the predecessor's parent set to be the *bestParent* (line 23 and 24). This order of first computing the possible *rhs*-value and only afterwards running a line-of-sight check was chosen because the line-of-sight check is more expensive than the computation of the *rhs*-value. Another implementation detail worth mentioning is that the used NativeHashSet for the *collectedParents* is not the default NativeHashSet provided by Unity's *Collections*-package, as this does not support the usage of the *GetEnumerator* function (line 14) within jobs. The reason for this is the integrated safety system in the C# Job System. To be able to use NativeHashSets in jobs and iterate over them, the Collections package version 0.15.0-preview.21 was cloned and some of the safety system for the NativeHashSet removed.

The last major change between MT-D* Lite and AAMT-D* Lite is the updating of *rhs*-values of *static* and *dynamic neighbour* nodes. The full pseudocode for the *UpdateStaticNodeRhs* function can be seen in the Appendix in listing 29. As with *dynamic* nodes, first the minimum *rhs*-value based on the node's predecessors is computed from line 9–71 and afterwards it is determined if a *rhs*-value based on one of the predecessors' parents is lower in line 78–127. The main performance issue of computing the *rhs*-value of *static* and *dynamic neighbour* nodes is that they can have a lot of predecessors and determining the path costs to all predecessors and predecessors' parents and thereby the *rhs*-value is cost intensive. Therefore, first a *distanceBasedRhs* value is computed, using the distance between two nodes as their path costs (listing 29 line 34–36 and 96–98). Only if this distance-based *rhs*-value is lower than the current *rhs*-value, the correct *rhs*-value is computed using the correct path costs from the predecessor to the node (line 62) or a line-of-sight check is used to check whether the line-of-sight between the node and the predecessor's parent is free (line 121). As the distance calculation between two nodes is also not trivial, it was decided to always group the computation of the *rhs*-values for four nodes together. Listing 20 shows the relevant part of the *UpdateStaticNodeRhs* function responsible for determining the minimum *rhs*-value based on the predecessors' parents. Whilst iterating over all collected parents, the possible parents' x-, y-, and z-positions as well as *g*-values are stored in four-dimensional vectors using the *FillPossibleParentsValues* function shown in listing 30 in the Appendix (see listing 20 line 4 and 5). After all vector slots have been filled after each fourth iteration of the while-loop, the *distanceBasedRhs* value of type double4 is calculated for the four possible parents using the *GetDistanceBasedRhs* function shown in listing 21 (see listing 20 line 14–16). As can be seen the *GetDistanceBasedRhs* function computes the distance of the four previously stored parents in parallel, instead of sequentially as would have been done if the values had not been collected over the last four iterations. If the *minDistanceBasedRhs*-value returned by the *GetDistanceBasedRhs* function is lower than the currently lowest *rhs*-value, the while-loop continues immediately (see listing 20 line 17–20), otherwise for each value in *distanceBasedRhs*

lower than the current minimum *rhs*-value a correct *rhs*-value using the correct path cost or line-of-sight is determined and if required the currently best *rhs*-value updated (line 22–31). Even though this technique of collecting data over four iterations makes the code longer and harder to read, it increases its performance significantly.

```
1  while (collectedParentsEnumerator.MoveNext())
2  {
3      int predParent = collectedParentsEnumerator.Current;
4      FillPossibleParentsValues(counter, predParent, ref gValues, ref parents,
5                                ref predPosXs, ref predPosYs, ref predPosZs);
6      counter++;
7
8      if (counter != 4)
9      {
10         continue;
11     }
12     counter = 0;
13
14     double4 distanceBasedRhs =  GetDistanceBasedRhs(node.localPosition,
15                                 predPosXs, predPosYs, predPosZs, gValues,
16                                 out double minDistanceBasedRhs);
17     if (minDistanceBasedRhs < minRHS)
18     {
19         continue;
20     }
21
22     bool4 checkPred = distanceBasedRhs < minRHS;
23     for (int i = 0; i < 4; i++)
24     {
25         if(checkPred[i]
26             && LineOfSight(node.localPosition, nodes[predParent].localPosition))
27         {
28             minRHS = distanceBasedRhs[i];
29             bestParentNode = parents[i];
30         }
31     }
32 }
```

Listing 20: Determining a *dynamic neighbour* or *static* node's *rhs*-value based on the collected parents. Part of the *UpdateStaticNodeRhs* function in listing 29.

```
private double4 GetDistanceBasedRhs(int3 nodeLocalPosition, int4 predPosXs,
            int4 predPosYs, int4 predPosZs, double4 predGValues,
            out double minDistanceBasedRhs)
{
    float4 diffX = nodeLocalPosition.x - predPosXs;
    float4 diffY = nodeLocalPosition.y - predPosYs;
    float4 diffZ = nodeLocalPosition.z - predPosZs;

    float4 squareDistanceX = diffX * diffX;
    float4 squareDistanceY = diffY * diffY;
    float4 squareDistanceZ = diffZ * diffZ;

    double4 distance =
                math.sqrt(squareDistanceX + squareDistanceY + squareDistanceZ);
    double4 distanceBasedRhs = predGValues + distance;

    minDistanceBasedRhs = math.cmin(distanceBasedRhs);
    return distanceBasedRhs;
}
```

Listing 21: Computing distance based *rhs*-values

Lastly a "Fast" mode was implemented for AAMT-D* Lite, called AAMT-D* Lite Fast, similar to the "Fast" variation of Theta* described in chapter 5.2.3.2. Hereby, the expensive updating of a node's *rhs*-value described throughout this chapter is only used for *static* and *dynamic neighbour* nodes. For *dynamic* nodes the computations of *rhs*-values is the same as used in MT-D* Lite.

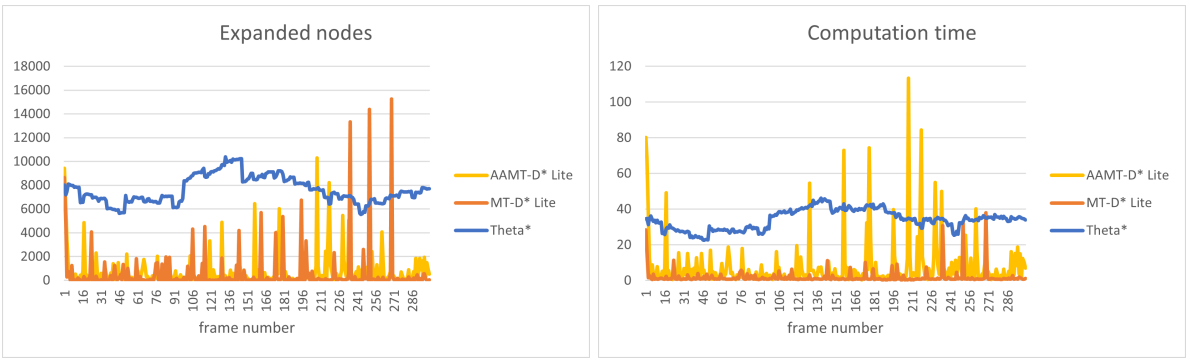
5.3.3 Performance analysis

To conclude this chapter about AAMT-D* Lite, the performance of this pathfinding algorithm is analysed and compared to Theta* and MT-D* Lite.

Given the very different nature of Theta* and MT-D* Lite it is difficult to compare these two algorithms to each others. Therefore, two different approaches for the analysis of AAMT-D* Lite and its comparison to Theta* and MT-D* Lite were chosen.

First the algorithms were observed over 300 frames in the test environment where the destination and obstacles are dynamic, but the AI agent is static. Figure 52 shows the computation time and number of expanded nodes over this time. The difference between the dynamic algorithms MT-D* Lite and AAMT-D* Lite and the non-dynamic algorithm Theta* are shown very nicely.

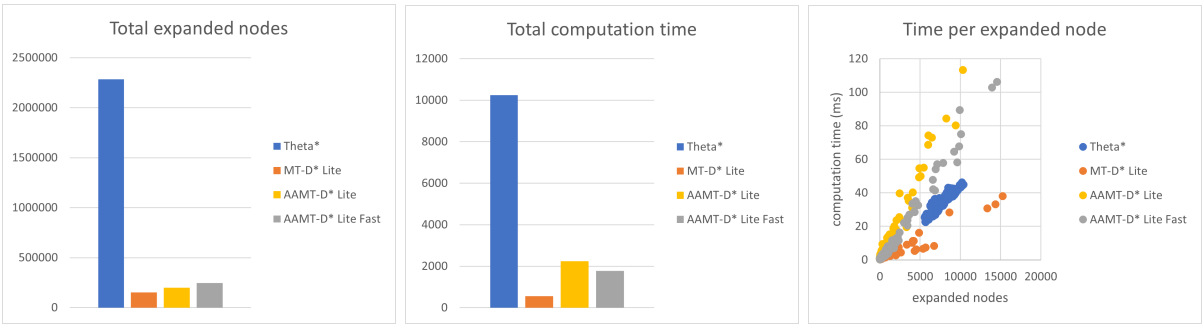
The computation time and number of expanded nodes of Theta* are constantly high, whereas the computation time and number of expanded nodes of the dynamic algorithms vary. It can be seen that AAMT-D* Lite and MT-D* Lite mostly expand fewer nodes than Theta*, but do feature very high peaks in these numbers. Hereby the peaks of AAMT-D* Lite are lower than the ones of MT-D* Lite. In contrast to this, the computation time of AAMT-D* Lite is overall higher than the one of MT-D* Lite, and at times even significantly peaks above the one of Theta*.



(i) (ii)

Figure 52: Comparison of the number of expanded nodes and the computation time (in ms)over 300 frames using the hybrid search graph, $\varphi = 1$, and $\varepsilon = 1$

For further analysis figure 53 shows the cumulated computation time and number of expanded nodes over these 300 frames, as well as the computation time relative to the number of expanded nodes. In this figure also the results of the "Fast" variation of AAMT-D* Lite is shown. For these measurements AAMT-D* Lite uses a multiplication factor $\varphi = 1$ for calculating the key value of under-consistent nodes, whereas the AAMT-D* Lite Fast uses a factor $\varphi = 0.5$. The lower multiplication factor for the "Fast" variation is needed, because its found paths are slightly longer than the ones of the normal version and therefore the issue of under-consistent nodes having too low priorities to be expanded occurs more often.



(i) (ii) (iii)

Figure 53: Comparison of the total number of expanded nodes, total computation time (in ms), and average path length (in Uu) using the hybrid search graph and $\varepsilon = 1$

Figure 53i and 53ii show very clearly that AAMT-D* Lite expands in total only slightly more nodes than MT-D* Lite, but has a distinctly larger total computation time. Furthermore, it can be seen that AAMT-D* Lite Fast expands slightly more nodes but has a lower total computation time than the normal AAMT-D* Lite version. It still has a higher computation time than MT-D* Lite, though. This relation between the number of expanded nodes and the required computation time is also reflected in the results shown in figure 53iii. MT-D* Lite has the lowest computation time relative to the number of expanded nodes, whereas AAMT-D* Lite has the highest. Interestingly, the computation time of Theta* relative to the number of expanded nodes is also lower than the one of AAMT-D* Lite. This explains why some of the peaks in the computation time of AAMT-D* Lite seen in figure 52ii are significantly higher than the time required by Theta*.

Next the average path lengths of the pathfinding algorithms in this setup are compared in figure 54. As expected MT-D* Lite finds longer paths than the any-angle algorithms. The difference in path length between Theta* and AAMT-D* Lite is only minimal. Furthermore, the paths found by the "Fast" variation of AAMT-D* Lite are only slightly longer than the ones of the normal AAMT-D* Lite version.

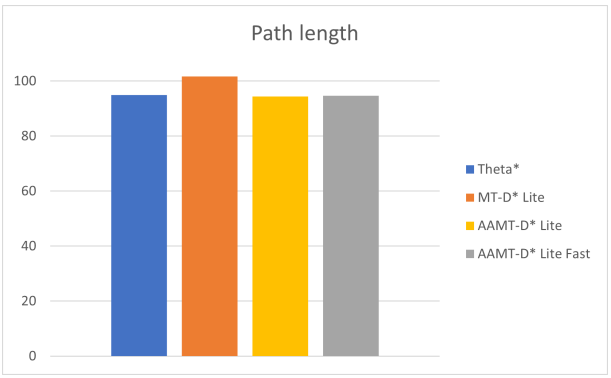
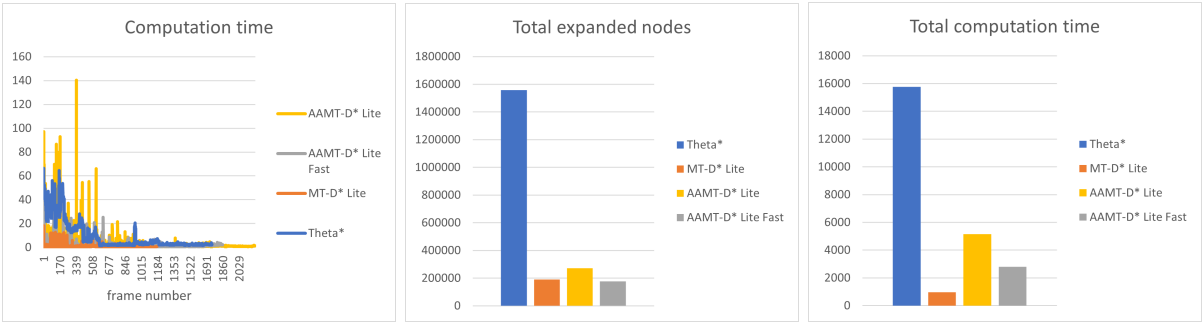


Figure 54: Average path length (in Uu) using the hybrid search graph and $\varepsilon = 1$

After analysing and comparing the performance of AAMT-D* Lite in an environment with a static AI agent, in a second approach the pathfinding algorithms are run in a fully dynamic environment and their computation time and number of expanded nodes are measured from the start until the AI agent reaches its destination. The results of these measurements are detailed in figure 55. Here again the normal version of AAMT-D* Lite uses $\varphi = 1$, whereas AAMT-D* Lite Fast uses $\varphi = 0.5$. Mostly the results are as expected based on the previous analysis in figure 52 and 53. One interesting observation in figure 55i, though, is that AAMT-D* Lite Fast hardly shows any extreme spikes in its computation time that are higher then the time required by Theta* and figure 55ii shows that AAMT-D* Lite Fast even expands fewer nodes than MT-D* Lite. Moreover, the difference in the total computation time between the "Fast" and the normal variations of AAMT-D* Lite has increased in comparison to the difference of the total computation time between the two version in figure 53ii.



(i) (ii) (iii)

Figure 55: Comparison of the computation time (in ms), total number of expanded nodes, and total computation time (in ms) in a full run from start to destination using the hybrid search graph and $\varepsilon = 1$

Overall AAMT-D* Lite, and especially its "Fast" variation, can be a good alternative to Theta* and MT-D* Lite. This pathfinding algorithm successfully combines the advantages of an any-angle algorithm, like Theta*, with the ones of a dynamic algorithms, like MT-D* Lite. It produces short and natural looking paths, and is able to reuse results from previous iterations to reduce its computation time. The analysis has shown that the paths found by AAMT-D* Lite are comparable to the ones of Theta*, but that AAMT-D* Lite most requires a significantly lower computation time than the non-dynamic pathfinding algorithm most of the time. From time to time, though, AAMT-D* Lite shows very high peaks in its computation time, which are higher than the time required by Theta*. Using the "Fast" variation can reduce this issue significantly. Developers should nevertheless be aware of these spikes when deciding to use AAMT-D* Lite. If spikes in the computation time will result in issues with the pathfinding system, depends on the overall requirements and setup of the application or game. In the test environment of this thesis, the overall low computation time and natural looking and short paths of AAMT-D* Lite outweigh the downside of the high spikes.

5.4 Pathfinding update loop and success rate

For the purpose of testing and measuring computation times up to this point the updating of the search graph and the execution of the pathfinding algorithms were executed on the main thread. Using the *JobHandler.Complete* command [Tec21y] it was waited for the jobs to complete before the main thread continues. This obviously defeats the purpose of jobs to run multi-threaded on worker threads in parallel to the main thread. Therefore, for the final implementation the jobs should be scheduled at suitable times in the player loop and their results waited for later in the loop. Hereby, it would be good to schedule the updating jobs of the search graph after all dynamic objects in the scene and the AI agent have moved and to schedule the pathfinding job after the updating jobs have completed, but before the AI agent moves the next time.

When exactly the scheduling of the updating and pathfinding jobs is best will depend on the game the system is used in. Deciding factors will be how the performance of the program is spread over a frame and when the results of the jobs are needed exactly. For this test environment, which features very little game code and hardly any expensive rendering, it was decided to schedule the updating jobs of the search graph at the end of the general *Update* method [Tec21r]. All dynamic obstacles as well as the AI agent are moved during the *Update* function, so that by the time the updating of the search graph is scheduled all changes to the environment have been completed. During the *PreLateUpdate* phase [Tec21v] marked by the green frame in figure 56 the results of the updating process are then returned to the main thread, so they can be used by the pathfinding algorithm, which is scheduled during the *PostLateUpdate* phase [Tec21u] marked by the red frame in figure 56. In the *EarlyUpdate* phase [Tec21m] of the next frame (blue line in figure 56), the progress of the pathfinding job is evaluated and if it is completed the shortest path is passed to the pathfinding AI, so it has all information available before taking the next step during the *Update* phase (orange line in figure 56). To utilize the different update phases of the *PlayerLoop* [Tec21t] the player loop was customised following the example in an article by Chris Gaudino on the website Grizzly Machine [Gau19].

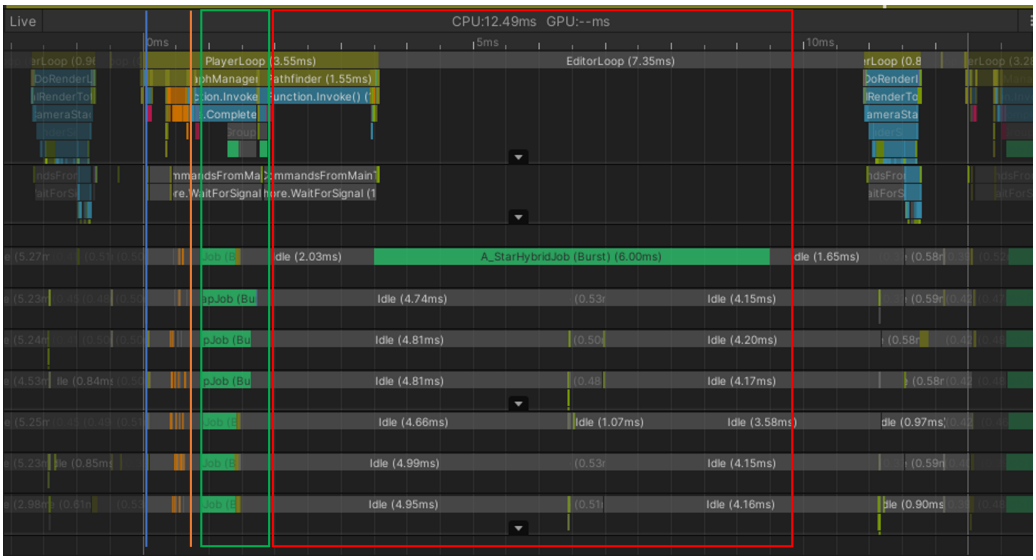


Figure 56: Representative frame of the pathfinding process. Blue line shows start of the *EarlyUpdate* phase, the orange line shows the start of the *Update* phase, the green frame shows the *PreLateUpate* phase, and the red frame shows the *PostLateUpdate* phase.

Using this implementation of the pathfinding loop a final performance test is run for A*, Theta*, Theta* Fast, (Basic) MT-D* Lite, AAMT-D* Lite, and AAMT-D* Lite Fast in the hybrid search graph, to determine their success rates. A run is rated as a success if the AI agent reaches its destination without colliding with any obstacle. Figure 57 shows the success rate of the pathfinding algorithm in the hybrid search graph using an inflation factor $\varepsilon = 1$, where AAMT-D* Lite uses a multiplication factor $\varphi = 1$ and AAMT-D* Lite Fast uses $\varphi = 0.5$.

It can be seen that the success rates of all pathfinding algorithms is very high. A*, MT-D* Lite, and Basic MT-D* Lite even have a success rate of 100%. The success rates of the any-angle algorithms is slightly lower with MT-D* Lite Fast having the lowest one of 96%. Interestingly, when the any-angle algorithms fail it is not due to the dynamic obstacles, but because the AI agent collides with static obstacles. Therefore, the most likely explanation for the lower success rate is that the used line-of-sight check does not account for the AI agent's size well enough, causing the agent on rare occasions to collide with obstacles even though the line-of-sight check returns true. To verify this assumption a second test was run with AAMT-D* Lite Fast using a significantly smaller AI agent. In this second run AAMT-D* Lite Fast showed a success rate of 100%.

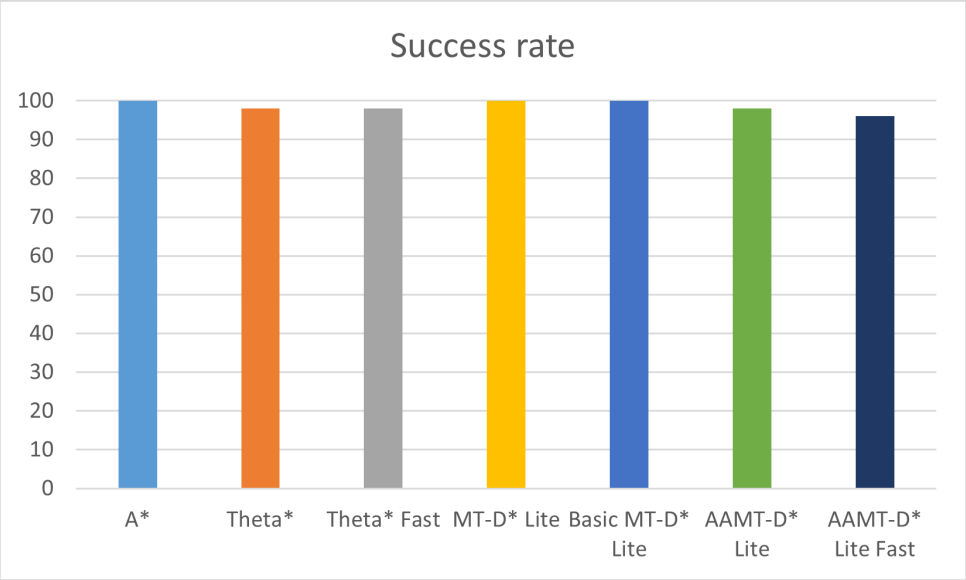


Figure 57: Success rate in percentage of pathfinding algorithms in the hybrid search graph with $\varepsilon = 1$. AAMT-D* Lite uses a multiplication factor $\varphi = 1$ and AAMT-D* Lite Fast uses $\varphi = 0.5$.

Overall, these final test runs show that the pathfinding system developed in this thesis, consisting of the hybrid search graph and one of the pathfinding algorithms listed in figure 57, is a great success. This pathfinding system enables the AI agent to travel smoothly through the dynamic three-dimensional environment and to reach its destination quite successfully. Hereby the different computation times of the algorithms and the rather large spikes in the dynamic algorithms' times do not seem to make a difference. For comparison to the previous success rates of the pathfinding algorithm implemented as part of the bachelor thesis [Kra19] one can look at figure 49 on page 74 of that thesis. Here the most successful pathfinding algorithm was A* with a success rate of 93.277% in the cell grid search graph using an inflation factor of $\varepsilon = 2$.

One interesting observation during this test is the AI agent's travelled distance through the dynamic environment from its start to end point. Figure 58 shows these travelled distances of

A*, Theta*, MT-D* Lite, and AAMT-D* Lite over multiple runs. Surprisingly the any-angle algorithms Theta* and AAMT-D* Lite often have a longer travelled distance than the non any-angle algorithms A* and MT-D* Lite. The reason for this is that all algorithms always use the next corridor position of the AI agent as their next starting point. In the any-angle algorithms the corridor points are further apart than in the non any-angle algorithms. In other words, the agent travels further distances between corridor points without getting any new information about the changed environment and thereby possible changed shortest paths. This for example causes the agent having to travel backwards longer distances in case their previously followed path gets blocked, because they have to reach the next corridor position first, before being able to turn around. When using any-angle algorithms in an actual application or game this issue should be addressed.

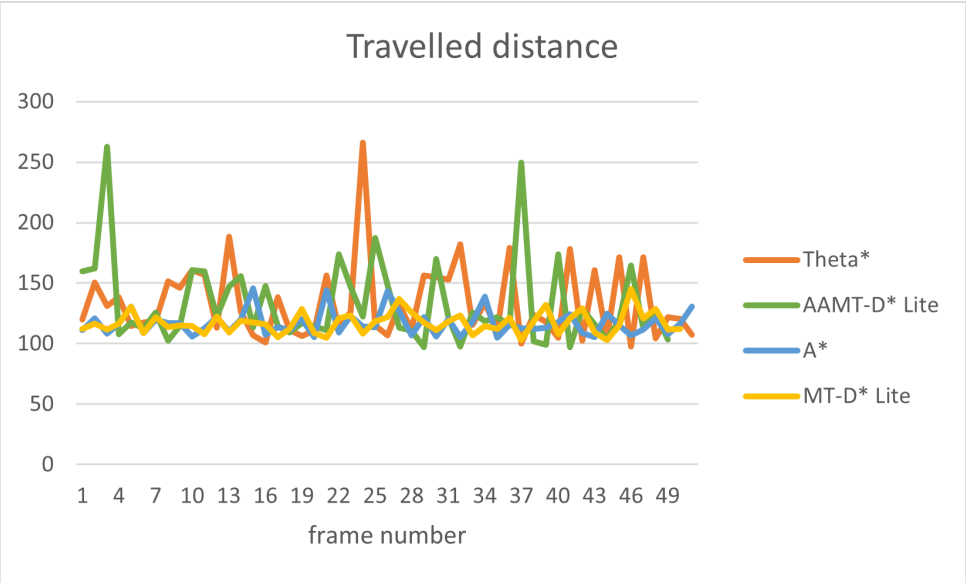


Figure 58: Travelled distance (in Uu) of pathfinding algorithms in the hybrid search graph with $\varepsilon = 1$ and a multiplication factor $\varphi = 1$

6 Evaluation

After having implemented and analysed optimised versions of a selection of pathfinding algorithms and an optimised search graph, the following chapter concludes this master thesis by giving a brief summary, drawing conclusions from the implemented optimisations, and taking a look at what further optimisations or extensions to the pathfinding system could be worked on.

6.1 Summary

The goal of this thesis as defined in chapter 1.3 was to implement a pathfinding system, which is able to meet the challenges of a highly dynamic three-dimensional environment, for example in a game. For this the pathfinding system should be able to update its search graph and recompute a shortest path in a maximum of 16 ms, as this matches the frame duration in a game with 60 FPS. Additionally, the pathfinding system has to run asynchronously, as to not block the main thread.

Before starting to develop such a pathfinding system, chapter 2 detailed important theoretical knowledge required throughout the development process. This included the pathfinding algorithms Jump Point Search (2.2) and (Basic) MT-D* Lite (2.3), as well as an introduction to Unity's C# Job System and Burst compiler (2.1), and the Separating Axis Theorem (2.4).

Afterwards, chapter 3 took a look at the old performance of the pathfinding algorithms and search graphs based on their implementation in the previous bachelor thesis [Kra19] (3.1) and identified the main performance issues (3.2). Hereby only the pathfinding algorithms A*, Theta*, and D* Lite were considered as they were the most promising ones based on the results of the bachelor thesis.

Based on the results of the performance analysis and identified performance issues a concept for optimising the pathfinding system was introduced in chapter 4. The concept included strategies to optimise the used pathfinding algorithms or replace them by more performant ones (4.1), as well as a concept for a new search graph (4.2).

The main work of this thesis was then done in chapter 5. Here the pathfinding system was optimised in multiple steps. First, the pathfinding algorithms were optimised in chapter 5.1 using Unity's C# Job System and Burst compiler, and by replacing some of the algorithm with more performant ones. Then a new, "Hybrid", search graph was introduced and implemented in chapter 5.2. Throughout the chapter its construction (5.2.1), updating process (5.2.2), and the adaptations of the pathfinding algorithms to the search graph (5.2.3) were explained.

Next, a new pathfinding algorithm called Any-angle Moving Target D* Lite was suggested in chapter 5.3, which combined the any-angle characteristics of Theta* with the dynamic advantages of MT-D* Lite. Lastly, chapter 5.4 detailed the final pathfinding update loop and the success rate of the developed pathfinding system.

6.2 Conclusion

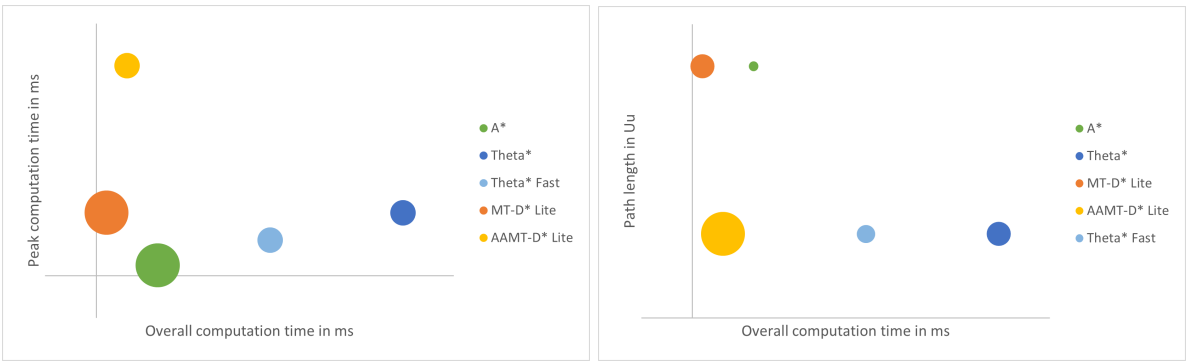
After implementing and analysing different optimised pathfinding algorithms in a cell grid and hybrid search graph the following conclusions can be drawn.

First, using Jump Point Search to optimise the runtime of A* in the cell grid search graph was not successful. The core element of JPS is to find jump points instead of expanding direct neighbours. Even though this element enables JPS to expand significantly fewer nodes than A*, its computation time is much higher than the one of A*. The reason for this is that finding these jump points is very expensive. Even the lower number of expanded nodes can not compensate this performance overhead.

Secondly, in the test environment used in this thesis the hybrid search graph has to be preferred to the cell grid search graph. As discussed in chapter 5.2.2 and shown in figure 45 the hybrid search graph is able to adapt to the changing environment and update its path costs very fast. With about 1.35 ms it is more than 150 times faster in updating than the previously used cell grid search graph. In addition, using the hybrid search graph the pathfinding algorithms A*, Theta* and its Fast version, and (Basic) MT-D* Lite find shorter paths, expand fewer nodes, and have a shorter computation time compared to the cell grid search graph as can be seen in chapter 5.2.3.2. A*, for example, has an almost three times lower computation time in the hybrid search graph than in the cell grid graph (see figure 46iii).

Lastly, the choice of the best pathfinding algorithm is not as clear as the choice of the search graph. The aim of this thesis was to implement a pathfinding system which is able to fully update itself within 16 ms. The only combination of pathfinding algorithm and search graph able to reach this goal reliably in the used test environment is A* in the hybrid search graph. Throughout the development it became clear, though, that meeting this 16 ms goal, is not actually the only factor to measure the suitability of a pathfinding system and especially the choice of the pathfinding algorithms has to account for other factors. Different pathfinding algorithms have different strengths and weaknesses and depending on the use-case a different algorithm might deliver the best results. For example A* is faster than Theta*, but also produces longer paths. Depending on the application one might need a fast algorithm or prefer one producing shorter and more natural looking paths. As an alternative to A* one could also choose

to use (Basic) MT-D* Lite. Both algorithms produce the same path lengths, but differ in their computation times. (Basic) MT-D* Lite has a lower average computation time than A*, but it also has higher peaks in its computation time, which might be problematic in certain scenarios. If one requires an algorithm with a very steady and reliable computation time, then A* is the better choice. If the overall computation time is more important and occasional longer computation times are irrelevant, then (Basic) MT-D* Lite is the better choice. A great compromise between an any-angle algorithms and a dynamic algorithm is AAMT-D* Lite, which combines Theta* with MT-D* Lite and is explained and implemented in chapter 5.3. It produces shorter paths than MT-D* Lite, similar to the way Theta* produces shorter paths than A*. Additionally, it has an overall lower computation time than Theta*, similar to how MT-D* Lite has an overall lower computation time than A*. Due to expensive line-of-sight checks, though, AAMT-D* Lite occasionally shows very high peaks in its computation times. Therefore, here a similar consideration has to be made as before. If one needs an any-angle pathfinding algorithm with a steady and reliable, but slightly slower computation time, then Theta*, or its "Fast" version as described in chapter 5.2.3.2, are good. If the overall computation time is more important, though, and the peaks are not an issue, then AAMT-D* Lite, and especially its "Fast" version as described in chapter 5.3, are a great choice. Figure 59 shows the different strengths and the trade-off between the algorithms. In 59i the algorithms' overall computation time is plotted on the x-Axis, their peak computation times on the y-Axis, and the lengths of their found paths is represented by the size of the bubbles. In 59ii the bubble size represents the peak computation time and the overall computation time and path length are plotted on the x- and y-Axis.



(i) The bubble size shows the path length (in Uu)

(ii) The bubble size shows the peak computation time (in ms)

Figure 59: Overall computation time (in ms), peak computation time (in ms), and path lengths (in Uu) of different pathfinding algorithms using the hybrid search graph, $\varphi = 1$, and $\varepsilon = 1$

Overall, it can be said that the results of this thesis are very positive and the tests run in chapter 5.4 have shown that the implemented pathfinding algorithms in the hybrid search graph have a very high success rate in the used test environment. As already discussed, every use-case will have its own challenges and requirements, therefore it will most likely not be possible to use

an exact copy of the pathfinding system developed in this thesis for every game or application. Instead, the results of this thesis can be used as a guideline and reference to implement a suitable pathfinding system in a highly dynamic 3D space.

6.3 Future work

Even though the optimisations done throughout this thesis are already quite good and have improved the pathfinding system's performance immensely compared to its previous implementation in the bachelor thesis [Kra19], the computation time of the algorithms is not perfect yet. Therefore, the following chapter concludes this thesis by discussing some possible next steps and future work to improve the pathfinding system even further.

One interesting future work would be the optimisation of the JPS pathfinding algorithm by reducing the overhead caused by finding jump points. In their paper "Improving Jump Point Search" [HG14] Daniel Harabor and Alban Grastien propose a number of possible optimisations for JPS including one to find jump points more efficiently. Their proposed solution would require a slightly different representation of the cell grid search graph, though, and whether JPS could be adapted to be used in the hybrid search graph would need to be investigated carefully.

Another interesting future work would be the further optimisation of line-of-sight checks. These checks are needed for any-angle algorithms such as Theta* and AAMT-D* Lite. Even though the method of determining a line-of-sight used in this thesis is a lot more performant than the physics-based one used in the previous bachelor thesis [Kra19], it is still the main performance issue of the any-angle pathfinding algorithms. Not only does its expensive computation cause longer computation times of the algorithms, but it also has difficulty to account for the AI agent's radius causing it to collide with obstacles in rare edge cases.

Next to optimising the runtime of line-of-sight checks, another approach to optimise any-angle algorithms is to reduce the number of required line-of-sight checks significantly or eliminate them all together. One interesting approach would be to transfer the pathfinding algorithm "Anya" presented by Daniel Harabor, Alban Grastien, Dindar Öz, and Vural Aksakalli in their paper "Optimal Any-Angle Pathfinding In Practice" [HG+16] into 3D space. According to the authors this any-angle pathfinding algorithm has shown good results in a 2D search grid. It has not been developed for 3D, yet, though.

One general issue observed during the run of the AI agent through the test environment is that the found paths only ever show the shortest path for the current moment. By the time the AI agent has followed that path a few steps, a dynamic obstacle might block the way, causing the agent to turn around and walk backwards. This could be avoided or minimised if the pathfinding

system not only had information about the currently shortest path, but also accounted for the dynamic objects' movement during the path computation. For this it would be worth taking a look at the concept of time encoded maps presented by Tim Edmonds, Antony Rowstron, and Andy Hopper in their paper "Using Time Encoded Terrain Maps for Cooperation Planning" [ERH98].

Lastly, one might consider implementing the pathfinding system using the GPU more heavily than the CPU, for example by using compute shaders.

Bibliography

[And19] Dave Anderson. Unity-2D-Pathfinding-Grid-ECS-Job: ECS Burst Job System 2D Pathfinding, 2019. Retrieved 06.11.2021 from: https://github.com/Omniaffix-Dave/Unity-2D-Pathfinding-Grid-ECS-Job.

[Ans16] Unity Answers. JsonUtility array not supported?, 2016. Retrieved 09.09.2021 from: https://answers.unity.com/questions/1123326/jsonutility-array-not-supported.html.

[Bit10] William Bittle. SAT (Separating Axis Theorem), 2010. Retrieved 25.09.2021 from: https://dyn4j.org/2010/01/sat/.

[BK+19] Mirza Beig, Bill Kapralos, Karen Collins, and Pejman Mirza-Babaei. G-SpAR: GPU-Based Voxel Graph Pathfinding for Spatial Audio Rendering in Games and VR. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019. Retrieved 06.11.2021 from: https://ieee-cog.org/2019/papers/paper_27.pdf.

[dv03] Gino Johannes Apolonia den van Bergen. *Collision Detection in Interactive 3D Environments*. The Morgan Kaufmann Series in Interactive 3D Technology. Elsevier/Morgan Kaufman, 2003. Retrieved 24.09.2021 from: http://www.r-5.org/files/books/computers/algo-list/game-development/Gino_van_den_Bergen-Collision_Detection_in_Interactive_3D_Environments-EN.pdf.

[Ebe99] David Eberly. Dynamic Collision Detection using Oriented Bounding Boxes, 1999. Retrieved 25.09.2021 from: https://www.geometrictools.com/Documentation/DynamicCollisionDetection.pdf.

[ERH98] Tim Edmonds, Antony Rowstron, and Andy Hopper. Using time-encoded terrain maps for cooperation planning. *Advanced Robotics*, 13(8):779–792, 1998. Retrieved 09.09.2021 from: https://www.cl.cam.ac.uk/research/dtg/www/files/publications/public/files/tr.2000.16.pdf.

[Fre19a] Andreas Fredriksson. Intrinsics: Low-level engine development with Burst - Unite Copenhagen, 2019. Retrieved 27.08.2021 from: https://de.slideshare.net/unity3d/intrinsics-lowlevel-engine-development-with-burst.

[Fre19b] Andreas Fredriksson. Intrinsics: Low-level engine development with Burst - Unite Copenhagen, 2019. Retrieved 03.10.2021 from: https://www.youtube.com/watch?v=BpwvXkoFcp8.

[Gau19] Chris Gaudino. Maximizing the Benefit of C# Jobs Using Unity's New PlayerLoop API. *Grizzly Machine*, 2019. Retrieved 06.10.2021 from: https://www.grizzly-machine.com/entries/maximizing-the-benefit-of-c-jobs-using-unitys-new-playerloop-api.

[Gra19] Graceful Algorithms. User's manual for "PathFinder 3D": Version 0.4, 2019. Retrieved 06.11.2021 from: https://gracefulalgs.com/wp-content/uploads/2019/01/User_39_s_manual.pdf.

[Ham19] Lee Hammerton. Getting started with Burst - Unite Copenhagen, 2019. Retrieved 03.10.2021 from: https://www.youtube.com/watch?v=Tzn-nX9hK1o.

[HG11] Daniel Harabor and Alban Grastien. Online Graph Pruning for Pathfinding On Grid Maps. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, volume 2, pages 1114–1119. Association for the Advancement of Artificial Intelligence (AAAI), 01 2011. Retrieved 15.08.2021 from: https://www.researchgate.net/publication/221603063_Online_Graph_Pruning_for_Pathfinding_On_Grid_Maps.

[HG14] Daniel Harabor and Alban Grastien. Improving Jump Point Search. *Proceedings International Conference on Automated Planning and Scheduling, ICAPS*, 2014:128–135, 2014. Retrieved 03.09.2021 from: https://users.cecs.anu.edu.au/~dharabor/data/papers/harabor-grastien-icaps14.pdf.

[HG+16] Daniel Damir Harabor, Alban Grastien, Dindar Öz, and Vural Aksakalli. Optimal Any-Angle Pathfinding In Practice. *Journal of Artificial Intelligence Research*, 56:89–118, 2016. Retrieved 27.10.2021 from: https://jair.org/index.php/jair/article/view/11004/26163.

[Huy08] Johnny Huynh. Separating Axis Theorem for Oriented Bounding Boxes, 2008. Retrieved 22.09.2021 from: https://www.jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20Oriented%20Bounding%20Boxes.pdf.

[KL02] Sven Koenig and Maxim Likhachev. D* Lite. In *Eighteenth National Conference on Artificial Intelligence*, pages 476–483, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence. Retrieved 18.01.2019 from: http://idm-lab.org/bib/abstracts/papers/aaai02b.pdf.

[Kra19] Carina Krafft. Implementation and comparison of pathfinding algorithms in a dynamic 3D space, 2019. Retrieved 15.08.2021 from: https://github.com/CarinaKr/Pathfinding-in-dynmic-3D-space/blob/master/Implementation_and_comparison_of_pathfinding_algorithms_in_a_dynamic_3D_space.pdf.

[Kra21] Carina Krafft. Pathfinding-in-dynmic-3D-space, 2021. Retrieved 02.12.2021 from: https://github.com/CarinaKr/Pathfinding-in-dynmic-3D-space.

[Kry20] Gerald Krystian. NativeHeap, 2020. Retrieved 25.08.2021 from: https://github.com/Amarcolina/NativeHeap.

[LLV21] The LLVM Compiler Infrastructure Project, 2021. Retrieved 19.11.2021 from: https://llvm.org/.

[LW+17] Sikang Liu, M. Watterson, K. Mohta, Ke Sun, S. Bhattacharya, C. J. Taylor, and V. Kumar. Planning Dynamically Feasible Trajectories for Quadrotors Using Safe Flight Corridors in 3-D Complex Environments. *IEEE Robotics and Automation Letters*, 2:1688–1695, 2017. Retrieved 07.11.2021 from: https://ieeexplore.ieee.org/ielaam/7083369/7875382/7839930-aam.pdf.

[Rie13] Calvin Rien. MiniJSON.cs, 2013. Retrieved 09.09.2021 from: https://gist.githubusercontent.com/darktable/1411710/raw/d7c8c5fd25d86031e52883f4e69c31234b5e735c/MiniJSON.cs.

[She16] Kevin Sheehan. jps, 2016. Retrieved 03.09.2021 from: https://github.com/kevinsheehan/jps.

[Sta21] Jannik Staub. DungeonBurst (Building DungeonKeeper with Unity ECS and Burst) Part 2 – AI and Pathfinding, 2021. Retrieved 06.11.2021 from: https://tech.innogames.com/dungeonburst-building-dungeonkeeper-with-unity-ecs-and-burst-part-2-ai-and-pathfinding.

[SYK10] Xiaoxun Sun, William Yeoh, and Sven Koenig. Moving Target D* Lite. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, pages 67–74, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems. Retrieved 22.02.2019 from: http://idm-lab.org/bib/abstracts/papers/aamas10a.pdf.

[Tec20] Unity Technologies. Unity - Scripting API: Unity.Jobs.JobHandle.ScheduleBatchedJobs, 2020. Retrieved 03.10.2021 from: https://docs.unity3d.com/2019.3/Documentation/ScriptReference/Unity.Jobs.JobHandle.ScheduleBatchedJobs.html.

[Tec21a] Unity Technologies. DOTS packages| Unity, 2021. Retrieved 03.10.2021 from: https://unity.com/dots/packages.

[Tec21b] Unity Technologies. Unity - Manual: Burst User Guide, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Packages/com.unity.burst@1.3/manual/index.html#synchronous-compilation.

[Tec21c] Unity Technologies. Unity - Manual: C# Job System, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Documentation/Manual/JobSystem.html.

[Tec21d] Unity Technologies. Unity - Manual: C# Job System Overview, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Documentation/Manual/JobSystemOverview.html.

[Tec21e] Unity Technologies. Unity - Manual: C# Job System tips and troubleshooting, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Documentation/Manual/JobSystemTroubleshooting.html.

[Tec21f] Unity Technologies. Unity - Manual: Creating jobs, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Documentation/Manual/JobSystemCreatingJobs.html.

[Tec21g] Unity Technologies. Unity - Manual: NativeContainer, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Documentation/Manual/JobSystemNativeContainer.html.

[Tec21h] Unity Technologies. Unity - Manual: Navigation and Pathfinding, 2021. Retrieved 05.11.2021 from: https://docs.unity3d.com/Manual/Navigation.html.

[Tec21i] Unity Technologies. Unity - Manual: ParallelFor jobs, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Documentation/Manual/JobSystemParallelForJobs.html.

[Tec21j] Unity Technologies. Unity - Manual: Scheduling jobs, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Documentation/Manual/JobSystemSchedulingJobs.html.

[Tec21k] Unity Technologies. Unity - Manual: The safety system in the C# Job System, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Documentation/Manual/JobSystemSafetySystem.html.

[Tec21l] Unity Technologies. Unity - Manual: What is a job system?, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Documentation/Manual/JobSystemJobSystems.html.

[Tec21m] Unity Technologies. Unity - Scripting API: EarlyUpdate, 2021. Retrieved 06.10.2021 from: https://docs.unity3d.com/ScriptReference/PlayerLoop.EarlyUpdate.html.

[Tec21n] Unity Technologies. Unity - Scripting API: JsonUtility, 2021. Retrieved 09.09.2021 from: https://docs.unity3d.com/ScriptReference/JsonUtility.html.

[Tec21o] Unity Technologies. Unity - Scripting API: Mesh.normals, 2021. Retrieved 22.09.2021 from: https://docs.unity3d.com/ScriptReference/Mesh-normals.html.

[Tec21p] Unity Technologies. Unity - Scripting API: Mesh.triangles, 2021. Retrieved 22.09.2021 from: https://docs.unity3d.com/ScriptReference/Mesh-triangles.html.

[Tec21q] Unity Technologies. Unity - Scripting API: Mesh.vertices, 2021. Retrieved 22.09.2021 from: https://docs.unity3d.com/ScriptReference/Mesh-vertices.html.

[Tec21r] Unity Technologies. Unity - Scripting API: MonoBehaviour.Update(), 2021. Retrieved 06.10.2021 from: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html.

[Tec21s] Unity Technologies. Unity - Scripting API: Physics.OverlapBox, 2021. Retrieved 07.09.2021 from: https://docs.unity3d.com/ScriptReference/Physics.OverlapBox.html.

[Tec21t] Unity Technologies. Unity - Scripting API: PlayerLoop, 2021. Retrieved 06.10.2021 from: https://docs.unity3d.com/ScriptReference/LowLevel.PlayerLoop.html.

[Tec21u] Unity Technologies. Unity - Scripting API: PostLateUpdate, 2021. Retrieved 06.10.2021 from: https://docs.unity3d.com/ScriptReference/PlayerLoop.PostLateUpdate.html.

[Tec21v] Unity Technologies. Unity - Scripting API: PreLateUpdate, 2021. Retrieved 06.10.2021 from: https://docs.unity3d.com/ScriptReference/PlayerLoop.PreLateUpdate.html.

[Tec21w] Unity Technologies. Unity - Scripting API: Profiler, 2021. Retrieved 03.11.2021 from: https://docs.unity3d.com/ScriptReference/Profiling.Profiler.html.

[Tec21x] Unity Technologies. Unity - Scripting API: SpherecastCommand, 2021. Retrieved 18.11.2021 from: https://docs.unity3d.com/Documentation/ScriptReference/SpherecastCommand.html.

[Tec21y] Unity Technologies. Unity - Scripting API: Unity.Jobs.JobHandle.Complete, 2021. Retrieved 03.10.2021 from: https://docs.unity3d.com/Documentation/ScriptReference/Unity.Jobs.JobHandle.Complete.html.

[Tec21z] Unity Technologies. Unity - Scripting API: Unity.Mathematics, 2021. Retrieved 03.10.2021 from: https://docs.unity.cn/Packages/com.unity.mathematics@1.2/api/Unity.Mathematics.html.

[Ver17] Steve Vermeulen. How to use Async-Await instead of coroutines in Unity3d 2017, 2017. Retrieved 08.09.2021 from: http://www.stevevermeulen.com/index.php/2017/09/using-async-await-in-unity3d-2017/.

[Ver18] Steve Vermeulen. Async Await Support - Asset Store, 2018. Retrieved 08.09.2021 from: https://assetstore.unity.com/packages/tools/integration/async-await-support-101056.

Appendix

```
private void UpdateNeighboursOfNode(Node node)
{
    foreach(Node neighbour in node)
    {
        double minRHS = Mathf.Infinity;
        Node bestParentNode = null;

        foreach(Node predecessor in neighbour)
        {
            double rhs = predecessor.gValue + predecessor.pathCosts[neighbour];
            if(rhs < minRHS)
            {
                minRHS = rhs;
                bestParentNode = predecessor;
            }
        }

        neighbour.rhsValue = minRHS;
        neighbour.parent = bestParentNode;
    }
}
```

Listing 22: Updating *rhs*-values of a node's neighbours

```
private void UpdateNeighboursOfNode(Node node)
{
    foreach(Node neighbour in node)
    {
        double rhs = node.gValue + node.pathCost[neighbour];
        if (rhs < neighbour.rhsValue)
        {
            neighbour.rhsValue = rhs;
            neighbour.parent = node;
        }
}
```

Listing 23: An optimised implementation of updating *rhs*-values of an over-consistent node's neighbours

```
private void UpdateNeighboursOfNode(int index)
{
    double3x2 newRhs = nodes[index].gValue + nodes[nodeIndex].pathCosts;
    double3x2 oldRhs = new double3x2(nodes[nodes[index].neighbours[0][0]].
    rhsValue,
            nodes[nodes[index].neighbours[1][0]].rhsValue,
            nodes[nodes[index].neighbours[0][1]].rhsValue,
            nodes[nodes[index].neighbours[1][1]].rhsValue,
            nodes[nodes[index].neighbours[0][2]].rhsValue,
            nodes[nodes[index].neighbours[1][2]].rhsValue);
    bool3x2 useNewRhsValue = newRhs < oldRhs;

    for (int i = 0; i < neighboursCount; i++)
    {
        int2 pos = CellGridHelper.IndexToPosition(i, neighboursCountSize);
        if (useNewRhsValue[pos.x][pos.y])
        {
            CellNode tempNode = nodes[nodes[index].neighbours[pos.x][pos.y]];
            tempNode.rhsValue = newRhsValues[pos.x][pos.y];
            tempNode.parentNode = nodeIndex;
            nodes[nodes[index].neighbours[pos.x][pos.y]] = tempNode;
        }
    }
}
```

Listing 24: An optimised implementation of updating *rhs*-values of an over-consistent node's neighbours using burst compiler

```
public static NativeArray<CellNode> UpdateHeuristic(NativeArray<CellNode> nodes,
    float3 goal, float inflate)
{
    NativeArray<CellNode> updatedNodes = nodes;

    float4 goalX = goal.xxxx;
    float4 goalY = goal.yyyy;
    float4 goalZ = goal.zzzz;
    NodesGroup nodesGroup;
    for (int i = 0; i < nodes.Length; i += 4)
    {
        nodesGroup = new NodesGroup(nodes[i].globalPosition,
                                    nodes[i+1].globalPosition,
                                    nodes[i+2].globalPosition,
                                    nodes[i+3].globalPosition);
        float4 diffX = goalX - nodesGroup.posXs;
        float4 diffY = goalY - nodesGroup.posYs;
        float4 diffZ = goalZ - nodesGroup.posZs;

```

```
        float4 squDistanceX = diffX * diffX;
        float4 squDistanceY = diffY * diffY;
        float4 squDistanceZ = diffZ * diffZ;

        float4 distance = math.sqrt(squDistanceX + squDistanceY + squDistanceZ);

        nodesGroup.plainHeuristics = distance;
        nodesGroup.heuristics = distance * inflate;

        for (int j = 0; j < 4; j++)
        {
            CellNode node = nodes[i+j];
            node.heuristic = nodesGroup.heuristics[j];
            node.plainHeuristic = nodesGroup.plainHeuristics[j];
            updatedNodes[i+j] = node;
        }
    }
}


public struct NodesGroup
{
    public float4 posXs;
    public float4 posYs;
    public float4 posZs;

    public float4 heuristics;
    public float4 plainHeuristics;

    public NodesGroup(float3 pos1, float3 pos2, float3 pos3, float3 pos4)
    {
        posXs = new float4(pos1.x, pos2.x, pos3.x, pos4.x);
        posYs = new float4(pos1.y, pos2.y, pos3.y, pos4.y);
        posZs = new float4(pos1.z, pos2.z, pos3.z, pos4.z);

        heuristics = 0;
        plainHeuristics = 0;
    }
}
```

Listing 25: Updating the heuristic of all nodes in the search graph

```
private void FillPathCostsAndParents(int nodeIndex, int3x2 neighbours,
                                     int nodeParentIndex, ref double3x2 baseGValue)
{
    CellNode node = nodes[nodeIndex];
    CellNode parent = nodes[nodeParentIndex];
    CellNode neighbour;

    for (int i = 0; i < 2; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            neighbour = nodes[neighbours[i][j]];

            //neighbour is blocked or outside of grid
            if (node.pathCosts[i][j].Equals(math.INFINITY_DBL)
                || !neighbour.isFree
                || neighbour.isInClosedList)
            {
                continue;
            }

            //check if neighbour, node, and parent are on a straight line
            bool4 direction = false;
            direction.yzw = parent.localPosition == neighbour.localPosition;
            direction.yzw &= node.localPosition == neighbour.localPosition;
            int dir = math.bitmask(direction);

            //neighbour is on a straight line from the parent of the current
            //node or passes a line-of-sight check
            if (math.countbits(dir) == 2
                || LineOfSight(parent.localPosition, neighbour.localPosition))
            {
                newParents[i][j] = nodeParentIndex;
                float3 diff = (parent.globalPosition-neighbour.globalPosition);
                pathCosts[i][j] = math.sqrt(math.csum(diff * diff));

                baseGValue[i][j] = parent.gValue;
            }
        }
    }
}
```

Listing 26: *FillPathCostsAndParents* function of Theta*

```
private int GetFirstUsedNode(int index, int3 direction, ref int stepCount)
{
    stepCount++;
    index = (allNodes[index].localPosition + direction).PositionToIndex(
    searchSpace);
    if (index == -1 || !allNodes[index].isFree)
    {
        return -1;
    }

    if (allNodes[index].isStaticNode
        || allNodes[index].isDynamicNeighbourNode
        || allNodes[index].isDynamicNode)
    {
        return index;
    }

    return GetFirstUsedNode(index, direction, ref stepCount);
}
```

Listing 27: *GetFirstUsedNode* function with overload for one direction

```
private void UpdateDynamicNodeRhs(int nodeIndex)
{
    collectedParents.Clear();
    HybridNode node = nodes[nodeIndex];
    HybridNodePathCosts nodePathCosts = nodePathCosts[nodeIndex];
    int3x2 predecessors = node.predecessors;
    double3x2 gPred = GetGValues(predecessors);

    double3x2 rhsValues = gPred + nodePathCosts.pathCosts;
    double neighbourRhsValue = math.min(math.cmin(rhsValues.c0),
                                        math.cmin(rhsValues.c1));

    int4x3 edgePredecessors = node.edgePredecessors;
    double4x3 edgeGPred = GetGValues(edgePredecessors);

    double4x3 edgeRhsValues = edgeGPred + nodePathCosts.edgePathCosts;
    double edgeRhsValue = Minimum(math.cmin(edgeRhsValues.c0),
                                  math.cmin(edgeRhsValues.c1),
                                  math.cmin(edgeRhsValues.c2));

    int4x2 vertexPredeceesors = node.verticesPredecessors;
    double4x2 vertexGPred = GetGValues(vertexPredecessors);

    double4x2 vertexRhsValues = vertexGPred + nodePathCosts.verticesPathCosts;
    double vertexRhsValue = math.min(math.cmin(vertexRhsValues.c0),
```

```
                                    math.cmin(vertexRhsValues.c1));

    double rhsValue = Minimum(neighbourRhsValue, edgeRhsValue, vertexRhsValue);

    double minRHS = rhsValue;
    int bestParent = -1;
    if(!rhsValue.Equals(math.INFINITY_DBL))
    {
        CollectParents(predecessors);
        CollectParents(edgePredecessors);
        CollectParents(vertexPredecessors);
        collectedParents.Remove(-1);
        collectedParents.Remove(nodeIndex);

        var enumerator = collectedParents.GetEnumerator();
        while (enumerator.MoveNext())
        {
            int predParent = enumerator.Current;
            double rhs = nodes[predParent].gValue +
                Distance(node.localPosition, nodes[predParent].localPosition);
            if (rhs <= minRHS &&
                LineOfSight(node.localPosition,nodes[predParent].localPosition))
            {
                minRHS = rhs;
                bestParent = predParent;
            }
        }
        enumerator.Dispose();
    }

    int parent;
    if (minRHS.Equals(math.INFINITY_DBL))
    {
        parent = -1;
    }
    else if (bestParent != -1)
    {
        parent = bestParent;
    }
    else if (neighbourRhsValue.Equals(rhsValue))
    {
        bool3x2 isParentNode = rhsValues == neighbourRhsValue;
        int3 parentsColumn0 = math.select(-1, predecessors.c0, isParentNode.c0);
        int3 parentsColumn1 = math.select(-1, predecessors.c1, isParentNode.c1);
        parent = math.max(math.cmax(parentsColumn0), math.cmax(parentsColumn1));
    }
    else if (edgeRhsValue.Equals(rhsValue))
```

```
73      {
74          parent = GetParentToRhsValue(rhsValue, edgeRhsValues, edgePredecessors);
75      }
76      else
77      {
78          parent = GetParentToRhsValue(rhsValue, vertexRhsValues,
79                                       vertexPredecessors);
80      }
81
82      UpdateParent(node.index, parent);
83      node.parentNode = parent;
84      node.rhsValue = rhsValue;
85      nodes[nodeIndex] = node;
86      UpdateNodeInOpenList(nodeIndex);
87  }
```

Listing 28: Determining a *dynamic* node's *rhs*-value and parent in AAMT-D* Lite

```
1   private void UpdateStaticNodeRhs(int nodeIndex)
2   {
3       collectedParents.Clear();
4       double minRHS = math.INFINITY_DBL;
5       int bestParentNode = -1;
6       HybridNode node = nodes[nodeIndex];
7
8       //get minimum $rhs$-value based on predecessors
9       NativeMultiHashMap<int, int>.Enumerator enumerator =
10                                  predecessorsMap.GetValuesForKey(nodeIndex);
11
12      int counter = 0;
13      double4 gValues = math.INFINITY_DBL;
14      int4 parents = -1, posXs = 0, posYs=0, posZs=0;
15      while (enumerator.MoveNext())
16      {
17          int pred = enumerator.Current;
18          if (nodes[pred].gValue > float.MaxValue)
19          {
20              continue;
21          }
22          collectedParents.Add(nodes[enumerator.Current].parentNode);
23
24          FillPossibleParentsValues(counter, pred, ref gValues, ref  parents,
25                                    ref posXs, ref posYs, ref posZs);
26          counter++;
27
28          if (counter != 4)
29          {
```

```
30              continue;
31          }
32          counter = 0;
33
34          double4 distanceBasedRhs = GetDistanceBasedRhs(node.localPosition,
35                                      posXs, posYs, posZs, gValues,
36                                      out double minDistanceBasedRhs);
37          if (minDistanceBasedRhs < minRHS)
38          {
39              continue;
40          }
41
42          bool4 checkPred = distanceBasedRhs < minRHS;
43          for (int i = 0; i < 4; i++)
44          {
45              if(checkPred[i])
46              {
47                  double rhs = gValues[i] + GetPathCost(parents[i], nodeIndex);
48                  if (rhs < minRHS)
49                  {
50                      minRHS = rhs;
51                      bestParentNode = parents[i];
52                  }
53              }
54          }
55      }
56
57      for (int i = 0; i < counter; i++)
58      {
59          HybridNode parent = nodes[parents[i]];
60          if (gValues[i] + Distance(parent.localPosition, node.localPosition)
61                  < minRHS)
62          {
63              double rhs = parent.gValue + GetPathCost(parents[i], nodeIndex);
64              if (rhs < minRHS)
65              {
66                  minRHS = rhs;
67                  bestParentNode = parents[i];
68              }
69          }
70      }
71      enumerator.Dispose();
72
73      collectedParents.Remove(-1);
74      collectedParents.Remove(nodeIndex);
75      collectedParents.Remove(node.parentNode);
76
```

```
77      //get minimum $rhs$-value based on predecessors' parents
78      counter = 0;
79      gValues = math.INFINITY_DBL;
80      parents = -1;
81      predPosXs = 0; predPosYs = 0; predPosZs = 0;
82      var collectedParentsEnumerator = collectedParents.GetEnumerator();
83      while (collectedParentsEnumerator.MoveNext())
84      {
85          int predParent = collectedParentsEnumerator.Current;
86          FillPossibleParentsValues(counter, predParent, ref gValues, ref parents,
87                                    ref predPosXs, ref predPosYs, ref predPosZs);
88          counter++;
89
90          if (counter != 4)
91          {
92              continue;
93          }
94          counter = 0;
95
96          double4 distanceBasedRhs =  GetDistanceBasedRhs(node.localPosition,
97                                      predPosXs, predPosYs, predPosZs, gValues,
98                                      out double minDistanceBasedRhs);
99          if (minDistanceBasedRhs < minRHS)
100         {
101             continue;
102         }
103
104         bool4 checkPred = distanceBasedRhs < minRHS;
105         for (int i = 0; i < 4; i++)
106         {
107             if(checkPred[i] &&
108                LineOfSight(node.localPosition, nodes[predParent].localPosition))
109             {
110                 minRHS = distanceBasedRhs[i];
111                 bestParentNode = parents[i];
112             }
113         }
114     }
115     for (int i = 0; i < counter; i++)
116     {
117         HybridNode parent = nodes[parents[i]];
118         double rhs = parent.gValue +
119                          Distance(node.localPosition, parent.localPosition);
120         if (rhs < minRHS
121             && LineOfSight(node.localPosition, parent.localPosition))
122         {
123             minRHS = rhs;
```

```
124                 bestParentNode = parents[i];
125             }
126         }
127         collectedParentsEnumerator.Dispose();
128 
129         tempNode.rhsValue = minRHS;
130         UpdateParent(nodeIndex, bestParentNode);
131         node.parentNode = bestParentNode;
132         nodes[nodeIndex] = node;
133         UpdateNodeInOpenList(nodeIndex);
134 }
```

Listing 29: Determining a *dynamic neighbour* or *static* node's *rhs*-value in AAMT-D* Lite

```
1 private void FillPossibleParentsValues(int index, int parent, ref double4
      gValues, ref int4 possibleParents, ref int4 parentPosXs, ref int4
      parentPosYs, ref int4 parentPosZs)
2 {
3     gValues[index] = nodes[parent].gValue;
4     possibleParents[index] = parent;
5     parentPosXs[index] = nodes[parent].localPosition.x;
6     parentPosYs[index] = nodes[parent].localPosition.y;
7     parentPosZs[index] = nodes[parent].localPosition.z;
8 }
```

Listing 30: Writing information about a parent's position and g-value into matrices

Hereby I confirm that I completed the presented master thesis with the title:

"Optimisation of pathfinding in a dynamic 3D space"

independently without outside help and that I have not used any sources or aids other than those indicated. Passages taken verbatim from other works or passages borrowed in meaning are clearly indicated by source citations.

Hamburg, 08.12.2021 Carina Krafft

Hiermit versichere ich, die vorliegende Master-Thesis mit dem Titel:

"Optimisation of pathfinding in a dynamic 3D space"

selbständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.