NavNets: 3D Path-planning system

# NavNets: 3D Path-planning system

By Thomas Gwosdz,

*A Thesis Submitted to the School of Graduate Studies in the Partial Fulfillment of the Requirements for the Degree Masters of Applied Science*

McMaster University
Masters of Applied Science (2019)
Hamilton, Ontario (Department of Computing and Software)

TITLE: NavNets: 3D Path-planning system
AUTHOR: Thomas Gwosdz (McMaster University)
SUPERVISOR: Martin von Mohrenschildt
NUMBER OF PAGES: v, 105

# Abstract

The current state of 3D path-planning leaves room for improvement. To navigate a 3D environment, techniques which were developed for 2D navigation are used and slightly adapted to generate convincing motion. However, these techniques often constrict the motion to a single plane. This constriction is not only a limitation, but also increases the error. We created a new method to compute a path in a 3D world without a planar constraint. We will discuss the computation of a Navigation Volume Network (NavNet), and how it finds a path. A NavNet is the 3D generalization of NavMeshes, and holds boundary and connection information which is utilized when planning a path for motion. Similar to how NavMeshes allow path-planning by simplifying the ground meshes, the NavNet simplifies the search space by approximating the 3D world through sampling.

# *Acknowledgements*

I would like to thank my wife, who has been wonderfully supportive and patient during this time. My parents, who have encouraged me in my academic pursuit. And my supervisor, who helped me make this possible.

# Declaration of Authorship

I, Thomas Gwosdz, declare that this thesis titled, "NavNets: 3D Path-planning system" and the work presented in it are my own. I confirm that:

- I have written each chapter, with input and guidance of my supervisor.
- That I have developed the prototypes used for testing and result gathering.
- That I have performed the analysis of the data.
- That I have given credit where credit is due via references.

# Chapter 1

# Introduction

## 1.1 Introduction

Navigation Meshes (NavMesh) have become a standard tool available in simulation systems where simple artificial intelligence (AI) is used to control the movement of some entity Menard 2012. They are able to calculate a path efficiently by reducing the complexity of the world geometry. NavMeshes achieve this by representing a section of the ground terrain as a single convex polygon. This polygon approximates the general shape of the terrain section while greatly reducing the number of vertices. Any area of the ground that is not approximated by this polygon, will either be approximated by another simple convex polygon, or simply cannot be traversed. Areas without polygon cover will not be traversable. These polygons do not overlap, but rather will share a common edge where the polygons would intersect. They are connected via these shared edges, and will allow a path from one polygon to the next polygon, intersecting the edge. This allows a path to be calculated between two positions by means of this polygon mesh. The result is that path queries are calculated significantly faster when compared to using the more detailed environment mesh. Performance is also increased since the search space has been reduced. However, by doing so, NavMeshes limit the entity to a 2D plane of motion Alt et al. 2008.

For most applications of path planning, the limitations of NavMeshes are acceptable as movement is limited to the xy-plane due to gravity. Multi-story levels can be navigated with a stack of NavMeshes, one on top of the other Toll et al. 2011. These NavMeshes are able to provide information about the environment to facilitate both global as well as local navigation and static obstacle avoidance.

If the path however is not constricted by gravity, then NavMeshes cannot be used to reduce the complexity of the search space. Examples of such cases include drones, birds, schools of fish, or any kind of simulation occurring in space. Ideally, we would like to have a similar method which is able to offer information for navigation and obstacle avoidance in the above cases while also increasing performance by reducing the search space.

NavNets are our solution.

## 1.2 Application

The applications of a 3D pathfinding system is both currently limited, and at the same time potentially unlimited in the future. As mentioned above, most applications can be expressed as a 2D motion problem. Videogames use these pathfinders for the non-player characters to either fight or help the player. In architecture, crowd simulation is used to optimize people flow. And in history, battle outcomes are validated using simulations based on a factions tactics and equipment. However, in all these scenarios the third dimension is not used as humans cannot fly or levitate. And for video games where Zero Gravity motion is available, the non-player characters often behave too simple in my opinion. However, as space exploration becomes more accessible to the public, tourist space stations, space ports, and other zero gravity structures will need to be designed with a similar methodology as their earthly counterparts. In these scenarios, NavNets will be the prime candidate to facilitate pathfinding for these simulated entities. And in a closer timeframe, NavNets should allow for a wide spread increase in building of zero gravity video games, as a viable solution to the challenge of 3D pathfinding now exists.

## 1.3 Problem

Path-finding and path-planning are well understood problems in 2D Rabin 2017. Given a set of line segments which will act as boundaries, a starting point, and a target point, the shortest path can be calculated. Aside from the trivial case, where no obstruction exists between the start and end, the path will snake through the environment. As the path may not cross the boundary lines, the vertices are used to move around these line segments. A complete planar graph can be constructed, removing any graph edges which cross a boundary. When we express the problem space in terms of N edges, we will have $2^{2N}$ possible paths which can be taken from start to target; among them will be the shortest path. Many algorithms have been developed to reduce the runtime cost, and luckily we do not need to calculate $2^{2N}$ paths. Often a greedy algorithm will provide satisfactory paths which may not be the shortest path, but are good enough. Figure 1.1. illustrates a 2D example and show a subset of the problem space. It also illustrates how the good enough solutions, and correct solution vary by an insignificant amount amount.

In 3D however, this well understood problem becomes trickier. Boundaries are no longer simple edges, they are three dimensional triangles. Using the same logic as above, the problem space will be $2^{3N}$ if we express a boundary as a triangle and make use of its vertices. However, while this approach does provide us with many paths, it does not provide a complete set of paths. Figure 1.2. illustrates a simple 3D scene for pathfinding. Figure 1.2b. especially shows how even for only a single boundary of the obstacle, we

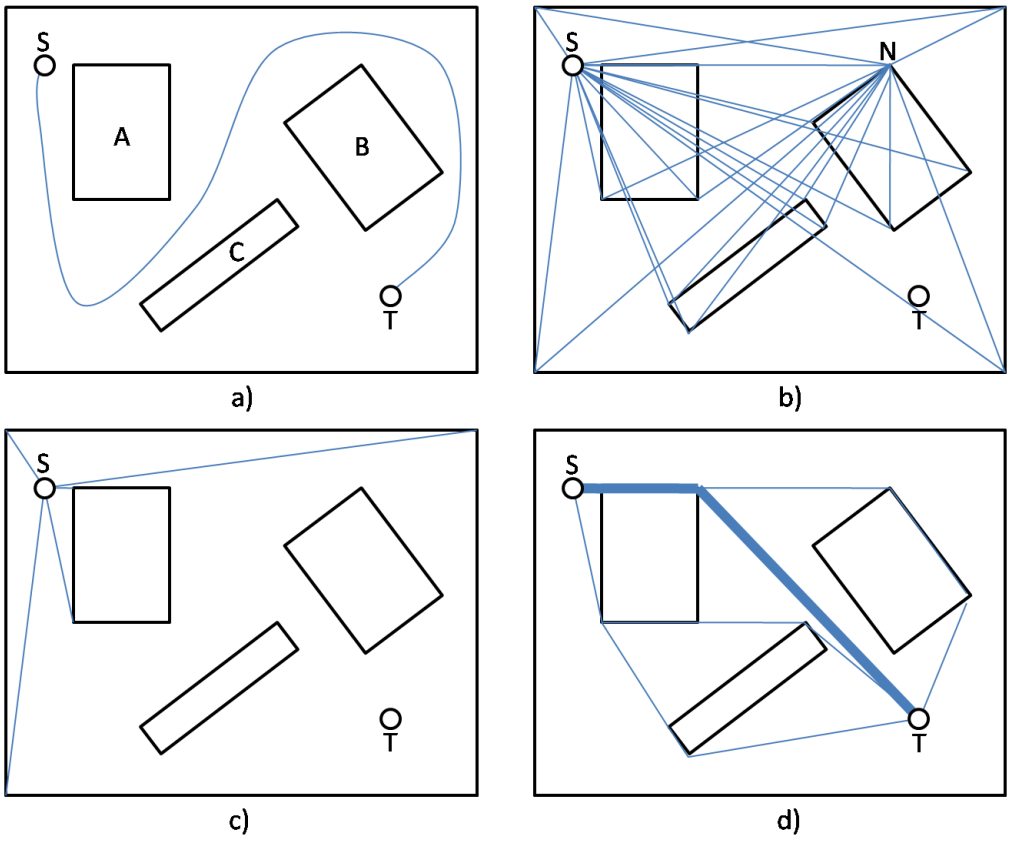


FIGURE 1.1: Finding a path in 2D. **a)** A simple 2D scene. A (S)tart point, a (T)arget point as well a number of obstacles (A, B, C) are placed in the scene. A path from S to T can be constructed and it goes through the scene, however it is not the shortest path. **b)** Using the vertices of the obstacles, a complete graph is generated. Here, only two nodes (the start node, and an additional node N) are connected to every other vertex in the scene. It is just a portion of the whole graph. **c) For illustration purposes, only the start node is shown. All edges which intersected with any boundary (edge) of an obstacle has been removed as it would be invalid.** . **d)** It can be concluded that the best candidate paths move from S to T and try to minimize the total length. In the scene, four paths are candidates (thin lines), however only one (**bold** is the shortest path. The vertices are used as turning points for path adjustment.

can in fact have infinitely many paths on the surface. A shortest path can fold over an edge of a triangle. We require more information to prevent this set explosion, as well as to find a shortest path in 3D.

To further increase difficulty, at the time of writing, there are no standardized benchmarks to evaluate the performance of a 3D pathfinding solution. There is no elegant

solution available as of now for this hard problem. In order to gain an understanding of the problem, we will implement several prototypes and compare and contrast their performance as outlined in section 7.1.

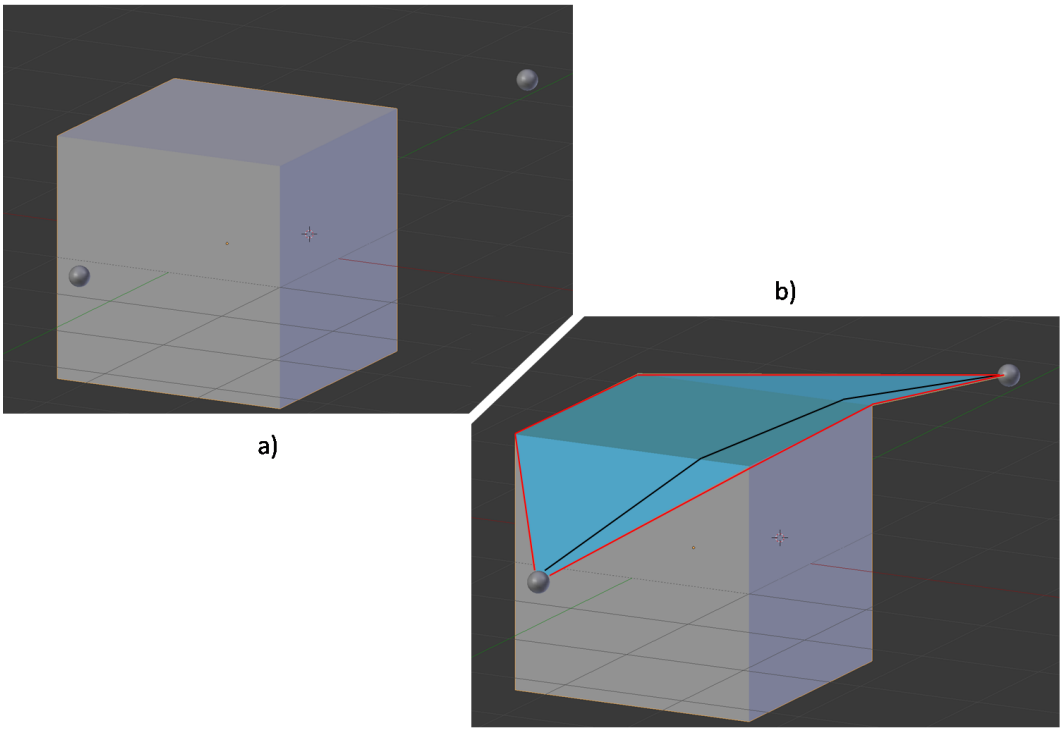


Figure 1.2: A simple scene in 3D. **a)** A start and target sphere are located on opposite sides of a cube. A shortest path has to go around the cube. **b)** If a node graph is constructed using the boundaries vertices, to mimic the 2D approach, paths which lie along the edges of the cube are discovered (red). For illustration purposes, a surface is formed and bounded by the red paths (blue surface). Neither red line is a shortest path. The true shortest path in 3D has to cross the edges of the cube, not using the vertices at all. The path will lie on the blue surface, however where precisely it needs to bend is not trivial.

## 1.4 Contribution

This work will contribute the following to the area of 3D pathfinding:

- A review of the current pathfinding methods in the literature. As well as defining why 3D pathfinding is a difficult problem to solve.

- A summery of where the current state of the art has shortcomings, and when they are advantageous.
- A clear theoretical framework used to describe and analyze the structures and algorithms which were developed to solve the pathfinding problem in 3D. This is in contrast to the implementation first approach often used in this area.
- A system capable of solving a 3D pathfinding problem efficiently for a zero-gravity environment.
- Several prototypes which are used to investigate and better understand the problem, and to test possible solutions or approaches to the problem.
- Tools that are used to read the intermediate data and provide visualization of it, as well as to display the solution.

# Chapter 2

# Literature

Autonomous navigation does not require human input for an agent to transverse an environment between two points. Such agents can be robots, they can be entities in a simulator, as well as non-player characters in video games. Autonomous navigation spans both the real world, as well as the virtual world. However, agents in a virtual environment can have a big advantage, they can have a global overview of the world Seemann 2004. Such an overview allows for the possibility to find the optimal shortest path, at the expense of computational cycles. However, robots are still able to discover and navigate a limited environment based on input of their sensors Milford 2008. A great deal of work has been put forward to allow local navigation, as well as building up a model of the environment for navigation. These concepts and approaches are of interest in the virtual world as they may allow for faster computation of highly complex worlds, where a global search may not be feasible Seemann 2004.

The purpose of this summery is to review the current literature with respect to pathfinding, calculating the shortest path quickly. We will also review path planning algorithms as a means of navigating environments. For path planning we will look at algorithms which have a global view as well as those which only have limited local knowledge to make path planning decisions.

We aim to provide a clear overview of the various algorithms and will be laid out in four main sections. Each section will be comprised of algorithms which follow a common theme. These sections are Path-Finding, Path-Planning, Dynamic Reaction, and Collision Avoidance.

## Path-Finding

Pathfinding, as the name implies, finds the shortest path between two points within an environment. This environment can be a 3D representation of a terrain made up of vertices, or it could be a set of nodes within a graph. A pathfinding algorithm attempts to return a path, which is an ordered set of points, between a starting point A, and

a target point B. However, often we are more interested in a path between two points which minimize some metric. Distance is the most common metric to minimize Seemann 2004. It is so common that it is sometimes referred to as shortest-path problem LaValle 2006. Sometimes a path should minimize some other metric, for example to maximize the speed of an entity. Other times the metric may be to visit the most amount of nodes for a given supply of fuel Milford 2008.
Clearly, there are a number of different approaches which pathfinding may take since different problems will be better suited by some approaches than others. Below are what I believe to be the most common approaches based on what I was able to find in the literature.

### Node/Graph Based

One method for calculating the shortest path is a network of nodes, which are connected by edges forming a graph. The distance between two nodes is usually the value of the edge. However, different weights can also be applied to edges, including negative. In these cases, a shortest path between will be the sequence of nodes one has to visit along their connecting edges from a starting node A, to a target node B. Different algorithms have been developed for different scenarios and structures of these graphs. Some may allow negative weights, others offer an advantage at the expense of another property or memory space. If an environment can be approximated with a network of nodes, then using this approach will yield acceptable results Seemann 2004. This method is also referred to as Waypoint Navigation Seemann 2004.

## 2.1 Bellman–Ford–Moore

A general algorithm which performs single-source pathfinding. It computes the shortest path from a start node A, to all other nodes in the graph. It is also capable of determining if a graph contains a negative cycle and will return true. Otherwise, it will return the shortest path, allowing negative weights in the graph. However, by allowing negative weights the computation speed slowed down. It is interesting to note that some environments may be better modeled with the use of a negative edge, such as a tunnel system *Introduction to Algorithms* 2009.

## 2.2 Dijkstra

Dijkstra is a faster pathfinding algorithm than Bellman-Ford-Moore for the single-source shortest path problem. It accomplishes this by restricting the input graph as a weighted directed acyclic graph where all weights are non-negative. All nodes in the graph are eventually visited, starting with the start node. Upon each iteration it is determined whether or not the distance to the current node along the path can be improved upon

and reduced based on previously visited nodes. In the process, all nodes of the graph are visited which can be quite expensive for large graphs. Once the target node is reached, and it is confirmed that the distance from the start node to the target node cannot be improved, the algorithm is finished, and the path sequence can be used *Introduction to Algorithms* 2009. Since the algorithm will visit each node during execution, it can also be used to find the shortest path tree rooted at the start node.

## 2.3 Floyd–Warshall

In a graph which contains no negative cycles (which can be confirmed with the above algorithm), it is possible to efficiently compute the shortest distances between any two nodes in the graph. Each distance is an estimate, and each iteration improves upon this estimate until it is optimal *Introduction to Algorithms* 2009. The original algorithm only reveals the distances between any two nodes, however with a common modification, a shortest-path tree, the actual path for any two nodes can be reconstructed *Introduction to Algorithms* 2009.

## 2.4 A*

An extension of Dijkstra's algorithm, it is noted for its efficiency and performance. It uses a heuristic to achieve better time performances, and uses a best-first search approach to transverse the graph *Introduction to Algorithms* 2009. It will find the least cost path from a given start node to a goal node. This is an important consideration, A* is able to have a set of target nodes called goal nodes. The algorithm combines knowledge of the cost of edges with a heuristic which must be admissible. It shouldn't overestimate the distance to the goal, otherwise a wrong result and sub-optimal path may be returned. A standard heuristic is to select the next node along a straight line from the current node to the nearest goal node. An advantage is that in the worst case, every node needs to be visited once, however the average case will visit significantly less nodes than the graph contains *Introduction to Algorithms* 2009. Because of this property, it has seen wide adoption in the Game industry for navigation of non-player characters Seemann 2004.

## 2.5 Potential Function Based

Up to now, the path was calculated on a graph. This is very intuitive, if we compare them to hallways, or roads. As such for most situations they are well adapted in my opinion. However, they are not the only approach available. Potential functions can also be used to calculate a path. Potential functions utilize forces of attraction and repulsion, similar to a molecular potential in an atom. These interactions can be used to find an optimal

path, local minimal and maximal, as well as collision avoidance. In these systems, our starting point is a set of coordinates. The targets potential will be set to negative infinity, and act as an infinite sink. Obstacles are assigned a value of positive infinity, as well as an area of influence. It is important to note that the target sink can have a limited area of influence, as well as an infinity large area of influence Seemann 2004. The agent can then calculate the cumulative potential at its location, as well as immediate surrounding and follow the highest gradient. Another approach would be to calculate the potentials in a grid, and then follow the gradient as well. Using this method, very smooth, optimal paths can be calculated, however it can be very computationally intensive for a large set of obstacles, or a large set of agents Seemann 2004.

## 2.6 Mesh Based

Ideally, for a path, we would like to find the globally shortest path, while being able to perform local path navigation to surround obstacles. In essence we would like to use as few as nodes as possible, while maintaining a high resolution for obstacle avoidance. Navigation Meshes (NavMeshes) was one of the answers Ramon Oliva 2011. The geometry of the environment was abstracted away, and what was left was a series of polygons, connected with edges. Shared edges can be used to transverse from one polygon to an adjacent polygon. We are able to perform a fast global shortest path search over the set of polygons using on of the graph search algorithms. Then use a different technique such as potential functions to navigate the cell to go from one side to the other, avoiding any obstaclesRamon Oliva 2011. Each polygon has a restriction that it must be convex, which allows it to be traversed in a straight line. Furthermore, NavMeshes allow not only for the shortest route, but also the most efficient by another heuristic. The size of the agent can now be taken into consideration, as well as turning radii of vehicles. Thus, using the underlying polygon, the path can be altered to improve not only upon the shortest route, but also prevent a different graph for each agent Geraerts 2010.

## 2.7 Funnel Algorithm

As outlined above, NavMeshes provide a powerful and flexible method for pathfinding. However, because of how they are constructed, it can be more difficult to find an optimal path within the NavMesh Kallmann 2010c. For this reason, finding a path through a NavMesh with an arbitrary clearance is under heavy research and many methods have been developed. One such method is to use the Funnel algorithm *Funnel Algorithm* 2010. It can be best visualized as a string of rope being pulled tight which has been pulled through a network of tubes. In order to minimize the amount of string in the network, the rope will find the most efficient positions within the network to further reduce the chosen path. The Detour software package uses a similar approach along with corridors to find the optimal path through a NavMesh Mononen 2009.

## Path-Planning

Path planning is closely related to pathfinding. I differentiate them by the domain. These approaches utilize a grid, and can be used when the terrain is not fully known, or many agents are using the same domain. Most of these approaches are inspired from autonomous robot systems which have to discover the surroundings, figure out where in relation to a goal it is located, as well as how to navigate to reach the goal.

## 2.8 Crowd Based

Crowd based path planning uses multiple agents who communicate together to understand the world. They explore the world around them and communicate their findings to the others. Boundaries in the world are discovered as well as if one agent discovers the target location, the others can then use the information gathered to plan their paths towards the goal using pathfinding methods. This approach is now also being utilized in real time strategy games "Path-Planning for RTS Games Based on Potential Fields" 2010. Where the movement is combined with potential functions.

## 2.9 D*

D* is another path planning algorithm, which is used by autonomous robots which have to move towards a goal position in an unknown terrain Milford 2008. The robot is aware of its own position relative to the goal position and has an internal model representation of the terrain. In the beginning it is assumed that there are no obstacles between it and the goal. As the robot moves towards the goal position, if an obstacle is detected by its sensors the internal map is updated. The map is then analyzed to determine if the obstacle is on the current path, or not. The robot will then continue on towards the goal, or attempt to circumnavigate the obstacle in an attempt to reach the goal position.

## 2.10 WaveFront

Wavefront uses a grid to model the surrounding terrain. Like the D* approach, the relative goal position is known. Sensory input provides a means to acquire information about the surrounding world, where are obstacles in the immediate surroundings. A grid is updated with values starting at the goal position, and radiating outwards like a wave. At each step, a value for the next cell is incremented. It propagates through the environment map like a wavefront. Once the wavefront reaches the position of the robot, the robot will repeatedly choose to move towards the cell with the lowest value. Thus moving towards the goal position in as few as steps as possible. The path can be considered optimal for this reason. If an obstacle is then placed into this model,

portion of the grid suddenly become inaccessible. a new wave starts to extend from the obstacles location, like ripples in a pond when a stone is thrown in. This grid, as far as I understand needs to have a well defined overview of its immediate surroundings for this method to be accurate.

## 2.11 Following

Path planning based on following is often used in virtual simulations where the agent does not have a specific goal to reach. Just like before, the terrain is not fully known, a globally optimal path is not necessary in this case. But rather, the agent has access to the immediate vicinity. This method of pathfinding is often utilized by non-player characters who are meant to reach either the player, or another agent if they cross paths. A path is planned by attempting to reach the acquired target, which can be achieved by means of nodes which the target has visited. In these cases, the global understanding of the environment is unnecessary, as only nodes within the immediate vicinity matter. Using this information, an interception path can be plotted, which will be adjusted as obstacles become known similar to how D* operates.

## 2.12 VFH+ Based

VFH+ is an improved version of VFH, which stands for Vector Field Histogram. VFH+ improves upon several issues of the original version. It is a method which allows for real-time local obstacle avoidance without any global knowledge Iwan Ulrich 1998. It was originally developed for a robotic aid called the GuideCane. The robot would be able to avoid obstacles and lead a visually impaired person along an obstacle free path. It works by building a polar histogram around the robot and identifies obstacles. Using this information, the heading is adjusted to move around the obstacles. It identifies a on obstacles center cell, and using polar coordinates computes the angles, as well as distance between the cells. If a path exists through which the robot may traverse, then the heading is adjusted. Otherwise the robot will have to circumnavigate the group of objects until a path towards the original heading can be found. With respect to GuideCane, it was equipped with dynamics as well, to allow for a smooth transition between headings. Using these dynamics, the velocity of the robot does not need to be adjusted in most cases as it moves through an area Iwan Ulrich 1998.

## Dynamic Reaction

In this section, I will discuss methods which are often used in combination with pathfinding, path planning, and autonomous agents to prevent the agent from settling in a local minimal. Such a settling could potentially lead to the agent to never reach the goal even

though a path may be known. Furthermore, these methods can also be utilized to add some variation to the movement especially in crowd simulations, where multiple agents need to avoid each other and cannot all follow an exact path.

## 2.13 Particle Based

Part reaction system, part path planning, a new approach to robot navigation through an unknown terrain employs Bayes Filters. These filters provide a powerful means by which the robot can map there environment, figure out where it is located, and where it needs to go and by what route Stachniss and Burgard 2014. So called particles each contain an aspect or partial representation of the world. As more data is obtained some particles weights will increase or decrease depending if their state representation matches or approximates the gathered information. As the robot spends time gathering, the particle representation will accurately build up a model of the environment around it. This model can then be used in conjunction with the previous methods to find the shortest path to the goal position.

## 2.14 Vision Based

Up to this point we have seen agents being able to query a path for navigation, build up a model of the environment when a global view is not available, as well as avoid obstacles in local space. Another way to react to changing dynamic environments is with the aid of computer vision. Here a path provided by another method can be augmented to avoid obstacles, as well as add new information into the model "Following a Group of Targets in Large Environments" 2012. An interesting iteration upon this concept is to use coherent targeting. Originally developed to follow a group of targets with a camera, this approach has also been adapted to track a set of interesting features as an aid to navigate a terrain and acquire an approximate location and movement. Although relatively new, it provides an interesting approach which appears promising. I believe that the combination of global knowledge systems in combination with local navigation system provides the most robust agents. Allowing for computer vision to approximate a location and relative movement to a set of features will further increase accuracy and robustness.

## 2.15 Logic Based

Another common method of reacting to dynamic situations is the use of a set of logical rules. Based on the input, they will trigger a reaction in hope to remedy the issue at hand. These systems work best in conjunction with other methods, and used as

an overall decision maker when either a conflict occurs, or not enough information is available for the other methods to produce a good solution.

## 2.16 FSM

Finite state machines are among the most powerful ways of modeling situations and behaviors. It consists of a set of states and transitions, where a certain input or stimuli will trigger a transition from one state to another. They can be used to control the higher level logic of what happens when a sensor detects an obstacle, or an obstacle moves toward the agent. The revaluation of the surroundings can then started, or perhaps a new path needs to be queried because a route has now been blocked.

## 2.17 Fuzzy Logic

Contrary to FMS, Fuzzy logic is not well defined with a set of transitions for a specific stimuli. Fuzzy logic can best be described as qualitative analysis vs. quantitative analysis. The exact values do not matter as much as the meaning behind these values. For example, a fuzzy problem may be that when a tall person enters a room, a sign lights up stating "Watch your head". But what exactly is tall? This qualitative analysis is what fuzzy logic tries to address and appears to have great success Seemann 2004. They allow a more versatile reaction, especially in crowds, where a property may have slightly different meaning for each agent. Each interpretation will have some overlap with other meanings, then it can be up to some probability function to determine if the value is enough to trigger one interpretation or the other. Because it adds some variance into the system, fuzzy logic adapts well to crowd simulation path planning which prevents every agent performing the same action and reasoning for a given situation.

## Collision Avoidance

Part of reacting to a dynamic environment is to avoid obstacles. In order to make navigation as quick as possible multiple levels of navigation are needed. A coarse high level, over which we can run graph based pathfinding to find an optimal route over connected cells in the environment. A middle level over which path planning can be applied where the exact details may not be known either because the data is not yet available, or storing it all would be not feasible. And finally a fine local level, where local obstacles are avoided to navigate through the region. Following are some of the most useful methods as they repeatedly appeared in literature.

## 2.18 Potential Function Based

As mentioned before, Potential Functions can be used to find a path by following a gradient towards a target. However, they are also able to provide a means of collision avoidance Seemann 2004. The centre of the obstacle will have a value set to a large positive number relative in size to the obstacle with the surroundings. As the entity approaches the obstacle, the gradient will push the entity away, thereby avoiding the obstacle Seemann 2004.

## 2.19 Influence Map Based

Similar to potential function based collision avoidance, the use of influence maps is a precomputed grid of influence values for a given object. They can be used to either attract or repel an agent. However, after the precomputation phase, influence maps use very little computation power, and are well suited for a large number of agents using the same map Seemann 2004. It is interesting to note that while a single map for the entire environment would be assumed, influence maps are actually small sub-grids, applied to the obstacle and its area of influence. When no influence map is found, the agent can assume free space around them. Once close enough to an obstacle, the agents position will enter the bounds of the influence map, and begin to change course. While less computationally intense for large number of agents compared to potential functions, influence maps can have very large memory requirements if the number of obstacles is large Seemann 2004.

## 2.20 Ray Tracing Based

Collision avoidance can also be achieved by means of ray tracing. In a virtual environment, a ray is constructed from the agent to a point at distance d from the robot and angle theta. This ray is then checked for an intersection with the environment, other agents, or obstacles. If no intersection occurs then free space is located between the agent and the ray endpoint. For best performance, multiple such rays will be used at different angles to generate a compete picture of the local surroundings for the agent. One can think of this ray tracing approach as the virtual counterpart to VFH+.

## 2.21 VFH+

As mentioned earlier, VFH+ was designed for obstacle avoidance for the GuideCane Iwan Ulrich 1998. A polar histogram is generated around the robot, and obstacles are identified by their distance from the robot. Once identified, the robot is able to avoid the obstacle by changing its heading.

# Chapter 3

# Technical Background

This chapter gives the technical background on the applicable techniques from the previous chapter as well as computer graphics that we will use in the remainder of this work. Some problems that at first sound quite simple, e.g. triangle intersection testing, are actually very expensive from a complexity or memory point of view. Sophisticated algorithms were developed, and data structures that are more memory efficient have been employed. As the datasets and search space is quite vast, some algorithms make use of specialized hardware, e.g. GPU acceleration, to perform these operations effectively.

What follows is a technical overview of major algorithms and data structures used in this work.

## 3.1 Dijkstra

Dijkstra is a shortest path algorithm which works on a graph. It explores the currently shortest path first as it visits each node in the graph. Although many variations exists, one of the more popular builds a shortest path tree of all the nodes in the graph as they are visited. That way, from a given source, we find the shortest path to every node in the graph Stein 2009.

The original complexity of Dijkstra over a graph $G$ with $E$ edges and $V$ vertices or nodes was:

$$O(|V^2|) \tag{3.1}$$

However, by adding a min-priority queue to determine which node should be visited next, the complexity drops to:

$$O(|E| + |V| \log |V|) \tag{3.2}$$

In order to compare Dijkstra to A* (next section) we will utilize a similar notation for both.

The current cost of a node $n$ will be the cost function $f(n)$.

The cost function will be comprised of the cost of the node itself, the graph cost $g(n)$ which is usually the current distance from the source to node $n$.

Depending on the algorithm, $f(n)$ may also include other metrics or functions in order to make a more informed decision as to which node to expand next.

As Dijkstra only utilizes the graph node cost, the cost function is the graph cost:

$$f(n) = g(n) \tag{3.3}$$

The value of $f(n)$ is used in the min-priority queue to select the next node to visit.

The algorithm works as following:

1. All nodes in the graph are marked as not having been visited yet. We also create a set of these unvisited nodes to keep track of which nodes still need to be visited.
2. As all nodes have not been visited yet, no distance information is known. Thus for each node, we set a distance value of infinity.
3. The start node receives a distance of 0 as we are currently located at it, and do not need to move anywhere. We add it to the min-priority queue.
4. We pop the first node in the min-priority queue and set it as the current node.
5. Using the graph $G$, we know which nodes are the neighbors of the current node. For each one of these nodes, we calculate their distance based on the value of our current node, and the weight of the edge connecting it. If this new distance is smaller than the value stored at the node, then the nodes value will now receive the new distance. Otherwise we do not modify the nodes value. We also mark in the neighbor node that the new smaller value came from the current nodes index so we know where the smaller value came from.
6. The discovered node is then added to the min-priority queue if it has not been visited before.
7. Once we have performed the above steps for each of the currents nodes unvisited neighbors, we mark the current node as visited by removing it from the unvisited set.
8. We now repeat steps 4-7 until all nodes have been visited.
9. The algorithm finishes when the target node has been visited.

To reconstruct the path, we build the path in reverse. We start at the target node, and repeatedly follow the previousNode value to move through the graph until we reach

the starting node. Then we flip the node order and we have our path which can be used for way-point navigation.

A brief visual overview of Dijkstra can be seen in Figure 3.1.

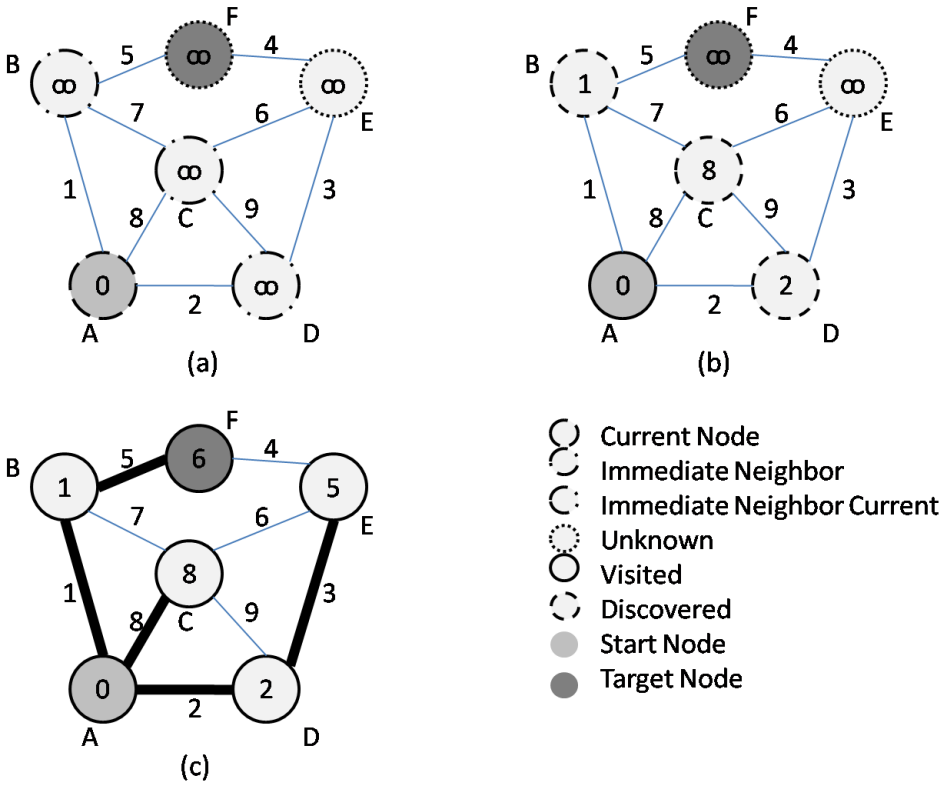


Figure 3.1: Overview of Dijkstra. a) A graph with start node A and target node F. All nodes are undiscovered, and their path costs are infinite. b) After a single discovery session. Node A has discovered its neighbors, and their path costs have been calculated. They have been added into a min-priority queue for expansion. Next step would be to expand node B. c) Dijkstra has finished and all nodes have been visited. A shortest path to target node F has been discovered, as well as the shortest path from A to any other node in G. The minimum paths have been bold-ed.

## 3.2 A*

The A*, pronounced "A Star", algorithm can be seen as an extension of Dijkstra's algorithm. It is noted for its efficiency and performance. It uses a heuristic function

to achieve better time performances, and uses a best-first search approach to transverse the graph Finney 2005.

The cost function $f(n)$ is comprised of the graph cost $g(n)$, and the cost of the heuristic function $h(n)$ for node $n$ in graph $G$.

$$f(n) = g(n) + h(n) \tag{3.4}$$

It will find the least cost path from a given start node to a goal node.

One of the advantages of A* is that it is able to have a set of target nodes called goal nodes. The algorithm combines knowledge of the cost of edges with a heuristic function.

A standard heuristic function is to select the next node along a straight line from the current node to the nearest goal node.

An advantage is that in the worst case, every node needs to be visited once, however the average case will visit significantly less nodes than the graph contains Stein 2009. This results in A* having a complexity of:

$$O(|E|) \tag{3.5}$$

Because of this property, it has seen wide adoption in the Game industry for navigation of non-player characters Seemann 2004.

The algorithm works as following:

1. Similar to Dijkstra, all nodes in the graph are marked as not having been visited yet. We also create a set of these unvisited nodes to keep track of which nodes still need to be visited.
2. As all nodes have not been visited yet, no distance information is known. Thus for each node, we set a distance value of infinity.
3. The start node receives a distance of 0 as we are currently located at it, and do not need to move anywhere. We add it to the min-priority queue.
4. We pop the first node in the min-priority queue and set it as the current node.
5. Using the graph $G$, we know which nodes are the neighbors of the current node. For each one of these nodes, we calculate their distance based on the value of our current node, and the weight of the edge connecting it. If this new distance is smaller than the value stored at the node, then the nodes value will now receive the new distance. Otherwise we do not modify the nodes value. We also mark in the neighbor node that the new smaller value came from the current nodes index so we know where the smaller value came from.
6. The discovered node is then added to the min-priority queue to become a current node eventually and be expanded. However, the value which is used to sort is

not the node value, but rather the sum of the node value $g(n)$ and the heuristic $h(n)$. By doing this, the actual distance is not changed, but we are able to steer the expanding of nodes to the general direction of the target node and potentially prevent moving into the opposite direction.

7. Once we have performed the above steps for each of the currents nodes unvisited neighbors, we mark the current node as visited by removing it from the unvisited set.
8. We repeat until all nodes have been visited, or we have reached any goal node.
9. The algorithm finishes when the target node has been visited.

After we have found a path, we start at the found goal node, and build the path by going backwards until we reach the start node.

Figure 3.2. has a visual overview of A* applied to the previous example graph. Of particular note is step b. The double arrows represent the value of the heuristic function. Even though the value of nodes B, C, D are 1 , 8, 2 respectively because of the heuristic function the order of exploration would actually be B (1 + 1 = 2), C (8 + 2 = 10), and finally D ( 2 + 18 = 20). Because we were going to explore node D last, we actually end up never visiting node D or even discovering node E because we are able to form a path to F much earlier.

A*'s ability to avoid wasting computation makes it an excellent choice for exploring densely connected graphs or grids. Figure 3.3. illustrates a very densely connected graph where the nodes are organized in a grid. In the case of Dijkstra a wave of discovery moves over the entire block and nodes which are the furthest away from the target node are explored prior to closer nodes. A* on the other hand leaves many nodes in the graph undiscovered as they have been essentially eliminated due to the heuristic function. These nodes are at the very end of the min-priority queue. We can see that due to the heuristic influence, nodes which are moving generally in the right direction are given preference and the target node is discovered much sooner and with fewer calculations.

## 3.3 Funnel Algorithm

The funnel algorithm is another pathfinding algorithm which works on connected nodes *Funnel Algorithm* 2010. However, whereas Dijkstra and A* work on generic graphs, the Funnel algorithm works with a mesh of triangles, made up of vertices and edges. This mesh has a perimeter boundary, internal edges, and navigation is to be achieved between a vertex in the mesh to another vertex in the mesh. Figure 3.4. illustrates one such mesh. The Funnel algorithm tries to find the shortest distance between these two vertices in the mesh, without crossing the bounding perimeter. Crossing of internal edges is however allowed, and indeed necessary Demyen 2007.

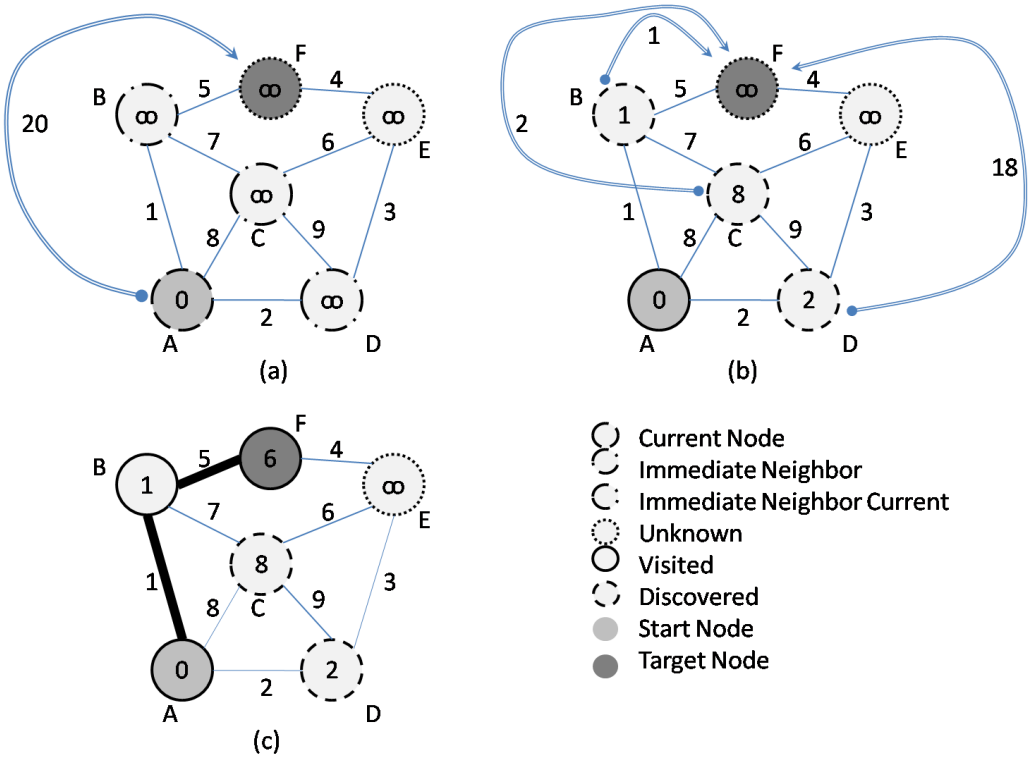


FIGURE 3.2: Overview of A*. a) A graph with start node A and target node F. All nodes are undiscovered, and their path costs are infinite, the heuristic cost to F from A is 20. b) After a single discovery session. Node A has discovered its neighbors, and their path costs have been calculated. The nodes with values $g(n) + h(n)$ have been added into a min-priority queue for expansion. Next step would be to expand node B, C, and thirdly D. c) A* has finished as a goal node has been visited (Node F has value of $f(n) = 6 + 1 = 7$ and will be placed before node C). A shortest path to target node F has been discovered, and unnecessary nodes have not been expanded. The minimum path has been bold-ed.

An analogy would be to pulling a string tight which was placed in a hall. As the string gets shorter and shorter, it will rest on corners. Similar, the Funnel algorithm does the same. Figure 3.5. illustrates the first few step of how the Funnel algorithm finds a path. We explore neighboring edges one at a time. If we cross a boundary, then we have gone as far as we can for that particular path, when no exploration path can proceed, the shortest path will become the first segment.

Commercial implementations include the Detour package which applies the Funnel algorithm to NavMeshes.

The Funnel algorithm by itself does not really help us, however when paired with a mesh based on the world, then the system as a whole is called a NavMesh.

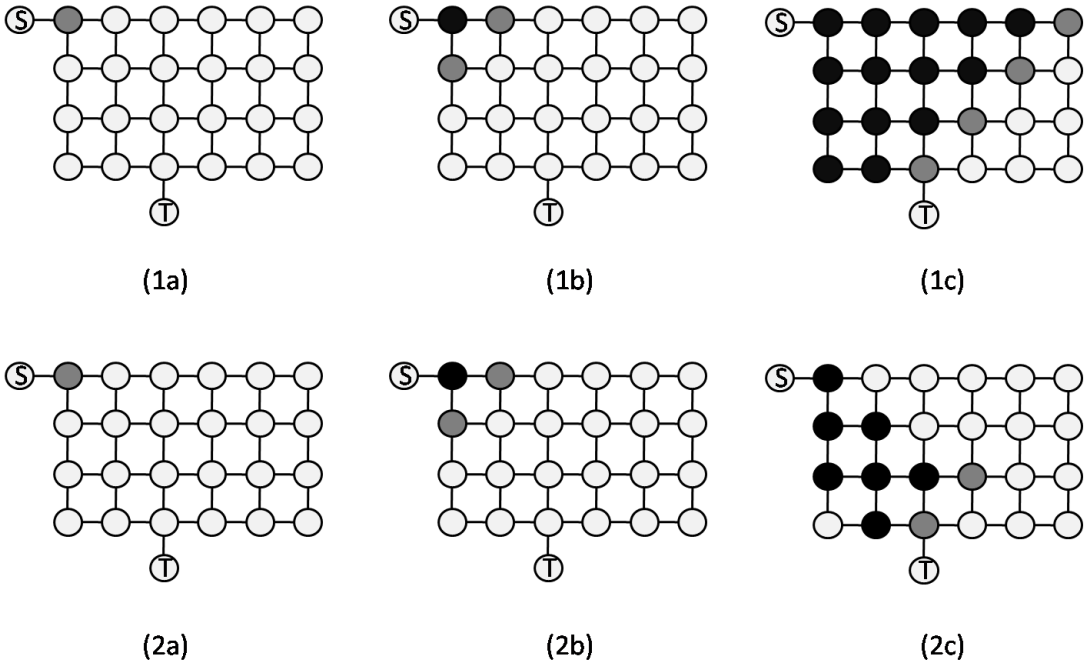


Figure 3.3: Dijkstra vs. A*: 1a) Initial neighbour of start node S. 1b) Next step, one visited node (black), two discovered nodes with equal values. 1c) nth step, many nodes expanded at the top, very far from target node T. 2a) Same grid with A*. 2b) Next step, still added immediate neighbors. 2c) Heuristic function has pushed many nodes to the bottom, expanded nodes approach target node T and will find a path much sooner compared to Dijkstra.

---

How the algorithm works:

1. Begin at the starting vertex in the mesh. It becomes the current vertex
2. For each neighbour of the current vertex calculate the distance.
3. In min-priority queue, pick vertex with smallest distance, and move to its neighbor, then calculate the Cartesian distance to the current node. If this line would cross the perimeter, do not save the distance, and move to the next vertex in the queue.
4. When the queue is empty, all vertices in the mesh which have a direct line of sight to the current node will contain a distance.

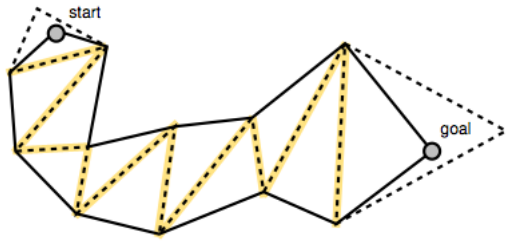


FIGURE 3.4: Illustration by: Mikko Mononen. A simple mesh. A starting and goal vertex has been selected and we would like to find the shortest path through the mesh.

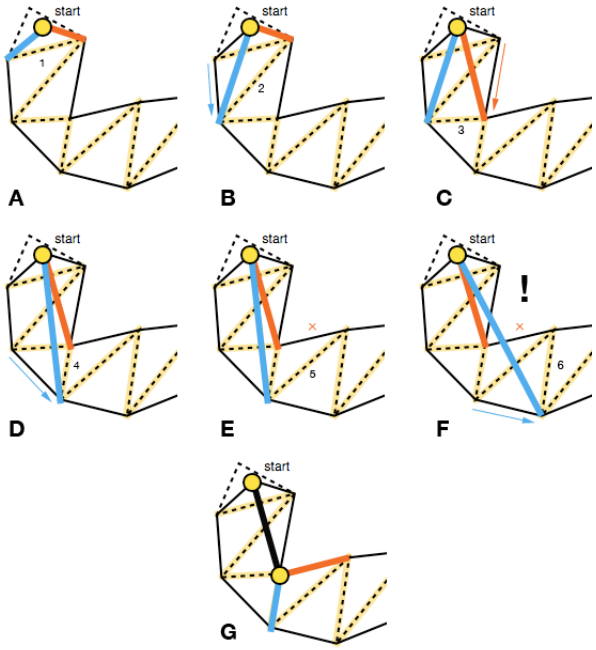


FIGURE 3.5: Illustration by: Mikko Mononen. A) The neighbors (i.e.,connected vertices) of the start vertex are explored. B) we can trace the following vertices on either side and a direct line will result in the shortest distance. E) The red path cannot proceed to the next vertex along the perimeter, the corner is as far as it can go. F) Similarly, blue cannot move to the next vertex with a straight line. G) The shorter of the red and blue path is selected to become the first segment (black) and a new path node has been inserted at the corner vertex of the mesh. Now we repeat the process until we reach the goal vertex.

5. The vertex which has the smallest distance value becomes the next current node.
6. Repeat until goal node is reached.

## 3.4 2D: Nav-Mesh

NavMeshes are a powerful addition to a motion strategy, however they can be more difficult to find an optimal path Kallmann 2010a. Luckily, finding a path through a NavMesh with an arbitrary clearance is under heavy research. A NavMesh is a 3D mesh which is usually planar. For an environment which has gravity and actively pulls an entity to the ground, NavMeshes are ideal as the degrees of freedom the entity has will be limited to the ground plane. An entity cannot levitate upwards by itself. So even though the environment and the entity is a 3D object, their motion is limited to two dimensions.

Ideally, for a path, we would like to find the globally shortest path, while being able to perform local path navigation to circumnavigate obstacles. In essence we would like to use as few nodes as possible, while maintaining a high resolution for obstacle avoidance. Navigation Meshes (NavMeshes) was one of the answers *Navigation Mesh Reference* 2012. The geometry of the environment was abstracted away, and what was left was a series of polygons. Figure 3.6 shows such a NavMesh. Shared edges can be used to traverse from one polygon to an adjacent polygon. We are able to perform a fast global shortest path search over the set of polygons using one of the graph search algorithms. Then use a different technique such as potential functions to navigate the cell to go from one side to the other, avoiding any obstacles *Navigation Mesh Reference* 2012. Each polygon has a restriction that it must be convex, which allows it to be traversed in a straight line. Furthermore, NavMeshes allow not only for the shortest route, but also the most efficient by another heuristic. The volume and dimensions of the entity can be taken into consideration and the path be adjusted. Thus, using the underlying polygon, the path can be altered to improve not only upon the shortest route, but also prevent a different graph for each entity *Navigation Mesh Reference* 2012.

The NavMesh reduces the complex environment into a simplified approximation. A large flat portion can be replaced with a single quadrilateral without loss of travel information.

Another popular aspect of NavMeshes is that they can be automatically generated from the world geometry Menard 2012. Triangles which have a normal aligned with the up vector of the world (plus/minus a tolerance) are projected downward onto a 2D plane to form a new mesh. After this, triangles are grouped into larger convex sets to reduce the number of triangles. However, at times sub-optimal meshes are generated with thin triangles which can cause some issues during pathfinding Menard 2012.

The actual algorithm used to find a path in a NavMesh is based on the Funnel algorithm. Packages such as Detour add modifications to them (i.e., path corridors), to increase some aspect of the algorithm or address a specific implementation detail or specification.

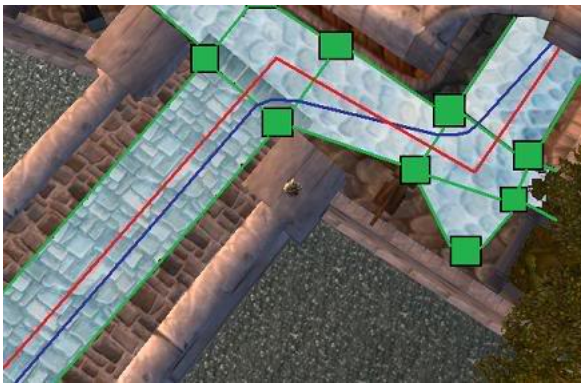

Figure 3.6: 2D: NavMesh Navigation Shi 2011

## 3.5 3D: Layered Nav-Mesh

Although a NavMesh is very powerful, it does lack a 3rd dimension. To address this shortcoming as well as "fake" 3D pathfinding, in some situations multiple layers of NavMeshes which are stacked on top of each other can be used.

Figure 3.7 shows an example of layered NavMesh navigation. The different NavMeshes are located at different elevations, as well as above one another, and actions such as jumps can be used to move from one NavMesh to another. These connections can take the form of physical geometry (vertices from NavMesh 1 are connected to vertices of NavMesh 2) or programmatically (extend a ray upwards or downwards, and move to a point that the ray intersect with the closest NavMesh that is not my current NavMesh). Its not hard to imagine that layers of NavMeshes could be used in a space battle game. Game play motion would occur on a plane, however the entity would be able to move up and down in the layer stack to give the illusion of 3D motion.

As with a normal NavMesh, layered NavMesh also make use of the funnel algorithm within each NavMesh layer. However, they add another mechanism to allow the motion (and pathfinding) between layers.

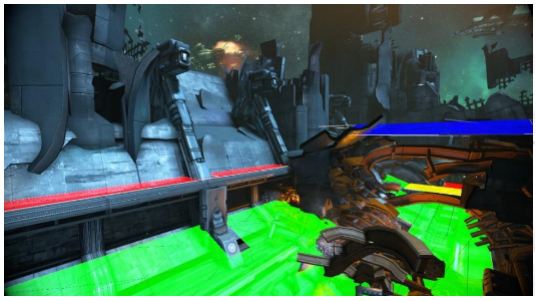

Figure 3.7: 3D: Layered NavMesh, Green, Red, and Blue regions represent different NavMeshes which are stacked Brewer 2015

## 3.6 3D: Triangle Mesh

Before NavMeshes were widely used, early 3D space games used several techniques to give the illusion of path planning Seemann 2004. These ranged from pre-defined paths to way-points, usually combined with line of sight algorithms to give them reaction to a user. However, there also were attempts to utilize the 3D mesh of the world directly for path planning. It turned the vertices of the mesh into nodes of a graph, edges remain edges, and the weight was the edge length Seemann 2004. A starting and target node were selected and then using the mesh graph a shortest path was calculated through the world using a graph algorithm like Dijkstra or A*. A variation exists where the shortest path is refined by checking if selected path nodes can be reached shorter by forming a line of sight shortcut Finney 2005. To prevent a shortcut from going through the mesh, a line-triangle intersection test is utilized. However, as the game worlds increased in the number of triangles, pathfinding at runtime became no longer feasible Seemann 2004.

## 3.7 3D: Sparse Voxel Octree

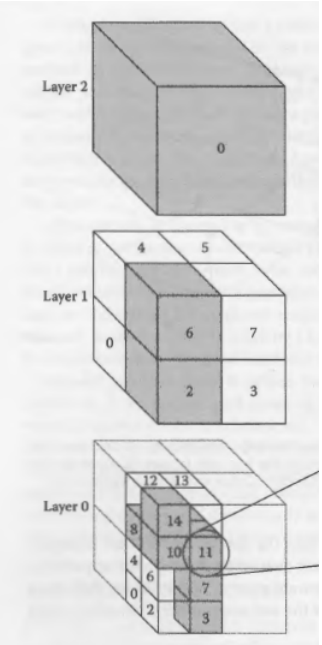


Figure 3.8: Octree and SVO breakdown Rabin 2017

Sparse Voxel Octrees (SVOs) is one method of true 3D pathfinding. In a world where the entity can navigate to any position in space, and is not confined to a two dimensional set of motion, SVOs provide a mechanism to capture traversable volumes of space Rabin 2017. Through these portions one could use a simple waypoint navigation

motion strategy or line of sight. However the interesting part is boundary definition and how a path is formulated as it snakes through the environment.

Daniel Brewer has applied SVOs to areas where he maps and breaks them down into large empty convex portions which can be freely traversed Rabin 2017. He employs them to provide pathfinding for space navigation and combat for the video game Warframe Rabin 2017.

SVOs are a slightly modified version of a normal octree. In addition to the normal parent - child links, they also contain information about their immediate neighbors. It allows for fast, constant $O(1)$, lookup of adjacent cells without having to traverse the node tree Rabin 2017.

SVOs also only store a small 4x4x4 grid of voxels in the leaf nodes. These voxels are cubical in shape and represent a portion. If a triangle exists in a voxel, then it is solid; otherwise a voxel is empty. The small size of a leaf node greatly reduces their memory footprint, while capturing the important difference between empty space and intersection with the environment. There is some pre-processing involved as an environment is analyzed and the SVO is build. Once it has analyzed, the SVO will be a connected graph representing the free space that entities can traverse Rabin 2017. Figure 3.8. Illustrates the breakdown using SVOs, as well as how the leaf nodes such a node #11 will then be a 4x4x4 grid as mentioned above.

At this point a graph search algorithm, such as A*, can then be used to find a path. For A*, the heuristic function is still the distance to the target, but also a penalty is applied for exploring a node unnecessarily Rabin 2017. Figure 3.9 highlights a path of different sized nodes wherein lies the path used to move from one end to the other.

Looking up a point is identical to an octree. In this case, if a point is in a childless nodes then it is in an area of empty space. Otherwise we can calculate which voxel inside the leaf contains the point. These voxels also determine whether there is free space or a collision Rabin 2017.

Where SVOs are different is that when the region needs to be explored, we can simply utilize the neighbour links instead of having to traverse the octree again to visit adjacent cells Rabin 2017.

At a high level, pathfinding using SVOs works in the following manner:

1. A starting point and end point are queried against the SVO to determine in which leaf node each one resides in.
2. Using the tree structure, a nodes neighbor links, as well as whether or not a node is a leaf, we use an algorithm such as A* to move from the start node to the target node.
3. If a node has child nodes and we need to traverse the whole node, rather than going depth-first into the node and move across, we skip over it. In Figure 3.9.

the two smaller interior cubes are part of a larger node. Since we have to traverse the whole node, we do so at the parent level.

4. When the target node is reached, we can construct a path to follow through the environment to the target node. An example strategy would be to place a waypoint at each nodes center.

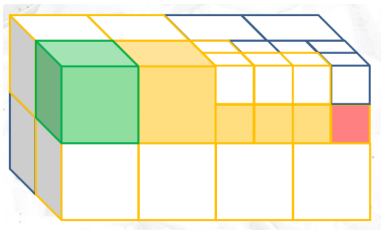

Figure 3.9: 3D: Pathfinding with SVO: Green is a lower resolution starting partition, Red is a higher resolution target partition, Yellow are partitions part of the path (i.e., the path will lie within the space bounded by those partitions) Brewer 2015

**The following techniques are used in Computer Graphics, and have been used directly or as an inspiration for the automatic generation of NavNets.**

## 3.8 Marching Cube for Voxelization

Voxelization refers to transforming a 3D model based on triangles into a 3D approximation based on voxels. Using a picture as an analog, the equivalent process would be pixelation. The image has lost some detail, however the color information is retained. A voxel is a 3D pixel, a unit cube. An example where 3D voxel art is primarily used is a video game called "3D Dot Heroes". The naive voxelization defines a bounding volume for the model, as well as a voxel size, the bounded space is divided up into voxels and each is checked against the original model. If a voxel location intersects with any triangle of the 3d model, than the voxel at the specific location is marked as intersecting and will be solidly displayed. When the process has finished each voxel is rendered as a solid cube. Its complexity is $O(TV)$ where T are the number of triangles of the model and V is the number of all voxels comprising the space. There is an optimized voxelization method which is based on **Marching Cube** Lorensen and Cline 1987 Kaufman and Shimony 1987. It looks at each triangle, one at a time, calculates the bounding box of the triangle and only performs a box-triangle check with the voxel location inside the triangle specific bounding box. By doing this, the number of checks can be significantly reduced. As before, if they do intersect, then a voxel is displayed at the specific location. Color information is also passed onto the voxel based on the triangle color. Although the complexity is still $O(TV)$, in practice it behaves faster, however it is entirely dependent on the geometry of the model.

## 3.9 Ray-Box Intersection Test

Testing if a ray intersects a box is a common task in computer graphics. It is used for ray-tracing, mouse-picking, and collision detection. To that extend many techniques have been developed, each one improving some aspect of a previous method. To test if a ray intersects a box can be as easy as solving the equation for t:

$$ax + by + cz + d = p + td \tag{3.6}$$

Here we set the equation of a plane in 3D equal to the equation of a line in 3D.

1. For the line we have a known point and a direction
2. For the plane we have the normal of the plane. The plane in this case is a single face of the box.
3. Solving for t we have a point on the plane and on the line.
4. If this point lies within the boundaries of the boxes face, then the line intersects this face and therefore the box.

The above steps are repeated for each face of the box and ultimately decide whether the ray intersects, touches, or misses the box Williams et al. 2005.

There also exists a more optimized version of this method called the Slab test. It makes use of half boxes and is able to perform the tests without floating point arithmetic. Because of its implementation, it has improved execution speed compared to the naive method.

The Slab test uses the boundaries of the box to hone in on an end point and a start point on the ray. These points are placed at infinity and negative infinity respectively along the ray. Then the boundaries of the box are used to move the points closer to the box.

If the start point has a smaller t value on the line compared to the end point then there is an intersection. Otherwise the ray and box do not intersect.

## 3.10 Ray-Triangle Intersection Test

To test for the intersection between a ray and a triangle in $\mathbb{R}^3$, we make use of the Moeller-Trumbore intersection algorithm. It is a method to calculate the intersection without having to pre-calculate the plane equation of the triangle Möller and Trumbore 1997. It is very useful for ray-tracers.

The algorithm uses parameterization and projections of a point onto a triangle to determine if a point lies on the line and the triangle.

## 3.11 Axis-Separation Theorem

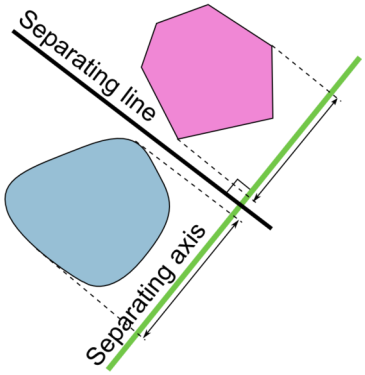


Figure 3.10: Separation between two objects along a separation axis Oleg Alexandrov n.d.

The axis-separation theorem belongs to the hyper-plane separation theorem. It states that for two convex objects, if there exists a line (or plane in 3D) on which the projections of the two objects do not intersect, then the two objects do not intersect either. Figure 3.10. illustrates an example of a separation line between two convex objects.

## 3.12 Triangle-Box Intersection Test

Also known as Triangle-Axis Aligned Bounding Box (Triangle-AABB) test, is an algorithm to test whether an axis aligned bounding box intersects with a given triangle in 3D. Because we are sampling the world with a sample grid, all grid cells will be aligned.

We might be inclined to formulate a simple approach where we test if each vertex of the triangle is inside the box, or at least one edge intersects with any face of the box. However, in the case of the box residing entirely withing the triangle, no simple and fast calculation can determine an intersection. The keywords are simple and fast. Because the sampler will perform this test a large amount of times, the execution cost will add up.

A better approach to test for the intersection between a triangle and a box in $\mathbb{R}^3$ is to make use of the separate axis theorem Akenine-Moeller 2001.

The test involves testing for 13 axis *Game Physics Cookbook* 2017.

- Three axis are the face normals of the AABB.
- One axis is the face normal of the triangle.
- Nine axis are cross-products of the edged of each primitive.

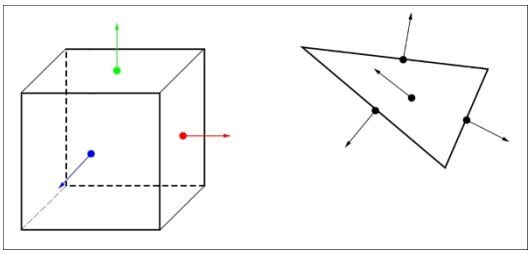

FIGURE 3.11: Some of the axis used to determine the intersection between a triangle and an axis aligned bounding box using the separate axis theorem *Game Physics Cookbook* 2017.

If any test case confirms that a separating axis exists, then the intersection is negative. However, if all 13 test cases cannot find a separating axis, then the triangle does intersect with the axis aligned bounding box. Figure 3.11. illustrates some of these axis.

# Chapter 4

# Our Approach

In this chapter we define the concepts of what a NavNet is, what are its parts, and how they are constructed theoretically. This section serves to outline how NavNets are formulated mathematically to study the problem as well as determine complexity.

## 4.1 Definitions and Properties

This section will be used during implementation to define how a specific algorithm should be reflected in the actual code. As well as how it should behave conceptually at runtime.

Although care has been taken to have a logical flow of terms, it may still be necessary to use a concept in an explanation before it is defined. In such a case, the definition of the concept will follow the section.

**1. System**
The system refers to the collection of all parts which are needed for a simulation, a video game, or a batch process. It may include an environment, objects which can be controlled directly by a human, objects which are controlled by a computer, obstacles to avoid, along with anything else which exists at a specific instance in time.

**2. Entity**
An entity $A$ in the system can move from one position to another position in 3D space defined as $\mathbb{R}^3$ and has some volume associated with it. An entity is not infinitesimally small and thus cannot be represented by a point. It is an object which is controlled by a computer. In video games it is also known as a non-player character (NPC) or as an opponent. An entity is some 3D object which is able to move through the world, and requires a means to navigate the environment in the system.

For an entity, the following properties are defined:

- It cannot go through any wall in the environment. The walls are a finite set of surfaces, which are composed of one or more triangles. In other words, the walls of the environment and surfaces of obstacles are solid and the entity cannot go through them.

- The entity is not restricted to a 2D plane of motion (i.e., restricted to the floor due to gravity). This is an important property as such a restriction would turn a NavNet into a NavMesh. Finding a path for an entity which has complete freedom to move in any direction in a 3D environment is the key purpose of the NavNets. An example of such an entity might be a school of fish for an underwater environment, or a bird. Even a robot in space is a valid candidate to meet this requirement.

**3. World**
The world represents a 3D environment. It can be either digitally constructed from blueprints, or be a surfaced point-cloud from a 3D scan. The world can be composed of different objects, walls, floors, stairs, etc. However these objects are all defined by a list of triangles. In the 3D world, everything will be constructed of triangles, and the world will be a collection of all triangles.

- The world W is constructed from N triangles, so that $W = \{trig_1, trig_2, ..., trig_N\}$, and
- each triangle exists in the 3D space, $trig_i \subseteq \mathbb{R}^3$

**4. Discreet Sample Grid**
Using the world triangles directly to determine where a boundary exists in the world is a valid form of path navigation. However depending on the world geometry and the level of detail used, the number of triangles which need to be checked can significantly reduce efficiency of finding a path. The issue is not only present in pathfinding, but also in computer graphics. Particularly near real time graphics, such as video games, or simulation which need to react to input. In these cases, only world triangles which are in front of a camera are actually drawn and only if they are also not too close or too far away from the camera. In order to achieve acceptable frame render time, the world is simplified to a level where we do not lose any information, but rather filter out unnecessary information.

Similarly, for NavNets, using the world geometry directly would result in inefficiencies as a path to a target point may traverse the entire world. The concept of rendering what is in front of the camera does not apply. However, because an entity cannot go through walls, or the bounding triangles of the world, we can reduce the search space by means of an approximation. Furthermore, we need to be able to differentiate between what is traversable space, and what space is either fully bounded or enclosed, or technically inside a wall and thus cannot be traversed by the entity.

To achieve this, we will take little portions the systems 3D space and test them to see if they intersect with the world geometry, the 3d environment. If they do, then this sample portion will be marked as non-traversable. Otherwise, it is space which an entity can move through provided it, or a continuous portion is large enough to accommodate an entities volume and dimensions.

A 3D grid is used to keep track of which portions of $W$ are empty space and which contain geometry. These portions will be non-overlapping. Let $S$ be a finite discreet 3D sample grid.

- $r$ represents the resolution with which $W$ will be sampled; each grid cell will be of uniform size $r$.
- We define the sample grid as the collection of these grid cells with their x, y, and z indices, $S = \{S_{ijk}\}$
- Each grid cell exists in the real space of the system and thus is a sub region of $\mathbb{R}^3$.

**5. Free Space**
The free space is the portions of the world where we could place an entity at, and this entity would not intersect with any triangles of the world. The free space may be disjoint and non continuous, NavNets are able to handle this use case.

The *Free Space* $F$ of W with respect to sampling $S$ is an approximation of the volumes which an entity could be positioned in.

- The entity does not intersect with any triangles in W while inside $F$, and
- It is constructed such that the free space is the union of all grid cells which do not intersect the world. $F = \cup S_{ijk}$ where $S_{ijk}$ is free($S_{ijk}$). Figure 4.1. illustrates this construction with a simple 2D example.
- We define free($S_{ijk}$) if $S_{ijk}$ does not intersect with any triangle in W.
- We can refer to $S_{ijk} \in$ F via $F_i$, i.e.: $F_1 = S_{i_1 j_1 k_1}, ..., F_n = S_{i_n j_n k_n}$

**6. NavVolumes**
NavVolumes are one of the components of NavNets, and they represent the large convex regions in $W$ that do not intersect with any triangle with respect to sampling $S$.

A NavVolume is a simplification of a large number of sample grid cells to further reduce the search space. Because we would like to traverse through space, these large convex regions can reduce the number of checks to a single instance. If the entity is inside a NavVolume, then by construction the entity cannot intersect any triangle in the world.

Let $N$ be the set of all NavVolumes so we can reference them.

- NavVolume $N_i$ is maximally convex; adding any other non-empty subset of $S$ will make $N_i$ concave, and
- we define two NavVolumes as *connected* when:
  connected($N_i$,$N_j$) is true iff ($N_i \cap N_j \neq \{\}$). Or in other words: two NavVolumes are connected when they intersect and have an overlap of sample cells belonging to them.
- By inspection we can see that $\cup N_i = F$

**7. Intersections**
As mentioned above, two NavVolumes are connected if they intersect. The intersections will play a vital part during pathfinding and thus we give them focus. We are interested in these intersections as we will use them to move from one NavVolume to another.

- Let $I_{ij}$ be the intersection between NavVolumes $N_i$ and $N_j$,
- they are symmetric: $I_{ij} = I_{ji}$.

**8. Navigation Volume Network**
The Navigation Volume Network or NavNet is our solution to tackle the problem of unconstrained 3D pathfinding.

It is a weighted undirected graph $G = (I, N, w)$ composed of the NavVolumes and the intersections. Analogous to NavMeshes, the graph represents how the NavVolumes are *connected* to each other through the intersections.

- The weights are calculated based on the intersections. We are using the Cartesian center point of each intersection and set the weight equal to the distances between intersections.
- The NavVolumes are being used as the edges in the graph.

**9. Volume Path**
In order to find a 3D path efficiently, we reduce the search space. Once we have sampled the world and approximated it using NavVolumes, we can query the NavNet for a path. However, here too we would have to spend a great deal of time to return a path. Instead, we will use the NavNets to find a Volume Path.

This volume path is an alternating sequence of NavVolumes and intersections to outline the corridor in which the path exist.

Given a starting and end point, a starting NavVolume and end NavVolume are calculated. The we use a shortest path algorithm such as Dijkstra or A* to find a route through the graph. This route is a volume path $P_{\text{Vol}}$.

- The starting point is in $N_0$, the end point is in $N_n$ and
- connected($N_0$,$N_1$) AND connected($N_i$,$N_{i+1}$) AND ... AND connected($N_{n-1}$,$N_n$)

## 4.2 Computing NavNets

Up to now, we have only discussed the theoretical construction of NavNets. However, computing them efficiently is vital as the problem space can explode. In our work we compute the NavNets by approximation using a down-sampling of the 3D world. Doing so we are able to significantly reduce the complexity. The NavNets are computed during pre-processing and saved for use in subsequent path planning queries.
Computing a NavNet consists of four stages.

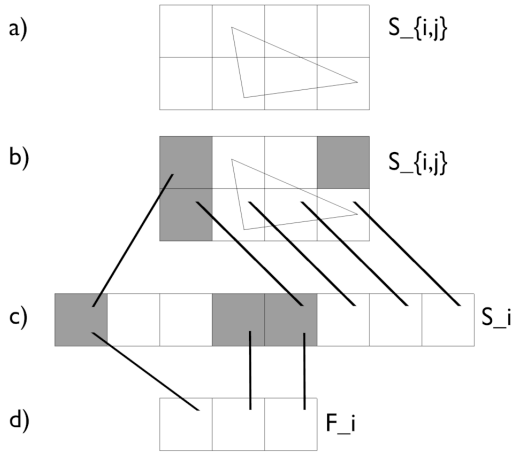


Figure 4.1: Free space being constructed from the sample grid. a) The sample grid as a 2D example, with a triangle. b) Sample grid cells which do not intersect with the triangle are highlighted. c) the 2D sample grid is referenced using single index i. d) Free space $F$ is constructed to keep keep track of which sample grid cells do not intersect with the triangle.

1. Compute a discretization by sampling.
2. Compute the NavVolumes as maximal convex sets.
3. Calculate the intersections between NavVolumes.
4. Build the NavNet using the NavVolumes and intersections.

### 4.2.1 Targeted Sampling

We need to determine which of the sample cells do not intersect with any triangles. Although traditional sampling would solve this, each sample cell would check for an intersection against each triangle in $W$. We approach this problem under the assumption that there are significant less triangles than sample cells. As such, we drive the sampling by the triangles: for each triangle we find the cells it intersects with. After all the sample cells have been determined, the free space $F$ will contain all other cells. The actual test is performed using the box-triangle intersection test described in section 3.12.

The approach borrows the idea from the marching cube method in section 3.8. In addition, we also use an octree to break down the sampling box of the triangle to aid in determining which cells cannot possibly intersect with the triangle, further reducing the number of intersection tests we need to perform.

The naive version will have a complexity of $\mathcal{O}(|S||W|)$ in 3D, as every grid cell in $S$ is tested against every triangle in $W$.

Although our approach has the same worst-case complexity, it offers an observable run-time performance increase as we have greatly reduced the number of checks necessary. Table 7.1. compares the number of operations of this method to the naive version. Please note that the triangle/box intersection step (labeled as trig/box check) is the expensive step.

As with all sampling, we also have the issue of the origin for the sample grid. It is unfortunately possible at this point that the combination of resolution and origin can miss volumes which an entity should be able to visit. This is one of the key problems for future work. Figure 4.2. illustrate how a shift in origin can prevent an area from being correctly identified as free space.

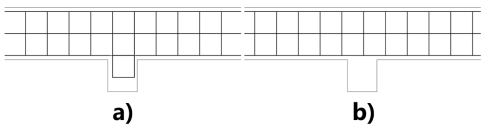


FIGURE 4.2: Impact of sample grid origin. a) Offshoot is correctly sampled. b) Grid does not align correctly and offshoot is not sampled.

### 4.2.2 NavVolume Computation

The main idea is to reduce the search space, and we accomplish it by finding the largest convex subsets in S. Since a set of $n$ elements has $2^n$ subsets, calculating our NavVolumes is an exponential problem. However, we do not need to check them all, nor do we need to back-trace our selection choices like in a knapsack problem. A significant portion of the subsets do not meet our requirements. So to compute these relatively few subsets directly, we use an iterative technique of "growing". Because of sampling, NavVolumes are rectangular boxes, and we only need to check the immediate neighbors during a "grow" step; and since the NavVolumes are allowed to overlap, the order of NavVolume computation is irrelevant.

During implementation we found it necessary to add support structures. Each $S_{ijk}$ keeps track of which NavVolumes it belongs to, as well as if it belongs to F. We also added a queue $Q$ for seed elements which take priority over elements of S.

"Growing" a NavVolume is an efficient method to find one maximally convex subset in S for a given starting sample cell $S_{ijk}$.

**Grow($S_{ijk}$):**

1. If $S_{ijk}$ does not belong to any NavVolume, we create a new NavVolume $N_i$, add $S_{ijk}$ to it, and add $N_i$ to N else we are done.

2. We are only interested in the sample cells around $N_i$, so we use the indices of S to keep track of the convex hull.
   - We initialize BottomCorner and TopCorner to $(i, j, k)$.
3. We determine the indices of the adjacent neighbours by adjusting the BottomCorner and TopCorner variables.
   - BottomCorner = BottomCorner + $(-1, -1, -1)$.
   - TopCorner = TopCorner + $(1, 1, 1)$.
4. The adjacent neighbours (A) of $N_i$ are defined as:
$$A = (\cup_{(i,j,k)=BottomCorner}^{TopCorner} S_{ijk}) - N_i$$
5. By checking which adjacent neighbours are *free()*, we can determine how $N_i$ grows to a larger convex set.
6. We repeat steps 3-5 until every subset of A would make $N_i$ concave.[1]
7. $N_i$ is now convex and maximal by construction, and we add the members of A from the last step to Q as potential seeds.

Now that we understand how a NavVolume "grows", we can write down how all maximal subsets are computed.

1. If $Q$ is empty, we start at the first element of $S$; if it is *free*() we push it onto $Q$, else we continue to the next element of $S$.
   - We also mark where in S this occurred, so that we only have to traverse the elements of S once.
2. We pop a seed $S_{ijk}$ off $Q$, and
3. Grow($S_{ijk}$).
4. We repeat steps 1 - 3 until every element in S has been visited once.

By construction, the NavVolumes will form intersections as well as a unique breakdown regardless of the "growing" order or seed locations.

A complexity analysis of the actual algorithm shows that the "growing" method does offer us a performance increase:

- For $n$ starting locations in $S$: $\mathcal{O}(n)$
  - Do a max of $n$ "grow" steps: $\mathcal{O}(n)$
    - Find the neighbors. $\mathcal{O}(1)$
    - Check corner neighbors. $\mathcal{O}(1)$

[1] Please note, the selection and check process prevents from having to compute $2^n$ subsets of A

- Check edge neighbors. $\mathcal{O}(n)$
- Check face neighbors. $\mathcal{O}(n^2)$
- Add neighbors to NavVolume. $\mathcal{O}(1)$

Worst case: $\mathcal{T}(n) \sim \mathcal{O}(n) * \mathcal{O}(n) * (\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(1)) \sim \mathcal{O}(n^4)$

Figures A1.1. - A1.6. illustrate the growing of NavVolumes (to make the pictures simple to understand we only show a normal construction).

### 4.2.3 Calculate the intersections

When all NavVolumes are generated, we compute the intersections. To prevent checking $\frac{n(n-1)}{2}$ pairs of NavVolumes, we scan over all $S_{ijk} \in F$ and use the NavVolume reference to make a short list of the intersections which actually occur. This removes significant amounts of computation since most NavVolumes will not intersect each other. The intersections will be a subsets of $S$, and contain additional information of which two NavVolumes intersect.

Figure A1.7. shows this stage in a simple example.

### 4.2.4 Building the NavNet

The Navigation Volume Network (NavNet) $G$ can now be constructed as it represents how the NavVolumes are connected to each other through their respective intersection. Each intersection $I_{ij}$ is a node in the graph. Edges are formed by a NavVolume $N_i$ common to two intersections, with a weight equal to the Cartesian center point distance between the intersections.

During implementation we also had to add another support structure. It keeps track of the intersections each NavVolume is associated with. Although $I_{i,j} = N_i \cap N_j$, this list allows us to efficiently look up the start and end intersections directly without computations.

Furthermore, once the NavNet has been determined the sample cells are no longer needed. We utilize the bounding box of each NavVolume and intersection and the intersection list, thereby reducing the run-time memory footprint.

Figure 5.6. / A1.8. show a simple NavNet alongside the support structure.

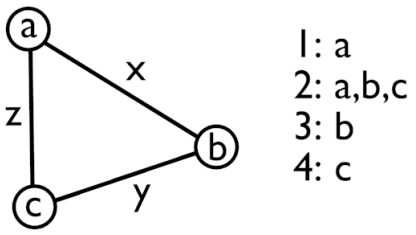


Figure 4.3: An example of a simple NavNet from Appendix A. Intersections a, b, and c are the nodes. The edge weights are calculated values x, y, and Z. To the side is the list of NavVolumes 1 through 4, with a map indicating which intersections occur in each.

# Chapter 5

# Using NavNets

Whereas the previous chapter was primarily focused on the mathematical definition and construction of the NavNet, this chapters primary goal is to define the path query algorithm which makes use of a NavNet. The algorithm takes a NavNet, and two points contained within the world, and calculates a path if possible.

Although the resulting path will be the minimum path within the NavNet, it is important to note that the resulting path will be the shortest path within the sampled space and not necessarily the shortest path through the environment. As the sampling resolution becomes finer, the NavNet shortest path and the true shortest path will converge. This difference in the result comes from the NavNet being generated from an approximation of the environment. Because of this there is a chance that the true shortest path between the query points lies outside the NavNet.

## 5.1 Using NavNets

Once the NavNet $G$ has been constructed according to section 4.2.4., pre-processing has finished and the NavNet can be saved for future use. At this point, path queries can also be performed against the pre-computed NavNet. The path itself is calculated similar to how the Funnel algorithm (section 3.3) calculates a path using a NavMeshes. Path queries will result in the shortest path within the NavNet.

One notable difference between the NavNet pathfinding and Funnel algorithms is the number of dimensions each algorithm operates in. The Funnel algorithm focuses on two dimensions. Figure 3.5. illustrates the algorithm on a NavMesh., and Figure 5.1 is a zoomed in portion of Figure 3.5. The illustration clearly demonstrates how the edges of bounding triangles can be crossed via a straight line. However, wherever the path is required to bend, such a bend has to be located on a vertex. These vertices are finite and well defined in the NavMesh. However, when extended into three dimensions, a computational problem arises. The previously defined vertex, is extended to a line. However, the path will still bend at a singular point in space. Even though the path has to bend around this edge, there are infinite many locations on the line. This edge is identified as a 'Folding Edge' as the path will need to 'fold' over the edge to adjust the

direction. As any point on the folding edge can be expressed as a linear equation, we will use a sequence of linear equations to find where on each folding edge the path will bend.

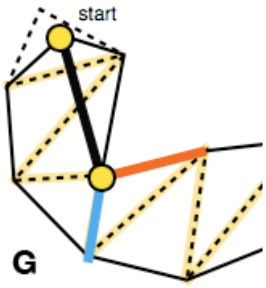


Figure 5.1: Illustration by: Mikko Mononen. Zoomed in portion of Figure 3.5.

The path algorithm has 3 stages:

1. Compute a volume path using the NavNet.
2. Refine the volume path into a parametric path using identified folding edges.
3. Reduce the mathematical path to a single shortest path within the NavNet by arg minimizing over the arguments using dynamic programming.

### 5.1.1 Computing a volume path

The volume path is the sequence of NavVolumes in the NavNet which must be traversed in order to reach the target point. It is analogous to the sequence of hallways a visitor to a university building would have to visit in order to arrive at a specific room. As a given point may lie within multiple NavVolumes, and could be approached from any direction, we need to find all NavVolumes which contain a point.

The first step in finding a path from start to target is therefore to find the volume path through the NavNet. As well as determining in which NavVolumes the start and end points are located in.

1. To avoid checking N NavVolumes, a previously generated octree is used for queries to determine which NavVolumes could potentially contain the point.
2. For each potential NavVolume, we check if the point is inside the bounding planes. If it is we add the NavVolume to a set of solutions.

$$\text{solution} = \{\} \tag{5.1}$$

$$\forall N_i \in N. \begin{cases} N_i.x_{min} <= p.x <= N_i.x_{max} \\ N_i.y_{min} <= p.y <= N_i.y_{max} \\ N_i.z_{min} <= p.z <= N_i.z_{max} \end{cases} . \text{ solution} \cup N_i \tag{5.2}$$

The above two steps are performed for both the start and end points. The start and end point will each produce a set of NavVolumes which contain the respective point. These two sets are useful for the algorithm, as when the algorithm explores a NavVolume which is part of either set, any point within the NavVolume will have a line of sight to the start or target point.

Since the NavNet is a graph where the intersections are the nodes the set of starting NavVolumes is used to determine the set of starting nodes in the graph. For each NavVolume in the set, its intersections are added to the starting nodes. Similar, the set of target NavVolumes, is used to determine a set of target nodes.

The A* algorithm is then performed on the NavNet, once for each node in the set of starting nodes. The algorithm then calculates a volume path through the NavNet until it encounters a node which is also in the set of target nodes. Each potential volume path is then evaluated and returns the sequence of NavVolumes which minimize the distance in the NavNet. This sequence is the volume path which will be then be used to compute the shortest path.

The main purpose of the volume path is therefore to reduce the number of possible paths to explore to a smaller subset.

The heuristic function of the A* algorithm uses the Cartesian distance from the nodes corresponding intersection center point to the provided target point of the path. The purpose of the heuristic function is to try to explore nodes in the graph which are moving in the right direction first, and potentially not have to visit nodes which would move away from the target point.

Calculating a volume path still requires further fine tuning. There is still a chance that A* will assign a false positive to a specific volume path which may not contain the shortest path. Figure 5.2. outlines the problem graphically, however it comes down to a limitation of line of sight. The size of NavVolume A produces intersections which lead away from the target point even though the NavVolume allows for a much shorter path to be refined later. This bias causes a lower score during evaluation of the volume paths. There are additional checks in the implementation which try to ensure that such false positives do not occur.

Figures A1.9. - A1.11 illustrate a set of NavVolumes and intersections, how these intersections are connected, and finally given a start and end point, which intersections (and by extension NavVolumes) are part of the volume path.

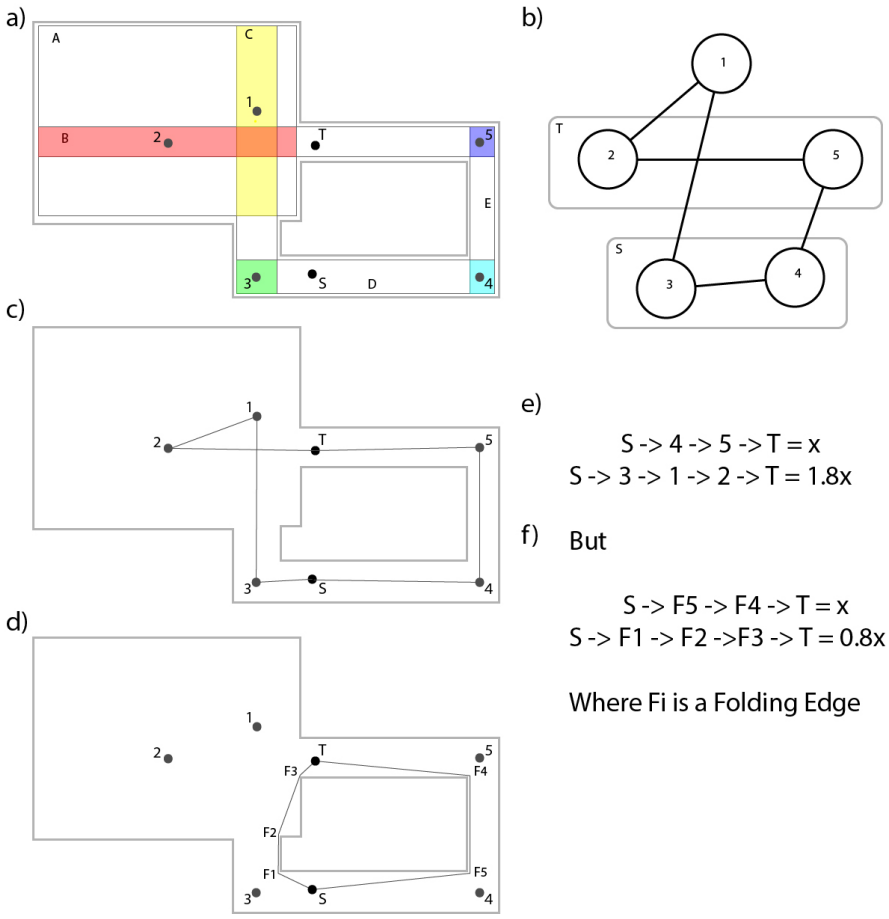


FIGURE 5.2: a) A simple test world, with the NavVolumes (A-E) and intersections (1-5) outlined. b) The resulting NavNet, nodes which can access T or S have been grouped for illustration purposes. c) Only the Cartesian center point of each node/intersection is visible. d) Two paths through the world using folding edges, e) The volume path generated by A* is shorter than the alternative path, however it is incorrect due to the use of center points. f) Path refinement on both volume paths results in the previously rejected volume path containing the shortest path.

### 5.1.2 Using Folding Edges

When all intersections in the volume path have been processed, the shortest path will intersect with the folding edges on the boundaries of the intersection. The entity will follow the path by moving along the path from the start point, through the sequence of points which intersect with the folding edges, towards the target point.

The intersections of the volume path are processed to identify the folding edge. As mentioned earlier, the shortest path will intersect with at a point on each folding edge.

A mathematical set of potential paths is formulated using the boundaries of the intersections. Then, using the geometric layout of the intersections in the volume path, it is determined which edges minimize the distance. Figure 5.3. illustrates a folding edge AE. For any two points which belong to different NavVolumes and are not connected by a line of sight, the path will have to move through the intersection of these NavVolumes if such an intersection exists. To minimize the total distance, straight lines are used with at least one bend.

Figure 5.3. shows a simple path using the convex hull (ABCDEFGH) of the intersection to connect the start and end points. The path bends over edge AE. In the figure the ideal point on the edge has already been determined at $P_i$, which will minimize the paths length.

Figure A1.12. & A1.13. highlight the folding edges in the volume path, and a visual representation of the set of potential paths respectively.

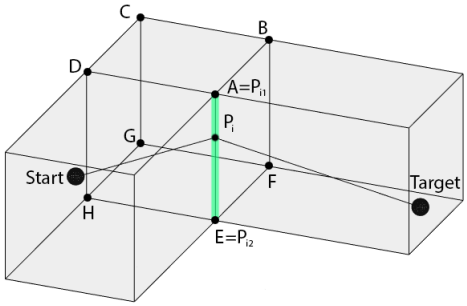


Figure 5.3: Selection of the folding edge between two intersecting NavVolumes.

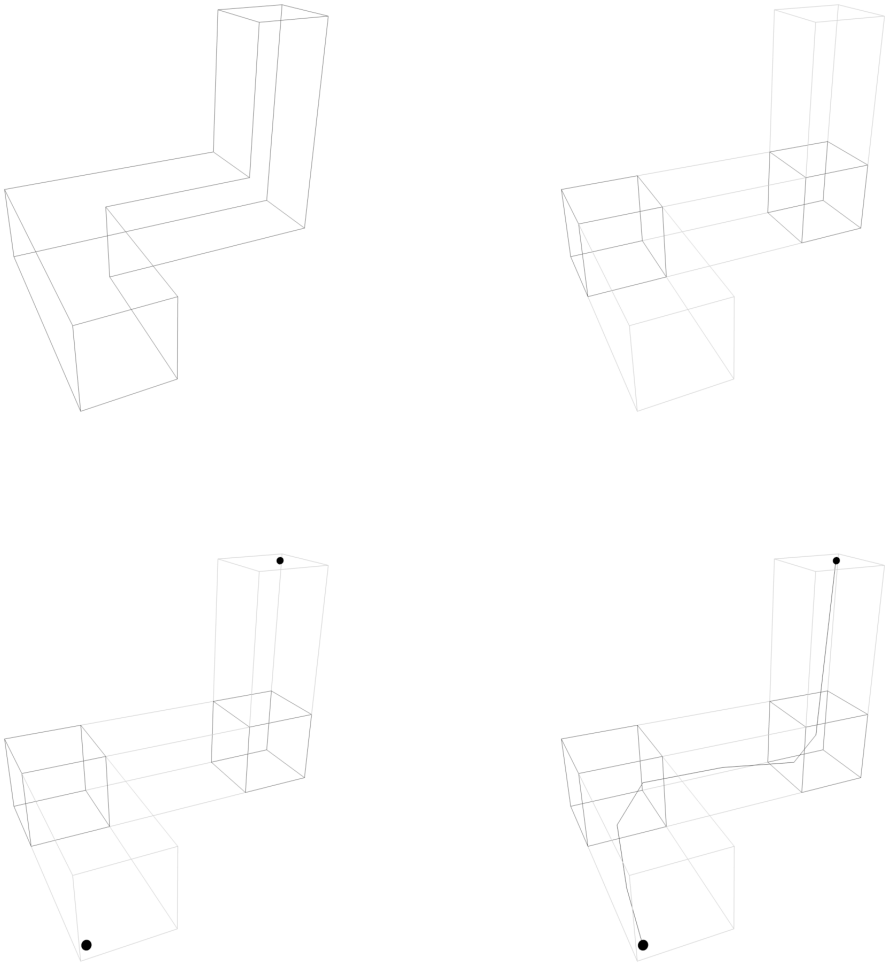

FIGURE 5.4: **Top-Left:** The sampled representation of a simple environment. Here the individual sample grid cells are joined to better illustrate the shape of the free space. **Top-Right:** The sampled environment has been broken down into NavVolumes and their intersections have been calculated. The darker outline represents the bounding box of each intersection. **Bottom-Left:** Two points are highlighted within the sampled world. A shortest path through the world connecting these two should be calculated. **Bottom-Right:** Many paths of various lengths can exist, snaking through the world.

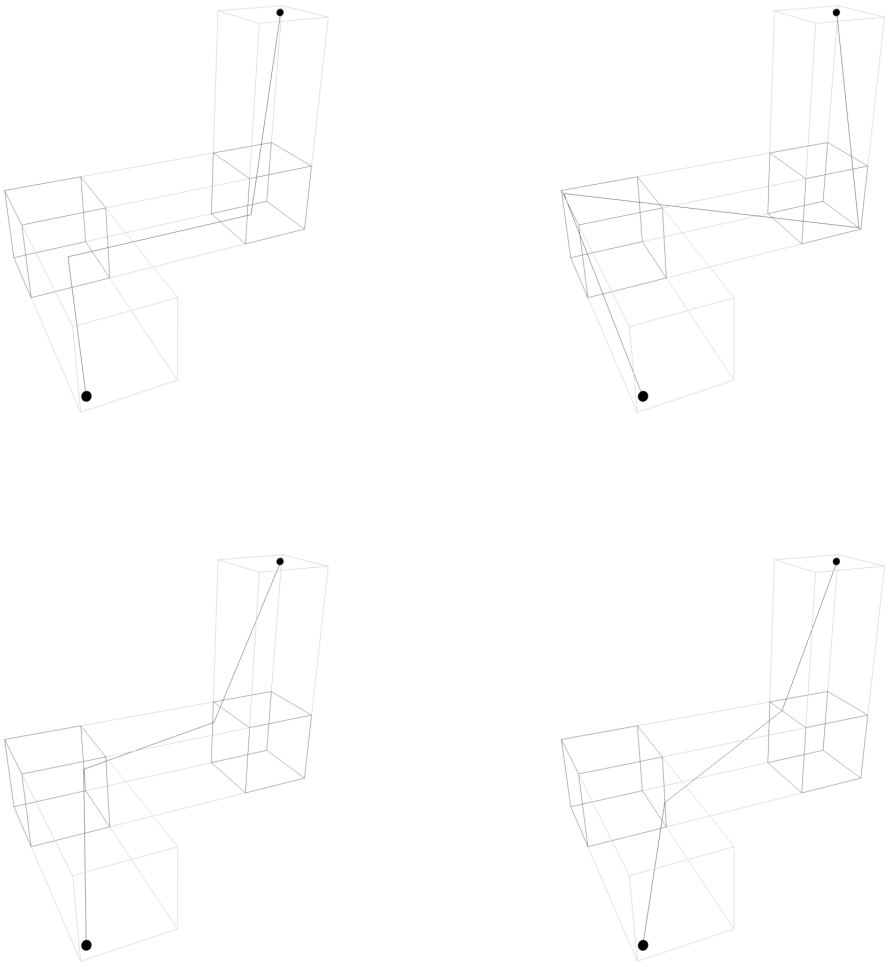

FIGURE 5.5: **Top-Left:** Since the shortest distance is a straight line, the number of turning points are reduced to their minimum. **Top-Right:** Even with fewest possible turns, its possible to have a long path through the sampled environment. **Bottom-Left:** The path is shorter, however we can do better by using the geometric layout of the space. **Bottom-Right:** By minimizing the path length the shortest path through this example connecting these points can be calculated.

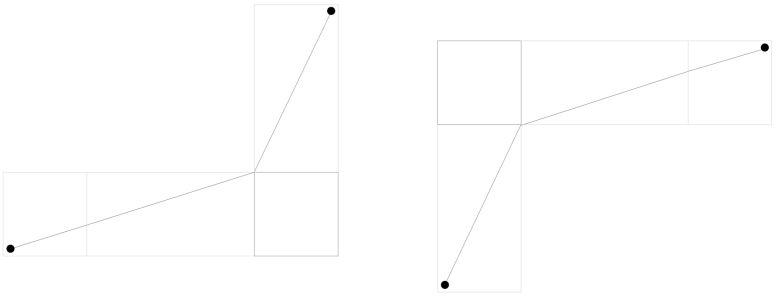

FIGURE 5.6: **Left:** From the side profile of the previous figure, the path has a turn which cannot be straighten out based on the location of the end points as well as the layout of the environment. **Right:** A similar observation can be made from the top view.

### 5.1.3 Path calculation

Once the folding edges have been determined, their corresponding linear equations are used to determine the path points $P_i$. See Figure 5.7.

To find where the optimal point is on each folding edge which results in the shortest path, we use dynamic programming:

Given $n$ pairs of end points $(\rho_{i_1},\ \rho_{i_2})$ for the $n$ folding edges, compute the set of intersection points $P_i, 0 \leq i \leq n$

$$\text{Path } P = P_{\text{ start}}, P_0, P_1, ..., P_n, P_{\text{ target}} \tag{5.3}$$

where

$$P_i = \rho_{i_1} + \lambda_i(\rho_{i_2} - \rho_{i_1}) \tag{5.4}$$

$$0 \leq \lambda_i \leq 1, i \in \{0..n\}, \tag{5.5}$$

We have the length of the path as our cost function subject to:

$$\begin{aligned} J(n) = & \; ||P_0 - P_{\text{start}}|| + \sum_{i=0}^{n-1} ||P_{i+1} - P_i|| + ||P_{\text{target}} - P_n|| \\ = & \; ||(\rho_{0_1} + \lambda_0(\rho_{0_2} - \rho_{0_1})) - P_{\text{start}}|| \\ & + \sum_{i=0}^{n-1} ||(\rho_{(i+1)_1} + \lambda_{i+1}(\rho_{(i+1)_2} - \rho_{(i+1)_1})) - (\rho_{i_1} + \lambda_i(\rho_{i_2} - \rho_{i_1}))|| \\ & + ||P_{\text{target}} - (\rho_{n_1} + \lambda_n(\rho_{n_2} - \rho_{n_1}))|| \end{aligned} \tag{5.6}$$

$$\text{ARGMIN}_{\lambda_0 \ldots \lambda_n} J(n) \tag{5.7}$$

Once a solution has been determined by computing the $\lambda_i$'s, a path has been found. We solve it with help from the Math.NET Numerics package for C#. Figure 5.7. shows such a path through a simple world.

An analogy to what we are doing is of pulling a string tight inside the volume path.

## 5.2 Summery

The majority of computational time is done during the pre-processing stage. The pathfinding using a NavNet is very fast, and breaks down into 3 steps. Using A* a course volume path is calculated over the NavNet. This volume path is then refined by finding a mathematical definition of the folding edges where the shortest path will intersect. This results in a set of potential paths defined by a graph. The final step is to use dynamic programming to find the shortest path inside this new graph and results in the shortest path in NavNet.

At each stage, the search space is reduced to allow very fast path calculations in 3D environments.

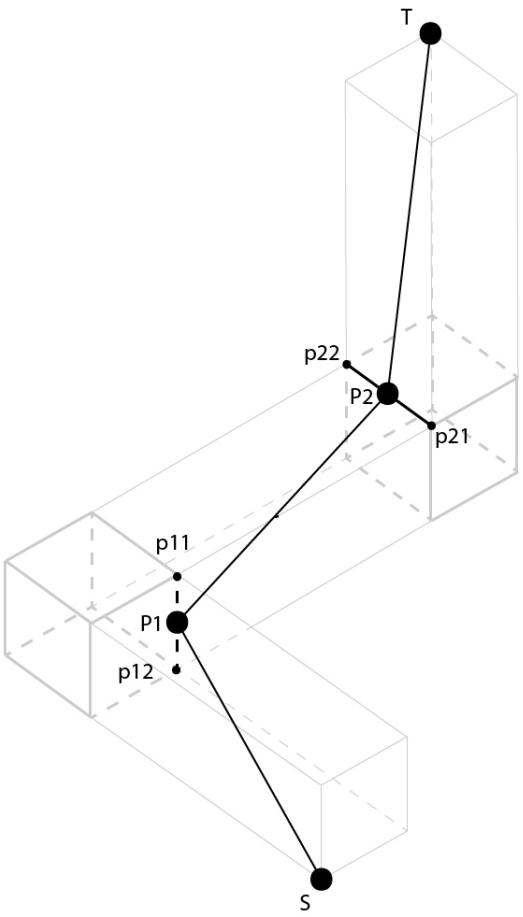


Figure 5.7: Once folding edges have been selected, the paths internal point can lie anywhere on the edge, we solve this system of equations to find the single path which has the minimum total distance out of all potential lambdas, all potential paths which could lie on these edges between S and T

# Chapter 6

# Implementation

This chapter provides the details of the implementation of the NavNets. Experiments made use of small prototypes, often focusing on one aspect. They allowed us to gather preliminary data as well as acquire an intuition about how the theoretical constructs of the NavNet would need to be implemented. The cumulation of these experiments, as well as the theoretical construction have shaped the final version of the NavNets.

In order to efficiently analyze the 3D environment, we use techniques used in Computer graphics. Their data structures, intersection tests, as well as optimizations with respect to 3D environment have greatly aided in shaping how the implementation executes. We combine these techniques with sampling to significantly reduce the search space during runtime, while minimizing error.

During the implementation, we had to add additional structures, methods, and support algorithms to achieve in practice what the theory in Chapter 4 sets out. The mathematical definitions which were able to be expressed in a single line, required many more lines of code to actually function as intended. As such, the implementation follows the spirit of the theory of NavNets.

The NavNets make use of a preprocessor and a runtime component. During preprocessing, the environment is analyzed, sampled, and then used to construct the NavNets. At runtime, the NavNets are used in pathfinding which provides navigation information to a motion strategy of an agent in the environment.

## 6.1 Dataflow and processes

### 6.1.1 Preprocessor: Sampling

Sampling represents the first step of generating a NavNet. During sampling we preprocess the world in such a way that we can more easily represent it for finding maximally convex volumes within an environment. Because it utilizes sampling to achieve this, the result will be an approximation of the true environment. As such, there is a certain error

associated with the NavNets. However, for real world applications, the approximation is acceptable.

Figure 6.1. illustrates the classes being utilized during the sampling stage. As well as the dataflow of how the input of a 3D environment results in an output of an approximation through the sampling process. The 3D environment is composed of 3D meshes, each of which is composed of triangles. Each triangle is defined by 3 vertices which are represented as points.

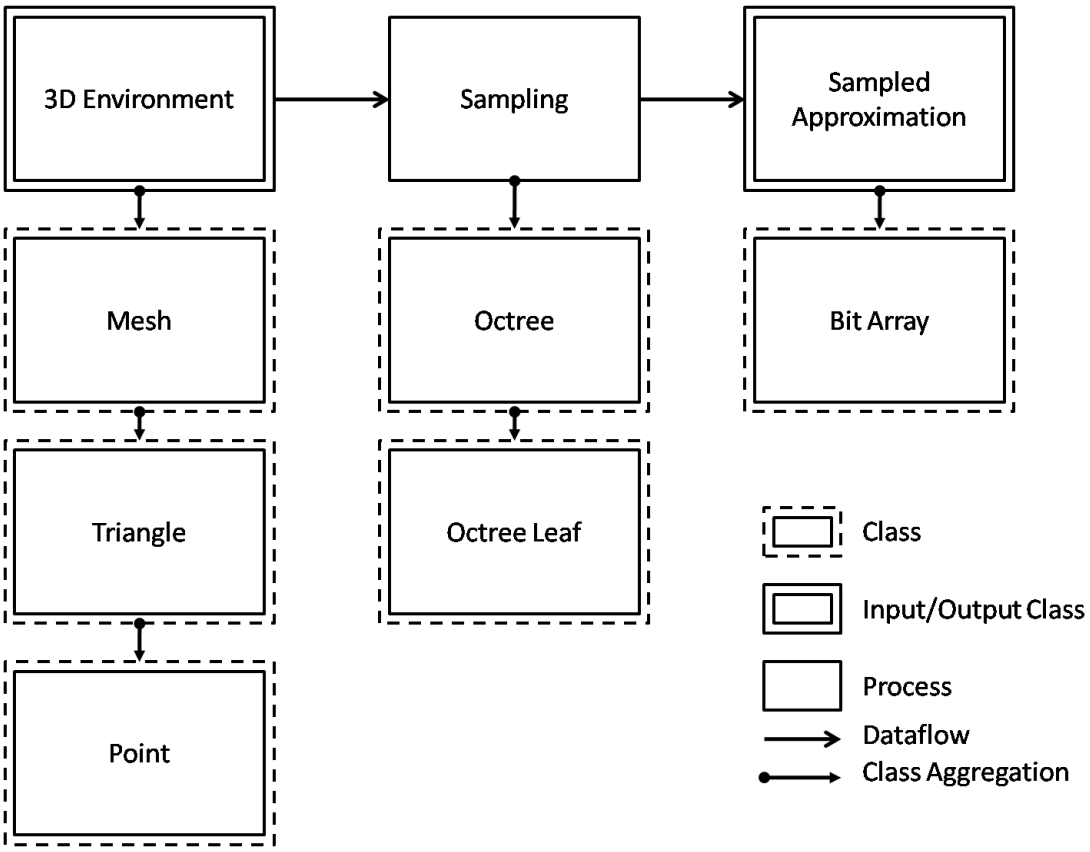


FIGURE 6.1: Dataflow and Classes used in Sampling. Illustrates what comprises the 3D environment input of the sampling process. The output results in a sampled approximation which makes use of a BitArray

The sampling process itself has six portions. Figure 6.2. visually shows these stages and the dataflow between them. The sample grid makes use of an octree to increase performance as well as gain locality information. Triangles of the 3D environment will be stored in the leaf nodes of the octree. Accessing triangles in a neighbouring subtree is achieved by moving up one level to the parent tree, and then down the next child subtree. Once an octree has been initialized, every triangle of every mesh in the environment is inserted into the octree.

The environment is divided based on the locality of its triangles, we use a 3D sample grid which spans the whole octree in terms of size. The number of grid cells are determined by the sampling resolution, and we start with a blank slate, no grid cell has an intersection.

Because the number of grid cells typically outnumber the number of triangles by a significant magnitude, checking each grid cell against all triangles would be a waste of time. A better approach was borrowed from Computer Graphics using the marching cube concept. Each point of the triangle is used in a min/max calculation on a per component level. An axis aligned minimum bounding box is established around the triangle, and to select the sample grid cells which intersect with this box is trivial because of point to index mapping. By using this method, we significantly speed up the intersection test by eliminating most of the grid cells from needing to be checked. Although mathematically the time complexity is the same, in practice our method executes much faster.

To perform the actual intersection test we use a Triangle-Box intersection test. At its core it uses the Axis Separation Theorem, which breaks the test into 13 test cases. Figure 6.3. is a visual representation of the Triangle-Box intersection test and shows the dataflow of the inputs, as well as the groups of test cases.

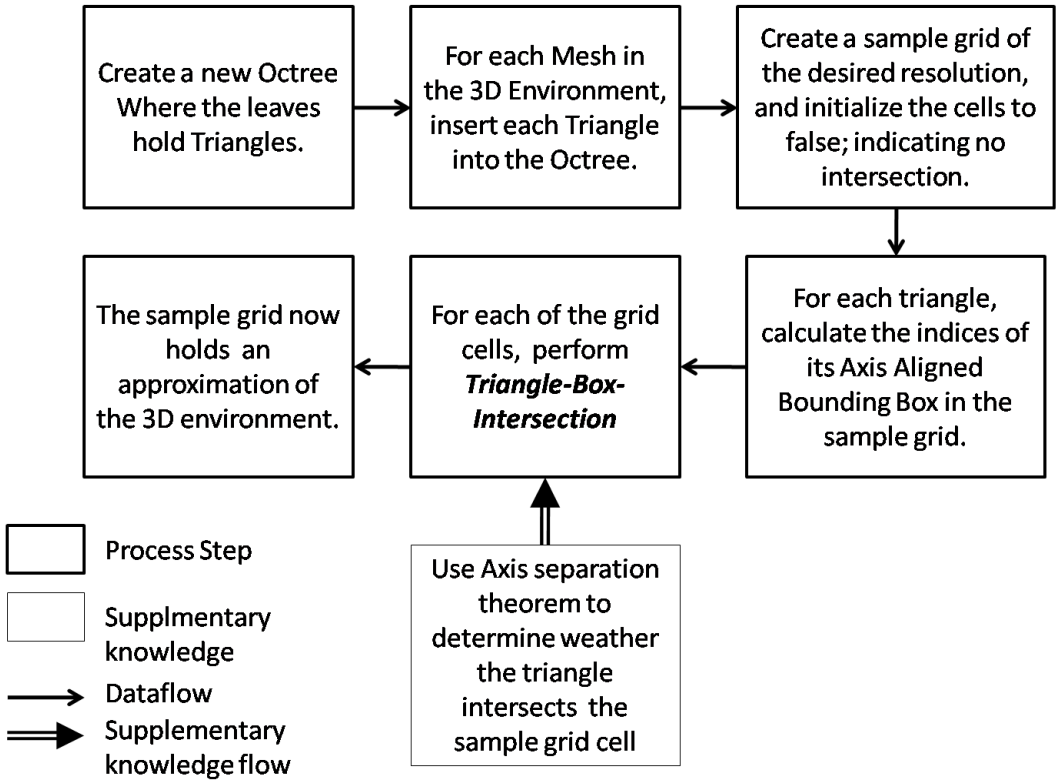


FIGURE 6.2: In Detail View: Sampling Process. Here the stages are illustrated of the sampling process. It outlines when and how additional structures are created, how the inputs are used, as well as which theorem is used to aid in determining if a sample grid cell is empty or intersecting.

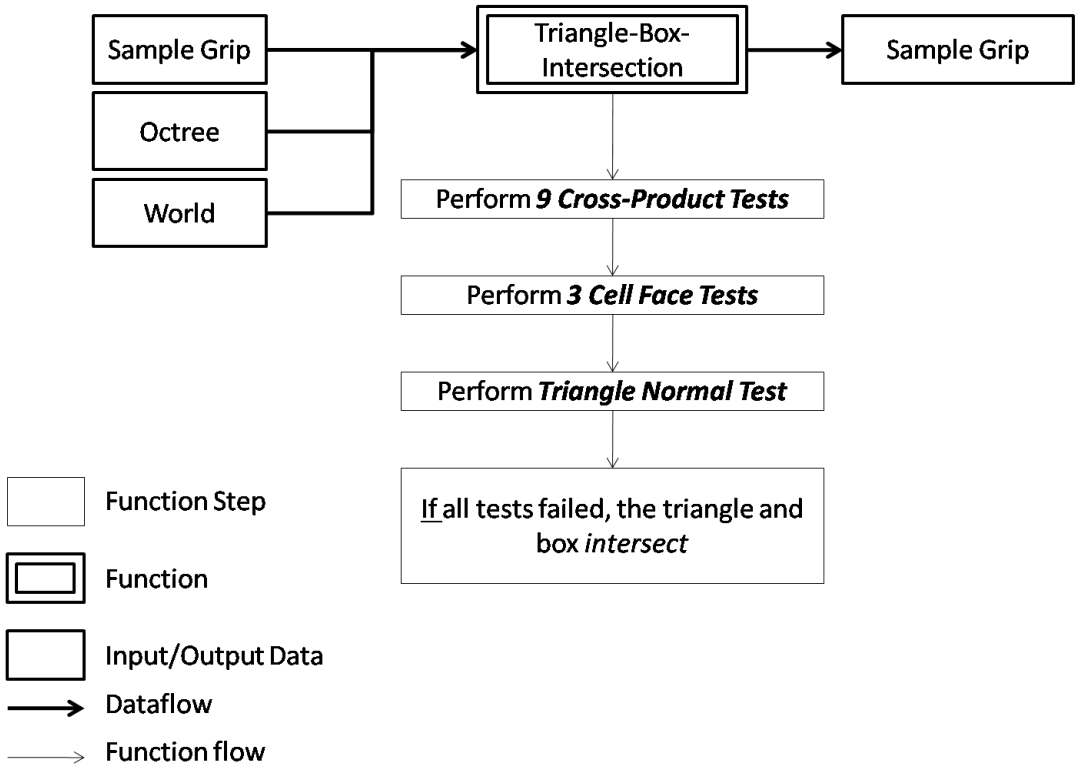


Figure 6.3: Dataflow of the intersection test. The world and samplegrid supply the box and triangle, then using 13 tests it is determined if the box and triangle intersect.

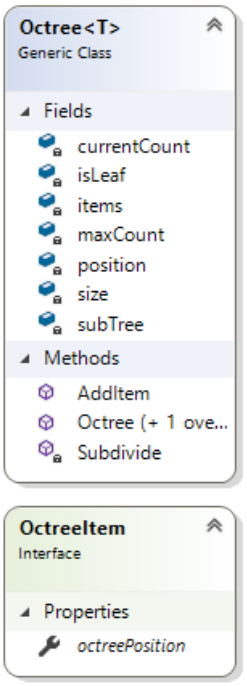


Figure 6.4: Octree and octree leaf classes, used during sampling to reduce runtime.

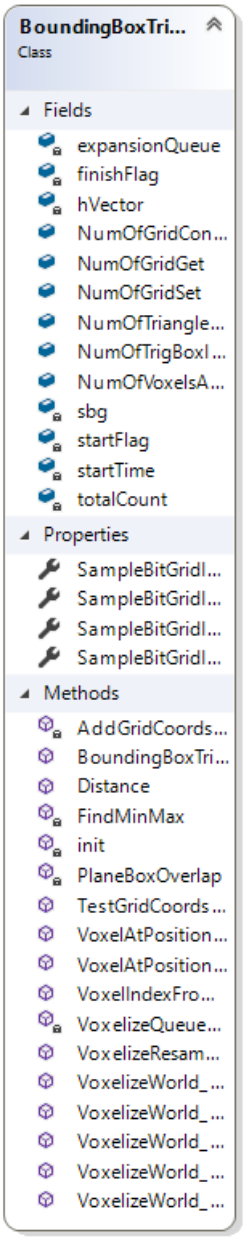


FIGURE 6.5: The BoundingBoxTriangleIntersection class is a static class which is used to check a grid cell against a world triangle.

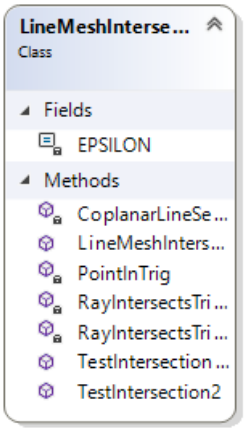


Figure 6.6: Line Mesh Intersection class. It is a static class which contains helper functions used to check for an intersection between a triangle and a line segment in 3D space. It also contains methods to visualize the result.

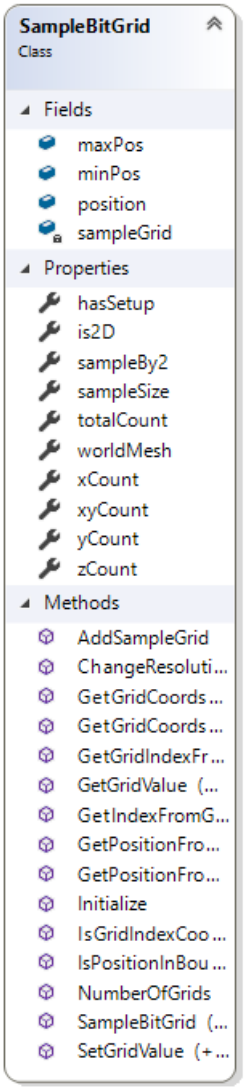


Figure 6.7: The implemented class of the sample grid. The grid itself is finite, and contains a position as well as boundary information. The data is stored in a BitArray, and is manipulated through its functions. The class also contains utility functions to convert index formats to use a one dimensional index or a three dimensional index.

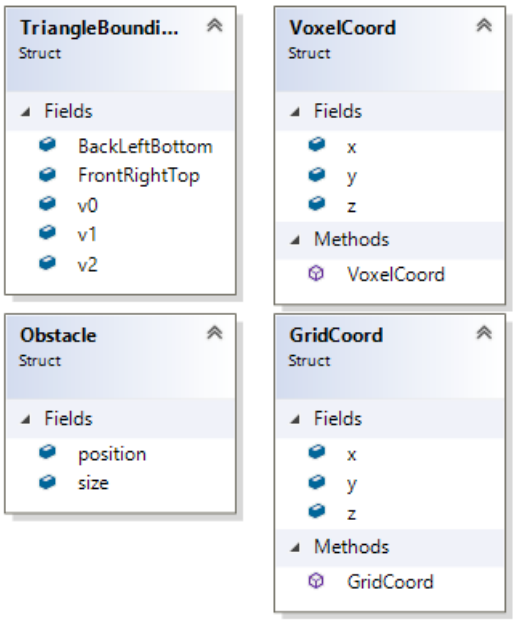


Figure 6.8: Various structs used to wrap data which is common but separated for easier reuse of the testing classes.

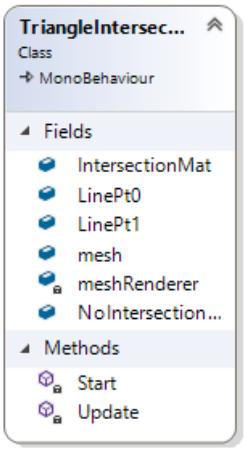


Figure 6.9: Visualization class, used to visualize a triangle which intersects a line segment. The data is passed into a static utility class, this class serves only to provide visual feedback within the Unity3D engine.

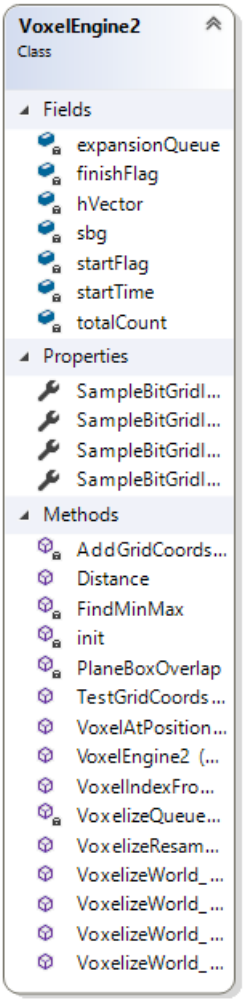


Figure 6.10: VoxelEgnine class, a class from a previous prototype. Before targeted sampling was utilized, we discovered the freespace by growing outwards. This class handled the logic.

### 6.1.2 Preprocessor: NavNet Generation

Once the world has been sampled, we will use the approximation for all further steps. Ideally the number of grid cells which intersect with the environment is smaller than the cumulative number of all triangles of the world. However, even when that statement is true, generally the number is still too large to perform any pathfinding over the grid cells effectively. It is possible, using an algorithm such as A* over the sample cells, where each cell is connected to its adjacent cells. However the resulting graph would be very dense, and this method resulted in extraordinary long run times. Most experiments were aborted after 2 hours as we would like to not block the game or simulation while finding a path for the agent to follow.

In order to achieve near-real time execution speeds (we will be aiming for around 60 frames per second) the search space needs to be further reduced. This reduction is achieved by grouping the grid cells, which do not intersect with the environment, into maximally convex regions. These regions are the NavVolumes and they can overlap one another, that is to say that the bounded portion of space of a sample grid cell can be part of multiple NavVolumes.

Once all NavVolumes are determined, we calculate how they intersect one another, and use these intersections as nodes in a graph. The graph can then be used by a node algorithm, such as Dijkstra, to find a path through the graph when supplied a starting intersection and a target intersection. This graph is called the NavNet, and is used to calculate a volume path which then gets refined into an actual path using waypoints for an entity to follow.

Figure 6.11. outlines the dataflow of the input of a sampling to the resulting NavNet. It also demonstrates the hierarchy of NavVolumes and intersections as well as their common parent the NavBox class. This hierarchy and NavBox is an implementation feature only and there is no mention in our theoretical definitions. Its purpose is to reduce code duplication.

Figure 6.12. outlines the three stages of the NavNet generator, as well as what takes place in each of the stages. The first step is to take the sample grid and find the NavVolumes, the maximally convex regions which the entity can traverse. To reduce some overheads in tracking and reduce unnecessary checks we created a supplementary array *visited*, this is also an implementation feature which has no impact on the theory or complexity, however it does reduce the execution time at runtime. The actual process through which the NavVolumes are generated is the process represented by the *Convex Expansion*. We will further describe how NavVolumes are generated below.

The diagram also outlines how the intersections are based on the boundaries. During the implementation our NavBox is defined by eight bounding vertices, forming a rectangular prism. Calculating the bounding vertices of the intersections is trivial since we have access to the bounding vertices of the NavVolumes and we can calculate the overlap region.

Finally, as mentioned above, the intersections are used as the nodes in the navigation graph, where the NavVolumes form the edges. We can then calculate a volume path over this graph quite easily using any shortest path algorithms which make use of a graph. The graph is the NavNet, and it has a few more methods and functions compared to a regular graph. It contains methods to determine which NavVolume a point lies within, and from there we can determine a list of source nodes. Similarly, we can do the same for the list of sinks and the target point.

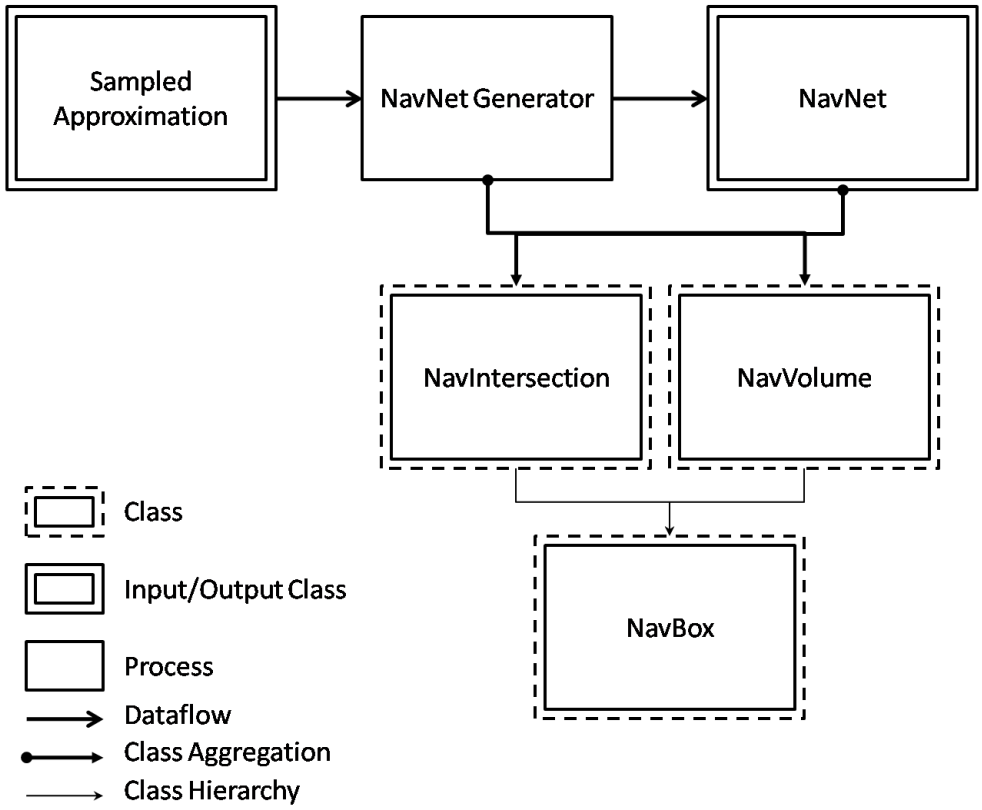


Figure 6.11: Dataflow and Classes used during the NavNet Generation. The NavNet generator takes a sampled approximation from the previous step as an input, and outputs a NavNet. It also displays the hierarchy of intersections, NavVolumes, and NavBox classes

### Grouping into NavVolumes

Where most of the NavNet generator is straight forward, or contains some trivial component, the grouping of sample cells into maximally convex sets is quite complex. In this section we will take a closer look at how a NavVolume is generated as it may be the single most important aspect of generating a NavNet.

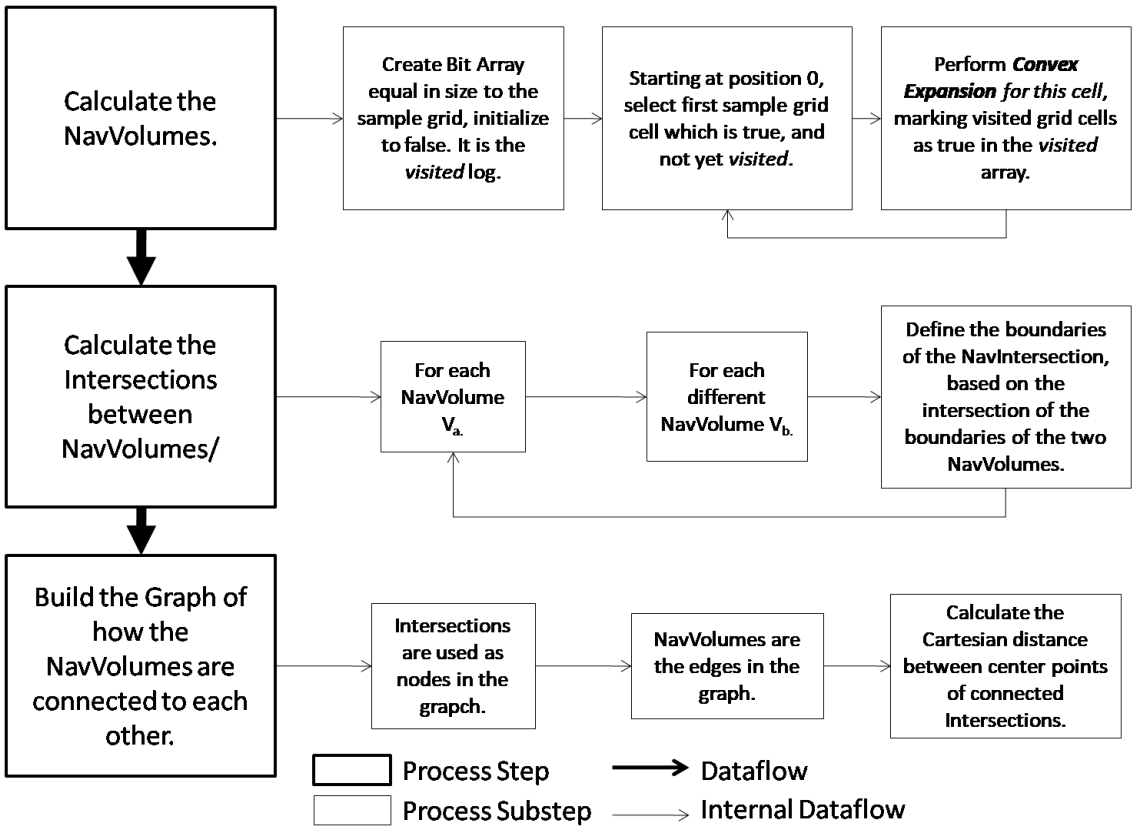


FIGURE 6.12: This diagram shows the major stages in the NavNet generation process. As well as, the logic flows to complete each stage. First NavVolumes are calculated using Convex Expansion. Then intersections are determined and saved. Finally, the NavNet is defined using the NavVolumes and their intersections.

Figures 6.13., 6.14., and 6.15. visually break down the process of finding a maximally convex set within the sample grid.

The start of a NavVolume is a sample grid cell which does not intersect the environment. This grid cell will act as a seed to 'grow' and expand outwards filling the space in a convex manner. Once the NavVolume can no longer expand in any direction, the NavVolume is maximally convex in the sample grid.

The logic flow is shown in Figure 6.13. The function has access to the sample grid, a shared visited array to keep track of visited grid cells, as well as a reference to the list of all NavVolumes, and a starting location: a seed. If the seed has been visited before during a previous NavVolume expansion, it will not be expanded as it could potentially generate a duplicate NavVolume. Instead we will expand the first seed which has not been visited. During the expansion the process uses the indices of the sample grid.

We define the expansion space as two opposite corners of a rectangular prism. The NavVolume will then perform multiple shell expansions until any expansion would be concave.

The shell expansion still has access to the sample grid, the visited array, as well as the indices which define the rectangular prism. For each direction along the major axis, we check to see if its possible to move each boundary of the prism outward by one grid cell index.

This boundary expansion uses each component (x, y, z) of the two bounding points. From there we use the sample grid to check if any of the grid cells have intersected with the environment.

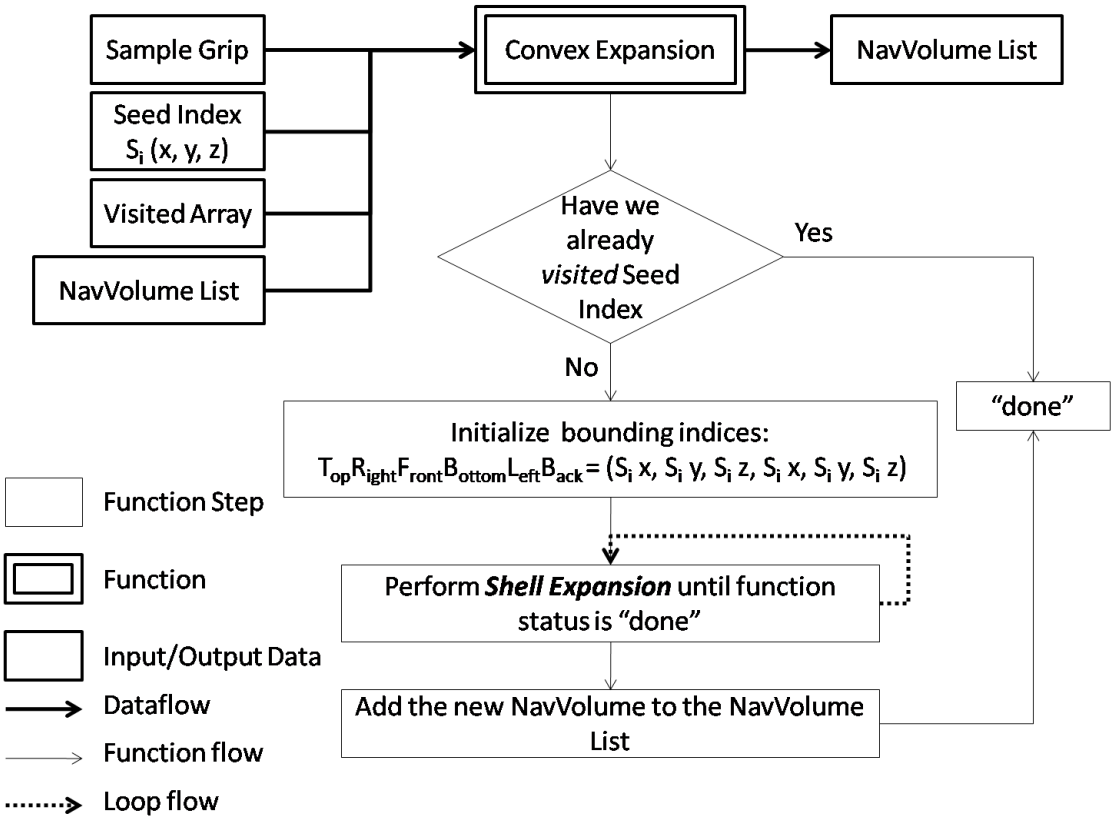


FIGURE 6.13: The Convex Expansion process uses the input data and calculates bounding parameters, from there it repeatedly expands the seed until any further expansion would result in a concave shape. It adds it to the list of NavVolumes, and exits.

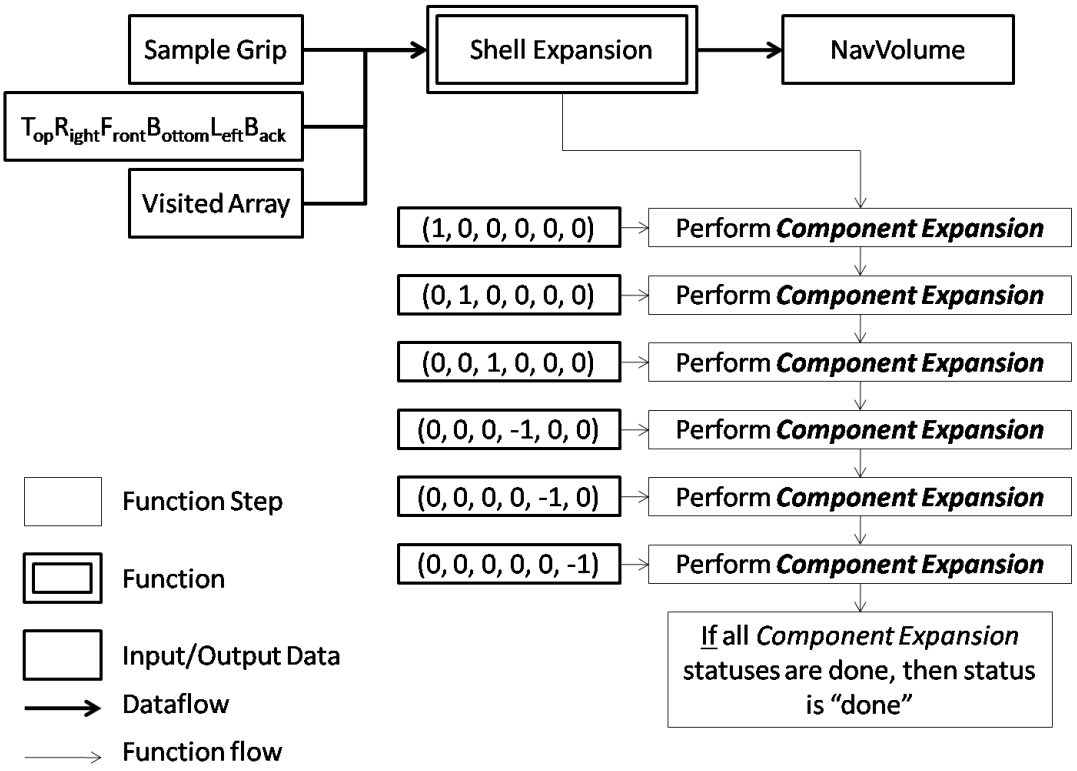


FIGURE 6.14: The dataflow of a shell expansion step. This is a sub process of convex expansion. The shell expansion performs a component expansion for each bounding surface of the new NavVolume, along the -x, -y, -z, x, y, z axis.

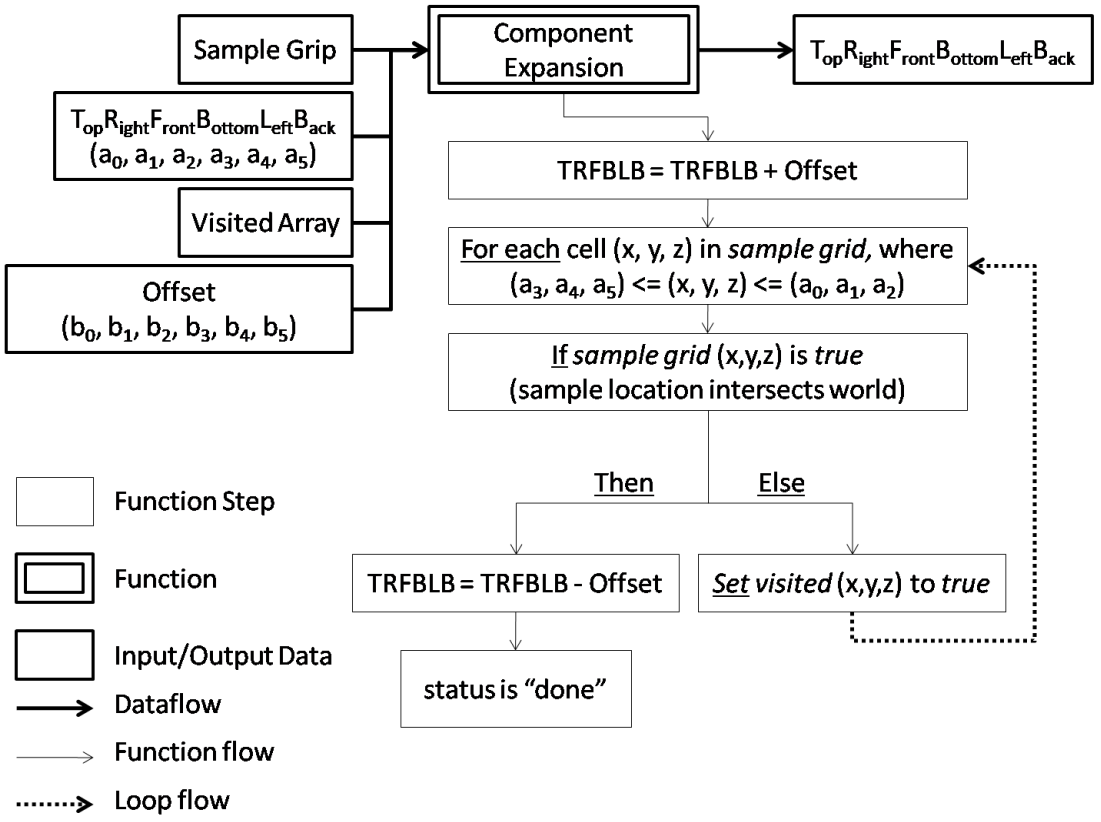


FIGURE 6.15: The component expansion process is a sub process of the shell expansion process. It outlines the logical flow of how it is determined if a bounding surface can be expanded along a world axis, or if it would form a concave shape.

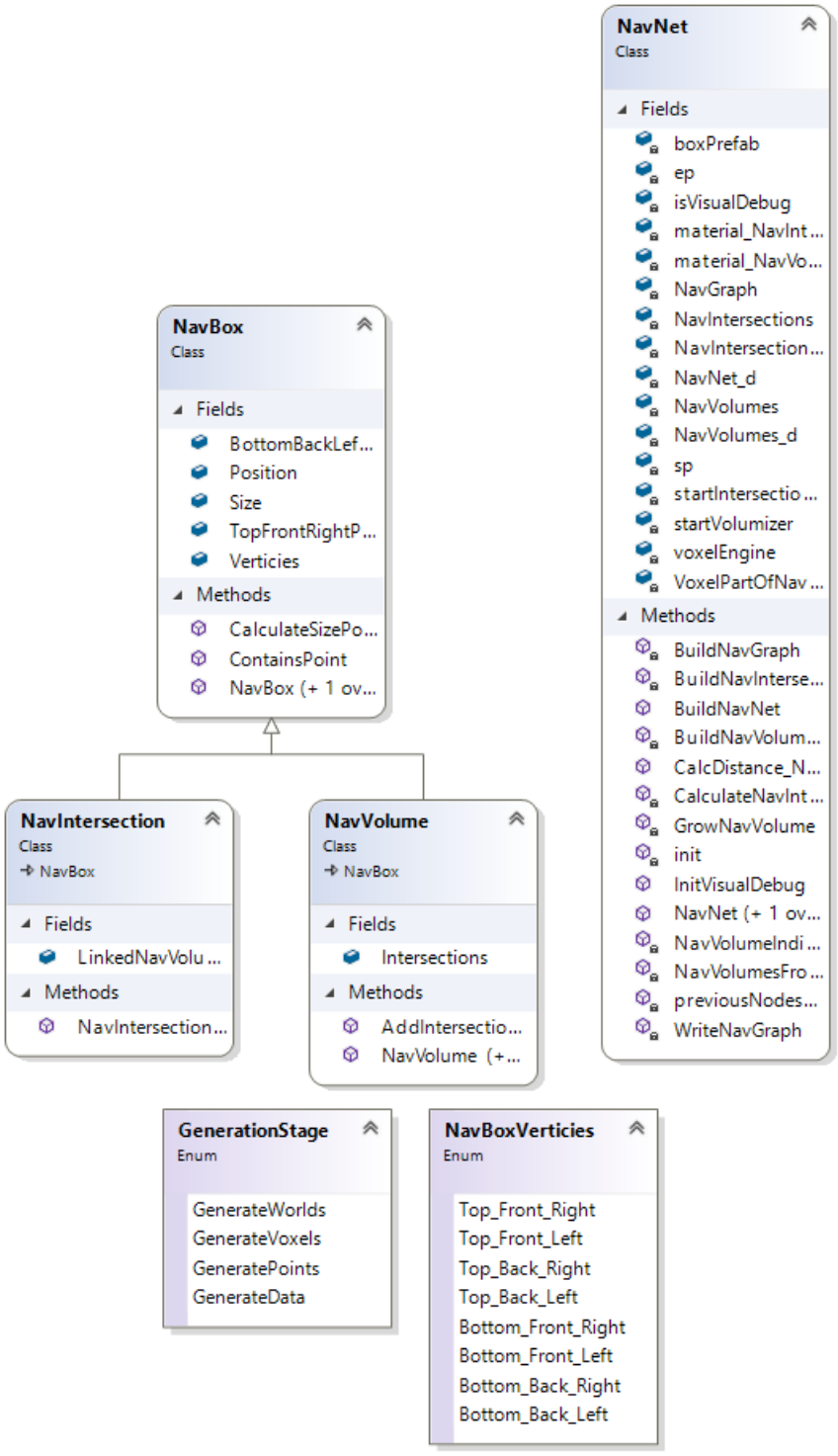


Figure 6.16: Class inheritance of the NavVolumes and their intersections, NavNet class implementation, control enums for testing as well as references boundary corners

### 6.1.3 Runtime: Pathfinding

Up till now, we have not yet found a path of any kind. We have been focused on simplifying the environment by reducing the world geometry through sampling. This effort is done in advance, and the resulting NavNet is then saved for further use. The NavNet only deals with static meshes and obstacles, as such, once we have preprocessed an environment the NavNet is ready to be used for all future path queries. During runtime, we will query the NavNet to find a volume path. It will then be further refined to a single path within this volume path. The path refinement approach allows us to greatly reduce the runtime complexity of finding a 3D path through the environment.

Figure 6.17. outlines the dataflow of the path solver. The solver requires a NavNet as an input. It was calculated during the preprocessor stage and contains the data on how the NavVolumes are connected to one another. It also takes two point objects as inputs. These will become the start and target points of a path, if a path exists. These points are also used to determine the start and target NavVolumes in the NavNet. Since each NavVolume has a bounding box associated with it, it is trivial to check whether the coordinates of a point lies within these boundaries. The Solver will then calculate a course volume path over the NavNet, and then further refine it to a path comprised of a sequence of point objects. These point objects will act as waypoint markers for the motion strategy.

The pathfinding process is further explored in Figure 6.18. The process involves three stages which refine the NavNet into a single path marked with waypoints at direction changing points. The first stage involves actually calculating the volume path. In the implementation we added a helper method to each NavVolume which takes in a point and returns either true or false depending on whether or not the point lies within the NavVolume. A point is defined as true if for each component it is within the bounding points of the NavVolume. Recall that during NavVolume generation a bounding vector **TopFrontRightBackLeftBottom** was used to keep track of the index location of the **top-front-right** corner and the **back-left-bottom** corner of the bounding box of the NavVolume. We can make use of this property to determine if a point lies inside a give NavVolume. For a point p(x,y,z) it lies within the NavVolume if and only if:

$$\textbf{Left} \leq x \leq \textbf{Right}$$

$$\textbf{Bottom} \leq y \leq \textbf{Top}$$

$$\textbf{Back} \leq z \leq \textbf{Front}$$

Once we have the starting and end NavVolumes, we can use them to find the corresponding intersections. Recall that the intersections are the nodes in the NavNet graph. As such, we will not find a path from a single start or source node, to an end or sink node, but rather we will have an array of start nodes and an array of end nodes. From there we expand each node and calculate a shortest path on this graph. It is important

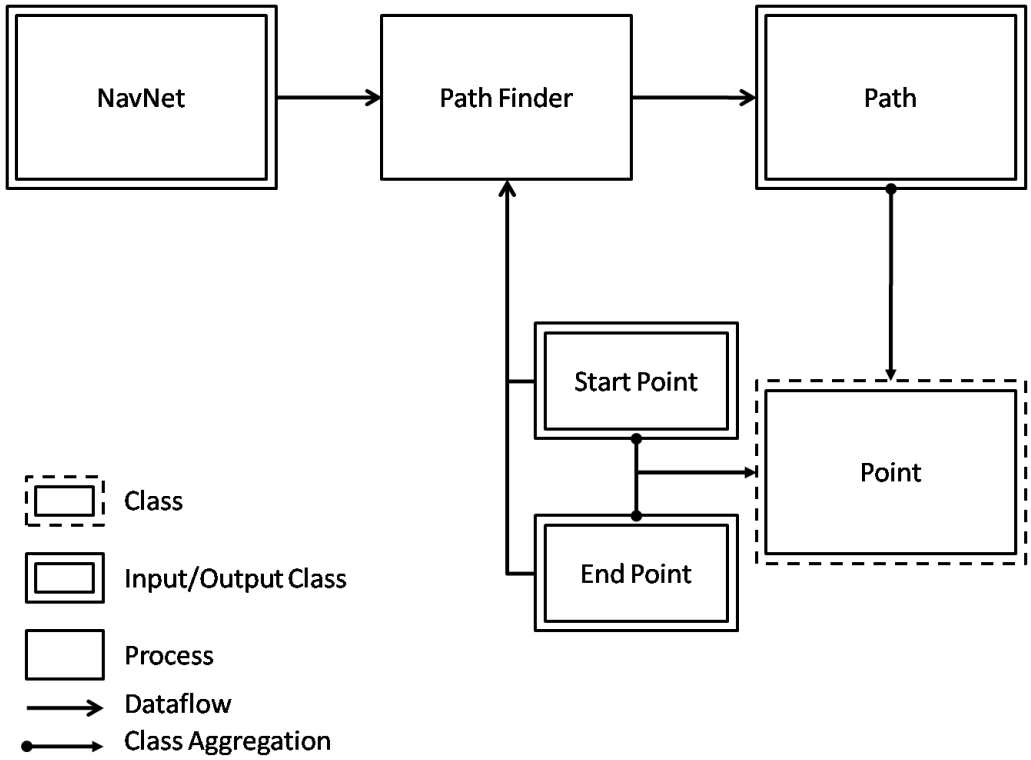


Figure 6.17: Dataflow and Classes used in Pathfinding. The path solver takes a NavNet object as an input, and will provide a path as an output which is comprised of a list of point objects. The Solver also takes two point objects which will be the starting point and the end point of the path which we would like to find.

to state that the shortest path in the NavNet is not the shortest path in the environment. The NavNet produces a volume path which will contain a path that minimizes distance. Since A* algorithm uses a heuristic function to determine the final distances, care had to be taken to find a function which works well. Through experiments we discovered that the volume of the intersection works very well as our heuristic function. The Cartesian distance between connected intersections is augmented with the volume of the intersection. The benefit of this heuristic function is that it helps to remove some bias towards smaller intersections when use of a larger intersection yields a better volume path. When the volume path is calculated, then the sequence of NavVolumes are passed into the refiner stage.

In order to quickly find a path in 3D we do not rely on the world geometry or the use of a densely packed node graph, instead we preprocess an approximation of the free space through sampling. The approximation is used to determine a volume path, which

we will now be refined into an actual path. A path is a sequence of points where the entity can move from one point to the next. The movement depends on the motion strategy of the entity, but a waypoint motion strategy is a basic strategy for a sequence of points.

The first stage of path refinement is the identification of so called folding edges. These edges are the edges of the boundary of an intersection between two NavVolumes. Using two consecutive NavVolumes of the volume path, we identify the folding edge using the geometric configuration of how these NavVolumes intersect. Figure 5.3. illustrates this edge selection. A starting point in one NavVolume needs to terminate at a target point in the other NavVolume. In order to achieve this, there will be a bend of this line on the intersection. In order to minimize the distance we use the boundary of the intersection. In the case of Figure 5.3. edge AE will act as the folding edge as the resulting path start - point on AE - target will result in the minimal distance. Any other edge of the intersection will produce a sub optimal result. The folding edge is then expressed as a linear equation with one argument. Dynamic programming is then used to solve where on the folding edge the path intersects.

Figure 5.3. illustrates the base case of a volume path consisting of only two NavVolumes and one intersection. However, a longer volume path is not more difficult. First all folding edges from the start of the volume path to the end are determined. With this list of folding edges, we form a function with an argument count equal to the number of folding edges. This function is then argument minimized to find the shortest path through the volume path using dynamic programming.

This path is then returned and used to move the entity.

## 6.2 Data Structures

The NavNet makes heavy use a number of data structures. The following is a summery of the important data structures implemented.

**Point** A struct containing three floating point variables as intended

**Line** A struct containing two points, as well as colour information for each end.

**Triangle** A class which contains three points as the three vertices of the triangle. It also contains methods for rendering the triangle, and highlighting edges.

**ArrayLists** A provided class which is able to have constant access time, while still able to dynamically grow and shrink in allocated space. Any array used in Unity3D is an array list.

**Mesh** A class which contains an array of vertices, triangles, and indices which define which vertices belong to which triangle. It also contains arrays for texture coordinates and colour information. A mesh is used to store information about obstacles, the environment, and for rendering the entity.

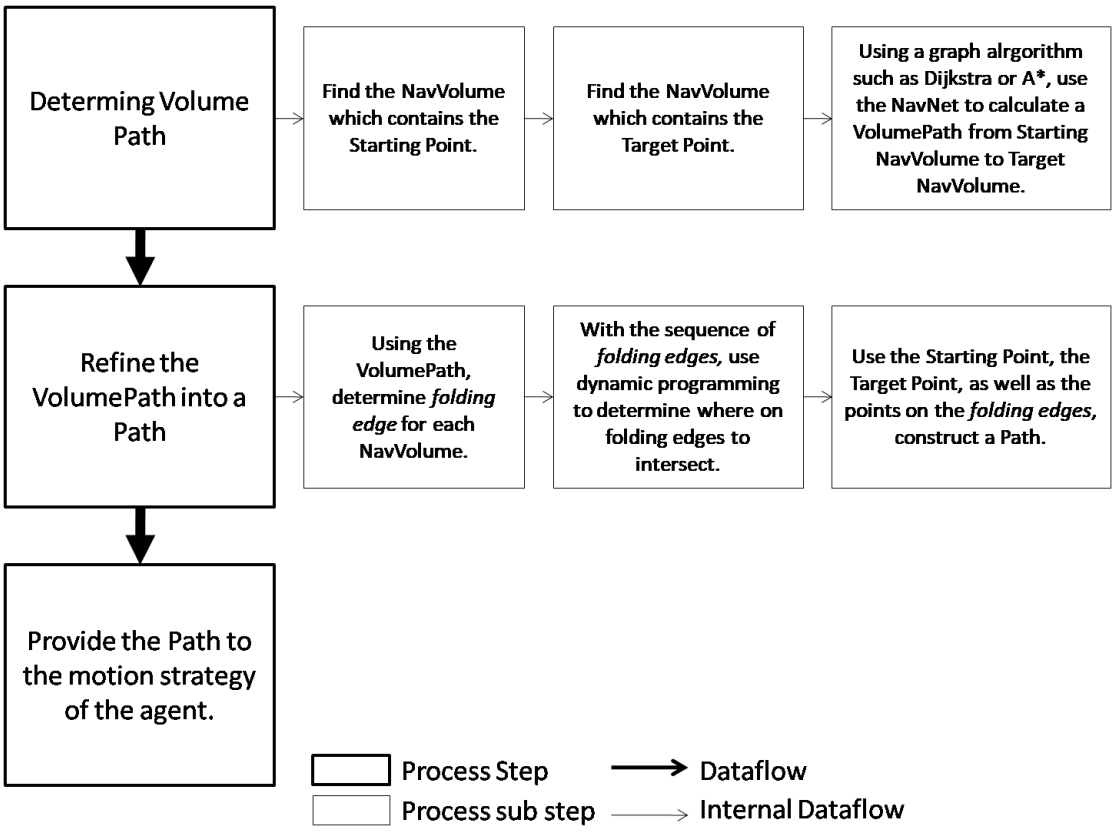


FIGURE 6.18: This diagram illustrates the dataflow of finding a path using a NavNet. **1.** Determine the starting and ending NavVolumes then calculate a volume path using the NavNet with A*. **2.** A path is refined from the volume path. The intersections are used to determine folding edges, edges which the path will intersect to cross from one NavVolume into the next. A sequence of waypoints will be calculated in this manner. **3.** The calculated path is submitted to the motion strategy for the entity to follow.

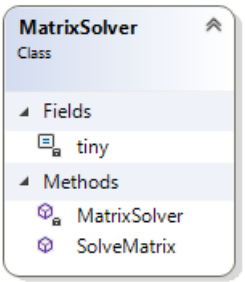


FIGURE 6.19: Matrix solver

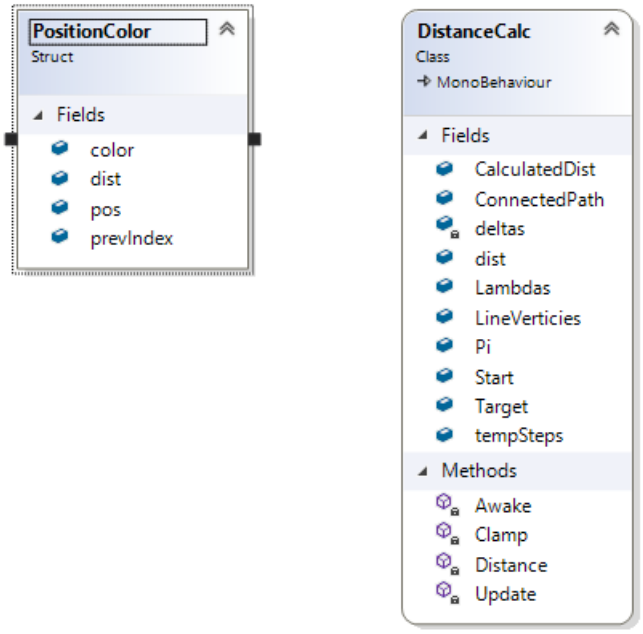


Figure 6.20: Distance calculation class, and position visualization struct for testing.

**Octree/OctreeLeaf** The octree is used as an optimization, it and its leaf are used to build up a spatial directory. The octree leaf contain position information and parent information. The octree node contains information about its neighbours and children. As well as, methods for subdividing an octet into eight children.

**BitArray** A more space efficient form of array. Each byte of computer system memory contains 8 bits. The bit array is able to store 8 indices of true and false information per byte using bit masks internally. Sampling and NavVolume generation make use of the BitArray.

**SamplegridCoord** A struct which is composed of three integers which hold the x, y, and z indices of a given sample cell. Used to pass grid cell location data between functions and classes, as well as represent the seed coordinates for **convex expansion**.

**NavBox** A class which acts as the base class for the NavVolumes and intersections. Both share how they are defined and what information such as position, size, eight bounding vertices.

**NavVolume** A class which inherits from the **NavBox** and adds additional parameters used during generation of the NavVolume. It maintains a list of index positions of

the sample grid which is part of this NavVolume. This is done to reduce execution time at the expense of memory.

**NavIntersection** A class which inherits from the **NavBox**. In addition to the properties of the base class, an intersection also contains references to the two NavVolumes which intersect. the intersection is a concrete representation of the intersection between two NavVolumes for fast lookup and use within a NavNet. The intersection also contains the volume for the heuristic function during pathfinding.

**NavNet** The NavNet is a class which contains the list of NavVolumes, the intersections, as well as precomputed weights for the edges between the nodes.

**Color** A struct containing colour information for visualization. Properties are Red, Green, Blue, and Alpha used for transparency. It also contains variables which set the internal colour blending method, and display shader.

## 6.3 Methods

The following methods are used within the NavNet system. Some may belong to multiple classes with slightly different implementations. Others are static helper methods.

**Octree - AddItem** Method which takes an object and a position and creates a new OctreeItem. This leaf is then inserted into the Octree. The position is used to determine which octet the object is in. It is used instead of the objects center point to provide finer control.

**Octree - Subdivide** Method which, when the max leaf count has been reached, will subdivide the octree node, and distribute the current leafs into the eight child nodes.

**BoundingIndiciesFromTriangle** A method used to reduce the number of sample grid cells which require testing for intersection. It takes a triangle and based on its vertices, calculates the min/max values for the x, y, and z axis. It then determines the bounding indices in the sample grid where these min/max points lie inside of. It returns a SamplegridCoord.

**TriangleBoxIntersection** A static method which constructs a box based on the sampling resolution and the sample cell indices, and tests for an intersection between this box and a triangle. It makes use of the Axis separation theorem to perform 13 tests. It returns a Boolean.

**ConvexExpansion** Method which takes a SampleGridCoord as a seed, and expands to form a maximally convex NavVolume in the sample grid. It returns an ArrayList of NavVolumes.

**NavVolume FromPoint** Method which determines which NavVolume contains the input point. It returns an index to the specific NavVolume in the NavNets NavVolume ArrayList.

**CalculateVolumePath** Method which uses the NavNet, a starting point, and a target point to determine a shortest path on the graph using the defined edge weights as well as a heuristic function. The volume path is calculated using the A* algorithm.

**FoldingEdgeFromNavVolume** Calculates a folding edge based on two NavVolumes in the volume path. It returns a folding edge as a line.

**RefinePath** Method which refines a path based on a start point, target point, and a sequence of edges. It sets up a dynamic programming problem and solves it while minimizing distance. It determines precisely where on the folding edge the intersection point must lie. It returns a sequence of points, starting with the start point, and ending with the target point as well as the path distance in world units.

## 6.4 Development environment

The development, testing, and data gathering were done on a windows computer with the following specifications:

- Intel i7-4790k @ 4.4GHz (Water cooled with Corsair H80i v2)
- 32GB Ram
- Gigabyte NVidia GTX 660
- Gigabyte Z97 Gaming 3 Motherboard
- Samsung 850 Pro SSD 1TB
- Windows 10 Pro (Operating System)
- Visual Studio Pro 2013 (C# is used)
- Unity 3D 5.x (Handles visualization)
- Blender3D (Creation of the environments, and visual elements)

To aid in visualization as well as provide a 2D implementation of NavMeshes, the Unity game engine is used. By having it perform the graphics and visuals, as well as handle user interaction, and runtime logging, we can focus on development.

## 6.5 Challenges

The main challenge when we started was how to capture the volume which can be traversed. We had to determine an inside vs. outside space, as well as the volume set

out by the triangles of the environment. We had to test a bounded volume without checking for intersections and self-intersections continuously.

In the beginning memory was the largest challenge. To start, we used simple Dijkstra to get intuition and to better understand the problem. In 2D, a grid can use Dijkstra to find a path by forming a node network. It was easy to extend it into 3D using voxels. However, we quickly ran into memory constraints. Through each iteration of the program, optimizations were applied to reduce the memory foot print, as well as gain insights into the nature of the problem. The end result was our approach using sampling and NavVolumes. The sample grid uses a BitArray but could be changed to a sparse matrix to be more memory efficient.

# Chapter 7

# Results

In this section, we will present the results of the NavNet system in terms of quality of volume coverage, as well as quality of the generated path between the agent and a target point. We will outline our testing methodology and explain any justifications.

## 7.1 Evaluation

As mentioned at the beginning, there are no standardized benchmarks to evaluate the performance of a 3D pathfinding solution.

Normally one would be able to perform regression tests against a know test suit with known results. And all improvements could be judged against a known set of values. In order to do our own regression tests and be able to evaluate our ideas, the NavNet system was implemented as a set of components. These components could be compared against each other to test improvements between iterations. The NavNets can be seen as a movement controller component which is attached to an entity. The data gathering was automated as much as possible, however a human operator is still required to supervise the execution and at times intervene. Using the Unity3D game engine, we created a test bed which allowed us to quickly switch the input data for a test. The input data is comprised of a test world, a starting point location, a target point location, as well as desired resolution. Test points were selected for each world so that we would be able to create specific regression tests at several resolutions. The NavNets would be used by an entity which we called the 'hunter' to attempt to reach the target or 'prey'.

Each test world is analyzed to determine the total bounded volume which we would like to capture with the NavNets, as well as the worlds complexity. The complexity is determined via the number of triangles the world contains, but also their sizes, shape (i.e., is it a sliver, or an equilateral), as well as noise. A large plane comprised of millions of triangles is not very complex; however a mesh with the same number of triangles where each triangle has a different normal vector can be very complex.

The evaluation of the NavNet is done in two portions:

**Quality of the Volume:** To evaluate the quality of the sampled volumes the following metrics are logged and then evaluated:

- The run time to generate the NavVolumes and intersections.
- The number of base function calls.
- The number of NavVolumes.
- the volume of all NavVolumes (ensuring that overlapping regions are only counted once)

The quality of the NavVolumes will be determined by the NavNet volume compared to the analyzed volume for a test world. Additionally, the number of NavVolumes generated will also be taken into consideration.

**Quality of the Path:** To evaluate the quality of the generated path the following metrics are logged:

- The run time to generate the path.
- The path length.

Based on the logged metrics we calculate:

- The difference in value between the computed path using the NavNet, and the analyzed path. This is the absolute error.
- The relative size of the difference compared to the entire path.This is the relative error.

Better quality paths will have negligible absolute and relative error.

We believe that these metrics capture a good set of values for comparison.

## 7.2 Methodology

In order to gather reliable data between program iterations, the steps of data gathering need to be solidified. This data will be used both for performance evaluation and comparison between iterations.

Since there is a graphical aspect to the NavNet generation and certain geometric environments can be intuitively broken down by a human operator, for certain environments we will also qualitatively judge the quality of the produced NavNet. This check is solely meant as a human intuition check.

The evaluation will follow the following methodology:

For each test world:

1. Generate a set of start and end points.

2. For each start/end pair, compute the distance between them and save the value with the points when analyzing the test world. The distance is calculated using a brute-force algorithm.
3. Compute the volume of the space meant to be captured by the NavNet.[1] The values obtained will be considered the correct values.
4. Generate a NavNet.
   (a) Log the number of triangles.
   (b) Log the number of sample cells, (total and actually visited sampled since the majority will not be computed)
   (c) Log number of NavVolumes and intersections.
   (d) Log the runtime.
   (e) Calculate the total volume by the sum of all sample cells which do not intersect any triangle. This value represents the volume of the entire NavNet without overlap.
   (f) Calculate the difference in volume between the sum of the sample cells and the brute-force computed volume of the world.
5. For each start/end pair
   (a) Compute the path length using the NavNet.
   (b) Calculate the difference between the NavNet path and the brute-force path. Also, calculate the relative size of this difference when compared to the distance of the path.
   (c) Log the runtime necessary to compute a path.

## 7.3 Results

The NavNet system was tested using 5 test worlds. These are as following:

- Three Worlds based on video game levels: 3D Maze Escape, UT3[2]:DM-KBarge, UT3:DM-Morbias.
- Two Worlds based on buildings on the University Campus: ITB[3] and ABB[4].

[1] We use Blender 3D, an open source 3D modeling application, to compute the value to make use of its powerful tools.
[2] Unreal Tournament 3
[3] Information Technology Building
[4] AN Bourns Science Building

The results of generating a NavNet for each of the test worlds are summarized in Table 7.2. and Table 7.3. Worlds 1, 2, and 3 are based on video games, which try to have as few triangles as possible for faster rendering. Worlds 4 and 5 are based on blueprints from two buildings on the university campus.

It is also interesting to note that world 2 has the incorrect value for some sample cells. This was traced back to the actual sampler. At a finer resolution, this error does get corrected.

For all worlds, as the sample resolution decreases in value, we often come within 2% of the true volume. Although we did have to go as low as 0.02 for world 4 to achieve it. Only world 5 continued to have a margin of 9%, we believe that is due to the many small nooks in the blueprint which are not properly captured at these resolutions.

This is an interesting find, as it directly links the sampling resolution to the noise in the environment. The many smaller nooks can be looked as a high frequency signal. To reliably sample the environment, we need to sample at twice the highest frequency as outlined in Shannon's theorem.

As expected, as the resolution becomes finer, new regions become connected as detailed NavVolumes can be constructed. A small surprise was when we saw that at times the number of intersections do not increase as NavVolumes increase.

Another surprise was that at finer resolutions the space between walls also gets sampled and forms a disconnected NavVolume. Contrary to our intuition, it is not a problem. It is a result of the sampler checking for an intersection with the environment. It has no knowledge of how the sample grid cell exists in the world. Although these separate nets are still valid, an entity will not be able to utilize this NavNet.

Presumably a starting and end point will be located in the free space, and not at a location which is inside a wall. As such, these disconnected graphs will not be taken into consideration during pathfinding. Our algorithm is robust enough to handle this scenario gracefully. We have looked at using other techniques such as the parity test for raster graphics to differentiate which NavNets are to be used, however it was discovered that such a test had no noticeable impact.

We also found that in which way the sample grid aligned with the test world was important to a certain extend. In test world 5, a resolution of 1 and 0.5 produced the exact same breakdown of NavVolumes. When we looked closer, each sample cell at res = 1 was cut into 8 sub-cubes at res = 0.5. The two sample grids were perfectly aligned with each other. It also outlined an issue that at larger resolutions details could be missed because of how the sample grid aligns with the world. Figure 4.2. visually illustrates this concept.

When all the data was summarized, it also became evident that the number of NavVolumes tends to differ only slightly, and are nearly constant. We believe it is due to the NavVolumes capturing the general shape of the test world relatively early at larger resolutions, and as the resolution becomes finer, these NavVolumes are able to capture

more of the test world, however how the spaces break down into maximally convex regions tends to stay nearly identical.

Overall, though we are satisfied with the breakdown of the NavVolumes and the much lower number of NavVolumes and intersections compared to the triangle count or sample grid count for a given world.

Table 7.1. outlines algorithm operation during the sampling of the test world. It combines a naive brute force algorithm against our sampler with and without an octree for sample cell lookup. As expected, our sampler provides several orders of magnitude fewer operations.

| Method | Naive | MC | MCO |
|---|---|---|---|
| # of triangles accessed | 200,518 | 100 | 100 |
| # of trig/box check | 200,518 | 2,298 | 1,984 |
| # grid cell accessed | 134,724 | 928 | 815 |

Table 7.1: Sampling operation comparison between naive, marching cube, and marching cube with octree, world = 100 triangles, resolution = 0.5

| World | N | Sample Resolution (m) | $S_\#$ | $N_\#$ | $I_\#$ |
|---|---|---|---|---|---|
| 1 | 422 | 1 | 11,482 | 20 | 19 |
| | | 0.5 | 96,681 | 20 | 19 |
| | | 0.25 | 803,433 | 22 | 20 |
| | | 0.1 | 13,229,340 | 24 | 20 |
| 2 | 284 | 1 | 94,073 | 19 | 18 |
| | | 0.5 | 677,984 | 31 | 40 |
| | | 0.25 | 5,383,485 | 47 | 64 |
| | | 0.1 | 82,961,158 | 73 | 98 |
| 3 | 858 | 1 | 55,571 | 35 | 48 |
| | | 0.5 | 468,015 | 35 | 48 |
| | | 0.25 | 3,895,481 | 35 | 48 |
| | | 0.1 | 62,646,027 | 35 | 48 |
| 4 | 4860 | 1 | 10,520 | 74 | 12 |
| | | 0.5 | 83,040 | 98 | 134 |
| | | 0.25 | 698,991 | 98 | 134 |
| | | 0.1 | 12,134,413 | 123 | 135 |
| | | 0.05 | 100,634,410 | 126 | 143 |
| | | 0.02 | 1,613,401,901 | 126 | 147 |
| 5 | 10,548 | 1 | 10,235 | 191 | 19 |
| | | 0.5 | 81,880 | 191 | 19 |
| | | 0.25 | 706,212 | 195 | 241 |
| | | 0.1 | 12,173,839 | 195 | 241 |
| | | 0.05 | 100,731,058 | 204 | 257 |
| | | 0.02 | 1,613,603,519 | 207 | 258 |

Table 7.2: All runtimes were insignificant ($<$ 1 minute), W is the test world used, N is the number of triangles in the test world, $r$ is the sample resolution, $S_\#$ is the number of sample cells which do not intersect any triangle, $N_\#$ is the number of NavVolumes, $I_\#$ is the number of intersections

| World | Volume ($m^3$) | Sample Resolution (m) | Volume of $\cup V_i$ | $\text{Volume}_{error}$ |
|---|---|---|---|---|
| 1 | 13,408.66 | 1 | 11,482.00 | -14% |
| | | 0.5 | 12,085.13 | -10% |
| | | 0.25 | 12,553.64 | -6% |
| | | 0.1 | 13,229.34 | -1% |
| 2 | 83,332.33 | 1 | 94,073.00 | 13% |
| | | 0.5 | 84,748.00 | 2% |
| | | 0.25 | 84,116.95 | 1% |
| | | 0.1 | 82,961.16 | -0.45% |
| 3 | 64,000.00 | 1 | 55,571.00 | -13% |
| | | 0.5 | 58,501.88 | -9% |
| | | 0.25 | 60,866.89 | -5% |
| | | 0.1 | 62,646.03 | -2% |
| 4 | 13,141.11 | 1 | 10,520.00 | -20% |
| | | 0.5 | 10,380.00 | -21% |
| | | 0.25 | 10,921.73 | -17% |
| | | 0.1 | 12,134.41 | -8% |
| | | 0.05 | 12,579.30 | -4% |
| | | 0.02 | 12,907.22 | -2% |
| 5 | 14,217.80 | 1 | 10,235.00 | -28% |
| | | 0.5 | 10,235.00 | -28% |
| | | 0.25 | 11,034.56 | -22% |
| | | 0.1 | 12,173.84 | -14% |
| | | 0.05 | 12,591.38 | -11% |
| | | 0.02 | 12,908.83 | -9% |

Table 7.3: Volume Comparison: Volume of the test world as calculated in Blender 3D, $\cup V_i$ is the total volume of the sum of all sample cells which do not intersect any triangle, $\text{Volume}_{error}$ is the margin of error to the true volume of the test world. Negative numbers represent we are under the total, positive numbers mean we have too much volume

The results of the second portion of the evaluation are summarized in tables 7.4. to 7.6. Please note that not all data is presented in these tables, rather only a sliver of the data is shown to illustrate the general trends. To contrast the results of the NavNet generated paths, a Convex Hull path finder was used. It is capable of handling 3D environments, however the runtime was several magnitudes larger than the runtime of the NavNet, often times resulting in a single frame being rendered in several minutes. The NavNet path finder on the other side was able to output a steady 60fps throughout testing. As such, these runtimes have been removed as they do not add any meaningful insight.

In tables 7.4. to 7.6, we can observe that a general trend of the difference between the calculated path and the computed path becoming close to 0 as the sample resolution becomes finer. We can also observe that the same sample resolution produces a different

relative size differences among the test worlds. It further enforced that the environment geometry needs to be taken into consideration when determining the sample resolution.

Also in tables 7.4. to 7.6, we compare the NavNet paths to another method of finding a simple path in 3D, the convex hull path. Its paths appear usable, however are quickly out performed by the NavNet paths. Furthermore, its performance is directly linked to the geometry and number of triangles, providing no way of knowing how good a path actually is. It also is able to produce paths which are physically impossible to follow by the entity, which resulted in positive values in world 1.

Finally, table 7.7 outlines the summarized values for all paths for test world 5. Figure 7.1. visualizes this data and a clear asymptote can be observed. A trend line was calculated for the values to further outline the patter.

| World | Sample Resolution (m) | Convex Hull (m) | NavNet (m) | Actual (m) |
|---|---|---|---|---|
| 1 | 2.00E-01 | 6.386633 | 6.417246 | 6.395221 |
| | 1.25E-01 | 6.386633 | 6.411351 | 6.395221 |
| | 1.00E-01 | 6.386633 | 6.421031 | 6.395221 |
| 2 | 5.00E-01 | 13.18393 | 13.0344 | 12.8994 |
| | 2.50E-01 | 13.18393 | 13.13478 | 12.8994 |
| 3 | 5.00E-01 | 22.87502 | 23.8884 | 22.6945 |
| | 1.25E-01 | 22.87502 | 22.76432 | 22.6945 |
| 4 | 7.50E-01 | 65.64091 | 63.4 | 61.263 |
| | 5.00E-01 | 65.64091 | 63.995 | 61.263 |
| | 2.50E-01 | 65.64091 | 63.647 | 61.263 |
| 5 | 5.00E-01 | 14.888 | 15.02492 | 14.557 |
| | 2.50E-01 | 14.888 | 14.8243 | 14.557 |
| | 1.25E-01 | 14.888 | 14.6379 | 14.557 |
| | 6.25E-02 | 14.888 | 14.558 | 14.557 |

Table 7.4: Path length comparison: A snippet of a single start/end pair for each world. The true path length, as well as a fast approximation, and the path produced from the NavNet

| World | Sample Resolution (m) | Convex Hull Error (m) | NavNet Error (m) |
|---|---|---|---|
| 1 | 2.00E-01 | 8.59E-03 | -2.20E-02 |
| | 1.25E-01 | 8.59E-03 | -1.61E-02 |
| | 1.00E-01 | 8.59E-03 | -2.58E-02 |
| 2 | 5.00E-01 | -2.85E-01 | -1.35E-01 |
| | 2.50E-01 | -2.85E-01 | -2.35E-01 |
| 3 | 5.00E-01 | -1.81E-01 | -1.19E+00 |
| | 1.25E-01 | -1.81E-01 | -6.98E-02 |
| 4 | 7.50E-01 | -4.38E+00 | -2.14E+00 |
| | 5.00E-01 | -4.38E+00 | -2.73E+00 |
| | 2.50E-01 | -4.38E+00 | -2.38E+00 |
| 5 | 5.00E-01 | -3.31E-01 | -4.68E-01 |
| | 2.50E-01 | -3.31E-01 | -2.67E-01 |
| | 1.25E-01 | -3.31E-01 | -8.09E-02 |
| | 6.25E-02 | -3.31E-01 | -1.00E-03 |

Table 7.5: Error comparison: A snippet of a single start/end pair for each world. This table calculates the difference between the computed path and the calculated path at various sample resolutions.

| World | Sample Resolution (m) | Convex Hull Error (%) | NavNet Error (%) |
|---|---|---|---|
| 1 | 2.00E-01 | 0.13% | -0.34% |
| | 1.25E-01 | 0.13% | -0.25% |
| | 1.00E-01 | 0.13% | -0.40% |
| 2 | 5.00E-01 | -2.21% | -1.05% |
| | 2.50E-01 | -2.21% | -1.82% |
| 3 | 5.00E-01 | -0.80% | -5.26% |
| | 1.25E-01 | -0.80% | -0.31% |
| 4 | 7.50E-01 | -7.15% | -3.49% |
| | 5.00E-01 | -7.15% | -4.46% |
| | 2.50E-01 | -7.15% | -3.89% |
| 5 | 5.00E-01 | -2.27% | -3.21% |
| | 2.50E-01 | -2.27% | -1.84% |
| | 1.25E-01 | -2.27% | -0.56% |
| | 6.25E-02 | -2.27% | -0.01% |

Table 7.6: Error comparison: A snippet of a single start/end pair for each world. This table calculates the relative size difference of the calculated path and the computed path at various sample resolutions.

| | Sampling Resolution (m) | | | |
|---|---|---|---|---|
| | 0.5 | 0.25 | 0.125 | 0.06125 |
| Max | 7.415% | 4.145% | 0.5913% | 0.012% |
| Avg | 3.133% | 1.988% | 0.4988% | 0.008% |
| Min | 0.694% | 0.34% | 0.4553% | 0.005% |

Table 7.7: Summarized Data for all paths of test world 5 with absolute value of relative error.

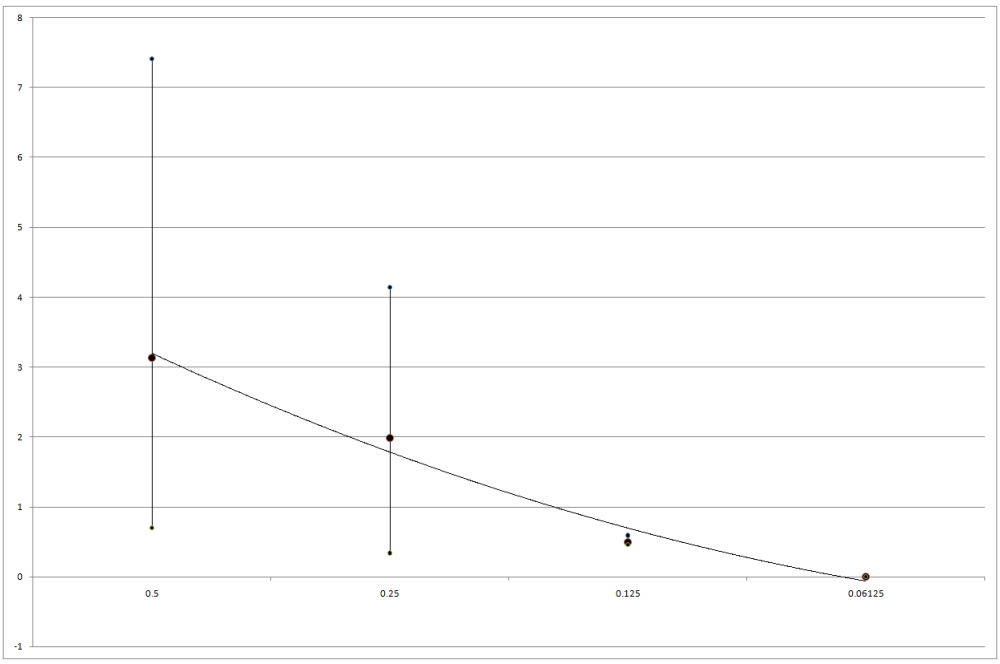


Figure 7.1: Edge Selection

# Chapter 8

# Conclusion

The purpose of this paper was to present a new way of finding a path in a 3D environment where motion is not constricted to a 2D plane. Although pathfinding is a well understood problem in 2D Rabin 2017, that is not the case with complex pathfinding in 3D. Often it has relied on casting the problem as a 2D problem if possible. Otherwise, brute-force approaches had to be utilized, time complexity works against these algorithms due to the explosion of the search space. We have presented NavNets and their use in finding a path in 3D. As it stands we are satisfied with the results. We are able to significantly simplify the search space and efficiently plan a path through the world (compared to any other method we know off). We believe it is a better approach over using the triangle data directly, as the NavNet captures enough useful information about the world and reduces the problem size. Our method produces a breakdown of the world into maximally convex regions.

This work contributes the following to the area of 3D pathfinding:

- Review of the current literature with respect to pathfinding, and identifying why 3D pathfinding is a problem.
- Where the current state of the art has shortcomings and strengths.
- A clear theoretical framework which is used to describe the structures and algorithms developed to solve the problem. As well, as to analyze them.
- A system to solve the problem of 3D pathfinding in a no-gravity environment.
- Implementation of several prototypes used to better understand the problem and calculate a solution. A final system which solves the problem of 3D pathfinding, and numerous tools used to read the intermediate data and provide visualization of the data and solution.

Navigation Meshes have become a standard tool when pathfinding needs to be applied to a world where motion is confined to a 2D plane. It is our hope that NavNets will become similar synonymous when 3D pathfinding is required. NavNets reduce the search space allowing run time pathfinding for most simulations and video games.

We have demonstrated that NavNets are a viable method because they give a significant reduction in complexity.

NavNets are our solution to the problem of unrestricted 3D pathfinding.

## 8.1 Future Work

During the development of NavNets, both the theoretical approach and the implementation new ideas emerged. Some of these ideas have been implemented and solved a specific problem, however a great many had to be shelved due to time constraints as well as focus constraints. These ideas however are excellent starting points for a next step for NavNets or even how to further improve them. What follows are ideas or concepts for future work with NavNets:

- Resolution calculation: The sample resolution has to be selected carefully to some extend. In the future we would like to analyze the environment and determine the best sampling resolution to prevent loss of information of the environment as outlined in Figure 4.2.
- Similar, can we calculate the minimum sample resolution to sample without loss of details based on an environments highest detail frequency. Can we determine detail frequency?
- Ability to find the best rotation to align the sample grid with the world to reduce the amount of missed details.
- Comparison for 2D systems: How do NavNets compare to NavMeshes for environments where the entity is confined to a 2D plane of motion. Are we able to use NavNets?

# Appendix A

# Chapter 1 Supplement

## A1 Appendix A: Generating NavNet

We illustrate a simple example of generating a NavNet for a simple world.

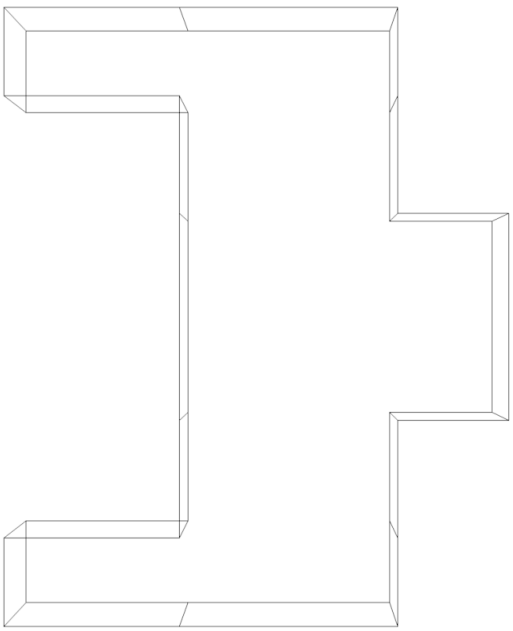

FIGURE A1.1: A simple 3D world.

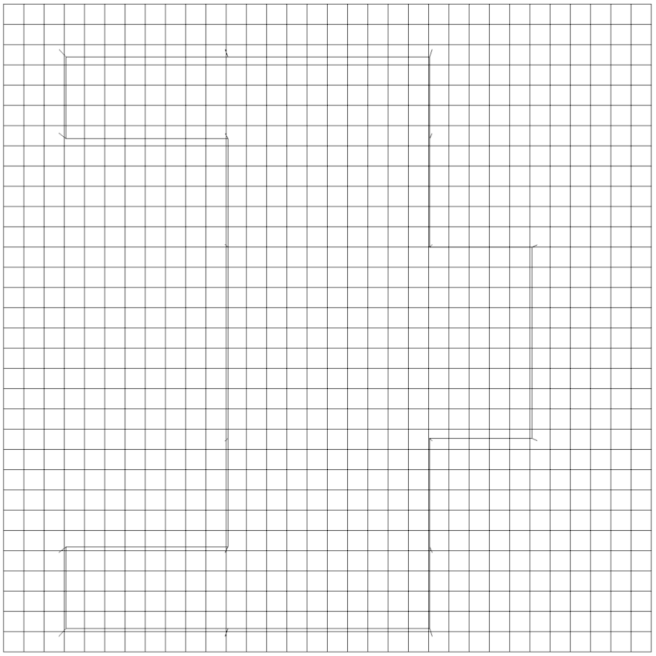

Figure A1.2: A 2D slice of the world taken and the sample grid.

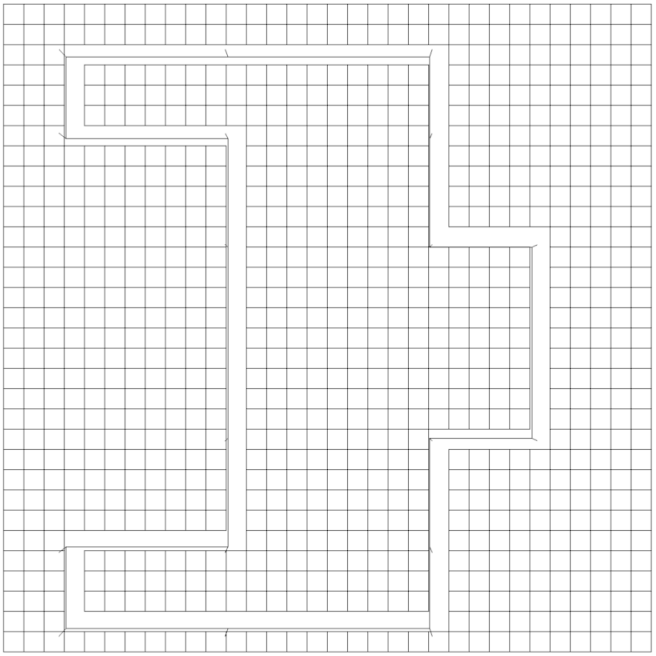

Figure A1.3: Grid cells which intersect the world have been removed for illustration purposes. It represents the Free Space.

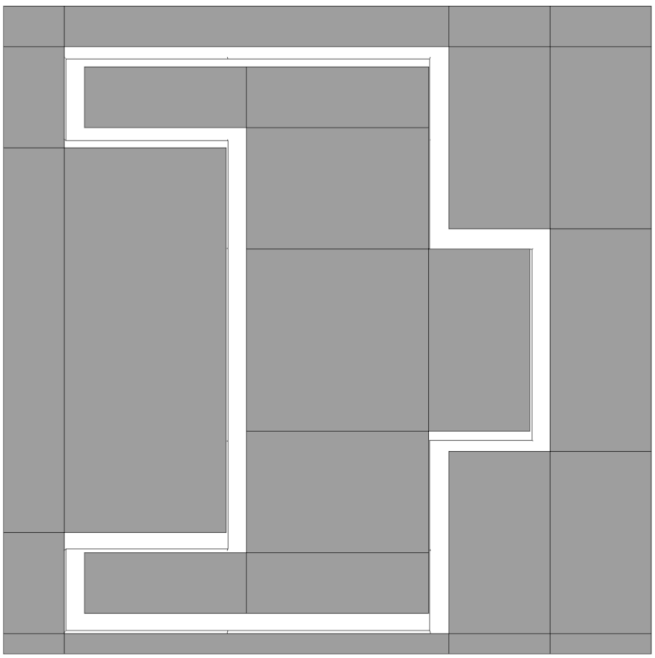

Figure A1.4: All NavVolumes having been calculated.

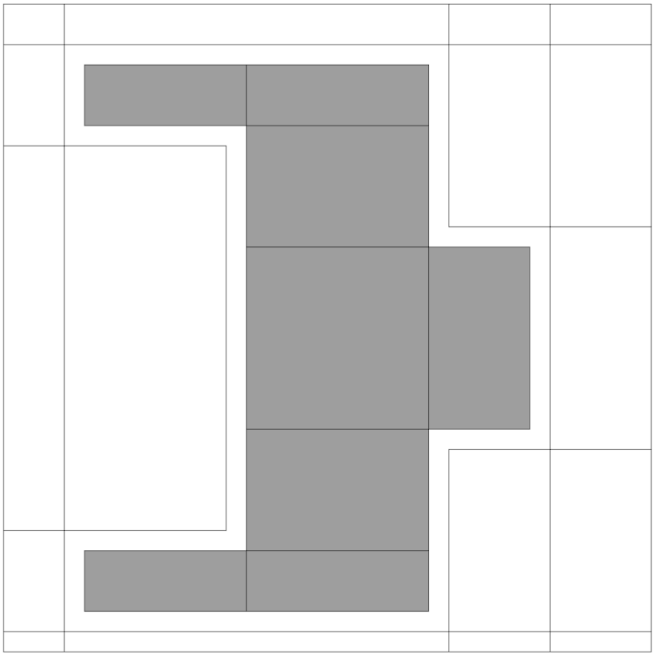

Figure A1.5: Focus on the interior portion.

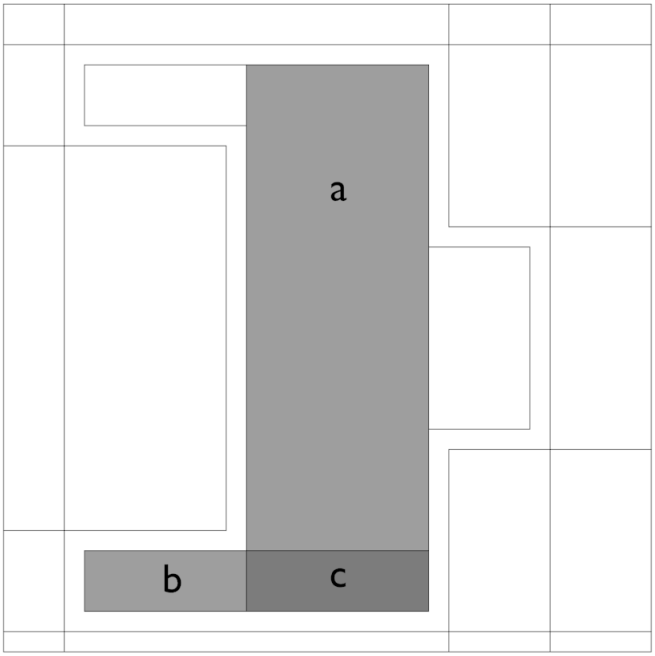


Figure A1.6: In the interior portion, two NavVolumes are highlighted (a,c) and (b,c). They intersect in portion c.

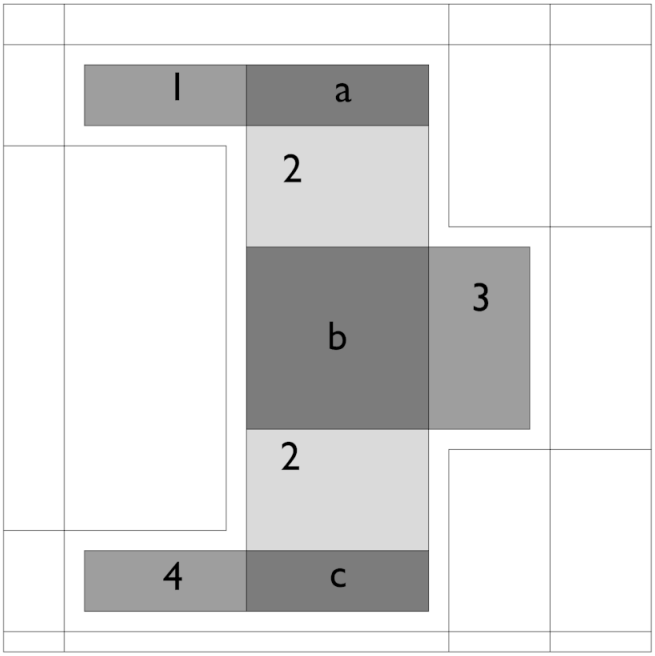


Figure A1.7: The interior portion has 4 NavVolumes and 3 intersections (a,b,c).

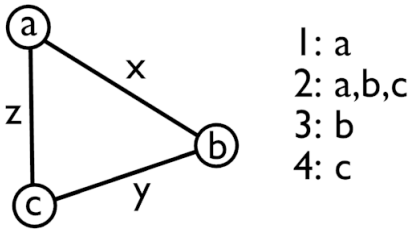


Figure A1.8: The NavNet is formed, intersections a,b,c are the nodes and we calculate the Cartesian distance between their center points. During implementation, we have a support structure which keeps track of which intersections each NavVolume has.

## A2 Appendix B: Using NavNet

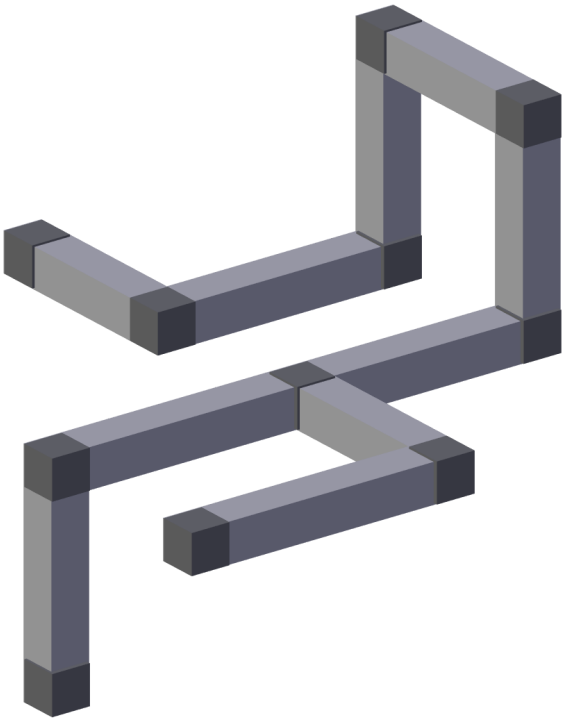

Figure A1.9: A simple NavNet with intersections highlighted. Original environment has been removed for illustration purposes.

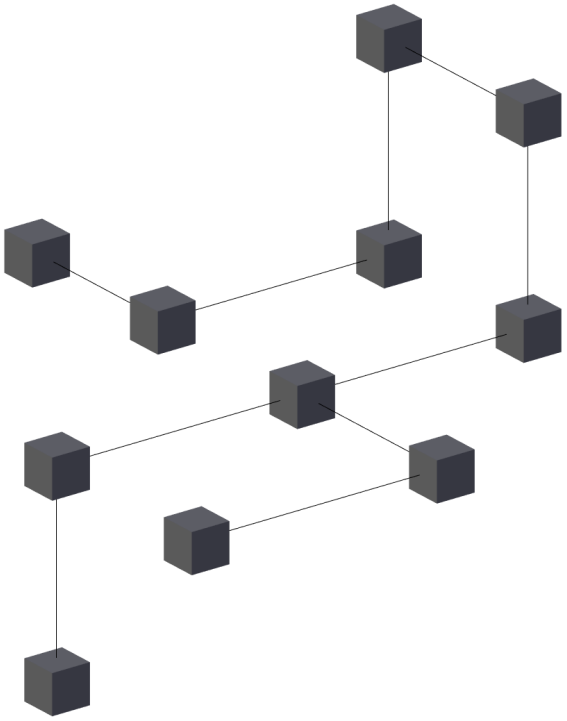

Figure A1.10: NavVolumes have been removed to further illustrate how the NavNet uses the intersections as nodes. Visualization lines have been added to show how the intersections, as nodes in the NavNet, are connected and form the graph.

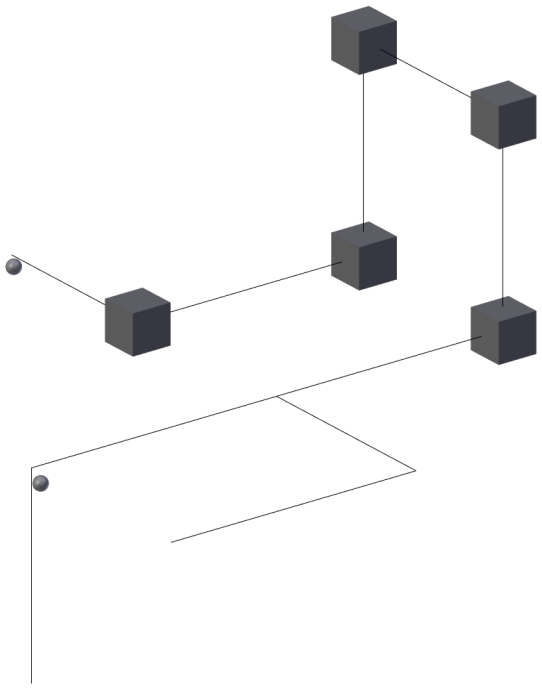

Figure A1.11: A volume path was calculated and only intersections which are part of it are illustrated.

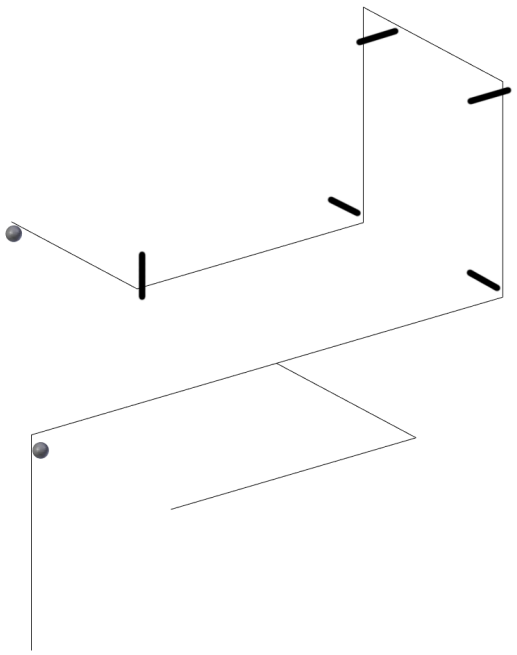

Figure A1.12: Using the NavVolumes and intersections, the folding edges are selected for further path refinement.

Figure A1.13: Visualization of all possible paths using the selected folding edges.

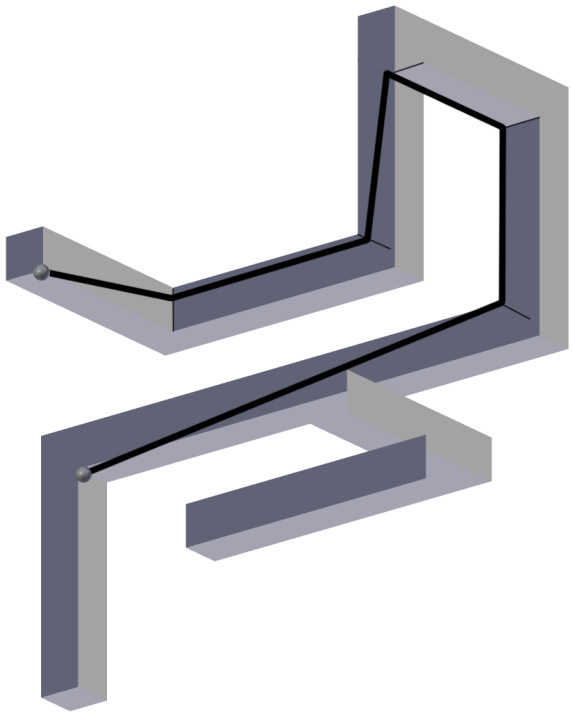

Figure A1.14: After solving a minimization problem using dynamic programming, intersection points on each folding edge are known. Along with the start point and the target point, these intersection points form the path.

# Bibliography

(2011). In: *Motion in Games*.

Akenine-Moeller, T. (Mar. 2001). *Fast 3D Triangle-Box Overlap Testing*. Tech. rep. Updated June. Chalmers University of Technology.

Alt, G., Dill, K., Isla, D., Orkin, J., Russel, A., Tozour, P., and Zubek, R. (July 2008). *Fixing Pathfinding Once and For All*. Game AI Blog.

Amato, N. M. (2010). Toward Simulating Realistic Pursuit-Evasion Using a Roadmap-Based Approach. In: *Motion in Games*.

Amitava Chatterjee Anjan Rakshit, N. N. S. (2013). *Vision Based Autonomous Robot Navigation*. Springer.

Basten, B. J. van and Egges, A. (2009). Path Abstraction for Combined Navigation and Animation. In: *Motion in Games*.

Bespamyatnikh, S. (2003). Computing homotopic shortest paths in the plane. In: *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 609–617.

Borenstein, I. U. J. (May 1998). VFH+: Reliable Obstacle Avoidance for Fast Mobile Robots. In: *IEEE International Conference on Robotics and Automation*. IEEE, 1572–1577.

Brady, M., Jung, K., Nguyen, H., and Nguyen, T. (July 1998). Interactive volume navigation. *Visualization and Computer Graphics, IEEE Transactions on* 4(3), 243–256.

Brand, S. and Bidarra, R. (2011). Parallel Ripple Search – Scalable and Efficient Pathfinding for Multi-core Architectures. In: *Motion in Games*.

Bredon, G. E. (1993). *Topology and geometry*. New York: Springer-Verlag.

Brewer, D. (Mar. 2015). *Getting of the NavMesh*.

Chazelle, B. (Aug. 1984). Convex partitions of polyhedra: A lower bound and worst-case optimal algorithm. *Society for Industrial and Applied Mathematics* 13(3), 488–507.

Cooper, J. L. and Ballard, D. (2012). Realtime, Physics-Based Marker Following. In: *Motion in Games*.

Cuesta, F., Ollero, A., Arrue, B. C., and Braunstingl, R. (2003). Intelligent control of nonholonomic mobile robots with fuzzy perception. *Fuzzy Sets and Systems* 134(1), 47–64.

Cuesta, F. and Ollero, A. (2005). Bifurcations in Simple Takagi-Sugeno Fuzzy Systems. In: *Intelligent Mobile Robot Navigation*. Springer, 51–77.

Curtis, F. E., Gould, N. I., Robinson, D. P., and Toint, P. L. (2013). An interior-point trust-funnel algorithm for nonlinear optimization. *submitted for publication*.

Demyen, D. J. (2007). Efficient Triangulation-Based Pathfinding. MA thesis. University of Alberta.

*Does local convexity imply global convexity?* (May 2012). Math Stack Exchange.

E. Lee, P. V. (n.d.). *Structure and Interpretation of Signals and Systems*. Addison Wesley. ISBN: 0-201-74551-8.

Finney, K. C. (2005). *Advanced 3D Game Programming: All in one*. Thomson Course Technology.

Following a Group of Targets in Large Environments (2012). In: *Motion in Games*.

*Funnel Algorithm* (2010). Lecture Notes.

*Game Physics Cookbook* (2017). Packt.

Geraerts, R. (2010). Planning Shrot Paths with Clearance using Explicit Corridors. In: *IEEE International Conference on Robotics and Automation*.

Hernández-Reindel, A. (2012). *Visually Realistic Tunneling of Volumetric Terrain in Real-Time*. Kongens Lyngby.

(2014).

*Introduction to Algorithms* (2009). 3rd. MIT Press.

Iwan Ulrich, J. B. (1998). VFH+: Reliable Obstacle Avoidance for Fast Mobile Robots. In: *IEEE International Conference on robotics and Automation*.

Jorgensen, C.-J. and Lamarche, F. (2011). From Geometry to Spatial Reasoning: Automatic Structuring of 3D Virtual Environments. In: *Motion in Games*.

Kallmann, M. (2010a). Navigation Queries from Triangular Meshes. In: *Third International Conference on Motion in Games*.

Kallmann, M. (2010b). Navigation Queries from Triangular Meshes. In: *Motion in Games*.

Kallmann, M. (2010c). Shortest Paths with Arbitrary Clearance from Navigation Meshes. In: *Eurographics/SIGGRAPH Symposium on Computer Animation*.

Kaplansky, I. (1972). *Set Theory and Metric Spaces*. Ed. by A. M. Society. AMS Chealsea Publishing.

Karamouzas, I., Geraerts, R., and Overmars, M. (2009). Indicative routes for path planning and crowd simulation. In: *Proceedings of the 4th International Conference on Foundations of Digital Games*. ACM, 113–120.

Kaufman, A. and Shimony, E. (1987). 3D Scan-conversion Algorithms for Voxel-based Graphics. In: *Proceedings of the 1986 Workshop on Interactive 3D Graphics*. I3D '86. Chapel Hill, North Carolina, USA: ACM, 45–75. ISBN: 0-89791-228-4.

Kelly, J. L. (1975). *General topology*. New York: Springer-Verlag.

Kim, H., Yu, K., and Kim, J. (2011). Reducing the search space for pathfinding in navigation meshes by using visibility tests. *J. Electr. Eng. Technol* 6(6), 867–873.

Kimmel, R. and Sethian, J. A. (1998). Computing geodesic paths on manifolds. *Proceedings of the National Academy of Sciences* 95(15), 8431–8435.

Kitamura, Y., Tanaka, T., Kishino, F., and Yachida, M. (Aug. 1995). 3-D path planning in a dynamic enviroment using an octree and an artificial potential field. In: *Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots', Proceedings. 1995 IEEE/RSJ International Conference on*. Vol. 2. IEEE, 474–481.

Klee, V. (1966). Paths on polyhedra. II. *Pacific Journal of Mathematics* 17(2), 249–262.

Kleinberg, J. (2005). An approximation algorithm for the disjoint paths problem in even-degree planar graphs. In: *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*. IEEE, 627–636.

Kolb, C. R. S. A. (Nov. 2005). *A Vertex Program for Efficient Box-Plane Intersection*. Tech. rep. University of Siegen, Germany.

Kruszewski, P. A. (2005). A game-based cots system for simulating intelligent 3d agents. In: *BRIMS'05: Proceedings of the 2005 behavior representation in modelling and simulation conference*.

Lau, M. and Kuffner, J. (2010). Scalable Precomputed Search Trees. In: *Motion in Games*.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

Lee, S., Han, J., and Lee, H. (2006). Straightest paths on meshes by cutting planes. In: *Geometric Modeling and Processing-GMP 2006*. Springer, 609–615.

Lengyel, E. (2004). *Mathematics for 3D Game Programming and Computer Graphics*. Second Edition. Charles River Media.

Li, F., Klette, R., and Morales, S. (2009). An approximate algorithm for solving shortest path problems for mobile robots or driver assistance. In: *Intelligent Vehicles Symposium, 2009 IEEE*. IEEE, 42–47.

Lipson, H. and Shpitalni, M. (Apr. 1996). Decomposition of a 2D polygon into a minimal set of disjoint primitives. In: *CSG96 Conference on Set-Theoretic Solid Modeling*. Winchester, UK, 65–82.

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. In: *Third International Conference on Motion in Games*. ACM Computer Graphics.

Luan, A. (n.d.). Shortest Path Approximation on Triangulated Meshes.

MacIver, D. (July 2004). *Filters in Analysis and Topology*. Tech. rep. Eris Free Network.

Manocha, D. (2008). Real-Time Path Planning and Navigation for Multi-agent and Crowd Simulations. In: *Motion in Games*.

Menard, M. (2012). *Game Development with Unity*. Course Technology CENGAGE Learning.

Milford, M. J. (2008). *Robot Navigation from Nature*. Springer.

Möller, T. and Trumbore, B. (1997). Fast, Minimum Storage Ray-Triangle Intersection. *Journal of Graphics Tools* 2(1), 21–28.

Mononen, M. (Mar. 2009). *RecastNavigation*.

*Navigation Mesh Reference* (2012). Unreal Engine 3. Epic Games.

Oleg Alexandrov, J. W. (n.d.). *Separating Axis vs Separating Line in 2D, including Projected Intervals*.

Oliva, R. and Pelechano, N. (2011). Automatic Generation of Suboptimal NavMeshes. In: *Motion in Games, Forth International Conference*. Springer.

Palmer, G. (2005). *Physics for Game Programmers*. APRESS.

Path-Planning for RTS Games Based on Potential Fields (2010). In: *Motion in Games*.

Pelfrey, B. (Jan. 2013). *Coding a Simple Octree*.

Peng, Z. D. W. C. H. B. H. Z. Q. (year?). *Real-time Voxelization for Complex Models*. Tech. rep. Zhejiang University, Hangzhou, China.

Plaki, E. (2012). Motion Planning with Discrete Abstractions and Physics-Based Game Engines. In: *Motion in Games*.

Porton, V. (2012). Filters on POSETS and Generalizations. *International Journal of Pure and Applied Mathematics* 74(1), 55–119.

Rabin, S. (2017). *Game AI Pro 3: Collected Wisdom of Game AI Professionals*. CRC Press. ISBN: 9781351647748.

Ramon Oliva, N. P. (2011). Automatic Generation of Suboptimal NavMeshes. In: *Motion in Games*.

Ren'e van den Berg, J. M. R. and Bidarra, R. (2009). Collision Avoidance between Avatars of Real and Virtual Individuals. In: *Motion in Games*.

Rodriguez, S. and Amato, N. M. (2011). Roadmap-Based Level Clearing of Buildings. In: *Motion in Games*.

Schreiber, Y. (2007). Euclidean shortest paths on polyhedra in three dimensions. PhD thesis. Tel Aviv University.

Seemann, D. M. B. G. (2004). *AI for Game Developers*. O'Reilly Media.

Shi, X. C. H. (Oct. 2011). Direction Oriented Pathfinding in Video Games. *IJAIA International Journal of Artificial Intelligence and Applications* 2(4).

Shi, X. C. H. (Dec. 2012). An Overview of Pathfinding in Navigation Mesh. *IJCSNS International Journal of Computer Science and Network Security* 12(12), 48–51.

Snook, G. (2000). Simplified 3D Movement and Pathfinding Using Navigation Meshes.

Stachniss, C. and Burgard, W. (2014). Particle Filters for Robot Navigation. *Foundations and Trends in Robotics* 3(4), 211–282.

Stein, T. H. C. C. E. L. R. L. R. C. (2009). *Introduction to Algorithms*. Third Edition. MIT Press.

Stentz, A. (1994). Optimal and efficient path planning for partially-known environments. In: *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*. IEEE, 3310–3317.

Sturtevant, N. R. (2012). Moving Path Planning Forward. In: *Motion in Games*.

Surazhsky, V., Surazhsky, T., Kirsanov, D., Gortler, S. J., and Hoppe, H. (2005). Fast exact and approximate geodesics on meshes. In: *ACM transactions on graphics (TOG)*. Vol. 24. 3. ACM, 553–560.

Tamassia, O. D. G. L. F. P. P. R. (Oct. 1998). *Checking the convexity of polytopes and the planarity of subdivisions*. Tech. rep. 3527. Institut national de recherche en informatique et en automatique.

Tarapata, Z. and Wroclawski, S. (2010). Subgraphs Generating Algorithm for Obtaining Set of Node-Disjoint Paths in Terrain-Based Mesh Graphs. In: *Motion in Games*.

Thomas Lopez, F. L. and Li, T.-Y. (2011). Space-Time Planning in Dynamic Environments with Unknown Evolution. In: *Motion in Games*.

Toll, W. van, Cook, A., and Geraerts, R. (Sept. 2011). Navigation meshes for realistic multi-layered enviroments. In: *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 3526–3532.

Van Toll, W., Cook, A. F., and Geraerts, R. (2011). Navigation meshes for realistic multi-layered environments. In: *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 3526–3532.

Walle, R. V. de (2011). Hybrid Path Planning for Massive Crowd Simulation on the GPU. In: *Motion in Games*.

Wash, C. (Jan. 2010). *Eliminate Branching (IF Statements) to Produce Better Code*. CapTech.

Willard, S. (1970). *General topology*. Addison-Wesley Publishing.

Williams, A., Barrus, S., Morley, R. K., and Shirley, P. (2005). An Efficient and Robust Ray-box Intersection Algorithm. In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH '05. Los Angeles, California: ACM.

Wong, K. Y. and Loscos, C. (2008). Hierarchical Path Planning for Virtual Crowds. In: *Motion in Games*.

Wouter G. van Toll, A. F. C. I. and Geraerts, R. (2012). A navigation mesh for dynamic environments. *COMPUTER ANIMATION AND VIRTUAL WORLDS* 23, 535–546.