Efficient obstacle avoidance using autonomously generated navigation meshes

Master thesis

By Sandy Brand

2009

TU Delft Delft University of Technology

Abstract

With the increasing demand for ever more depth and detail of modern video games, developers are faced with the problem of how to create and manage large amounts of content. One aspect of this is how to cheaply enable game entities to travel through their virtual worlds in a natural and realistic fashion. Game map sizes and complexity have however risen to such an extent that there is a strong demand for automization in order to relieve artists and designers from their medial tasks, and enable them to focus more on the creative aspect of game design. As a solution to this, we introduce systems for the autonomous generation of precomputed Navigation Meshes (NavMeshes) and their in-game application. These meshes contain abstractions of all walkable surfaces of a static map environment in the form of a set of convex areas and a matching graph topology. We discuss the pros and cons of generating them solely from map collision volumes using a 'lightweight' form of Boundary Representation (B-rep or BREP) algorithm, to help remove areas that cannot be reached due to the dimension of the traveling objects. This B-rep approach provides compact yet accurate NavMeshes abstractions that are ideal for classic path-finding algorithms. Natural movement around dynamic obstacles is achieved using a combination of 'fuzzy' whiskers sensory systems, and a 'deterministic' fall-back mechanism that temporarily enhances the resolution of the NavMesh graph locally. Further speed-ups are obtained by parallelizing classic A* algorithms for nowadays common multi-core architectures. We introduce the 'Parallel Bidirectional Search' that significantly outperforms traditional A* implementations.

Preface

This thesis contains the result of research in the field of real-time path-finding in video-games. The project was commissioned by both the Delft University of Technology and Triumph Studios as part of a graduation project for a masters degree in Computer Science. Most of the research and project execution has been performed at Triumph Studios.

During my full time employment the last few years I have acquired a great deal of experience that I find invaluable. My research was commissioned during the development of Overlord 1 and 2 by Triumph Studios in an attempt to improve their implementation of various path-finding techniques for future projects. Overlord 2 was released in June 2009 and was well received by the international gaming press (as was Overlord 1). I am very happy to have been able to take part in the conception of the Overlord series. Hopefully, some of my work in this thesis will help form and shape some of the future technologies we will need for projects to come.

I hereby thank Triumph Studios for making this research possible. In particular I'd like to thank Arno van Wingerden for his help and guidance. Further thanks go to my mentors Rafael Bidarra and Wim Bronsvoort at Delft University of Technology.

Index

1 Introduction

My work has been performed over a 3 year period during my full-time employment at Triumph Studios, a Dutch game development studio based in Delft, the Netherlands. During my employment I have also been closely working on their Overlord and Overlord 2 projects, which has given me new experiences and insights on how to tackle the many problems of game development. The actual time spent on my graduation project is probably half: roughly 1 and a half years.

1.1 Problem definition

Although video games are relatively 'young', we have already seen great developments and advancements in their complexity. This can be largely attributed to the increment in computing power. With it we have seen great improvements in their degree of 'realism'. Although the focus for this has long been on increasing the games' 'fidelities' such as graphics and sounds, we can see that recently efforts have also been shifting towards improvements of their 'artificial intelligence'.

A factor of providing better 'realism' in games is how game objects, such as characters and vehicles, move through these virtual worlds. There currently is a strong need for algorithms that can closely mimic the inherent 'fuzziness' of how creatures traverse their world, while still pertaining some degree of 'determinism'. From an engineering perspective, modern day games are astonishingly complex and require a lot of team effort to complete successfully. It is thus no wonder that developers are looking for algorithms that are still very deterministic in their nature because this will make them a lot easier to implement, verify and debug (reproducibility is key here). Also, from a game designer's stand point, it is important that character movement can be predicted to a certain degree so that they can create specific scenes and game-play mechanisms.

So the main focus of my research has been to find practical approaches for natural character movement through virtual worlds. This encompasses a number of aspects: path-finding, dynamic obstacle avoidance and group behavior. An interesting question here is how we can take advantage of the recent introduction of multi-core hardware architectures such as those on Microsoft's XBox 360 and Sony's Play Station 3 consoles. If it were possible to gain some performance boosts then this would allow us to free up processing time for other task (which again might involve more realistic/detailed behavior models for characters on other levels).

As an extra dimensions to all these problems, we should keep in mind what kind of impact the application of the related technology will have for all other contributors of the game development process as a whole. We should strive to keep the development cycles for level designers and graphics artist streamlined and short. This means we cannot rely on them to manually add large quantities of meta data that will be needed for our chosen algorithms, this would just be too costly and time consuming.

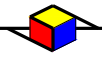
1.2 Goals

Following from this general problem description we can establish a number of concrete goals:

- Develop flexible, fast and reliable techniques for natural movement of groups of dynamically moving characters through virtual worlds.
- Obtain a suitable data format for storing navigation related meta data with a low memory footprint to accommodate for restricted available memory and streaming latency of the current generation of game consoles (so no more than 10 MB preferably).
- Determine how we can utilize current day multi-core architectures to speed up the process of path-finding (a 20% reduction would already make the extra effort worthwhile).
- The introduced techniques should reduce manual labor significantly, and with it chances on errors, for artists and level designers.

For fulfilling these objectives we can use Triumph Studios current state of technologies as a guideline for overall performance and ease of use. This will enable us to establish meaningful benchmarks and identify potential strengths and weakness in our new approaches. So the primary criteria are low memory consumption and CPU usage, notably for modern day consoles such as the Xbox 360 and PS3. Secondary criteria are of a more practical nature: ease of implementation, maintainability and user-friendliness for those who have to actually work with the game development tools.

All of these questions and issues will be addressed in this thesis. In it you will find both theoretical foundations of the methods I have been using, as well as discussions of the results I have obtained. We will start off in Chapter 2 with a summary of the research I have done in order to orientate myself into currently popular approaches to these problems (also see [3]). Here I will summarize the results I obtained and how these have influenced my design choices for the actual software implementations. Chapter 3 will begin with giving a description of a world representation format called 'NavMeshes'. It will deal with the advantages of this format and also show some examples which are the result of fully automatically generating them. Some of the difficulties I ran into in doing so will be discussed in Chapter 4, as well as how to implement future improvements of the quality of the NavMeshes. Chapter 5 then goes on to explain how to utilize the NavMeshes in-game and make a single objects safely travel through the static scenery of virtual worlds. Then, in Chapter 6, we will expand on this and add systems that can avoid collisions with dynamically moving objects. Having achieved the basics of natural travel motions we then take a step back in Chapter 7 and take a look on how to improve the actual path-finding algorithms via parallelization. Chapter 8 has been devoted to evaluate all the obtained results by doing benchmark comparisons with some of the systems that are currently used by Triumph Studios. In conclusion, Chapter 9 will give an overview and final discussion on all results that have been obtained. And finally, Appendix A will delve deeper into the more technical details of the NavMesh generator.



2 Research

My project started with a study of existing and popular approaches that are utilized in modern day video games (also see [3]). After extensive research I came to the conclusion that for natural movement of objects through a virtual world a number of different techniques have to be combined.

2.1 Static world representation

The first and most important step is that we need a good simplified representation of the world we want to travel through. The reason we need to simplify is because it is just too expensive to make travellers fully aware of all details in the world, this would cost too much processing power and too much memory. So we begin the process by constructing a simplification of the static world that only contains information that is actually relevant for a traveller. We can use this information to determine how it can move safely from a point A to point B using a path-finder module in an economic way. In general there are roughly 3 distinct approaches for creating such abstractions:

1. Rely on map designers to manually input a so called 'Path Lattice' that defines the 'accessible areas' in the map.
2. Use a 'discrete' representation of the world using a uniform grid that tell us where where walkable areas are.
3. Generate a simplified but accurate representation of all the traversable geometry, so called 'Navigation Meshes'.

The very first option is the easiest to implement: we only need to let map designers define way-points on the map which we will interconnect to form the path lattice (sometimes also known as ‘points of visibility’ or ‘meadow maps’, see [17]). The actual connecting could be automated by ray-casting between neighboring way-points to see if they should be connected or not, this way we can peel away connections that would otherwise run through walls. However, the simplicity of a path lattice is also its great weakness and there are a significant number of disadvantages. The first and foremost is that this approach does not scale well. For small maps it might still be doable but for medium to large maps the amount of markers that we would require becomes excessive. Also, human input is prone to errors which could result in severe ‘bugs’ or mistakes that are exploitable by a player. The level designer might accidentally add paths through a doorway that is actually too narrow for a specific game unit to pass through (this can also occur when our line-of-sight test is not robust enough because we want to reduce its overall execution costs). Finding these errors is not only cumbersome but also very fuzzy: it is impossible for any level designer to test if all conceivable paths are valid (as in: do not run through too narrow areas, etc.) The quality of the paths that a path-finder will generate using the lattice is also directly proportional to the number of markers that have been placed. If this number is too low, game units will walk through the terrain in a very ‘unrealistic’ fashion such as not cutting corners or ‘zigzagging’ for no apparent reason. Thus in general a level designer will have to place many more ‘sub-markers’ between various key areas in an attempt to suppress such anomalies. Ultimately, this large amount of markers will lead into a ‘combinatorial explosion’ because we have to test for each marker if it can ‘see’ any of the other markers. This means that our conversion tool will need to do increasingly more peeling before the final lattice has been obtained. Also note that the connectivity of a path lattice can fluctuate per node and can be far higher than strictly needed; this effect will be most notable at junctions (also see Figure 1).

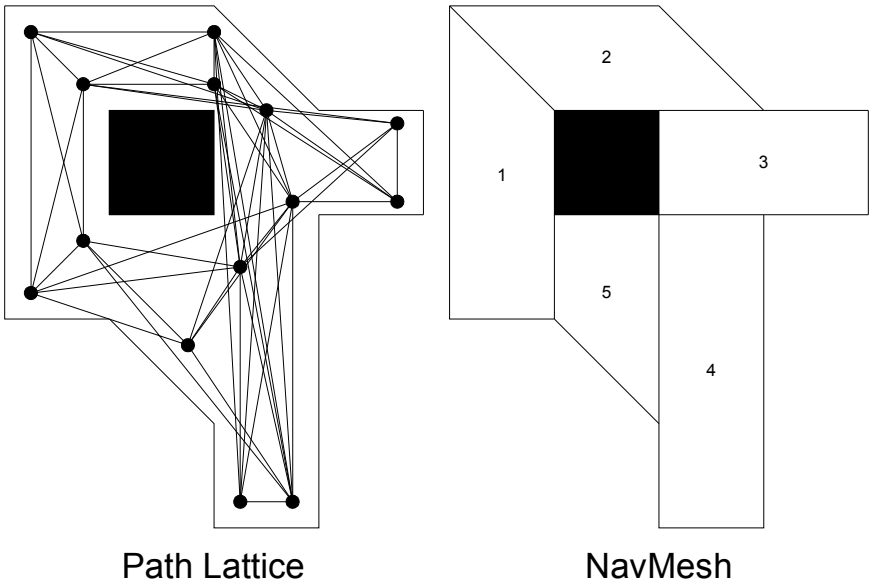


Figure 1: A NavMesh gives us a much cleaner and solid representation of the static world than a manually created Path Lattice.

The second approach is fairly straight forward in the sense that we can take samples of the virtual world at regular uniform intervals and determine if there is solid geometry there (= 'closed') or not (= 'open'). Once we know this, we can link all 'open' neighbors together in order to create a graph that can tell us how to get from one place to the other without clipping any solid geometry. The construction of this meta data can either be done automatically via sampling, or needs to be added manually by designers.

The third approach, using Navigation Meshes (or 'NavMeshes' for short), is the hardest but also the most rewarding. It is based around the idea that we can somehow recognize important geometric 'features' in the world which we need to navigate around. We then need to obtain all the walkable surfaces around these features and their adjacency (see Figure 1). These areas are then stored using polygons which means that our simplification will still be highly accurate. By connecting the individual polygons to represent a graph we can then use it to traverse the world.

I opted for the usage of NavMeshes for a number of reasons. First there is of course their flexibility, we can easily represent any sort of world because the retained accuracy is much higher than with a 'discrete and uniform' approach (see also Figure 2). Second is the fact that NavMeshes can be optimized so that the polygons represent large areas in the world such as flat roads, open town squares, etc. This will reduce the amount of memory they need, which in turn means that a path-finder will have less data to process in order to find routes through them. This makes them especially useful for platforms such as the current Xbox 360 and PS3 consoles which have tighter CPU and memory restrictions (also think of improved load times). This property almost automatically excluded the use of path lattices because these require a large amount of way-points. And finally, because the representation is polygon-based, we no longer pose any restrictions on level designers. They can potentially add any type of geometry they want with full degrees of placement freedom without having to do a lot of manual maintenance such as with path lattices.

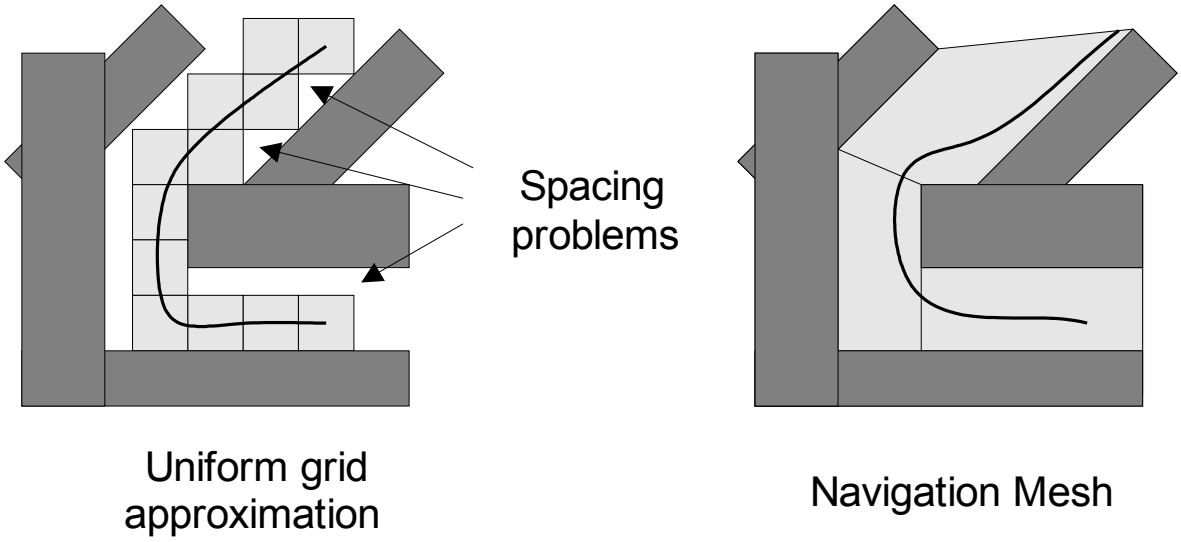


Figure 2: A drawback of uniform grids is that there is a limit to the approximation quality of the environment. To counter this, navigation meshes utilize polygons for the representation of walkable surfaces.

2.2 Path-finding

The question of finding high quality paths efficiently is a relatively 'old' one and has been the subject of much research. By far the most common approach is one of the many so called 'A*' variants, which are basically depth-first searches through directed graphs (also see [12]). A* is widely popular because it is fairly straightforward to implement and lends itself very well for fine tuning and adaptation for specific needs and restraints. The many existing A* variations basically deal with trade-offs between memory consumption versus CPU cost.

It was thus only logical to use A* myself. I was particularly interested to see if it was possible to take advantage of modern hardware and modify some A* variants that would lend themselves well for parallelization. During my research it became clear that 2 of these seemed to be most promising: Bidirectional search and the so-called 'Fringe Search' (also see [2]).

2.3 Obstacle avoidance

Although having a path that takes you from point A to point B safely through a static world is already quite an achievement, it is not yet enough. During our travels we are more than likely to run into dynamically moving obstacles such as other travellers. A natural behavior would be to avoid these and 'move around them' which is often a matter of side-stepping in opposite tangent directions. We can obtain this effect by using so-called 'whiskers' that are sensing ahead of the traveller up to a certain distance (see also Figure 3). If the whiskers detect an obstacle to the left then the traveller reacts by moving towards the right, and vice versa (also see [4], [10] and [14]). The distance that the whiskers measure towards an obstacle is also a measure of how strong the tangent avoidance force should be that is applied on the traveller. Multiple whisker samples are merged into a single avoidance force in order to deal with more than one obstacle. We could also add additional behavior logic depending on the type of traveller. For a car for example we could add a braking force that triggers when obstacles suddenly appear in front of the car.

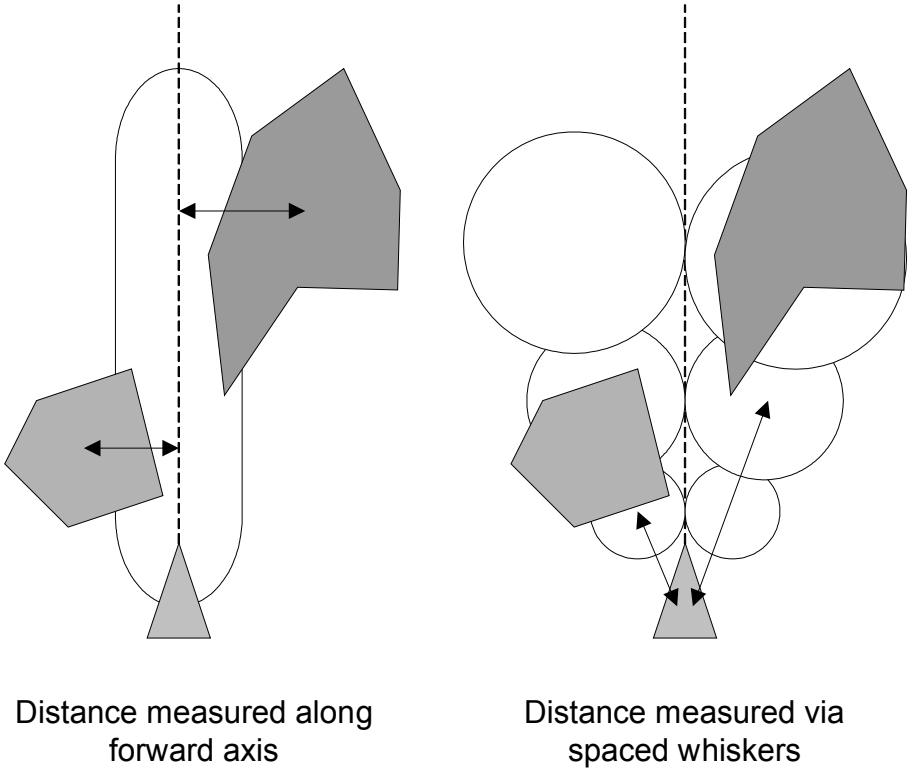


Figure 3: Different ways to implement sensory whiskers.

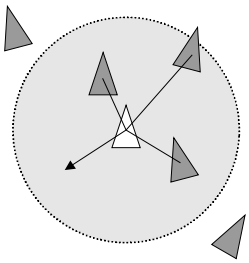
During my research I came to the conclusion that the use of whiskers is basically a subset of a larger system that controls flocking behaviors in groups of objects. This system uses so called 'Boids' which are objects that follow a couple of very simple rules and relies on 'emergent behavior' to control a group of them in a natural way (see [13]). These rules are depicted in Figure 4.

Each of these directives generates forces that influence the rotation and movement of an individual Boid in relation to its neighbors. The strength of each force is dependent on various parameters and attributes. For the Separation directive, for example, each force will be directly related to the distance between the neighbor and the Boid. This will make the Boid steer away from the neighbor which needs to be avoided most urgently. For the Cohesion directive we can, for example, increase the weight of specific Boids if we want them to fulfill a leading role. For video games we can use this to reflect the importance of special units or to force groups of units to match their speed to the slowest units.

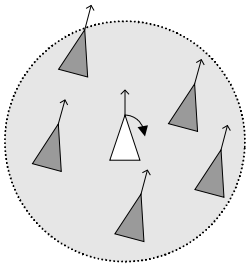
Note that directives 1 and 3 are in direct contradiction with each other. Directive 1 acts as ‘repulsor’ which forces the Boid to steer away whereas directive 3 acts as an ‘attractor’ which tries to pull the Boid closer. The result of these opposing forces will be a ‘natural’ minimal distance between all the Boids where the forces just about cancel each other out.

It is very simple to extend the flocking system in order to incorporate obstacle avoidance. All we have to do is represent all dynamic objects as 'repulsors' to the flocking system. The nice thing of incorporating the flocking system in my opinion is that it is now fairly easy to incorporate more advanced behaviors. We can for example mimic a 'hide and seek' strategy whereby we try to find

Separation: steer away from local neighbours to avoid crowding.



Alignment: rotate to match the orientation of local neighbours.



Cohesion: steer towards the average location of local neighbours to induce bonding.

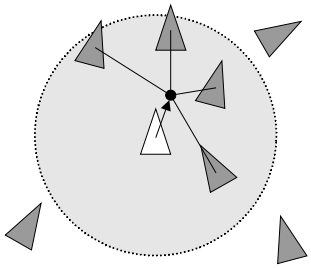


Figure 4: The 3 basic directives of a flocking system. The combined effect of these directives result in 'emergent behaviour' that closely mimic the natural behaviour of groups of animals and other autonomously moving entities.

safe-spots that are out of visual range of hostile objects. By representing these areas as attractors and the enemies as repulsors to the flocking system we can obtain some very naturalistic results.

One of the strengths of the flocking system is also a potential weakness. When we would strictly adhere to the solution that a path-finder has found for us, we would know that every step we take is 'safe'. If the flocking system forces us to deviate too much from this path, then we are no longer certain of our well-being. In my opinion it is best to rely on physics to make sure we do not walk through any walls and such when being driven off course. Using a physics model whereby travellers are represented by simple spheres, for example, is relatively cheap and is often already a necessity for a large number of game types, especially if the players are free to roam a 3D world and are allowed to jump. Because the flocking system generates 'forces', it is trivial to combine this with such a physics system. From my experience, this approach will automatically help to smoothen the movement of objects because they can then also push each other out of the way a bit and slide against walls and other smooth surfaces (one must not forget that because of all the summing and averaging in the flocking system, we will not always have perfectly 'valid' steering forces). As a bonus there is a very natural way to combine the flocking system with the path-finder: we only need

to run a strong attractor along the way-points of the found path solution, ahead of the traveller. We do not even need to smoothen the path using splines in order to make the object walk with curvature through corners, this will all be automatically done by the physics system.

So how can we make sure that the traveller does not accidentally stumble into a hazard such as fires, or walk off a cliff? Well, to begin with, all hazards should be represented as strong repulsors to the flocking system. That will deter the traveller from moving there in the first place. If we want to make absolutely sure that the traveller will not touch any dangers, then we also need to somehow represent them as physical obstacles. This could be done using 'invisible' walls, but we could also opt to link movement to the NavMesh somehow. We could for example mark certain polygons as 'impassable' and hard-wire the physics engine to prevent the object from moving into such areas. It depends on the kind of game what will yield acceptable results.

The use of whiskers will not strictly guarantee us that a traveller will always find its way around any dynamic obstacle it might encounter. It will however provide acceptable results in the majority of the cases. If we adopt a physics based system, then we can also apply some random 'noise' if an object appears to be blocked by moving it in an opposite 'random' direction for a short while. If the object is still not able to solve the problem locally, then we will need to fall back onto a deterministic approach that will put the traveller on the right track again. Because there can be many types of obstructions, I opted to apply a scheme that would try to re-plan the path using A* again. This will infuse some 'determinism' into the whole system again, which obviously has many practical advantages. In fact, this mimics natural behavior, for example: when a door suddenly gets closed and locked, a traveller will only notice this after 'bumping' into it. Only then will a traveller reassess how to fulfill its long term goal of reaching the desired end point.

In order to achieve this, I opted to use a scheme that temporarily enhances the resolution of the NavMesh by subdividing its polygons locally. We would first determine which of the obstacles is in our neighborhood and for which it is natural that the traveller would acknowledge its existence. We then progressively sub-divide the NavMesh's polygons in all the areas closely around these obstacles up to a certain minimum size. Next, all the obstacles will mark the new subdivided polygons they are resting on with 'movement penalties'. These movement penalties can for example be additional weights on the NavMesh graph edges that increase the movement cost towards the nodes that the polygons represent. The A* path-finder will then automatically avoid these nodes and find a path around them, or completely alternate routes if needed. If the path-finder fails to find a path at all then this means that it is no longer possible to reach the destination. It is up to the traveller to decide what to do next (this is basically a decision made by a higher order system). Often it is best to first apply a delayed retry system so that other obstacles can move out of the way (think of a small gateway that is 'clogged' with travellers that all want to move through it at once). If successive attempts fail, then the traveller should be switched into a sort of 'panic mode' whereby a new long-term goal is selected.

2.4 Group movement

So far we have only considered the routing goals of a single traveller. Will the scheme that we have just described also yield acceptable results if we let all group members travel all by themselves? No it will not, we can easily demonstrate this in Figure 5.

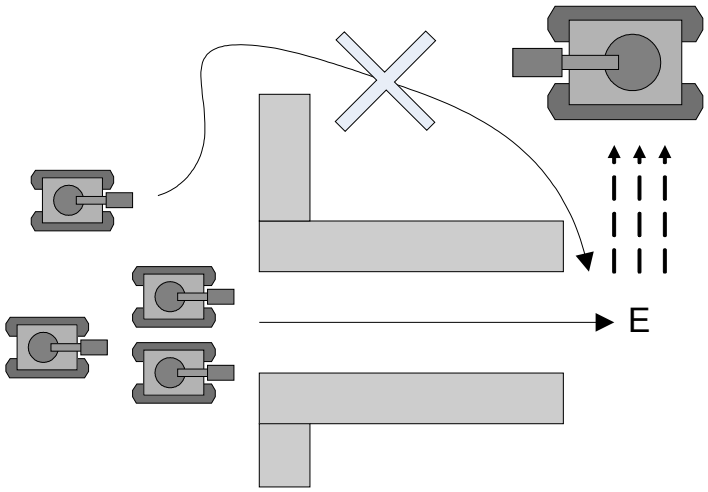


Figure 5: Locally optimal paths are not automatically globally 'optimal'.

A human player wants to knock out a big enemy tank using only a small group of less powerful but more agile tanks. Because he is out-gunned, he wants to use the element of surprise and approach the big tank unseen and then strike at it at close range before the big tank can swing its turret and reap havoc. Thus, the human player selects all his tanks into a group and orders them to move to position E. If each unit would simply use its own local path-finder, it is not difficult to see that some of the units will either:

1. Be spaced too far apart so that their optimal paths to the same goal are actually quite different.
2. Find their path blocked by units in the front and thus try to find alternative routes.

Such routes will however take them straight down the big tank's cross-hair which results in a lot of the smaller tanks getting destroyed instantly and alerting the opponent that more danger might be coming his way. Such events generate strong negative gaming experiences and frustrate the human player.

During my research I came across an algorithm that solves these problems in a way that can be nicely tied into the flocking system I had already proposed. The idea is to not generate individual paths for all the members of a group but to generate a single 'corridor' through which we will lead them all at once (the basics of which are described in [9]). The backbone of the corridor itself is constructed using any A* variant we would like to use, provided that it can find a path with a certain minimal dimension. In theory this could be the dimension of the largest group member, but we should try to use much larger dimensions first in order to improve realism. Next, the group of units will travel the corridor using flocking attractors that try to pull the group into a local region that keeps moving forward. This group region is basically a subsection of the corridor with a fixed surface area A. This constant A is a heuristic that is equal to the minimum area that is needed for all units to stand right next to each other multiplied by a 'magic number' to increase their room for maneuvering. Some simple 'look-ahead' logic is used in order to 'stretch out' the area that contains all the units at narrow choke-points so that in total it remains relatively constant in size. When attractors are about to travel through the choke-points they will increase their 'pulling' force (see Figure 6). This in turn will force the flocking units to 'fall in line' before they enter the choke-point

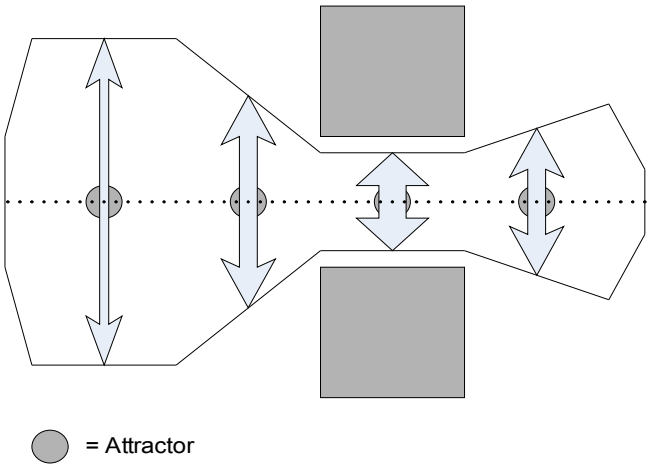


Figure 6: The corridor will deform itself by letting the attractors increase their 'pull' when nearby corridors.

itself so that all units can pass smoothly. In a sense, this mimics how a fixed volume of a fluid ‘traverses’ irregular surfaces, the natural spacing of the Boids mimicking its cohesiveness.

2.5 General approach

After having oriented myself on how to approach my research objectives it was clear that 'naturalistic movement' of objects through virtual worlds is not as easy as it looks and spans a multitude of disciplines. To bring all of it into practice was certainly outside the scope of the remainder of the project and I thus would need to focus on only a subset of them in order to execute them well.

Having worked at Triumph Studios for a while and being closely involved with the development process, I have learned that technical problems are often of a 'minor' nature compared to practical ones. Triumph Studios currently employs a path-finding system that is in essence discreet uniform grid and relies on quite a lot of manual labor to add path-finding related meta data. Graphics designers have to create building 'blocks' that contain both the geometry data and a rough description of walkable surfaces in the form of a local uniform height map grid. When a level designer places these blocks in the map using the level editor, their meta data is easily connected because these blocks can only be snapped into place in their own uniform 3D grid. Each block can only be rotated at 90 degree intervals around the vertical axis though. For NavMeshes this approach is no longer desired because it will undermine the whole principle of having polygon surfaces as path-finding meta data. Ideally you would want to be able to place any kind of model in the world and be able to rotate it unrestricted. For large models it will also be quite labor intensive to manually define all the NavMesh polygon surfaces. If we want full freedom of model placement then we automatically accept that models can intersect in many different ways once they have been placed in the map, which needs to be resolved somehow.

I therefor decided to write a software module that parses all the models in a map and automatically generate the NavMeshes at the push of a button. I figured that we could use all the models' hit-volumes for this purpose so that in the future a graphics artist only needs to add a minimum of meta data. The nice thing was that Triumph Studios already had a large library of models complete with

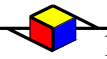
hit-volumes that define simplified representations of all their solid parts (these are needed for all sorts of collision detection during the game). So my goal was partially to reduce the work-load for a level designer, so that he or she can focus more the on creative aspect of level design and improve overall game-play quality.

I have observed that one of Triumph Studios tooling's most important aspect was the quickness of the edit and test-play cycle. The use of the a uniform block grid has the big advantage that each modification in the map will take almost no effort for updating the path-finding meta data. In fact, each map can immediately be played after saving and loading. This makes for very short 'trial-and-error' cycles which has absolutely a strong positive influence on the quality of the end result. So it was immediately clear to me that the speed with which we can generate NavMeshes is vitally important. I thought about using existing Boundary Representation (B-rep) libraries to aid in the NavMesh generation process in order to get rid of all the intersections, but I figured this was a bit of an 'overkill', and might be too slow in the end. I assumed that because my input data consisted mainly of relatively large and 'sparse' hit-volumes, that it would be better to write everything from scratch so that I could focus on execution speed as well. To me it seemed that the accuracy that professional CSG modules achieve would not be needed, I only needed to obtain a 'decent enough' abstraction of the virtual game worlds. By writing my own libraries, I figured I could take short-cuts whenever there was a trade-off between accuracy and speed.

Regretfully, it took a lot more effort than I had initially planned for to get the NavMesh generator stable enough. Therefore it was decided to simplify the whole natural path following problem by only addressing how to to generate paths and smooth them. This would save me the trouble of having to write and/or interface with existing physics libraries. However, this meant that unfortunately I would not be able to actually experiment with the flocking system and group coherency, which would have just taken too long. I figured I would still be able to achieve fairly decent results by implementing the 'footprinting' algorithm whereby the NavMesh's resolution would be locally enhanced so that paths can be found around any dynamic obstacle.

Although current day technology has evolved dramatically, so have quality standards. Especially for game development there is always the question if the software is fast enough. The last part of my work was thus focused on finding ways to improve the performance of the A* path-finder algorithm. The usage of NavMeshes is already a promising step because it greatly reduces the search graph and thus the work-load. So apart from 'better' input data, what else can we do to improve performance? Currently, multi-core hardware configuration have found their way into PCs and modern consoles, but these have thus far not been exploited to the fullest. I wanted to see if it was possible to speed up path-finding by parallelizing its algorithms, in order to obtain higher performance gains.

During the next chapters I will go into great detail on what the main problems were that needed to be resolved and all the results that have been achieved. The main focus has been on demonstrating that the chosen approaches are viable and usable for game development purposes. In many cases there remains a lot of additional future work to be done that would further enhance the quality of the obtained results. In these cases, I will give an overview of potential research directions and/or pitfalls that might need to be avoided.



3 NavMesh generator

A NavMesh is basically a simplified representation of all locations of interest to a game unit in a (virtual) world and their connectivity. This chapter will describe the exact layout of a NavMesh and how I have been able to generate these autonomously from raw user data.

3.1 NavMesh layout

NavMeshes are only concerned with the ‘walkable’ areas within a game world. With ‘walkable’ we generally refer to planes/polygons that are facing upwards and which do not have other geometry above them that would prohibit an object from being able to stand on it. The geometry of each individual walkable area (= convex polygon) is then represented by a single node in a graph, and the topology of the graph's edges defines which polygons are direct neighbors of each other. NavMeshes are thus stored ‘explicitly’ in memory as undirected (and often sparse) graphs together with a table of polygons (see Figure 7). Depending on further needs, all sorts of additional data is stored on for example the graph's edges, such as: movement restrictions (costs, prohibited access for certain entity types), which polygon edges does it crosses, etc.

The main advantage of NavMeshes is that they contain a very accurately representation of the virtual world through which objects want to travel. Normally they have a low memory footprint because large empty areas are represented by only a handful of equally large polygons. It is, however, not easy to generate NavMeshes and requires complicated software algorithms that can 'parse' the game worlds (which can be a quite lengthy operation).

3.2 Input data

At the time I started my project, Triumph Studios used custom design tools that allow designers to manually define path-finder data. This data was basically a uniform node grid with a half meter spacing. Each node contains some flags that define it to be either 'accessible' or 'blocked' in order to get a rough approximation of the static world. I found this data to be unsuitable for the resolution I needed, so I decided to obtain all path-finding data from the collider hit-volumes that are also part of the models that can be placed in the maps.

So the input data for the NavMesh generator is basically this:

- A straightforward height map consisting of a 2D matrix of height values. The terrain is simply converted into volumes by taking a terrain polygon and protruding it downwards.
- The hit-volumes of all objects placed in the world: this includes anything from walls to floor-tiles to stones and such. All hit-volumes are bounding-boxes that are freely oriented and positioned in the world.

3.3 Basic generation process

I decided to integrate the NavMesh generator into Triumph Studios' level editor because this would make it fairly easy to re-use its user interface and allow for short test-cycles. In the next paragraphs I will describe the phases the generator goes through to generate the resulting NavMesh. In later

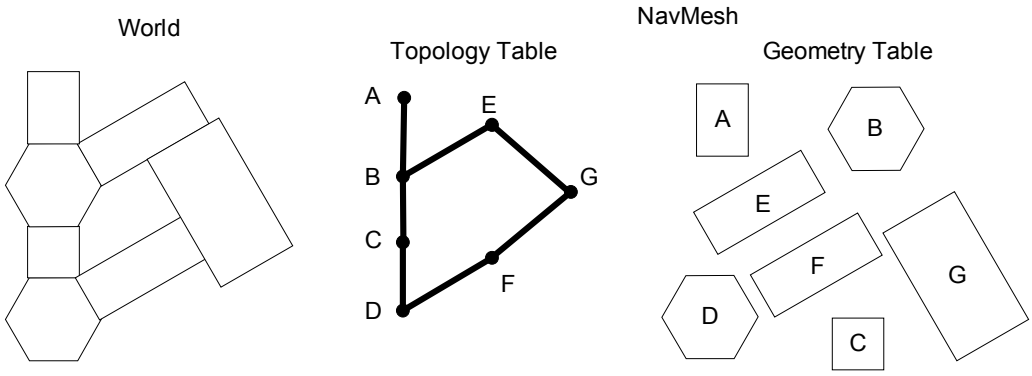


Figure 7: The definition of a NavMesh (often additional meta-information is present).

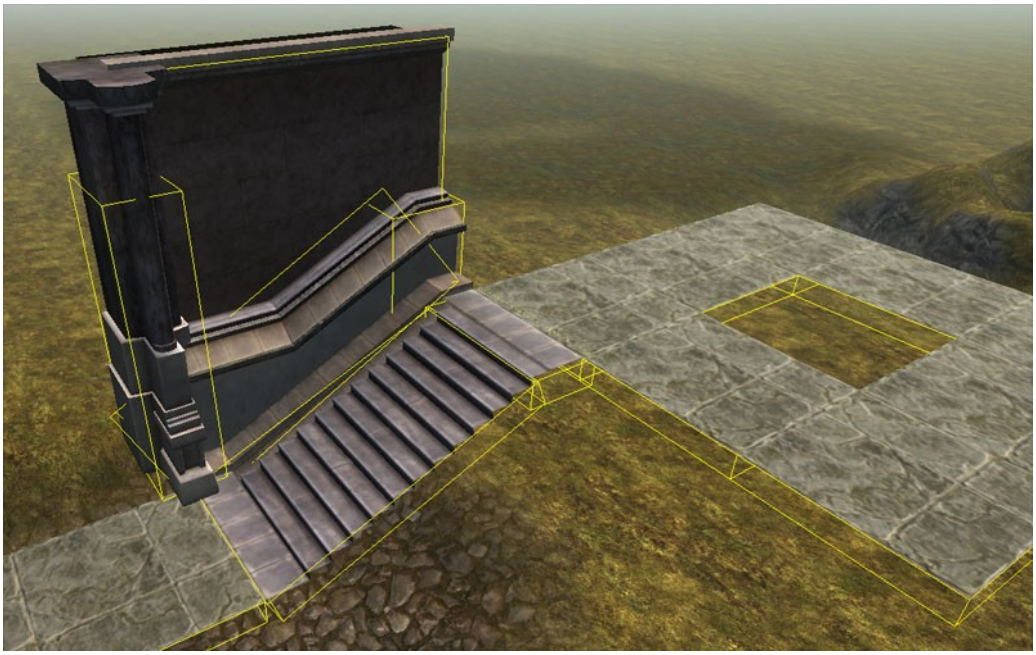


Figure 8: An example of the input data that we are using to generate a NavMesh: a collection of floor tiles and a simple staircase.

chapters we will see some refinements but for now the general idea behind the generation process will be explained. In short the following basic steps will be performed:

1) Convert world into mesh hulls.
2) Remove intersections.
3) Filter walkable surfaces.
4) Convert topology to graph.
5) Gap welding.

These steps will be illustrated using a simple input data set as shown in Figure 8. To keep things clear the terrain has been ignored.

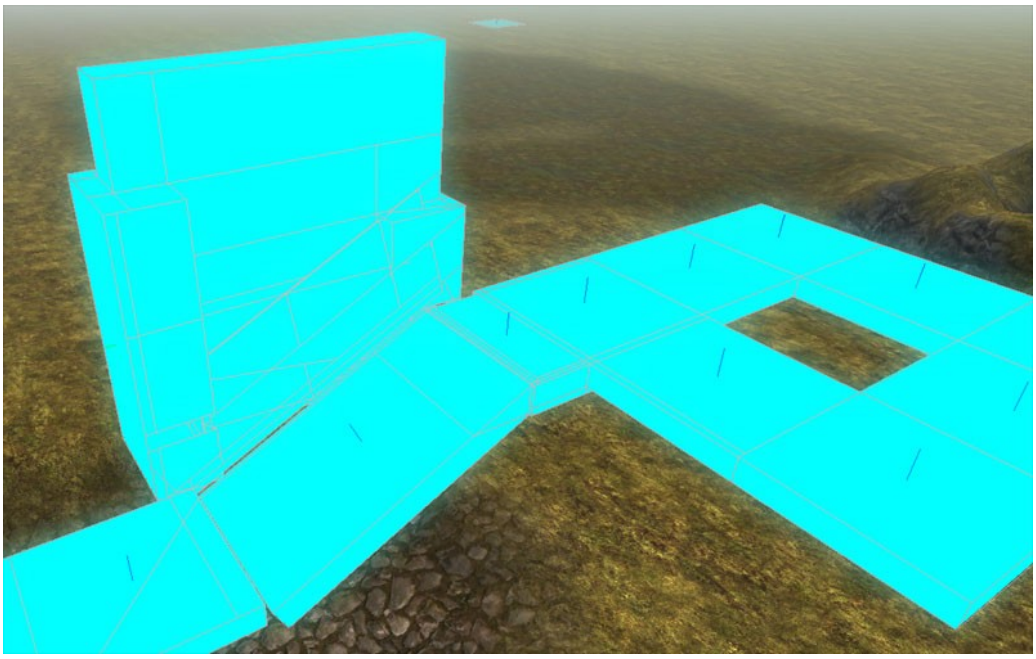


Figure 9: All hit-volumes have been converted into meshes and have their intersections removed.

3.3.1 Convert world into mesh hulls

We start the process by converting all the hit-volumes into mesh hulls, which involves generating polygons for all the sides of the hit boxes. A very important factor in this phase is to keep track of shared edges and surfaces, as later on it will enable us to determine which polygons are each other's neighbour and thus form a chain of polygons through which we can find paths. In the next step we also need to get rid of all intersections in order to remove of all the surfaces a potential traveller cannot stand on because they are inside other volumes. So during this first conversion step we need to generate the mesh hulls in such a way that we can represent such intersections. I have accomplished this by using a very rudimentary form of Boundary Representation that will add polygons one by one to the large world mesh. It will do this simply by taking a hit-volume's plane, use it to split all already existing polygons nearby and then construct a new polygon based on all the vertices and edges that were created during the cutting process. By repeating this for all the planes of the hit-volume we end up with a corresponding polygon mesh representation. This is a very CPU intensive process and severely suffers from degrading numerical precision, which will be discussed in greater length in chapter 4.

3.3.2 Remove intersections

Once all meshes have been generated, it is fairly easy to remove all intersections by taking the original hit-volumes and determine which polygons are inside it. When we remove all of these - making sure we do not remove any polygons that are on the outer surface of the hit-volumes - we have a derived data set as depicted in Figure 9. The current implementation is still very rough and has many unnecessary cuts that could be re-merged or avoided altogether.

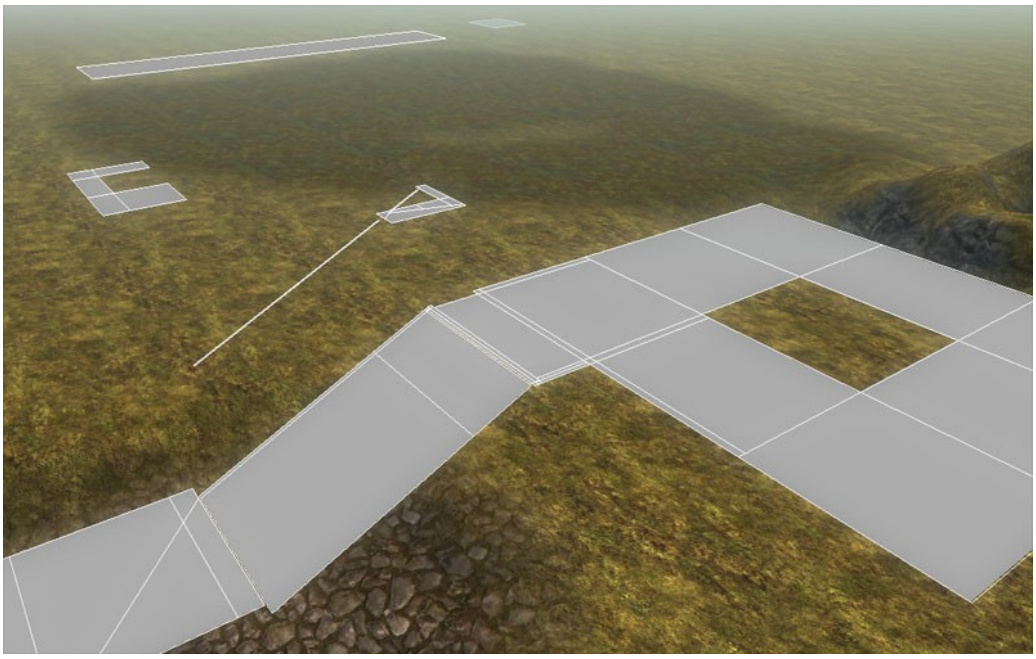


Figure 10: After removing all non-walkable surfaces we already have a good representation of the navigable sections of a virtual world.

3.3.3 Filter walkable surfaces

The next step in the process is to remove all the polygon surfaces that no object can walk upon, these are all surfaces that have a too high offset angle with an upwards normal vector. The result of this is depicted in Figure 10. We notice that a lot of surplus surfaces have remained, floating in mid-air. In this particular case these are for example part of the staircase railing and potentially an object could actually try and stand there. Depending on circumstances, we could enable level designers to also add meta data that specifies specific types of volumes that should also be avoided by travellers, e.g.: volumes of water, glass structures, etc. In this particular case they could also mark the stair-case railing (and related areas) so that the NavMesh generator will just ignore them.

3.3.4 Convert topology to graph

It is now very straightforward to convert the walkable polygon surfaces into the actual NavMesh. Because the generator has kept track of which polygon edges are shared between neighbouring polygons, it is easy to determine how to connect the NavMesh graph. Each polygon is first represented in the NavMesh graph by a new graph node. Then all these nodes are connected whenever we find a shared edge between the polygons they represent. The resulting graph is depicted in Figure 11. We can see that some graph edges appear to be crossing mid-air but this is just due to the representation of the graph by drawing lines between their centroids. For polygons that only share a small edge at a corner this will give this result. Later in section 5.2 we will see how a found path is converted into an actual smoothed and valid path without running the risk of being ordered to move through the air and/or other areas where we are not supposed to go.

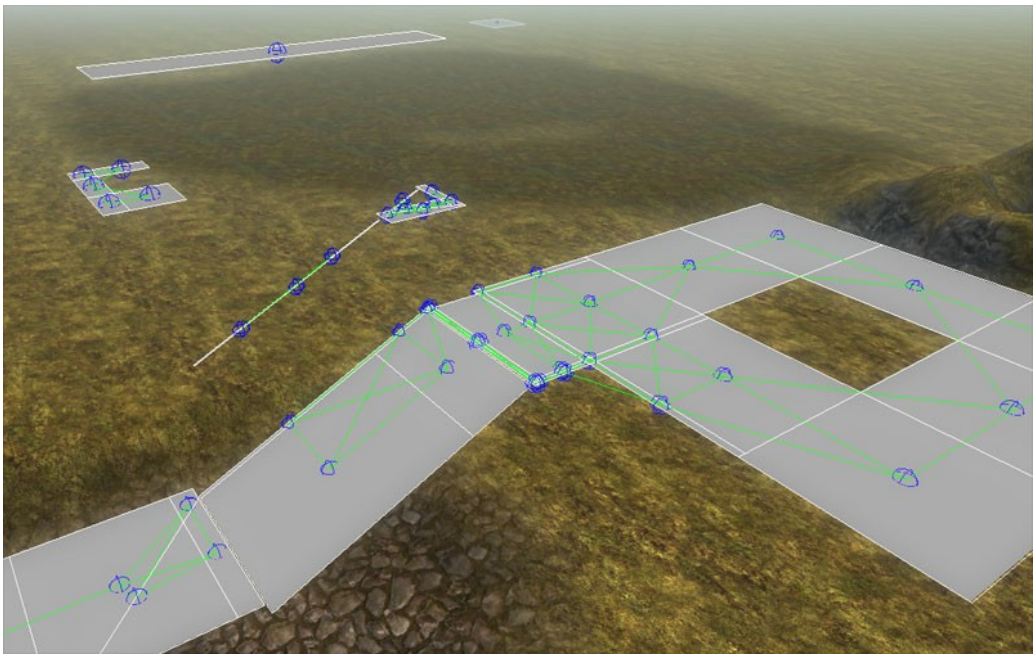


Figure 11: Creating the NavMesh graph is now just a matter of converting polygons into graph nodes and linking them wherever shared polygon edges are found.

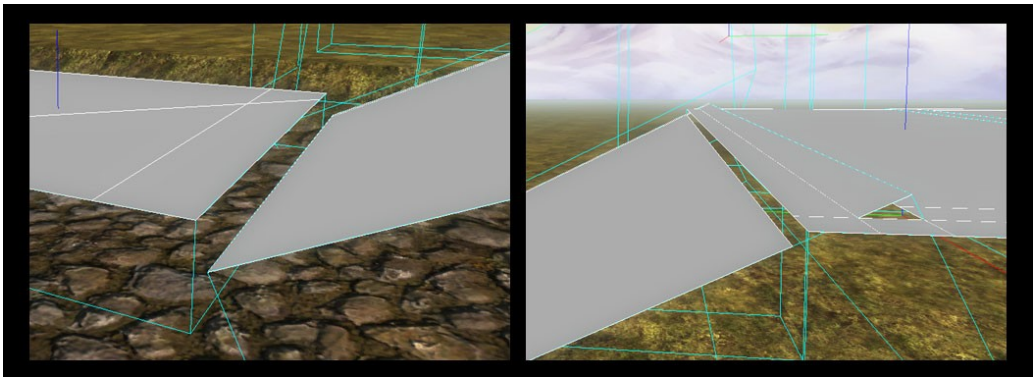


Figure 12: Upon closer inspection we see that there are 2 type of gaps. The geometry gaps as seen on the left are already present the original data set: a (minor) 'human error'. The gaps on the right have inadvertently been created while we removed non-walkable surfaces: a 'staircase defect'.

3.3.5 Gap welding

It is immediately clear that some walkable surfaces are not connected where we would expect them to be. On closer inspection we see that this is caused by 2 factors (also see Figure 12):

1. Input data errors: Small gaps between the original hit-volumes that were present in the original data set. Often these are hard to spot for a level designer because to the naked eye the geometry appears to be touching or intersecting each other, whereas in its internal (limited) numerical representation it is actually not.
2. Introduced errors: Staircases and such will have parts of their (semi-)vertical geometry removed. By doing this we will inadvertently loose some relevant topology information.

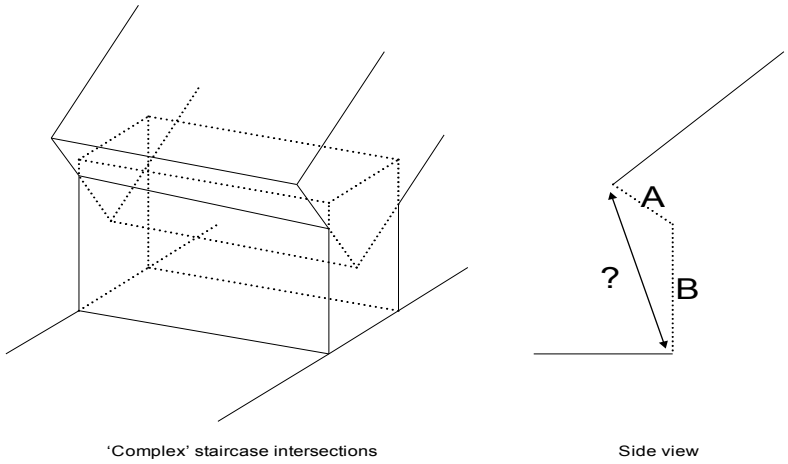


Figure 13: More 'complex' staircase intersection may be composed of several polygons that are not even coplanar and/or facing downwards.

At first I planned on fixing the second type of errors by determining in the previous phases of the generation process which polygons formed parts of staircases that connect parts of walkable surfaces. However, this turns out to be much more involved than it appears to be at a glance. Take for example Figure 13, where we see that staircase gaps can often span a multitude of polygons who are not all coplanar and might even be facing downwards. To make the generator parse something like this would have taken a lot of extra work and might still cause issues for more complex and unforeseen situations.

Furthermore, the solution of problems of the first type cannot be fixed automatically without doing some sort of guessing anyway. It is up to a level designer to specify some heuristics that define what maximum 'step distance' a traveller can perform, both in terms of height and 'forward' direction. These heuristics can then be used to find edges that are 'close enough' and appear to be 'parallel enough' in order to be considered adjacent and forcefully connected in the NavMesh. This type of welding will also automatically fix the staircase anomalies of the second type that were introduced. A generic post-processing step like this is overall less complex to implement and is also much more natural for a level designer to work with because it will yield far more consistent and predictable results. The new links created during the welding process can be seen in Figure 14. We now have an accurate description of the virtual world's navigable areas, which basically concludes the building of the NavMesh.

The welding approach is also easy to extend if we were to look for places whereby a traveller can potentially jump from one surface to another. During the welding process we would only need to add meta information that will define the corresponding NavMesh edge as either a 'continuous step' or a 'jump'. If for some reason a level designer wishes to create game-play where jumping should be prohibited (think for example about a wall of fire that a traveller should never attempt to jump trough, or a plain window) then we need to rely on meta data being manually added to tell the generator what to do in specific situations.

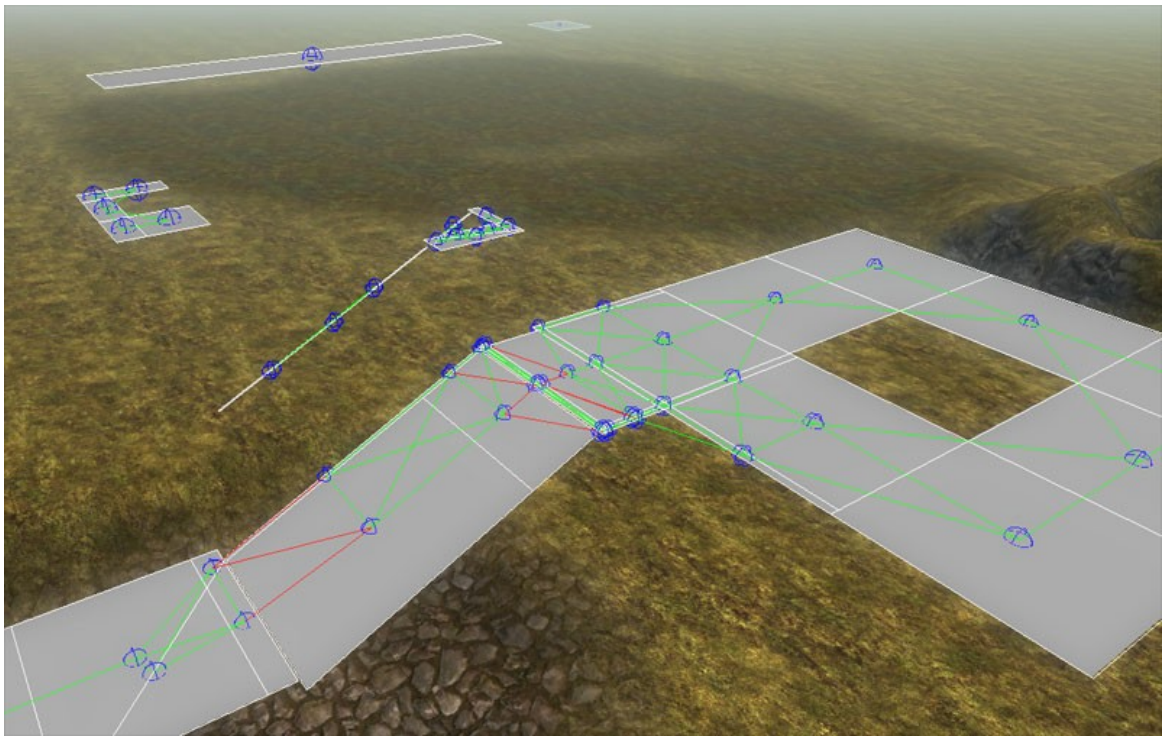


Figure 14: After welding, the NavMesh forms an accurate description of the virtual world. The green edges were constructed using topology information, the red edges are the result of welding.

Potentially, we could avoid a lot of the type 1 problems by letting volumes automatically 'snap and glue' together when they are interactively placed in the level editor. To the designers this would feel like as if placing 'magnetized' building blocks together. We could then add some visual feedback on where exactly the blocks have been glued together so that errors can be corrected early on. This idea is however beyond the scope of my thesis.

3.4 Compensating for traveller dimensions

So far the NavMesh generation process has only focused on finding areas that are potentially walkable. We must not forget though that any actual traveller is not just simply a dimensionless point-like object. This means that the actual area an object can stand on is smaller because we need spacing between walls and such. This of course, also needs to be reflected in the NavMesh. For simplicity we are going to assume that any traveller can be accurately represented by a bounding sphere. This will make the processes for incorporating this information a lot easier, we are mainly focused on the principles.

It is tempting to think that we can easily obtain enough information from the NavMesh polygons directly. This is deceptive, however, because there are many ways a potential traveller can walk across the polygons. Just looking at the length of a polygon's edge for example will not suffice to determine if the traveller can cross the polygon's area. In the left diagram in Figure 15 we can see that polygons might not always be part of valid paths, it completely depends on how the traveller is going to cross them. To make things more complicated, we can also see in the right diagram that it

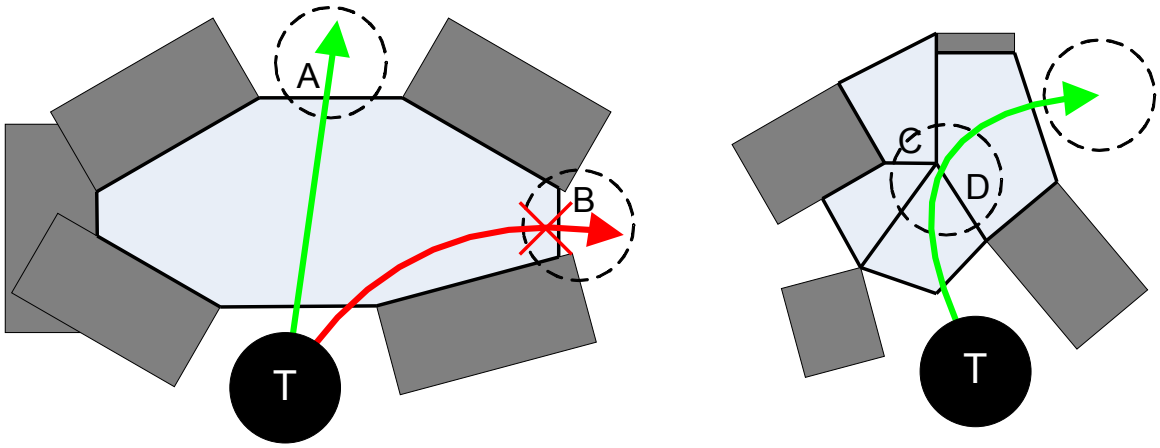


Figure 15: An essential path-finding problem, made harder due to the usage of NavMeshes. It is not easy to determine if traveller T could traverse any of these polygons, it is heavily dependent on the path it is following. On the left, for example, we can see that traveller T can easily cross edge A but not edge B. For the situation on the right it is even more complicated because traveller T cannot even stand on all the polygons individually but it is still able to pass because the polygons with edges C and D together provide enough clearance to pass.

might not be possible for a traveller to 'stand' entirely inside the boundaries of the polygon at all, whereas it should still be part of a path solution because there are neighboring polygons that provide enough clearance for the traveller to pass through. A naive approach would be to try and 'parse' neighboring polygons to see if we can ascertain through which edges a traveller can move freely without intersecting with any solid geometry, but this turns out to be very hard. The main problem is that there might be many polygons that are in turn 'fractured' in such a way that we can no longer make safe assumptions for such an algorithm to be correct (and it might even turn out to be really slow if there is a lot of solid geometry nearby). Also, we would then need to store a lot of extra meta data per polygon edge so that a path-finder can then figure out if it can travel from one polygon edge to an other.

How to compensate for traveller dimensions is in fact a core problem for path-finding in general, which is especially hard for world representations such as those provided by NavMeshes. My solution to this is to turn the problem the other way around. Instead of determining if there are enough neighboring polygons to allow for a traveller to stand on a particular polygon, we are going to 'cut-away' any surface parts that we know that traveller cannot stand on due to its dimensions. What we will end up with then is a NavMesh for which we know with absolute certainty that it describes all the areas where the traveller could stand on with its 'center of mass' (in this case the bottom of its sphere). This approach is much more solid and reliable, and will keep the act of path-finding itself much simpler because there is no extra meta data processing involved. Luckily we can do this with some fairly straightforward enhancements to the cutting algorithm.

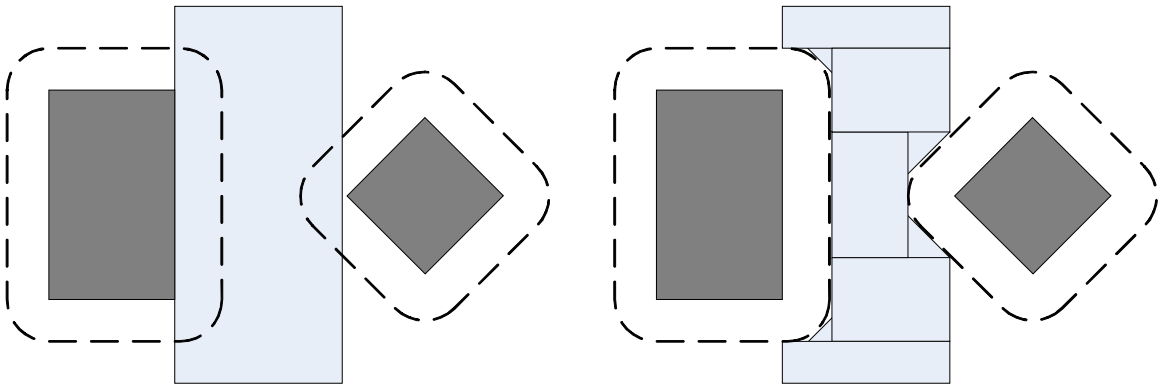


Figure 16: Top-view: surface areas that are close to walls need to be cut away from the NavMesh because travellers cannot actually stand there.

For a sphere with radius *R* we need to make the following adjustments to the 'Remove Intersections' phase:

- For each of the hit-volumes from the input data set:
  - Inflate the hit-volume distance *R* outwards for each of the boundary surfaces that the traveller cannot walk upon (later in this chapter we will refine on this).
  - Cut the meshes at the boundaries of the inflated hit-volumes.
  - Remove all polygon parts that are inside the inflated volume if they did not originate from the hit-volume during the previous mesh generation phase.

The idea behind the cutting is illustrated in Figure 16. Strictly speaking we should 'round' the corners of the inflated hit-volumes but this would make things more complicated and probably fragment cut polygons to a high degree. I have found that we can achieve acceptable results by simply 'uniformly' inflating the hit-volumes as long as sphere radius *R* is not 'too large' (say 10 meters). The generation process will run a lot faster and the whole process of cutting is a lot simpler this way (and more robust too). Also, because we are basically cutting away a bit too much surface area we can still be absolutely sure that we will not generate paths for travellers where they actually cannot physically pass.

Now, we cannot actually simply inflate the hit-volume in all directions because then we would run the risk of cutting away parts of neighbouring polygon areas that should remain intact. Think of a bunch of floor tiles for example that together span a single surface areas. If we would simply expand all vertical planes of their hit-volumes they would cut-away parts of their neighbours.

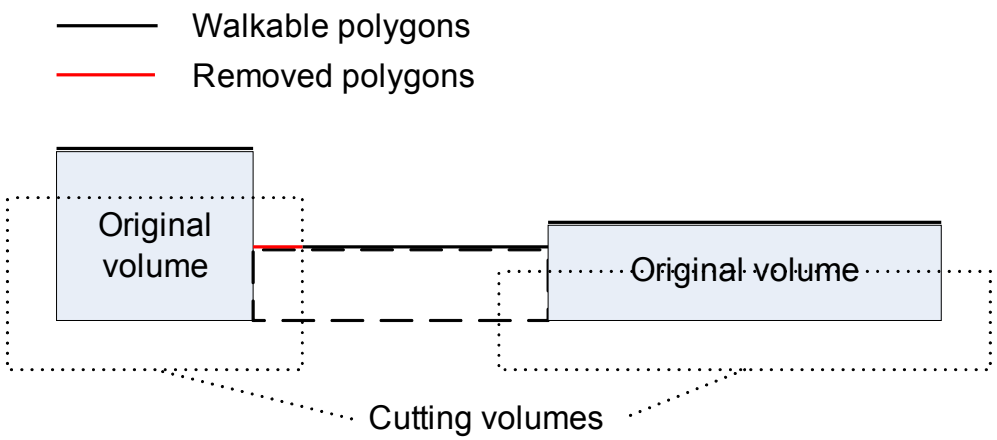


Figure 17: Side view: by 'deforming' the hit-volumes we can safely cut away polygon areas where a traveller cannot actually stand.

Because there are often many input hit-volumes I needed to find a fast and simple way to 'deform' the original hit-volumes into a valid cutting volume. An initial, and naïve approach is the following (see Figure 17):

- Walkable surfaces are actually moved inwards over a heuristic 'stepping distance' *SD* (see below).
- Non-walkable surfaces are moved outwards over distance *R*.

The value *SD* is basically chosen as to represent the maximum height distance a traveller can overcome when stepping upwards or downwards. By shifting the walkable surfaces inwards, we make sure that the cutting volume will be below any adjacent areas of walkable polygon surfaces and thus will not cut them.

However, if we look closely at Figure 18 we can easily spot problems with the algorithm in the vertical direction. If we pick value *SD* too large we will still cut away neighbouring surfaces when the original volume is tilted in any way. Similarly we notice that for surfaces that are aimed somewhat downwards, we should expand the cutting volume even more because we want to make sure that we cut away surface areas on which the traveller with its entire bounding sphere volume cannot stand on. That would imply that we should expand downwards aimed surfaces not with distance *R* but with the full sphere 's diameter: distance *2R*.

An improvement on the algorithm is thus to link the outwards deformation of a hit-volume's surface to its upwards orientation angle. So the deformation rules are then as follows:

- Walkable surfaces are actually moved inwards over a heuristic stepping distance *SD*.
- Non-walkable surfaces are moved outwards over distance $sin(\alpha/2)2R$ with α being the angle between the surface's normal and an upwards normal.

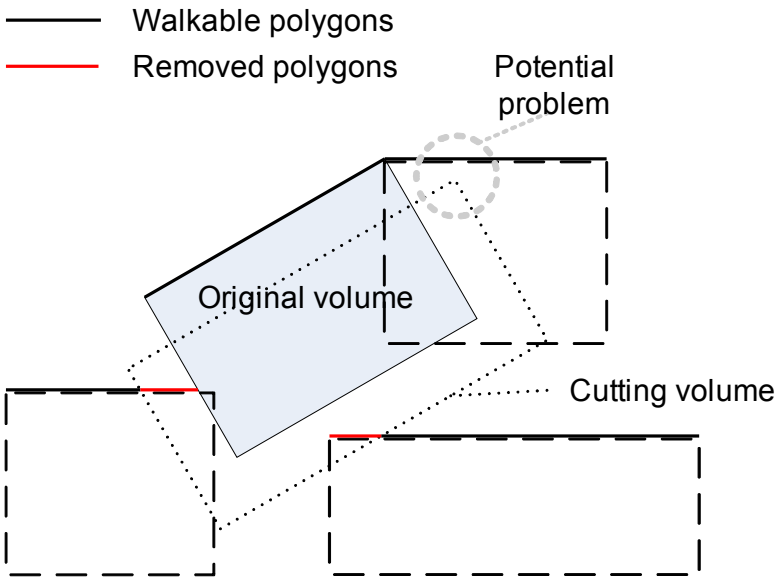


Figure 18: Side-view: the current deformation algorithm is quick but not without its potential flaws and needs refinements.

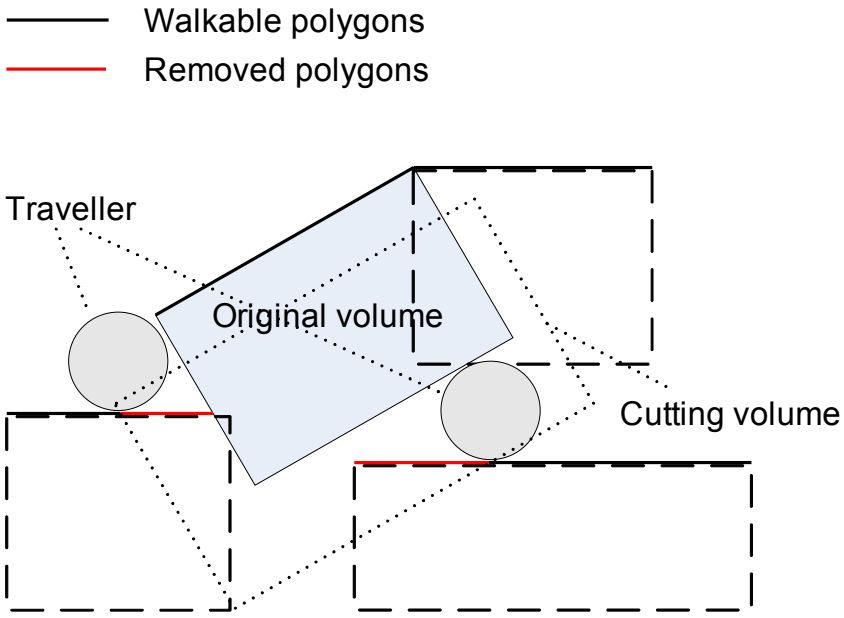


Figure 19: Side-view: The modified deformation now correctly cuts away all polygon areas where the traveller cannot stand.

Figure 20: The input data that is used to illustrate the cutting corrections for traveller radii.

The result of this modification is illustrated in Figure 19. We will now correctly cut away all polygon areas where a traveller cannot stand because it would otherwise intersect with the original collision volumes. Non-walkable surfaces that are aimed upwards will now be less extended outwards, and non-walkable surfaces that are aimed downwards will be more extended.

This simple deformation algorithm yielded satisfying results in my test-beds and I thus deemed them good enough for demonstration purposes. The result of cutting for different travellers' radii is illustrated in the next few screen-shots. The input data is a simple scene consisting of some horizontal tiles, some pillars and a doorway (see Figure 20). If we set the traveller radius to 0 we generate a NavMesh as was already demonstrated previously (see Figure 21). If we start to increase the radius of the traveller we will see more and more areas near the walls and pillars to be removed (see Figure 22 and Figure 23). Clearly, with a larger radius, the traveller can no longer pass through two pillars on the right for example. In the final figure (see Figure 24) we can even see that it is no longer possible for a traveller with a radius of 3 meters to exit the doorway in the center.

Because these areas have been removed it is easy to see that a path-finder that tries to find a route through the NavMesh will correctly avoid these impassable areas because there simple are no graph nodes passing through them. We can thus safely rely on the path-finder's output to send travellers through the virtual world without any collision problems.

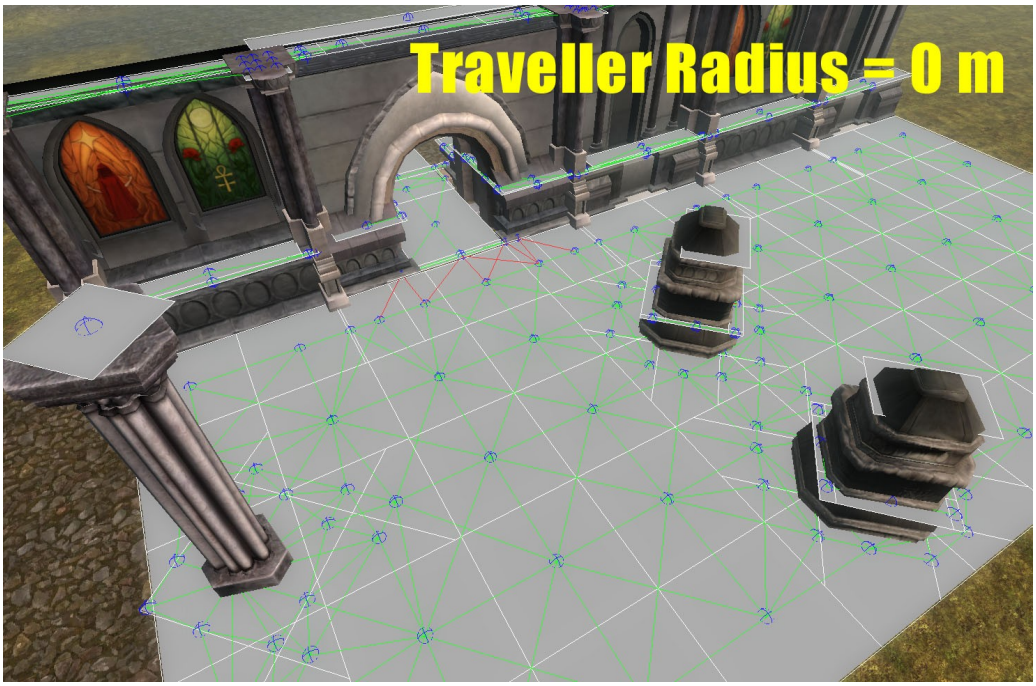


Figure 21: If we assume the traveller has a radius of zero then we obtain a NavMesh as was shown before.

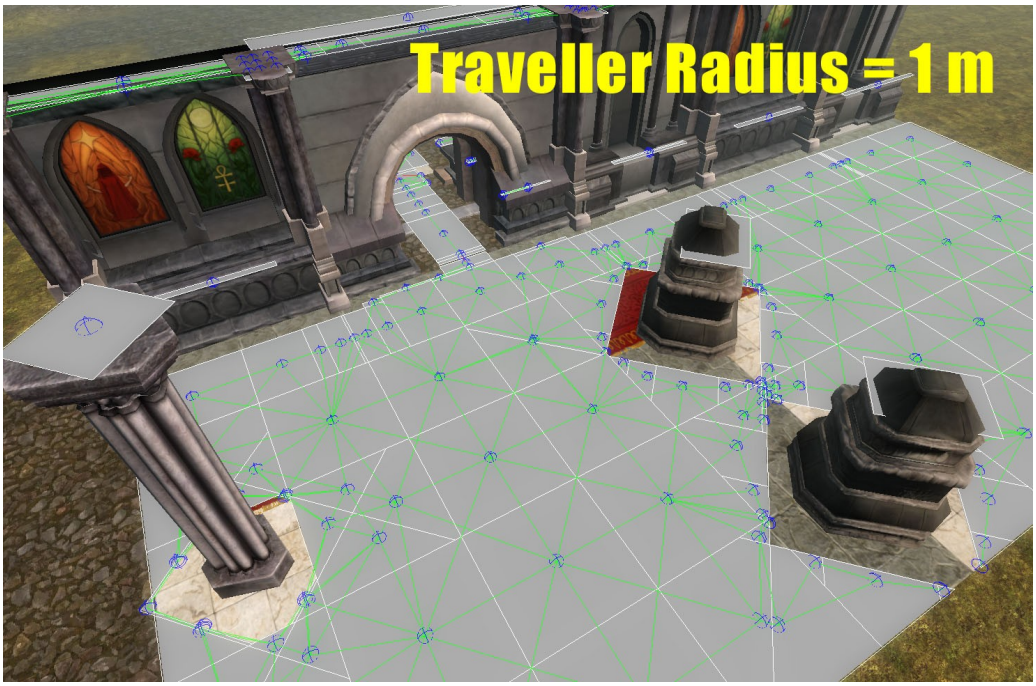


Figure 22: When we set the traveller radius to 1 meter we see areas being cut away from the walls and the pillars.

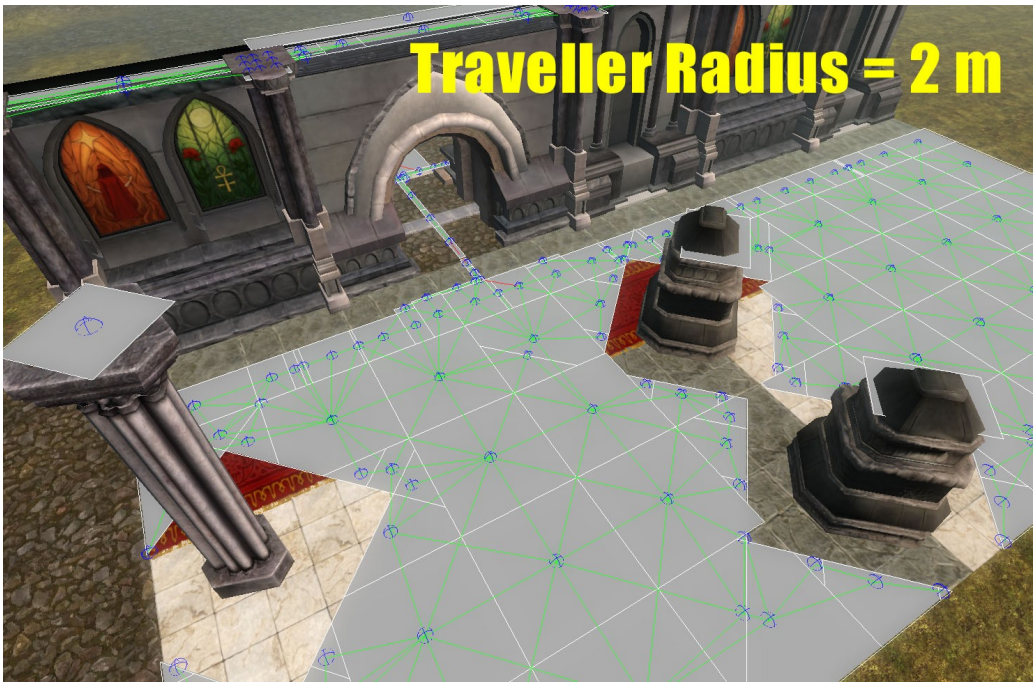


Figure 23: With a traveller radius of 2 we see more areas being cut away. We notice that the traveller is now too big to fit between both pillars on the right.

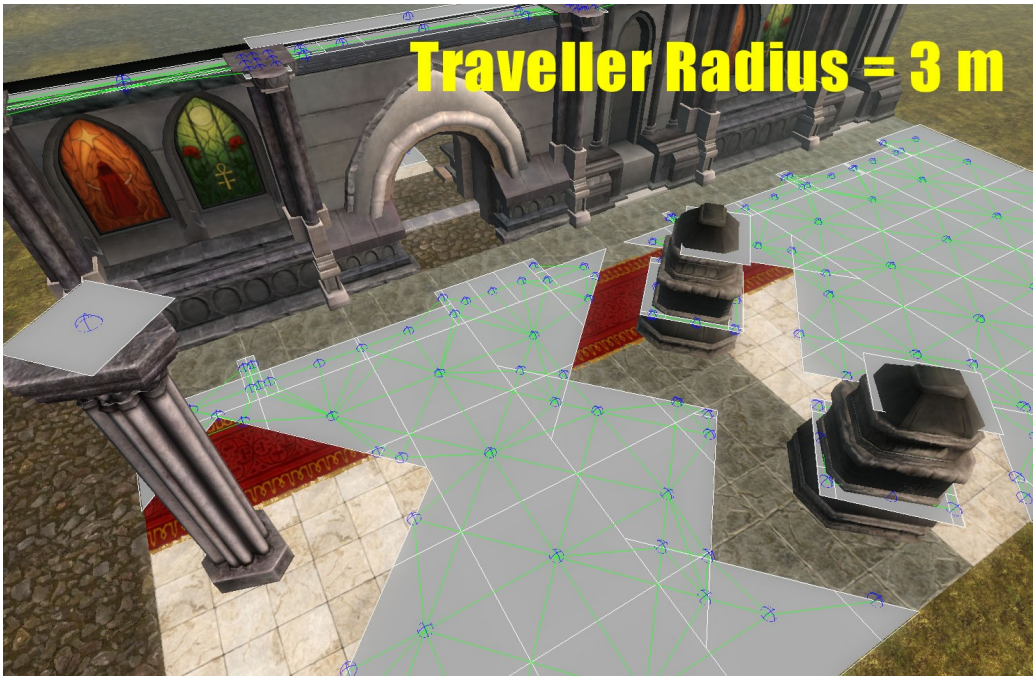


Figure 24: A traveller with radius 3 would no longer be able to pass through the doorway, the NavMesh no longer contains connections there.

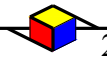
4 NavMesh implementation

Writing the NavMesh builder was not an easy task. The most important feature that it required was a way to create and manipulate intersecting polygon meshes. This turned out to be a lot more work than initially anticipated, and slowly evolved to a light-weight 'B-rep' (boundary representation) engine. Writing such an engine is quite a daunting task on itself, not only for its complexity but also because it is 'plagued' by many little problems that stem from limitations in floating-point representations. Debugging it is also harder because we often need to visualize problematic cases somehow in order to get a proper insight of the spacial context. As a result I have only been able to implement the very basics of the NavMesh generator. In this chapter I will give an overview of the complexities involved with construction and manipulating polygon meshes, and also discuss some features and strategies that could be implemented in the future to further improve and expand NavMesh quality and applicability.

4.1 Polygon mesh library

One of the most important parts of the NavMesh generator has been the construction of a library that could handle all sorts of operations on polygon meshes. When I started writing this, I was under the impression that it would be best to write one of my own that would be very 'light-weight'. This way I hoped I could make the generator fast enough for real-time editing, or at least not so slow that it would be prohibitive as a post-processing step. I have already explained much of my reasoning behind this in section 2.5; the turn-around time between making changes in an editor and testing the result in-game are of vital importance for level designers to obtain high quality maps. The implementation of a lot of the NavMesh generator processing steps mainly involve cutting and removing parts of polygon meshes. I figured that because a lot of geometry in games is by nature already 'low-poly' due to obvious memory and render limitations, this would not require a full-blown Boundary Representation (or B-rep) engine. For reasons that will become clear soon, this turned out to be a rather unfortunate decision on my behalf. Experience in retrospect has taught me that for even simple data sets its it very hard to do any kind of 'Boolean'-like operations while trying to maintain a consistent result. Because NavMeshes consist of polygons and adjacency information, I needed a way of detecting overlaps between hit-volumes from the models of the input data. This process involves converting them into polygon meshes as already explained in the previous chapter.

My approach was to treat vertices, line segments and planes not as singular/dimensionless/perfectly flat entities but to think of them as something that has a 'small amount of mass'. So points become spheres, line segments become capsules and (polygon) planes become nearly flat 'boxes' with rounded edges. Everything will thus have a 'minimal thickness' that I will denote with constant ε; this constant is chosen fairly small. After some experimentation I decided that the minimum distance between 2 points should at least be $\varepsilon = 0.001$, or otherwise they should be considered being singular. This way, we will actually see a 'physical' overlap if for example 2 boxes are positioned tightly against each other but do not actually share common vertices and edges precisely. Note also that due to limits to floating-point precision we might normally not actually always find a strict overlap between the sides of 2 boxes due to imperfect numerical representations. The usage of the ε constant will also help in suppressing such anomalies (but should by no means be considered a 'holy-grail'). Numerical instability will cause all sorts of subtle and unexpected problems as we will see later on. If we look at the specifications of the IEEE 754 standard, for example, we can see that there is a myriad of cases that generate unexpected noise in floating-point arithmetic (see [7]). A striking example is that half of all possible floating-point numbers are within the domain of [-1.0 ,



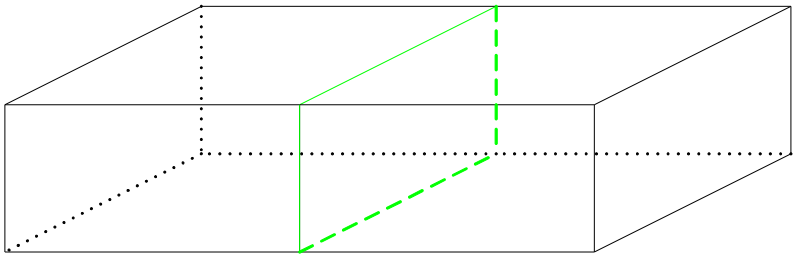


Figure 25: The 'PM_Space' container class will correctly identify areas/volumes that can be 'welded' together. The green edges are shared between the 2 box volumes.

+1.0], whereas the other half is outside it. So any projection of geometry from a small local-space into world-space can be affected by this.

4.2 The 'PM_Space' class

The base of the mesh library is a set of C++ templates that define the basics for vertices, edges and polygons. This made it very easy to define custom meta data for them whereby polygons even have 'front' and 'back' meta information. Central to the library is a special 'PM_Space' container class in which we store all the individual meshes and all of their atomic elements. A reference counting mechanism was added so that we can safely link elements together and easily determine when they can be discarded from memory. So logically, an edge contains 2 references to vertices, a polygon contains a clock-wise sorted set of edge links, and a mesh contains a set of links to polygon instances that span the boundary of a convex hull. To speed up a lot of the processes, octrees have been also implemented and 'parenting' cross-referencing has been applied (so an edge, for example, directly knows which polygons are sharing it).

The 'PM_Space' container class supplies a number of services and functions that enable us to create new meshes, cut through certain areas using split-planes and more. The code base for the 'PM_Space' library ended up being very large, its core is roughly 40K lines of well documented code (this also includes white-space, debug, error handling and visualization support code). It will take up too much time to go into all the complex implementation details in this thesis, which would only distract us from the more relevant topics. In appendix A, I will give a 'rough' review of some of the more important manipulation functions and how these were implemented. For specific questions and problems I would like to refer the reader to the code itself.

For now I will just stipulate again that one of the 'PM_Space' container class' primary goals was to adequately process volumes that are barely touching each other as seen in Figure 25. Not only will the 'PM_Space' container class detect shared edges, it can also ascertain that separate volumes are virtually 'welded' together via a shared surface. We find a lot of such cases in game maps because designers are using models that are designed as building block that fit neatly together. In the case of Overlord 1 and 2, the map editor actually utilizes a uniform grid onto which the models can be snapped into place.

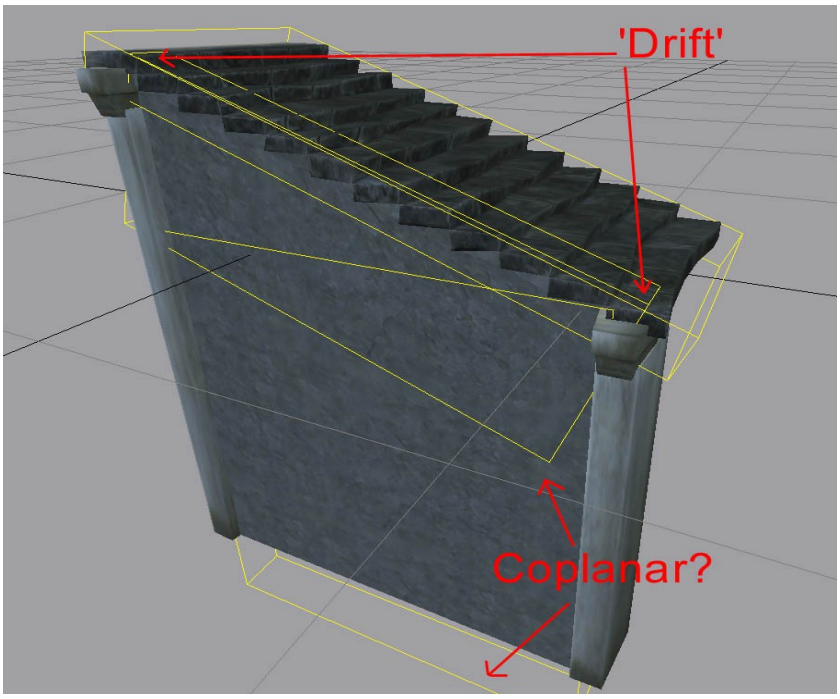


Figure 26: An example whereby a model's hit-volumes are not so tightly defined by the modeller as we would like. We can see a slight 'drifting' at the top whereby the boxes are not perfectly aligned. At the bottom we must wonder if both boxes are exactly coplanar at the side of the stair-case or not. These are potential problems for my 'PM_Space' class container when converted into polygon meshes.

4.3 Precision limitations: a big problem

Although I successfully implemented the polygon mesh creation and manipulation algorithms, I ran into considerable problems that were related to numerical precision and data representation. My initial assumption had been that the input data was already structured in a favorable way. Many of the 'building blocks' that a level designer works with are neatly segmented so that they can be fitted reasonable well together. My primary concern initially was how to detect shared edges and intersections between these building blocks in order to construct all the connectivity information needed for the NavMeshes. It was quite clear that the usage of the minimal thickness constant ε was the most obvious solution to this problem.

However, if we take a look at Figure 26 we see an example of an actual production model that uses but a few bounding boxes as its 'solid approximation'. These are all manually created by the modeler and thus these boxes can literally have the full spectrum of orientations and positions that a floating-point notation can provide. Now, the problem is not so much with all the volumes that intersect each other 'dramatically'. The problem lies in the fact that sometimes volumes that, to the naked eye, seem to be perfectly aligned to a grid, are actually slightly rotated and/or nudged from their perceived 'grid-locked' pose. This deceivingly minor problem suddenly caused all sorts of anomalies because of how I stored all my data structures.

This problem is illustrated in Figure 27 (be it a bit dramatized). Polygons *A, B* and *C* are coplanar and have been processed without any problems. As a result however, a diagonal edge *Q* has been constructed. When in turn coplanar polygon *D* is processed we apply more cuts but suddenly come

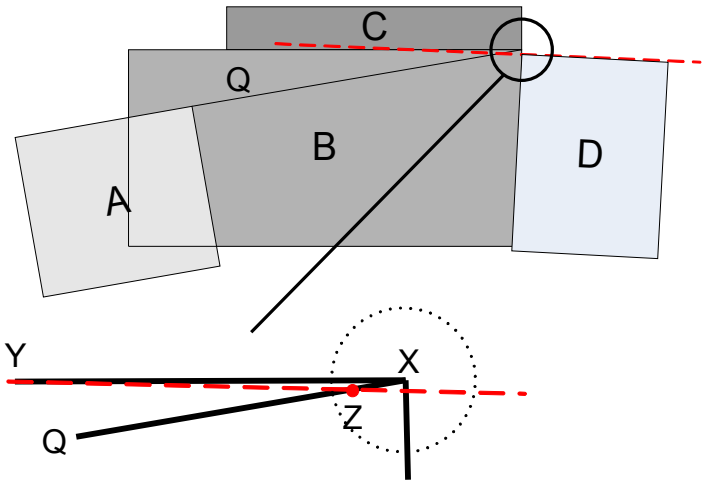


Figure 27: 'Dramatization' of a problem with cutting.

to the conclusion we cannot represent the results properly. The intended new polygon *XYZ* cannot be represented because the distance between vertex *X* and point *Z* is less than the minimal thickness constant ε. Correspondingly, we cannot correctly represent the intended contours of polygon *D* either. If we just ignore the problem and do not create vertex *Z* then we run the risk of edge creating functions not finding any intersections for polygon *D*'s problematic edge. It will just create a direct edge line-segment that will somehow run through polygon *B* without us knowing it. This means that the entire data representation in the 'PM_Space' container class will have become 'slightly invalidated'. Such a small anomaly can however easily cascade through the system in unforeseen ways.

I considered some strategies that might solve the problem adequately. Obviously, the simplest thing to do was to choose ever smaller values for minimal thickness constant ε. This however causes 2 additional problems. The first is that if ε is really small, we will not always detect shared edges between building blocks anymore even if they are actually touching each other 'physically'. This undermines the initial reason for using such a constant in the first place. The other problem is that all computations that are related to minimal thickness constant ε in some way suffer from the limits of floating-points precision. For example: if we want to know if a point *P* is 'singular' with an already existing vertex *V* we would need to compute their exact distance: *sqrt((P - V)● (P - V))*. Because of all the multiplications and square-roots involved we can easily introduce a large amount of numerical error. Even if we would try to compare the measured squared distance to ε^2 instead this will be very problematic, take ε = 0.00001 for example and ε^2 would already be as small as 0.000000001, which is roughly at the boundaries of what we can effectively represent with ordinary floating-points.

Another strategy I considered was simply to 'collapse' edges when they become too small. The problem then is however that you might end up with polygons that are 'dimensionless' because one of their edges has been folded into a single point. Furthermore, it will 'confuse' other algorithms that might, for example, expect to split a polygon into a front and back sub-polygon. Performing the operation and allowing edges to be collapsed might, for example, result in 'sub-polygons' that still stick out a bit through the split-plane. Also, treating split edges in such a way might disfigure polygons so that they are no longer perfectly convex (see Figure 28), or that their edges and vertices are no longer coplanar with the polygon plane that was originally represented. We could potentially

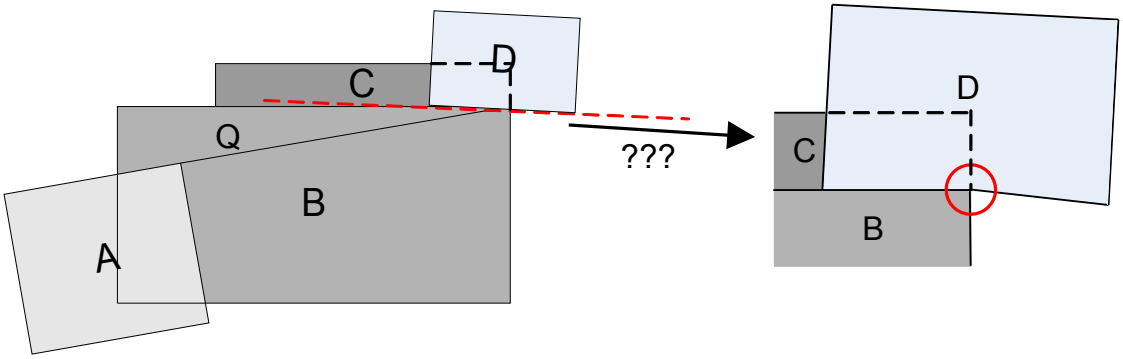


Figure 28: If coplanar polygon D would be created at the top-right corner, it can no longer remain perfectly convex if we just snap to existing vertices.

further subdivide non-convex polygons such as in the example of Figure 28, until we have multiple convex sub-polygons again. This could work, but we must realize that during the subdivision process we might run into similar problems on other edges: there are just polygons (small and/or very elongated ones) that we simply cannot represent at all. Also, we must think of the problem in 3D: if polygon *D* is a face of a convex box, and polygon *D* becomes distorted somehow, then all connecting faces will somehow also have to be corrected for this (meaning yet more sub-polygons). There is a limit to how far we can continue this before cumulative errors will make the system too unstable.

Admitted, this problem is also due to the nature of using the minimal thickness constant ε; it will pull points to very nearby vertices which means that new polygons that are created might invariably have their vertices 'pulled' out of their original edges and plane (see also Figure 29). Although this does not automatically mean that the polygon definition is always invalid (the polygon also is assumed to have a thickness), it will cause numerical noise in all computations such as convexity tests and 'inside-outside' checks. Another practical problem is that the amount of noise that is accumulated varies greatly with the order in which polygon meshes are constructed; in some cases we might end up creating sets of vertex positions in 3D space that appear to be more problematic than others. This kind of 'fuzziness' might make it harder for map designers to work with their tools.

Also, the limitation in floating-point precision sometimes slightly distorts the bounding box volumes in such a way that their local axes are no longer perfectly orthonormal in the raw input data. For example, some normal vectors might actually contain elements defined as 0.000001 instead of a perfectly rounded 0. So the input models are in a sense already slightly 'noisy'. Some of the inherit problems with floating-points can be improved by rearranging expression with multiplications and divisions in an attempt to suppress the cumulative error creep. In some cases it will help to use higher precision data formats if these are available, but these often suffer from major performance penalties (floats vs doubles in C++, for example). Ultimately however, I found that some problems will simply never solve themselves 'nicely', and you are just forced to implement some 'fuzzy' decision logic that uses 'magic numbers' a lot (regretfully). There is a broad field of studies that have analyzed the problems with numerical precision and how errors are accumulated, it is however far outside the scope of my research to address these problems at a fundamental level.

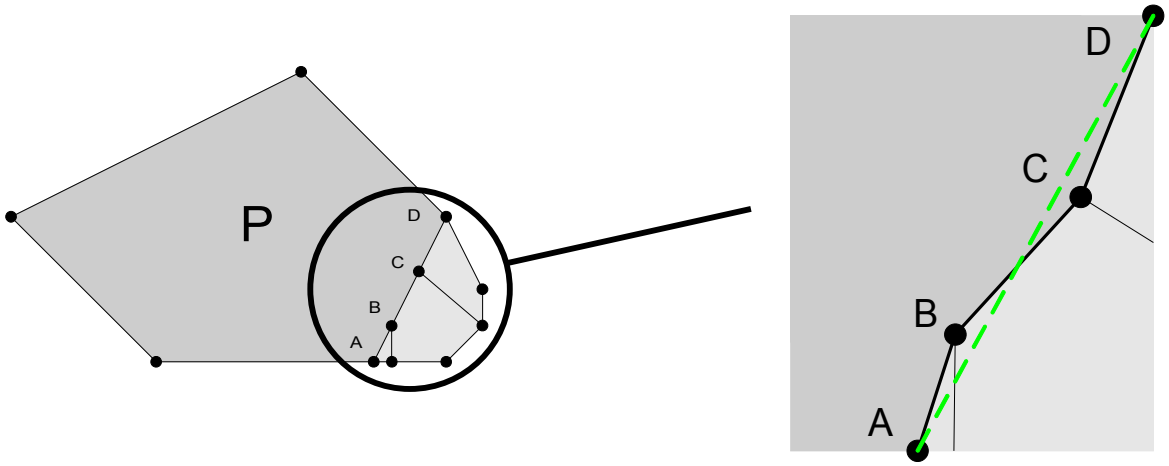


Figure 29: Example of how our data representations can be distorted in subtle ways. During the creation of polygon P's edge AD, the edge creation function apparently found vertices of already existing polygons. These vertices were deemed to be collinear with edge AD because their deviation was very small. As a result however, the polygon is actually no longer convex due to vertex B. Note that vertices B and C can actually have small deviations in all 3 dimensions, which means they might even be 'floating' above or below polygon P's plane.

From a strictly theoretical stand point, it was always clear to me that numerical precision was going to be a problem. Even if we would choose minimal thickness ε to be 0, we can always end up with edges that cannot be split because their vertices are so close together that we cannot represent it with a unique floating-point number between both their coordinates. I had not expected though that this is such a fundamental problem that you run into it quite quickly.

So my final solution to work around problematic cases was to filter and restructure the input data in such a way that we can almost guarantee that nothing 'seriously bad' will happen. This meant that all hit-volumes in the input data are snapped to fixed grid-positions with a 0.1 interval, and all their local orientation axes are matched to a single entry from a table of predefined allowed orientation axes (roughly at 22.5 degrees interval). It was my intention to numerically compute a 'perfect' set of allowed orthogonal axis with a given fixed grid-position and minimal thickness constant ε as parameters. In theory we should be able to construct the table by trying all sorts of combinations in the domain of [-0.1 , +0.1] in order to see if we get plane intersections that might cause problems. In an exhaustive search we should be able to ascertain how many planes we can safely add until intersections appear that are no longer well within distance of constant ε of an already existing intersection (say at a few percentages at max). So, in a sense, we would be trying to find the 'maximum intersection density' for which we can be sure that the system remains stable enough (see Figure 30). With 'stable enough' we should think in terms of 'magic numbers' with which we can successfully suppress and/or correct any accumulative errors without any problems. We could even go as far as to encode positions and orientations from the allowed sets and store them as meta data to edges and polygons, for example. If we derive unique hashing codes for these we can always perfectly ascertain if 2 edges are collinear: they then must have the same hash code.

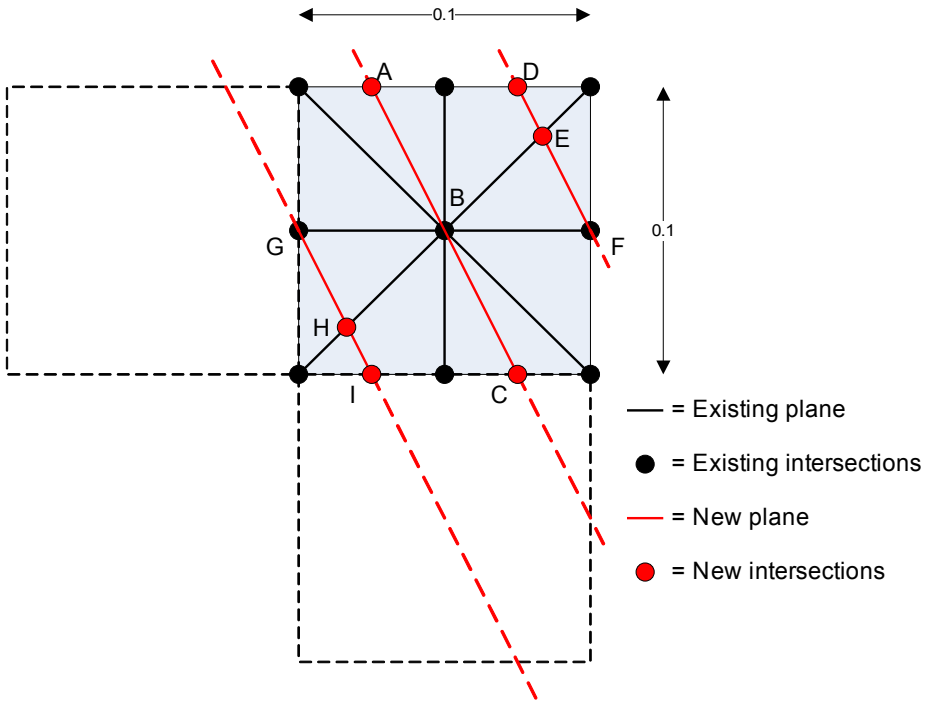


Figure 30: Demonstration of how to find the 'maximum intersection density' for a representative domain. For each new plane we create through the center of the cell at point B, in this case plane AC, we will need to find all intersections with already existing planes and those in neighbouring cells. Note that when the plane leaves at C it will enter an other cell at D. The same goes for F and G and finally I and A. We need to verify if new-found intersections A, C, D, E, H and I are not too close to already existing intersections. The new found intersections should either snap nearly perfectly onto existing intersections, or they should be spaced out 'far enough'.

However, this proved to be too much work and I thus had to concede with some restraints that I roughly tuned empirically (a lot of time was already invested in getting the 'PM_Space' library implemented). The obtained results are not as accurate as intended of course but the generated NavMeshes still proved to be useful. As long as the input bounding boxes are not too big, we will not notice any serious deviations (the hit-volumes are often rough approximations themselves, anyway). I think that with some serious tweaking and improvements, we might be able to obtain enough 'snapping' resolution that it will not be a big problem to modify all the hit-volumes in-game. In the case that this does not hold true, we could perhaps obtain a 'maximum degree of error' that we could compensate for by artificially increasing the radius of a traveller. This will give it some extra clearance and error-margin while traversing a path (see section 3.4). Actually, this is not a bad idea to begin with because there might be other sources of errors that could prove troublesome while traveling through the world, for example:

- Any errors accumulated by numerical imprecisions.
- Small implementation errors (there is a lot of code needed to generate the NavMeshes, there are bound to be subtle bugs somewhere).
- Path smoothing errors due to spline-fitting (these techniques will be discussed in section 5.2).

In retrospect, I should have perhaps used existing libraries for Boundary Representations and/or Boolean operations. I was afraid though that I might need some custom 'hacks' to cope with special meta data for ladders and staircases, such that it might be hard to do with 3rd party software. I assumed that 3rd party software also uses some sort of heuristics to deal with numerical imprecisions and/or implement short-cuts that will speed up the manipulation of meshes in such a way that such meta data might get lost or otherwise hard to track. By writing a light-weight B-rep engine myself I had hoped to be able to implement structural modifications that would greatly speed-up the process by only facilitating those features that are actually really needed; something that would be hard to pull off using existing libraries. Most of the models in my input maps just barely touch each other; so I figured that if I could focus on solving those cases efficiently the entire process might be fast enough. Perhaps these assumptions were ungrounded. And although the 'PM_Space' container class is fast, it turned out that it is nowhere near fast enough for reasonably normal sized maps. So, in that respect, a slower but professional B-rep library will not make that much difference; the map editing and testing cycle will still be (too) long to be of practical use on current PCs. Still, the obtained results are promising and I believe they could be greatly improved with the proper fine tuning and further optimizations, perhaps even made fast enough to support direct editing.

4.4 Future Improvements

Due to time restriction I was not able to implement all the features in the NavMesh generator that I had originally intended to do. Some of these features could significantly improve the quality and usefulness of generated NavMeshes in the future. So I am going to discuss them here with the intention to have given a complete overview of all required techniques and procedures for generating 'high grade' NavMeshes and corresponding path-finding solutions.

4.4.1 Polygon reductions

As has been demonstrated, my current NavMesh builder implementation already creates usable NavMeshes, but these are on closer inspection often a little 'rough around the edges', so to speak. Often near complicated geometry we will see that the NavMesh consists of a lot of smaller polygons that are fractured in myriad of ways. Next, we will discuss a number of approaches that could be used to further improve upon the quality of the NavMeshes by reducing the amount of polygons they contain.

4.4.1.1 Merging neighboring polygons

Very often, levels are designed with normal gravity in mind which means they will contain a lot of flat horizontal and sloped surface areas. This is especially the case when the maps represent any kind of buildings or other artificially created structures. A lot of these surfaces will consists of polygons that have virtually identical normal vectors (all aimed upwards). In a sense, these form an unnecessary 'triangulation' of an area that could also be represented by a single larger polygon. This forms the basis of a fairly easy optimization step, which in its essence is basically the reverse of what in computational geometry is better known as the ‘Optional Convex Partition’ problem. The best know solution for this runs in $O(n^3 \log n)$, which is ‘slow’. In order to speed things up, often a less optimal but much faster alternative is used called the Hertel-Mehlhorn algorithm (see [8] and [17]). This algorithm can run in near linear time and guarantees to give us a maximum of 4 times the minimal number of polygons attainable. The algorithm is actually used to partition a concave

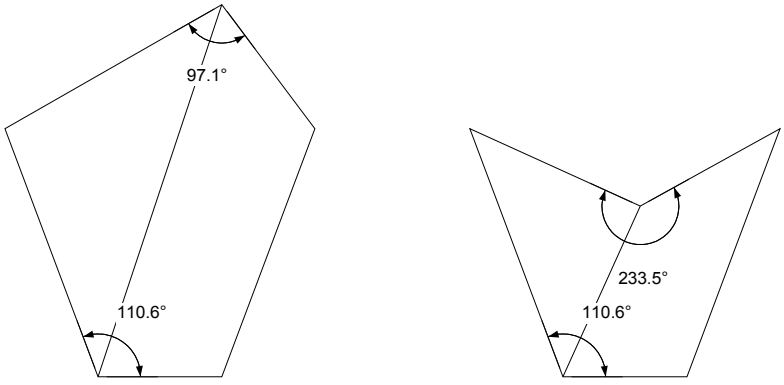


Figure 31: Hertel-Mehlhorn classifications of polygon diagonals.

1. **Find** a pair of adjacent convex polygons, *A* and *B*, that share a single edge and have nearly identical normal vectors (e.g.: $normal_1 \cdot normal_2 <= some\ small\ threshold\ constant$). Note that an edge in this context can consist of multiple collinear sub-edges (or in other words: the full length polygon edge is spanned by multiple vertices that are all on the same line).
2. Check if both polygons share the 2 vertices at both ends of the full length edge. **If** this is not the case **then** we cannot merge, **go to** 5.
3. Determine if the edge represents an inessential diagonal, **if** this not the case **then** we cannot merge, **go to** 5.
4. **Construct** a temporary polygon *C* that is a concatenation of both polygon *A* and *B* by eliminating the full length edge. This is done as follows:
5. **Find** the vertex a_n of polygon *A* on the shared edge that is the clockwise-most.
6. **Find** the vertex a_{n-1} of polygon *A* on the shared edge that is the clockwise-least.
7. **Copy** the vertices of polygon *A* in a clockwise order starting at a_n and ending at a_{n-1} (inclusive).
8. **Find** vertex b_n of polygon *B* on the shared edge that corresponds with a_{n-1} (this will be the clockwise-most in relation to polygon *B*).
9. **Find** vertex b_{n-1} of polygon *B* on the shared edge that corresponds with a_n (this will be the clockwise-least in relation to polygon *B*).
10. **Add** all vertices b_{n+1} up to b_{n-2} in a clockwise order to polygon *C* (exclusive).
11. **Repeat until** there are no more unprocessed edge candidates.

Algorithm 1: Hertel-Mehlhorn inspired polygon merging. It assumes that all polygons have their vertices sorted in a clockwise order

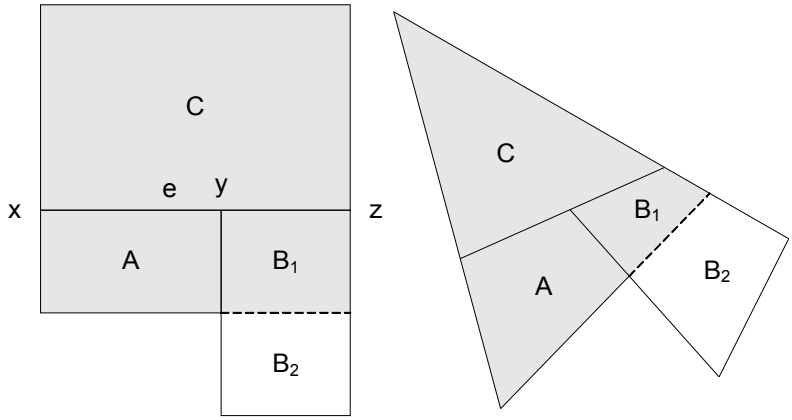


Figure 32: Example of 3 → 2 merging.

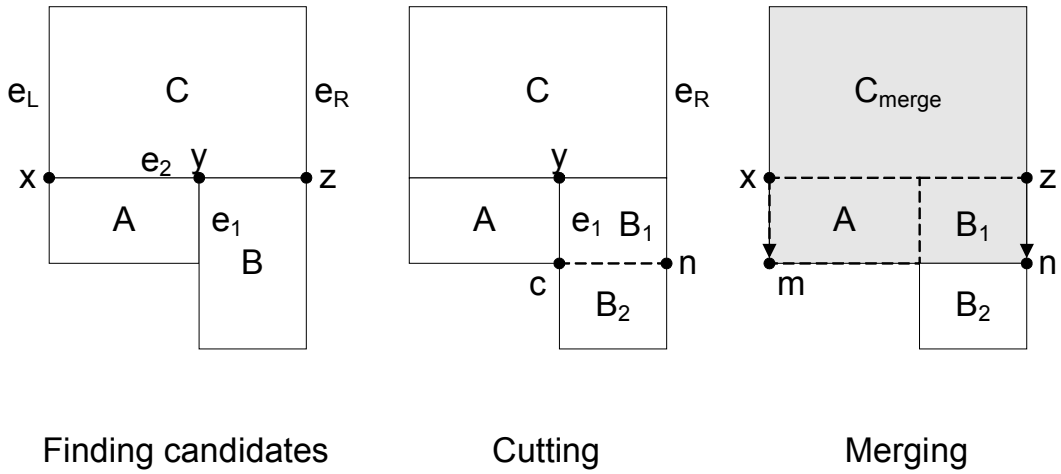


Figure 33: The various steps involved in the 3 → 2 merging process.

polygon into convex sub-polygons. This is not exactly what we need but we can easily reuse some of its concepts to aid us in the merging process (see also Figure 31):

- Given a (concave) polygon *P* and one of its vertices *V*: vertex *V* is **reflex** if its inner angle is larger than 180 degrees.
- Given a (concave) polygon *P*, one of its vertices *V* and a diagonal *D* to any of the remaining vertices that splits the polygon into 2 parts: diagonal *D* is **essential** if its removal would make vertex *V* reflex again.
- A diagonal is called **inessential** if it is not essential for both its endpoint vertices.

Our merging algorithm is rather straightforward and is formulated in Algorithm 1. Note that the resulting polygons will most likely have collinear vertices so we need to make sure our storage format can correctly handle such situations. We may also need to reformat the resulting polygons slightly so that all their vertices are once more coplanar (all sorts of 'numerical noise' can cause vertices to 'drift' out of the polygon's original planes).

1. Finding candidates:

   Find 2 adjacent polygons **that** share just one single vertex but also have a collinear edge (in our example these are polygons *A* and *B* with vertex *y* and edge e_1).

   Find a neighbouring polygon **that** has an edge on which vertex *y* lies (see polygon *C* with edge e_2).

   Verify that polygons *A* and *B* have edges that lie on top of edge e_2 and that each of them has a vertex that lies at an endpoint of the edge (see vertices *x* and *z*). In other words: edge e_2 must be spanned by starting at a vertex from *A* and ending at a vertex from *B* while crossing the shared vertex *y*. **If** this is not the case **then** reject it.

   Find the edges from polygon *C* **that** are connected to edge e_2 (see edges e_L and e_R).

   Verify that polygons *A* and *B* each have an edge that is collinear with either e_L or e_R (we need this constraint to make sure that the polygons that result after the merge are still convex). **If** this is not the case **then** reject it.

2. Cutting:

   Determine which of the polygons *A* or *B* has the vertex that lies on their collinear edge e_2 and is closest to shared vertex *y* (in our example this is polygon *A* with vertex *c*). We then know that we are going to cut polygon *B*.

   Find the edge in polygon *A* that ends in vertex *c* but is not collinear with edge e_1 (see edge e_C). This edge will be used to determine the cut direction.

   Cut polygon *B* **into** separate parts B_1 and B_2 and verify that the new vertex *n* is collinear with edge e_R and also lies on an edge of polygon *B*.

3. Merging:

   Merge the resulting polygons **by removing** any unnecessary edges such as e_1 and e_2 (basically we replace vertices *x* and *z* by vertices *n* and *m*).

Algorithm 2: How to perform a 3 → 2 polygon merging.

4.4.1.2 Optimization: 3 → 2 merging

In the previous subsection we have seen how we can merge all the 'easy' polygons, the next step is to merge the 'hard' ones. If we look at Figure 32, we can see that we could potentially merge the 3 polygons *A*, *B* and *C* into a larger surface area if only we could somehow cut polygon *B* into 2 parts. The entire process consists of roughly 3 steps as illustrated in Figure 33 and Algorithm 2. Afterwards, we could also opt on running another Hertel-Mehlhorn merge step on the newly created polygons to possibly further improve upon the result.

4.4.1.3 Culling trivial polygons

The merging algorithms we have seen in the previous chapters will undoubtedly improve the overall quality of the NavMesh quite a lot. However, we will most certainly have to deal with some edge-cases that just won't let themselves be 'ironed out', so to say. Later on we will see that notably very thin and elongated polygons will cause trouble for our path-finder and its smoothing algorithms

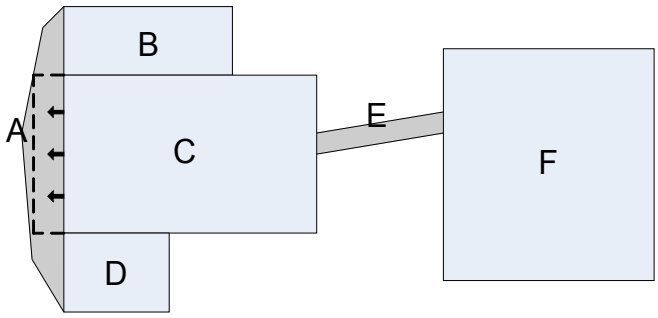


Figure 34: Polygon A does not reflect any significant map area so we could consider removing it, and possibly inflate polygon C a bit leftwards. We cannot inflate polygons B and D likewise because this will expose areas that are very likely not passable for a traveller. We cannot remove narrow polygon E because it forms a vital link between areas C and F.

For a given set of unique traveller radii *TR* all larger than 0:

Add 0 **to** set *TR* and **sort** in ascending order.

For each radius *R* **in** *TR*:

Mark all polygons by default **as**: 'accessible for radius *R*'.

Deform hit-volumes **into** cutting volumes for the given radius *R* (the step distance *SD* can be derived from *R* if needed).

Cut the polygon meshes **with** the deformed hit-volumes (do not remove intersections unless the radius *R* = 0!)

Mark all the polygons that are inside the deformed hit-volumes **as**: 'not accessible for radius *R*'.

Construct the NavMesh graph.

The maximum accessible radius for each NavMesh graph edge is the smallest one of the 2 polygons between which it is spanned.

Algorithm 3: Generating NavMesh topologies for multiple traveller radii. The reason we treat radius R = 0 as a special case is because it will speed up the entire process by getting rid of a lot of polygons very early on in the process

(just peek forward to Figure 49 on page 57 for a moment). In some cases, these polygons actually add little of value to the overall NavMesh and could just as well be removed entirely. Small polygons that run along walls for example will not cause any noticeable anomalies if these would be omitted. In some cases we might even allow a polygon to 'deform' a bit and just forcefully 'merge' them by removing a small polygon and inflating one of its neighborings a bit. Of course we must be very careful about doing this, the whole point of generating NavMeshes was to exactly know where a traveller could and could not stand. If we apply any deformations we must make sure that we do not break any 'rules'. So, in general, it is only safe to reduce the total area, but never expand it beyond its original polygon boundaries (also see Figure 34).

We must of course make sure that we never remove polygons that are vital links between areas because they are the only ones connecting them. Remember that we might end up with very narrow but vital polygons when we use larger traveller radii, shown for example in Figure 23 on page 26.

It is not easy to give a clear-cut culling algorithm. A lot will depend on finding some sensible heuristics and experimentation to find acceptable 'magic numbers'.

4.4.2 Multiple traveller radii

So far we have only generated the NavMesh for a single traveller radius. For a real-world scenario we would rather support multiple dimensions of travellers. In order for the NavMesh to support this, we need to add meta data that tells the path-finder the maximum allowed dimension of a traveller that wants to cross particular NavMesh graph edges. We can generate this meta data by repeating the cutting process for a given number of traveller radii on the polygon meshes, modifying the generation process described in section 3.4. This modified generation procedure for a given set of unique traveller radii is defined in Algorithm 3.

Path-finding trough the NavMesh itself is extended by first testing if NavMesh graph edges are 'wide enough' before the path-finder is allowed to flood across it. Supporting a very wide scale of traveller radii is also just a case of pre-computing a fixed number of radii and then finding a best match by rounding upwards towards the nearest fixed radius.

4.4.3 Meta data for smarter travellers

Having a graph data structure as a basis for the NavMesh has some other advantages compared to, say uniform grid representation. It is very easy to add (sparse) meta data to the graph edges and nodes that can tell a potential traveller more about how to travel across them. We can tell it for example where edges are that are part of ladders so that it knows that it should display a 'climbing' animation when it crosses them. Similarly we can tell it where doors are so the traveller can open them in order to reach its destination. This will greatly improve realism when for example chasing after a human opponent. We can extend this even further by telling it that there are certain conditions that need to be met before an edge can be traversed in a certain direction. For 2 rooms that are connected with an elevator we could tell the traveller that it needs to press a button first and then wait for the elevator to come down. The elevator cabin itself can then be represented by a special node with a rule attached to it saying it can only be entered if the door has opened.

The creation of this meta data needs some human intervention: a level designer will need to place markers at specific locations from which we can derive what the special rules will be. This marker should also cover a certain area that defines where the rules will be in place or where the traveller needs to go to in order to perform special actions (get close enough to the elevator button, for example). Similarly, for doors we need to know which area exactly can be toggled between 'opened' and 'closed' states in order to create the meta data. With these areas known, we can use their volumes to make extra cuts in the NavMesh polygons as part of a post processing step before the actual generation of the graph. These cuts will then be automatically converted into edges whereby we can store meta data on the nodes that are inside the special area, as shown in Figure 35.

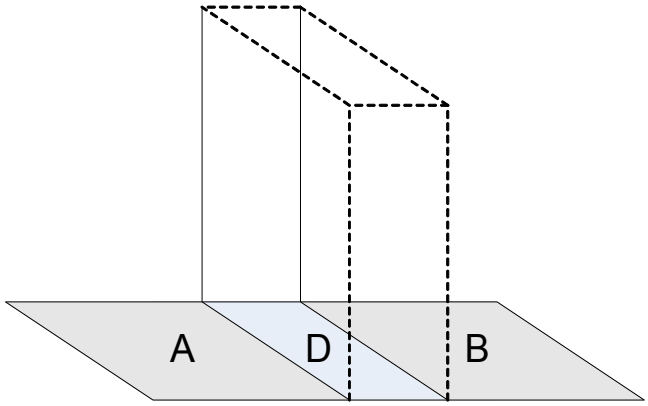


Figure 35: Example of how doorways can be defined and incorporated into the NavMesh. By cutting along the boundaries of the doorway volume we can create new polygons which we can mark with special meta information. During conversion to edges and nodes, the NavMesh generator can copy the meta information so that a traveller knows that polygon D has special traversal rules attached to it in relation to doorways.

4.4.4 Graph hierarchies

Because a NavMesh is already a nice geometrical abstraction using a graph, it should be possible to extract more information out of it. I think there is a possibility to create graph hierarchies which could potentially speed up any path-finding operations. If we first try to find a path through a high-level graph we can quickly bail out if no paths are available at all. This will save us a lot of computing power because if no paths exist then the path-finder will normally have to extensively flood the low-level graph before it is ultimately forced to give up. This potential speed up becomes very relevant when we consider that doorways and such can be opened and closed 'at random' in a game. By using a high-level graph that also includes a node representation of these doorways we can easily increase the intelligence of the travellers without any serious computational overhead. If a traveller is for example a dumb zombie that cannot open doors on its own, it will just idle without eating much CPU power until we determined that a high-level path suddenly became available. Similarly, if we include the meta data as was discussed in subsection 4.4.3 we can also create a light-weight decision process for a traveller when multiple means of travel are available to choose from. If we have a sky-scraper with stair-cases and elevators, for example, we could represent the average amount of time it would take to use either of them as part of meta data. Using a path-finder on a high-level graph with this kind of information we could easily let it determine if it should take the stairs, or wait for the elevator.

There are many ways to generate such hierarchies, both manually and automatically. I think that a Girvan-Newman hierarchy clustering algorithm might be a good candidate to do this automatically (see [6]). The algorithm takes as input a graph and corresponding edge weights that represent an abstract measure of 'connectivity'. Using this information it will cluster the graph which in turn we can interpret as high-level 'hubs'. The high-level graph is then easily constructed by representing the 'hubs' as nodes and linking them with edges if there are direct connections between them.

The 'connectivity' rating could be constructed in a number of ways. As said before, it could be obtained via limited user input if needed. If a user just defines a small number of 'hot-spots' in the maps with a special marker, then we could simply use the distance to that marker as a 'togetherness'

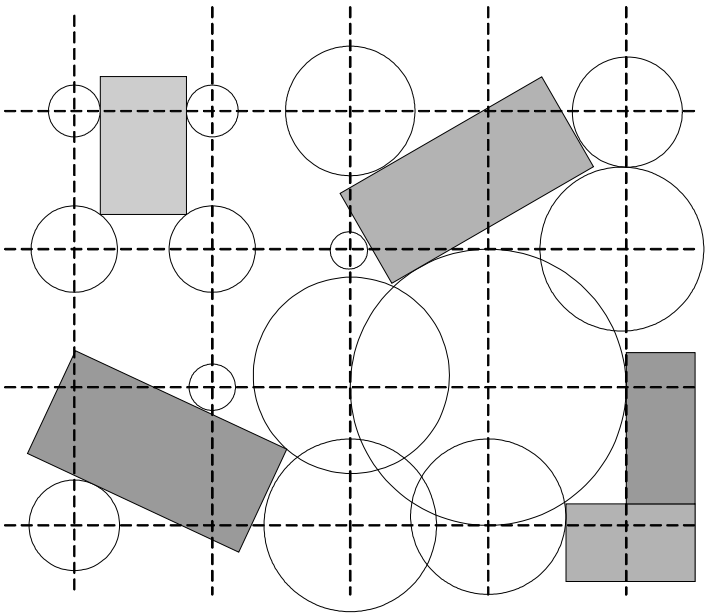
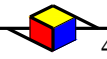


Figure 36: Example of a probabilistic roadmap sampling grid.

measure to form clusters. We could also opt for a completely automatic approach whereby we obtain the measure from the NavMesh itself. We could for example measure how far neighboring polygons stretch 'outwards' for a given polygon. If there appears to be a lot of room around it, then this probably means it is part of a larger chamber, a town square or perhaps a high-way junction. Giving these polygons a higher 'togetherness' radius might be enough to generate adequate clusters. To measure the degree of space around a polygon we could opt for a 'Probabilistic Roadmap Method' approach (see [1] and [2]). Such an algorithm just samples the maximum radius of a sphere it can place at uniform intervals, as illustrated in Figure 36. Perhaps the Probabilistic Roadmap Method could already generate information from which we directly can obtain high-level graphs. Or perhaps we can use polygon's medial axis to identify corridors and/or road-ways. This is, however, the topic of future experimentation.

4.4.5 Flying travellers

Although we have primarily focused on travellers that walk through the world, it is fairly easy to extend the use of the 'PM_Space' container class library for jumping and/or flying entities. Instead of filtering all the upwards oriented faces as a means to extract NavMesh data, we should instead start off with a single 'clearance' mesh that covers the entire map space. Then, each time when new hit-volumes are added to the 'PM_Space' container the clearance mesh will be fractured into smaller convex meshes. When all the volumes have been processed, we know all the areas that are 'open' and could be traversed by a flier of some sort. Because we keep track of all the adjacency information for polygons and such we can now construct a NavMesh in similar fashion. Instead of linking polygons with shared edges we now link meshes with shared polygons.



5 World navigation

Although a NavMesh gives us a nice and clean representation of a static world and a means to traverse it, this is only half of the solution. This chapter will focus on techniques that will help further shape realistic paths through the static world.

5.1 NavMesh path-finding

Path-finding through the NavMesh starts by finding the start and end polygons on which the start and end points are located. If we do not have any predetermined context then we will have to start from scratch and do a number of 'point-in-polygon' tests (using an octree here is obviously advisable). Normally though we should always keep track of a traveller's position on the NavMesh while it is moving around. That way we can easily reorient ourselves when all of a sudden a new path needs to be found from its current location.

The actual path-finding can be performed using any of the common path-finding algorithms that take directed graphs as input. My studies showed that the so-called 'A*' algorithm is by far the most popular today. A* is basically a flooding algorithm that uses a heuristic function to restrict the search space and to turn it into a 'best first' strategy (as in: always select the 'best' candidate node first). The algorithm is very intuitive and flexible, it can easily be modified for specific needs. The inner workings of A* will be discussed later in chapter 7, as well of some of the more common implementation variants. For now it will suffice to say that A* will enable us to construct the cheapest path through a directed graph, which in our case is the NavMesh graph.

The cost of each NavMesh graph edge is in its basis the euclidean distance between the polygon's centroids. Depending on the type of game, we may apply additional weighting schemes in order to influence the path-finder's behavior. We might for example incur extra penalties when connections are going 'up-hill' or give cost reduction for connections that are going 'down-hill'. The euclidean distance will in general suffice but might possibly cause anomalies for large polygons as depicted in Figure 37. In such cases the computed costs are actually higher than they intuitively should be; the path-finder might then decide that certain routes are too expensive and come up with other less obvious routes.

The only way to reduce the amount of error in cost estimation would be to subdivide the polygons more so that more centroids will appear and get closer to each other. This is however in direct contradiction with what we are trying to achieve using NavMeshes: less nodes/polygons to search through. During my experiments the potentially incurred error oddly enough never proved to be so problematic that it required some sort of countermeasure. The reason for this is probably that NavMeshes closely represent corridors, which means that the cost error will never be so large that the path-finder will erroneously select a different corridor, as the cost of these are often way too high. Also, my current NavMesh generator does not fully support polygon re-merging as part of a clean-up phase, so the polygons could just not be 'large' enough yet. Still, the impact of the problem could potentially be large and thus warrants a solution.

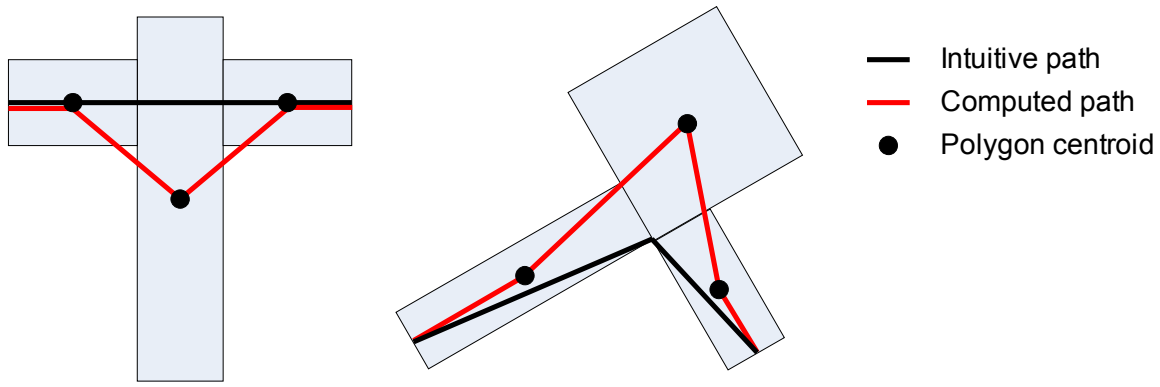


Figure 37: Some examples whereby computing costs as the euclidean distance between polygon centroids might contribute to noticeable errors.

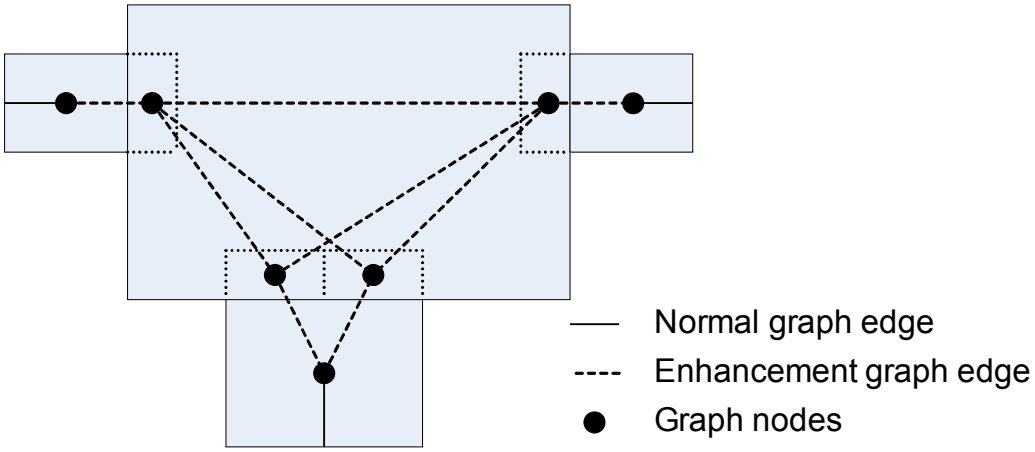


Figure 38: By adding more graph nodes and edges we can 'enhance' the resolution of the NavMesh polygon without actually subdividing it. We might even need to apply the enhancements near neighbouring polygons that are very wide (such as the one at the bottom).

A possible improvement could be to only change the graph representation of extremely large polygons in the NavMesh. We could create additional graph nodes near the entry points as a sort of 'resolution enhancement'. We do not need to subdivide the polygon itself, we only need to represent it in a different way to the path-finder so that it is tricked into finding more natural paths. By fully connecting these nodes, we can now more accurately model the costs for traversing the polygon in all possible ways (see Figure 38). Note that we might even need to 'subdivide' the entry points as well in case the neighboring polygon is very wide (think of wide roads). By only changing the representation of the polygon in the NavMesh graph we do not incur any 'noise' that might thwart the path-smoother, which will process the found path solution afterwards. We can just let the smoother ignore any graph edges that are not crossing any polygon boundaries. Also, we do not need to store more polygon information in the NavMesh, only just some more graph nodes and edges.

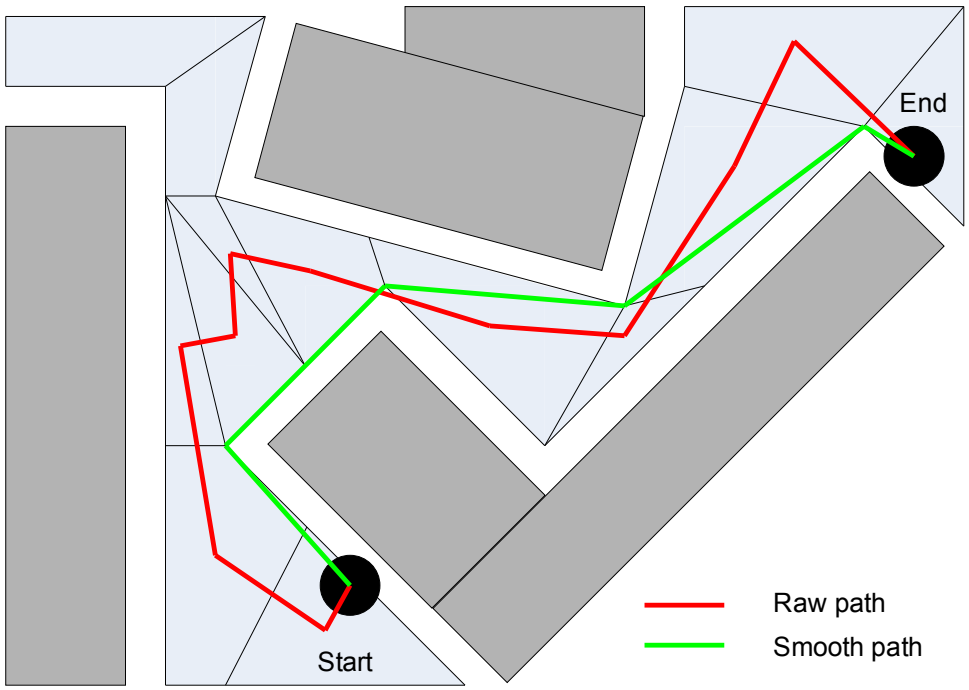


Figure 39: A raw path-finder solution connected through the centroids of the NavMesh polygons will give a very unnatural result. The path needs to be smoothed first by trying to 'touch' the outer edges of the polygons where needed. We know this is safe because the NavMesh has already been corrected for the traveller's dimensions; we will thus never accidentally come too close to surrounding geometry.

5.2 Path smoothing

The input for the path-smoother is just a straightforward solution set of NavMesh graph nodes and graph edges that the traveller should traverse in order to move from a point A to a point B. Although our path-finder has ensured us that it is the most optimal path, it is limited by the fact that the input it has taken is still but a simplified representation of the world in which to navigate. So additional data manipulation is required to make a traveller traverse the found path in a realistic manner.

5.2.1 Corner reductions

The first thing we need to do is to reduce the amount of corners a traveller has to make in order to traverse the found path. The raw path-finder solution data set only tells us how to move from one polygon to the next and which of the polygon edges to cross. Just 'connecting the dots' using the polygon's centroids will often result in a very erratic movement path, which is unacceptable, as we can see in Figure 39. Especially large polygons near corners can cause serious anomalies because their centroids can pull the traveled path far off course. In the 'real world', a traveller will usually cross a space by looking ahead and walk straight towards the next corner it can see. We now need to mimic this type of behavior by parsing the input set in a similar fashion.

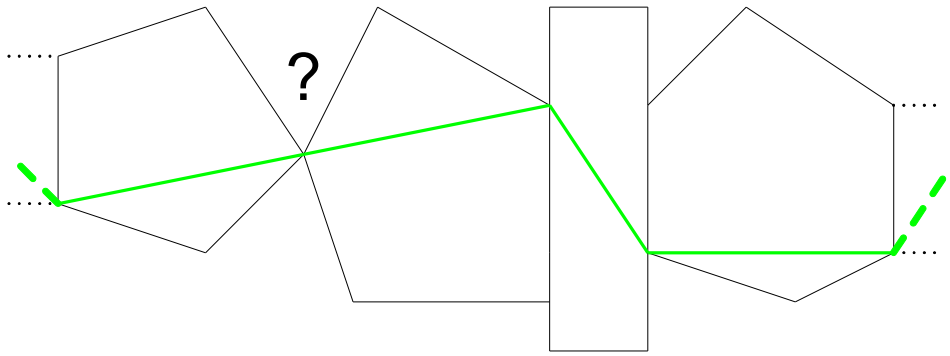


Figure 40: A naive approach would be to path-find through polygon vertices only. This however still leads to erratic paths and always leaves the question if polygons that only share a single vertex can be crossed by the traveller at all.

Now, we might be tempted to think that a lot of our problems can be solved by having the path-finder search through polygon vertices instead of polygon edges. That way we will automatically be able to generate 'smooth' paths around tight corners. Although this is true, we will still not end up with natural paths, as illustrated in Figure 40. Besides, it is always dubious if a traveller is allowed to move between polygons that only share a single vertex. We could partially amend this problem by only creating connections between such polygons if they have one ore more neighboring polygons that share the same vertex and together form an actual 'physical' link. That would still leave us with the problem of how to deal with polygons that we would rather cross by entering and leaving it via arbitrary points on its edges. For this we still need a more refined parsing algorithm that can process polygon edges instead of just vertices.

In order to created smoothened paths through a NavMesh we are going to need a special path smoother module. This module processes the raw path solution in 2 phases:

- Corridor detection.
- Way-point reduction.

During the first phase we will determine where exactly we should cross polygon edges when traveling them. During the second phase we will clean up the found way-points a bit so that we obtain a minimal set of way-points. This will help reducing additional costs when, for example, we are going to apply spline fitting to generate natural paths for travellers to follow. If a path would run across a number of small polygons that we could cross in a straight line, then having a dense set of way-points might unnecessarily force a spline to follow a tightly restricted path (more on this later in this chapter). It will also help us to determine where the actual points are where a traveller would need to turn into a new direction. This will assist us in all sorts of secondary actions whereby we might want the traveller to start looking in the new direction of movement, anticipate a reduction of speed for taking tight corners, etc.

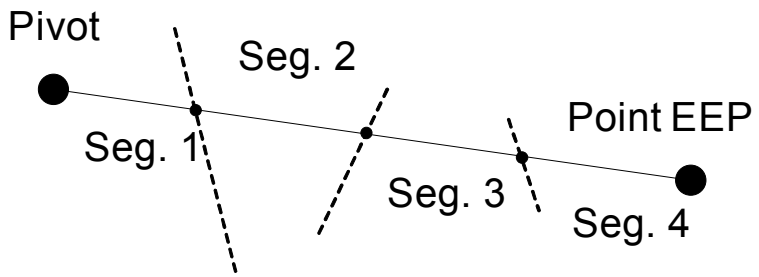


Figure 41: With 3 bumps we will generate a total of 4 segments between the Pivot point and point EEP. To find the intersections with all the bump line-segments we simply use a vertical plane. Later on we will see why it is guaranteed that each bump will have an intersection with the plane.

EmitTowards(end point *EEP*):

It is implicitly assumed that successive emissions form a path whereby segments are connected 'head to tail'. That is why we always start generating the new line-segments at the current *Pivot* point..

If *Pivot* = *EEP* **then** nothing will be emitted.

Else we will generate smoothed path segments in a straight line between the *Pivot* point to point *EEP*.

> **If** any bumps have been generated **then** we first **find** the intersection points with them using a vertical plane that intersects the *Pivot* point and point *EEP*. The new smoothed path segments will be generated in the order that we find bumps in the queue (see Figure 41).

After each emission we will restart the smoother for the next segment of the path: *Pivot* = *EEP*.

The queue of bumps will be cleared again.

Algorithm 4: The EmitTowards() function will emit actual path segments.

Now, the first phase of the process is by far the most extensive. During this process the smoother will keep track of a '*Pivot*' point from which smoothed path segments are constructed by scanning the raw path-finder solution in a forward direction. During this scanning process, the smoother will regularly store 'bumps' which are parts of polygon edges for which we only know we will certainly cross them, but not yet exactly where. A bump is simply represented by a pair of points in 3D space (which coincide with the edge vertices). Afterwards, during an actual 'emission' when we finally know the exact direction in which the smoother has moved we can convert these bumps into actual smoothed path line segments. The entire process utilizes a small number of functions which we will now discuss one by one.

The first function, called EmitTowards(), will generate actual path segments during the smoothing process (see Algorithm 4 and Figure 41).

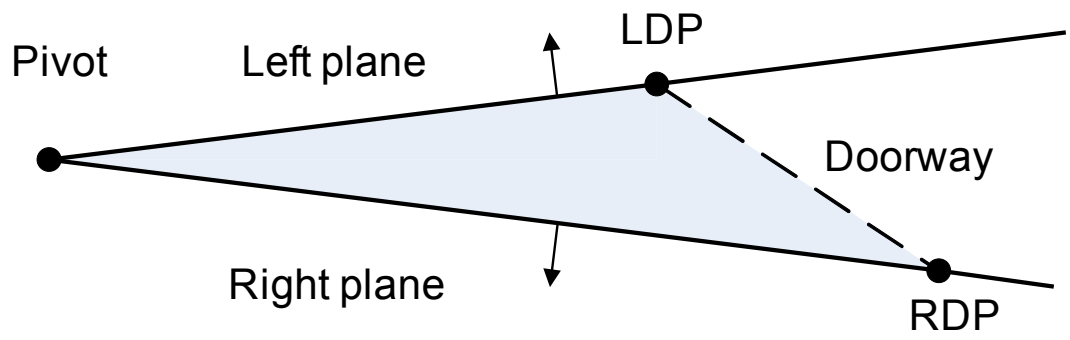


Figure 42: Top view of a corridor in the path smoother. The corridor consists of a pivot, a doorway line-segment and 2 planes that are always aimed away from each other.

DefineCorridor(point *P*, point *A*, point *B*):

P will become the new *Pivot* point, *LDP = A*, and *RDP = B*.

If it is not implicitly clear if *LDP* and *RDP* are specified in the correct order **then** this has to be ascertained using a default upwards vector and some cross-products.

In our case, points *A* and *B* are always polygon vertices, so as long as we can guarantee that the specified *Pivot* point is somewhere in this polygon then we automatically know *LDP* and *RDP* are correct when the NavMesh's polygon edges are defined in clockwise winding order.

Reconstruct the left and right planes (both of them run vertically in 3D space).

If *LDP* **or** *RDP* are the same as the *Pivot* point **then** immediately emit a new segment:

Depending on if *LDP* or *RDP* is furthest from the *Pivot* we will EmitTowards(*LDP*) or EmitTowards(*RDP*) respectively.

The state of the smoother module will be set to '**Singular**' again.

Else if both *LDP* **and** *RDP* are the same as the *Pivot* point **then**:

No actual corridor is defined so we need not do anything special.

The state of the smoother module will be set to '**Singular**' again.

Else:

Add a new bump to the back of the queue that spans from point *LDP* to *RDP*.

The state of the smoother is set to '**Corridor**'.

Algorithm 5: The DefineCorridor() function is used to specify search areas through which we will construct smoothed path segments.

CollapseCorridor(steering point *SP*):

Given the 'steering point' *SP*, assign a so-called 'closure point' *CP* based on:

If *SP* is in front of the left plane **then** *CP* = *LDP*.

Else if *SP* is in front of the right plane **then** *CP* = *RDP*.

Else *CP* = *SP*.

EmitTowards(*CP*).

The smoother's state is set to '**Singular**' again.

Algorithm 6: The CollapseCorridor() function is used to emit path segments when corners are being processed.

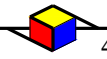
The smoother module itself is always in either of the following 2 states:

- **Singular**: The smoother only tracks the single *Pivot* point in the world. From this point we will try to construct new smoothened path way-points.
- **Corridor**: The smoother contains a 'corridor' definition that is basically spanned by 2 planes: a 'left' and 'right' one (see Figure 42). The intersection of both planes and one of the NavMesh's polygons define the start way-point of the corridor (which is the *Pivot* of course). Next to the planes we also keep track of a line-segment that defines the end/exit of the corridor which we will call its 'doorway'. The start point *LDP* of the doorway line-segment is always located on the left plane and the end point *RDP* on the right plane.

The **Corridor** state helps us to define the area in which a traveller could potentially move in a straight line unhindered. A corridor is always defined by 3 points using the DefineCorridor() function as described in Algorithm 5.

The main mechanism of the smoother is to try to generate successive corridors by using the polygon edges that we cross when we walk from one NavMesh graph edge to the next. Each time we add such a polygon edge it will become the new doorway of the corridor. If the polygon edge was (partially) smaller than the last doorway, then the corridor planes will come closer together. At some point we would inadvertently want to add a polygon edge that no longer fits in the corridor. At that point we are going to 'collapse' it, which entails the CollapseCorridor() function as defined in Algorithm 6.

Next we define the operations we let the the smoother perform to process the raw input data. We can either add a single point to it, or a line segment (needed to represent polygon edges). Depending on the current state of the smoother the AddPoint() and AddEdge() functions process successive polygons that are crossed (see Algorithm 7 and Algorithm 8).



AddPoint(point *NP*):

If smoother state = '**Singular** ' **then**: EmitTowards(*NP*).

If smoother state = '**Corridor**' **then**:

CollapseCorridor(*NP*).

EmitTowards(*NP*) (in case *NP* was not inside the corridor).

Algorithm 7: The path smoother uses the AddPoint() function to process single points.

AddEdge(start point *SP*, end point *EP*):

If smoother state = '**Singular** ' **then**

DefineCorridor(*SP, EP*).

If smoother state = '**Corridor**' **then**

Clip line-segment *SP* to *EP* using the corridor planes, the result is line-segment *SP'* to *EP'*. Note: it is possible that *SP'* and *EP'* become singular (also see Figure 44 later on).

If the clipped line-segment is inside the corridor planes **then**

Narrow the corridor: DefineCorridor(*SP', EP'*).

Else

If line-segment *SP* to *EP* is in front of the left plane **then**

CollapseCorridor(*LDP*).

Else line-segment *SP* to *EP* must be in front of the right plane so

CollapseCorridor(*RDP*).

Algorithm 8: The path smoother used the AddEdge() function to process polygon edges that need to be crossed.

Given a start point, end point and a sequential list of polygon edges that should be crossed:

Initialize smoother:

Set state to '**Singular**'.

Pivot = start point.

Add first 'bump' at start point.

For all polygon edges: AddEdge(edge)

AddPoint(end point)

Algorithm 9: The full path smoothing process.

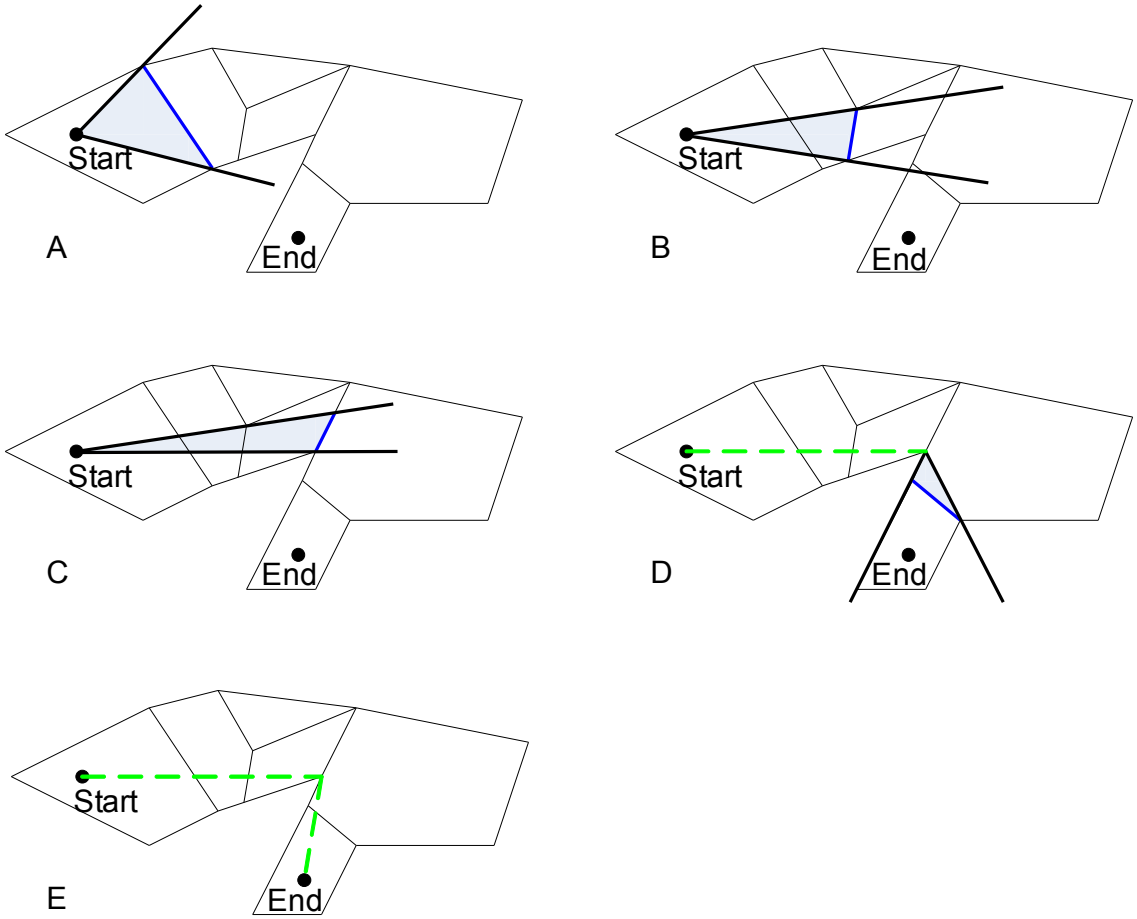


Figure 43: An example of how the path smoother uses the corridor (blue lines) and doorway (greyed areas) algorithm to traverse the raw polygon input. The large polygon on the right of the mesh now no longer pulls the traveller way off course. It is 'tightly cornered' instead because the corridor is collapsed in the transition from step C to D.

Having rigidly defined all the 'tricky' stuff, it is now very easy to describe the entire smoothing process (see Algorithm 9 and Figure 43). Note that if there are no polygon edges to traverse this means that both the start and end point are located on the same polygon. In this case the smoother will correctly return a single straight line-segment from start to finish. Also, if so desired, we could use AddPoint() when polygon vertices are actually part of the raw path-finder solution.

Note that it is allowed for the doorway to become 'dimensionless' (see Figure 44). This will occur when crossing tight corners whereby the next polygon edge just touches the corridor planes. When the corridor is then redefined both corridor planes will become coplanar. This is all expected behavior and will generate correct way-points.

This is all nice and well for 2D but how does this work in 3D when all the crossed polygon edges can have all sorts of orientations? The smoothing algorithm described so far has always kept its corridor planes vertically, which basically makes it a sort of '2D' projection. This does not cause any problems because normally a traveller cannot stand on vertical polygons anyway and thus we never

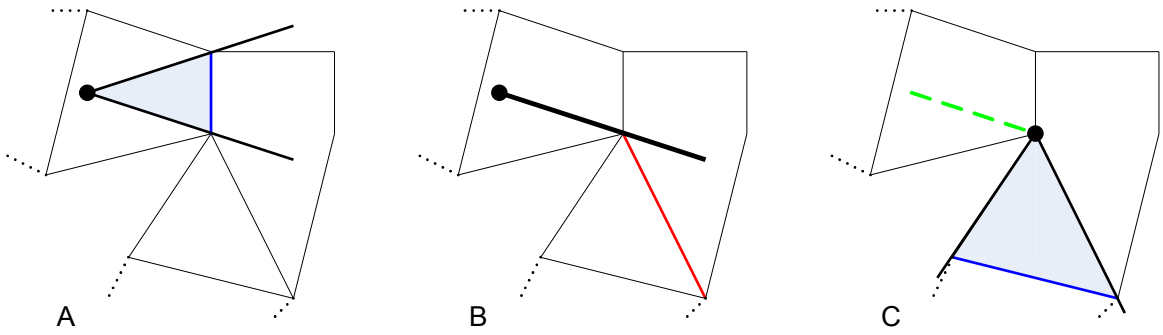


Figure 44: The red polygon edge only intersects the corridor at exactly one of its planes. This will temporarily compress the corridor's doorway into a single point. This is expected behaviour and will automatically generate the next way-point when the next polygon edge is added.

Given a maximum allowed horizontal deviation angle α and vertical deviation angle β:

Emit a way-point using the first bump.

For *i = 0* (= first bump) **to** *n* (= last bump).

Take bump *i* as pivot.

Construct a vector *V* towards bump *i + 1*.

Scan successive bumps *i + 2, i + 3,* and so on **until** we find a bump *i + j* that has too much deviation with vector *V* (check with threshold angles α and β) **or until** the last bump is found. Emit a way-point using bump *i + j - 1* or the last bump.

Algorithm 10: The path smoother will discard a lot of the 'bumps' and leave the actually relevant way-points.

expect to process vertical polygon edges either. However, each of the processed polygon edges might be in all sorts of positions and semi-diagonal orientations. So we have now arrived at the second phase whereby we will discard a lot of the 'bumps' we have collected so far and convert only the relevant ones into actual way-points. The algorithm for doing so is straightforward and is described in Algorithm 10 and Figure 45.

A small detail that we now might need to address is the fact that the NavMesh might contain graph edges that are the result of the gap welding stage. If we look at Figure 46, we see that such a graph edge might cause a 'glitch' in the movement path for staircases. The entire path smoothing algorithm that we just discussed will be able to deal with the welded gaps perfectly fine. Whenever we encounter such a welding graph edge, we will just add both polygon edges to the smoother: the edge of the polygon where we leave from and the one where we will arrive at. During the gap welding stage of the NavMesh generation we already figured out which edges were close enough for welding, if we store these as part of meta data in the welding graph edge then there will be virtually no additional processing costs. Because the gap welding distances should already be kept to a minimum, I found that the resulting path did not suffer any noticeable movement glitches for such unfortunate cases as already depicted in Figure 46. If one would want to be sure that potential

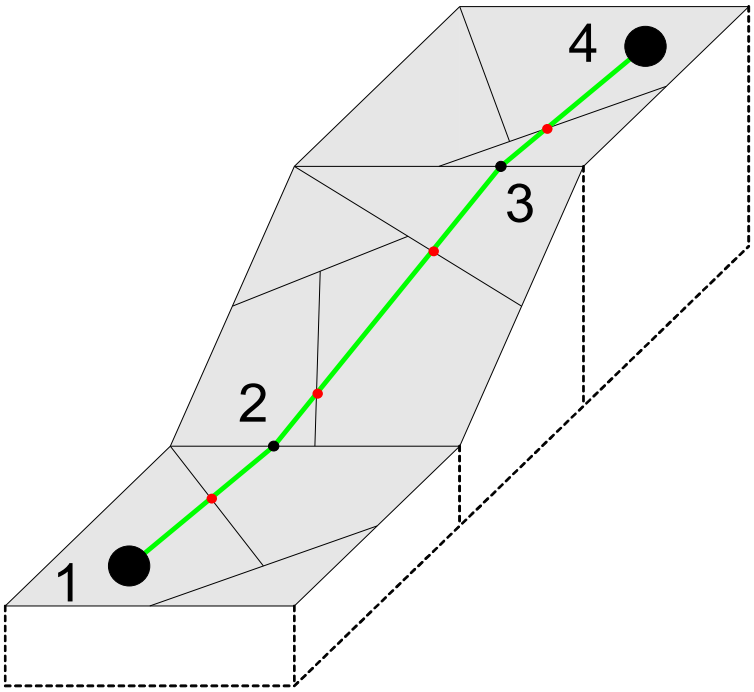


Figure 45: During the sanitization phase superfluous bumps have been rejected (red dots) which leaves the minimal amount of way-points (black dots) that define the curvature of the smoothed path through 3D space.

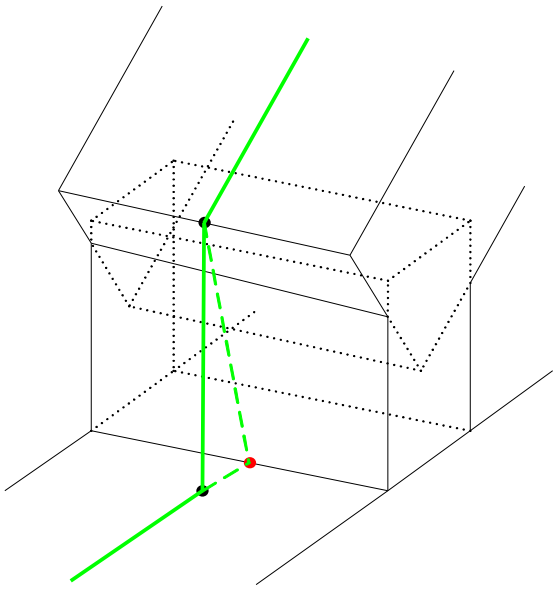


Figure 46: The NavMesh graph edges that were generated during the gap welding stage might now cause problems.

glitches are sanitized from the movement path, then one could try to drop a 'lead line' and find the intersection point with line-segments between previous bumps. We could then insert new bumps and remove the problematic ones before entering the actual way-point reduction phase. In many cases this should work well because during the corridor detection phase we automatically generate bumps that all form a coherent line when viewed from above. However, there is always the possibility that a gap welding edge is really close to a corner, or that other more exotic geometry configurations might prevent us from finding a clear point of intersection. In those cases we might need to resort to more fuzzy methods to try and sanitize the path in a satisfactory way.

It is important to note here that cases such as depicted in Figure 46 do not need to cause any problems at all. In the next chapter we will see that if the traveller is purely physics driven it will suffice to 'look ahead' along the smoothed path and find a 'magnet point' to pull the traveller towards. The unfortunate welding edges will then be skipped/traversed automatically.

5.2.2 Travel motion

The emitted route segments resulting from the smoother module will now represent a much more sanitized path to follow for a traveller. Depending on further requirements we will need to apply some additional 'post processing' in order to make the traveller walk through the world in an entirely natural way:

1. We could use the route to construct a spline curve such as a B-spline or Catmull-Rom spline so that the traveller does not 'criss-cross' linearly through successive way-points.
2. We could make the traveller completely physics-driven (with a certain amount of 'sluggishness' due to its mass), and just 'lure it along' the way-points of the route.
3. Just follow the route directly and use (turn) animation smoothing as polish.

Option 1 has the disadvantage that the traveller is pulled slightly off-track. Normally this is not a problem but it is not inconceivable that this might cause clipping with the surroundings due to the curving nature of the spline (see Figure 47). Usually, a bit of clipping with a wall is something we can live with as long as it is not too obvious. However, it might just as well bring a traveller too close to some sort of hazard and accidentally take damage from it. The amount of error that is introduced using splines is of course controllable by the 'order' of the spline: the amount of curvature it generates at corners. A trickier problem is that although it is natural to sway off course a bit in the horizontal plane, it might look a bit weird when this also happens in a vertical direction (see Figure 47 again). This problem is solvable by making the traveller scanning downwards continuously to find a solid point beneath its current path position (ray-casting into the world for example), but this will induce more performance penalties. Also, this might cause problems when traversing gaps that have been welded. In those cases we might be forced to linearly interpolate between the polygon edges of the gap in order to 'guess' an acceptable resting point.

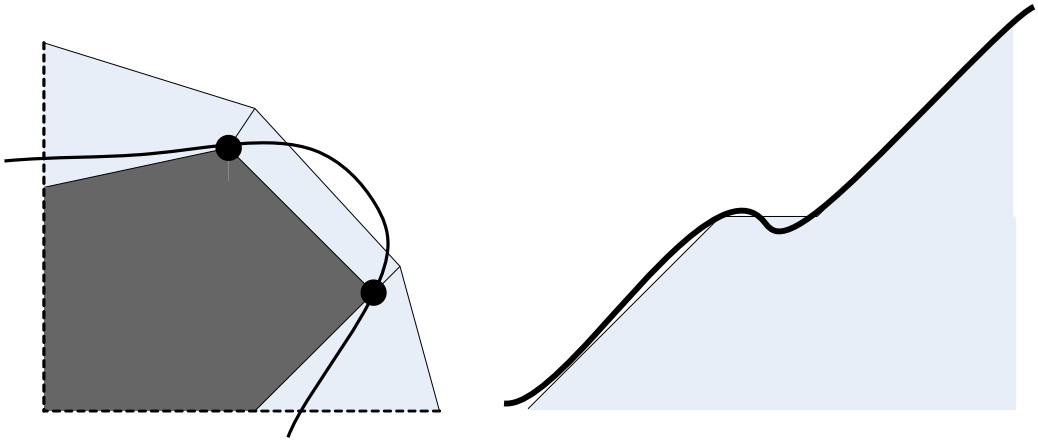


Figure 47: Curve fitting through found paths might result in awkward deviations.

The problem of path deviation in combination with hazard avoidance can be mended by keeping track of both positions on the linear path and the curved path. All collision and hit detection is then performed using the linear path position for which we know that the path-finder has made sure that it does not run through, for example lava, or other sorts of dangers. Usually, the path deviations are not so bad that this will cause inconsistencies in the game-play experience.

With option 2 we use actual physics to move the travellers around. We do this by simply generating a 'magnet' somewhere ahead of the traveller on the path it needs to follow. Then we apply a force in the direction of the magnet to push it towards it, generating successive magnet points as the traveller progresses down the path. How far forwards we should scan is a matter of heuristics. If we pick it too close to the traveller then it will still very strictly follow the linear path segments which sort of defeats the purpose. If we pick it too far then we will run the risk of forcing the traveller into taking short-cuts. In some unfortunate cases, this might also actually get the traveller stuck (see Figure 48). A solution to this is to temporarily decrease the forward scanning distance when the traveller appears to be blocked. This could be done quite naturally by linking the forward scanning distance linearly to the traveller's movement velocity (with a certain fixed maximum forward distance). The use of a magnet point can be extended further and also have the traveller apply braking forces when getting too close to the magnet. That way we can create more natural motion flows by decreasing the distance between the traveller and the magnet when detecting tight corners.

The usage of full physics will result in the most natural behavior but will also induce a greater amount of 'noise' (and of course a much larger performance penalty). We will have to face the same problems as with curve fitting because a traveller might stray into a fire hazard or even walk off a cliff. It all depends on the nature of the traveller if this is acceptable or not. If you are fighting a horde of dumb zombies, for example, this might actually be comical. For a lot of first-person shooters the enemies often already have a great deal of fuzziness in their behavior in order to make them look natural. They will have to be able to dodge incoming fire at any given moment which means that they are not solidly linked to a movement path to begin with. So the chances of them accidentally running into some sort of trouble are already a lot higher. The severity of mistakes

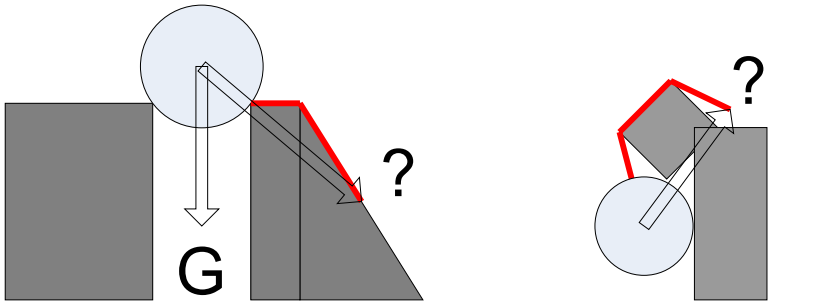


Figure 48: When using full physics, we should not pull the traveller towards positions too far ahead on the path or else it might get stuck near some unfortunate geometry (this is especially the case when other forces such as gravity come into play). This is also a reason why we should be careful on how far to apply gap welding during the NavMesh generation, we do not want to span gaps that cannot actually be crossed.

game entities are allowed to make is often related to the degree of dependency to the player. If this is high, then any noticeable errors becomes an irritation factor. You do not like it when you order your servant to crawl through a small gap in a wall to open the castle door from the inside only to have him accidentally drop into the moat and drown. There are, of course, always possibilities to solve the problem by adding 'invisible walls' near dangerous cliff edges and such. These will often require human intervention which is not at all uncommon. In fact, it is often up to the level designers to work around the weaknesses of the game engine and design their maps in such a way that it carefully avoids possible problems (and/or potential exploits).

I chose option 3 because it proved easiest to implement and still be acceptable for demonstration purposes. Triumph Studio's Overlord game relies heavily on the correct movement of travellers because it contains a lot of hazards such as firewalls and poison wells that the player and his little minions need to circumnavigate. A lot of the interaction with puzzle elements in Overlord is also indirect, in the sense that the player needs to send his minions to specific places and have them, for example, push a button. We found that it was fairly easy to polish the normally linear path movements by using various animation smoothing algorithms (including the turning of the characters themselves). Using option 3 was the best option to use in Overlord because the amount of entities the player can directly control is large and thus computational overhead needed to be minimized. Doing the same for my NavMesh version also gave satisfactory results which is why I stuck with it for now.

In Figure 49 we can see an example of an in-game raw and smoothed path. We can clearly see that the minion follows a natural path over the staircase towards the corner to the right. The NavMesh clearly needs some refinement because part of its path on the upper tiles is of poor quality due to very narrow and elongated polygons. This demonstrates the importance of the quality of the NavMesh clearly, the path smoother will only be able to clean up found paths to a certain degree. The solution to this is to apply a post processing step that will make sure that polygons are generally well proportioned (so that their vertex angles are roughly evenly distributed). This can be

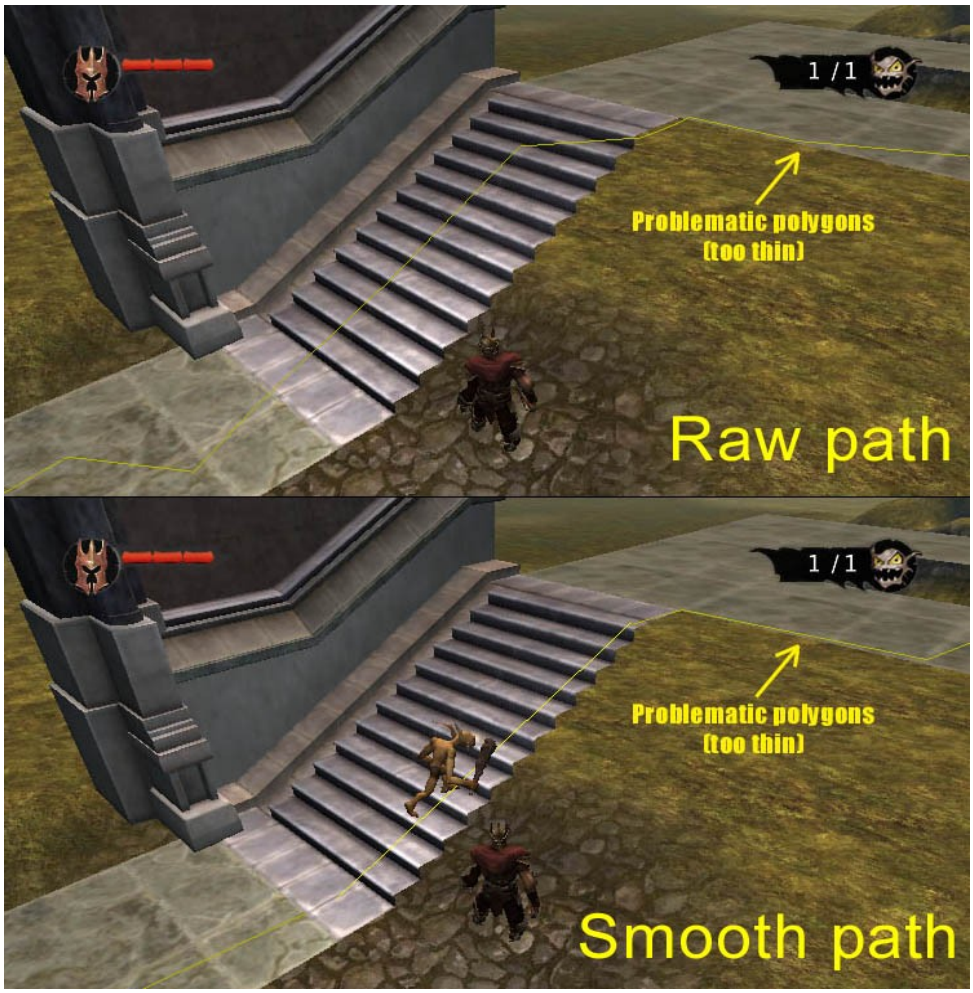
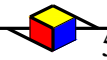


Figure 49: In-game screen shot of a raw and smoothed NavMesh path. Here we can clearly see that the quality of the NavMesh greatly influences the resulting paths. If the NavMesh contains narrow and elongated polygons then the smoother will not be able to generate more natural paths.

achieved by clever merging algorithms such as we have already seen in subsection 4.4.1. There we also discussed that in some cases we could consider removing very narrow polygons because they represent a small space which travellers will hardly ever traverse (provided that these polygons do not form crucial links between other areas). As a last resort we could forcefully subdivide strongly elongated polygons until they are more symmetrically proportioned.



6 Dynamic obstacle avoidance

So far we have only looked at how we can make a single object traverse through a static world. Now we need to determine what needs to be done to do this realistically for a multitude of objects that might bump into each other at any given time.

6.1 Repulsors

As already discussed in section 2.3, it was my intention to implement dynamic obstacle avoidance based on a Boids system utilizing repulsors and whiskers. Such systems have been implemented successfully in a large variety of games because they represent a very natural way to avoid nearby obstacles. They are also fairly easy to tie into meta systems that control group coherency (flocking and such) and goal driven behavior (Boid attractors). As a bonus, such systems are potentially parallelizable for modern multi-core systems because they are in essence a 'particle' engine that has to deal with a variety of push and pull forces. The only downside is that you will need to rely a lot on 'emergent behavior' and need to invest extensive amounts of time on tweaking the 'magic numbers' such as field of view, look-ahead distances, obstacle prioritizations, force strengths, etc. Also, ideally you need a good and solid physics system so that AI entities can easily be 'pushed around obstacles' during their travels.

For a more detailed description of the techniques that are involved for such systems I refer back to my research thesis that was used as a basis for the implementation of the various navigation systems (see [3]). Given time, I would have been able to implement these systems and be able to experiment with them. Due to time restrictions, though, I had to narrow down the work and could only implement a partial (but still vital) technique called 'footprinting', which will be discussed next.

6.2 Footprinting

A disadvantage of the repulsor system as described previously and back in section 2.3 is that it can only help solve very local problems in a rather 'fuzzy' way. It can make an entity not bump into something in front of it but there is never any guarantee that it will help to safely navigate the entity through a larger crowd of dynamic objects. For this you really need to be able to fall-back onto a system that can generate much more deterministic solutions.

6.2.1 Basics

The most obvious choice for a deterministic fall-back mechanism is of course the family of A* variants. Because we can easily control how the cost for graph edge traversal is computed, we can use this to artificially place movement penalties on certain areas. This idea forms the basis of the 'footprinting' mechanism: before we start an A* search session of a given traveller, we let all surrounding dynamic obstacles temporarily 'draw their footprint' by adding weighting values to nearby nodes. A simple example of this can be seen in Figure 50, whereby the graph is actually a uniform grid of nodes (as was the case for the Overlord games).

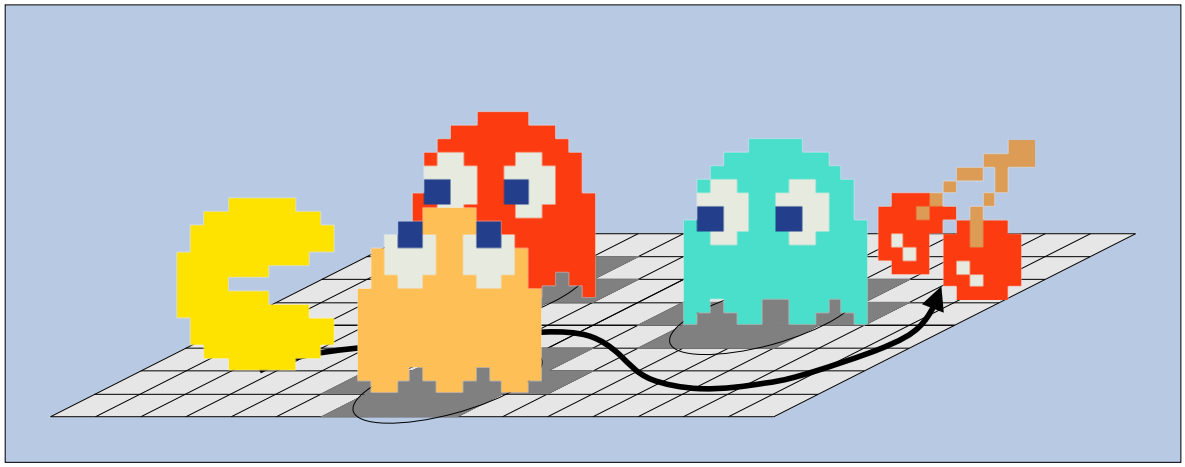


Figure 50: An example of dynamic obstacle footprints around which a path-finder will need to find a path.

Usually we will only place footprints on those graph node that are exactly underneath the dynamic obstacle in question, but we could potentially place weights on surrounding nodes as well so that the traveller will keep more distance to them. Depending on game-play requirements we can choose to make the footprints weighting 'solid' so that the node is actually marked as inaccessible, or just a weight value (> 1) that is multiplied to the cost of traveling towards the node. The latter has the advantage that in extreme cases whereby a traveller is literally stuck between some dynamic obstacles, we can allow it to 'cheat' and still move through them in emergencies. By using larger penalty values, as we get the closer to the center of a dynamic obstacles' footprint 'center of mass', we can force the A* path-finder to cheat as little as possible so that only nodes at the very edge of the footprints are traversed when needed.

It depends greatly on the type of game on how much 'cheating' is allowed. For Overlord, for example, we found that for the Minion creatures that are under direct control of the player it was beneficial to allow them to cheat quite a bit. This removed a lot of player 'frustrations' because it would mean that they would generally respond a lot 'snappier' and that they did not need to wait for friends and/or foes to move out of their way while on their way to important tasks. We experienced that the cheating actually goes largely unnoticed by the player because of how characters are rendered and how fast the Minions moved. Because the footprint volumes are already a fairly rough approximation of their body mass near their feet, you will automatically have some clipping when models are rendered. So when a character does decide to cheat to get itself out of a tight spot, it will just look like it just 'wriggled' its way past the opposing characters. For larger and clearly solid object such as trees and stones, the cheating was, of course, disallowed.

Note that the 'range' of the footprint session should not be too large. Otherwise dynamic obstacles that are rather far way from the traveller will also draw their footprints around which the path-finder will eventually curve. The chances are though that those dynamic colliders will have moved before the traveller has gotten near them; following the computed path will then suddenly make it seem to be 'drunk' because of needless 'zigzagging'. Although the same holds for nearby obstacles this is far less of a problem; by the time the traveller will actually reach the furthest obstacles they will likely

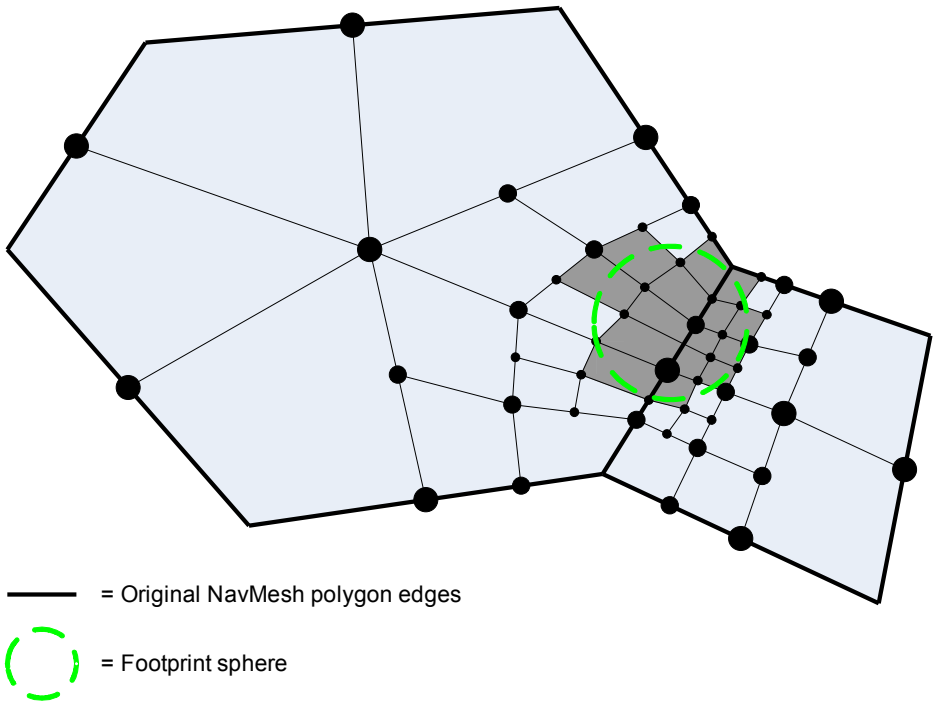


Figure 51: An example of how the resolution of the NavMesh is locally enhanced by subdividing its polygons. The resulting footprint is still 'rough' (= grey polygons) but will be good enough in combination with A path-finding and Boid-like repulsor mechanisms.*

have moved much further from the original footprint spot, which makes it much more noticeable. Often it is best to regularly do path-find sessions when in crowded places, or at least when a direct collision is imminent. This way we have the least amount of chance to do unnecessary zigzagging.

6.2.2 Polygon subdivision

For NavMeshes it is a bit harder to employ the footprinting sort of scheme. The problem is that the aim of NavMeshes is to represent as large as possible open spaces in order to reduce processing and memory costs. This means that the density of the NavMesh is often far too low to be able to accurately represent footprints. My solution to this has been to implement a new system that is able to locally enhance the resolution of the NavMesh by subdividing its polygons. In order to do this, all dynamic obstacles need to 'link' themselves using a special interface class with which they can define their 'footprint volume' (which in my case were simple spheres). The subdivision of polygons is a recursive process whereby we keep subdividing as long as polygons and/or sub-polygons are 'inside' the footprint volume. To speed the process up I used axis-aligned bounding-boxes in order to quickly determine if there were intersections between (sub-)polygons and the footprint volume. This approach is quite 'rough' but can be polished a bit by only applying 'weighting' on sub-polygons whose centroids are actually inside the spherical footprint volume (whereby the weight is proportional to the inverse of the distance between centroid and volume center).

For a polygon *P*, footprint volume *VOL* and a minimal subdivision size threshold constant *ST*:

SubdividePolyon(polygon *P*, footprint volume *VOL*)

If polygon *P* already has sub-polygons **then**:

For each sub-polygon *SP*:

SubdividePolygon(*SP*, *VOL*).

Return.

Create axis-aligned bounding-box *AABB* around polygon *P* (just process its vertices).

If the major dimension of *AABB* is smaller than threshold *ST* **then**

Return // Cannot subdivide any further.

Create centroid vertex *CV* for polygon *P*.

Create middle vertices halfway each edge of polygon *P* and store these in vertex list *MVL*.

For each vertex V_i of polygon *P*:

Create new sub-polygon SP_i.

Let edge E_i be the clockwise next edge starting at vertex V_i (apply wrapping if needed).

Let edge E_{i-1} be the clockwise previous edge ending at vertex V_i (apply wrapping if needed).

Let vertex MVL_i by the middle vertex that was created for edge E_i.

Let vertex MVL_{i-1} by the middle vertex that was created for edge E_{i-1}.

Create sub-polygon SP_i in clockwise order as: { V_i, MVL_i, *CV*, MVL_{i-1} } (we can also derive more hierarchy related meta data here if needed).

SubdividePolygon(SP_i, *VOL*).

Add sub-polygon SP_i **to** sub-polygon list of polygon *P*.

Algorithm 11: Definition of a polygon subdivision for a given footprint volume.

The subdivision itself is fairly simple: we compute a polygon's centroid, create a vertex for it, cut each of the polygon's edges in half by creating new vertices there also and then connect all the (new) vertices to form new sub-polygons (see Figure 51). The algorithm is summarized in Algorithm 11 for a polygon *P*, footprint volume *VOL* and a minimal subdivision size threshold constant *ST*.

This subdivision scheme has the advantage that it is very simple and will always result in quads that have fairly even dimensions. We can also directly control the accuracy of the footprints by lowering the minimal size threshold. From my experiments, it showed that we can limit this threshold to even 0.5 or so (it basically depends on the dimension of the traveller). The main goal of the subdivision is to provide more of a 'guideline' on how to navigate past dynamic obstacles so it is allowed to be 'rough' in places. The Boids repulsor system and its whiskers will ensure that the traveller will try to maximize its distance between all nearby obstacles; this will smoothen out any irregularities that

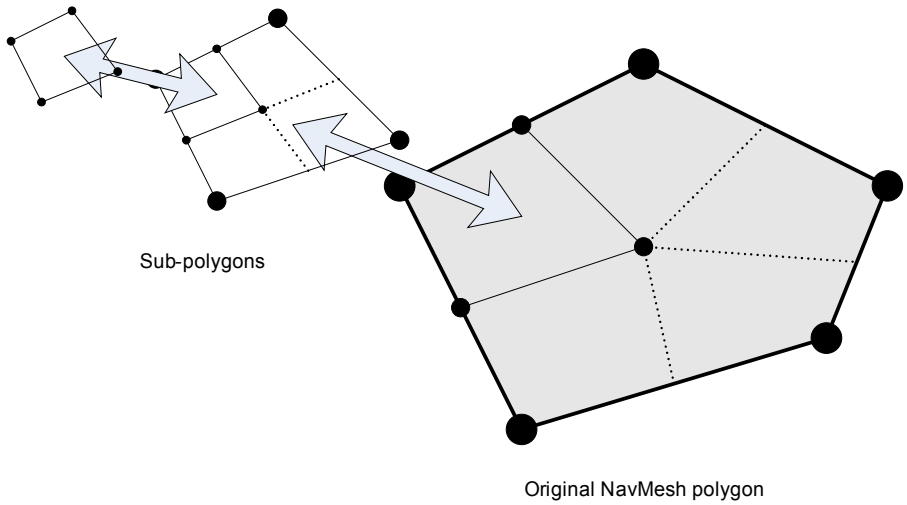


Figure 52: A lot of 'hierarchy' information can be stored while subdividing NavMesh polygons in order to speed-up the search for polygons that (partially) share edges with a given line-segment. Because we always split edges in half we can implement a sort of 'quadree' search to recursively scan all the sub-polygons inside the original NavMesh polygon.

CollectNeighboringSubPolygons(original NavMesh polygon *P*, target sub-polygon *TSP*)

Result sub-polygon lists *RPL* = {}.

For each child sub-polygon *CSP* **of** polygon *P*:

RPL += CollectNeighboringSubPolygonsHelper(*CSP*, *TSP*).

Return *RPL*.

CollectNeighboringSubPolygonsHelper(sub-polygon *SP*, target sub-polygon *TSP*)

Result sub-polygon lists *RPL* = {}.

If sub-polygon *SP* is sub-divided itself **then**:

For each child sub-polygon *CSP* **of** sub-polygon *SP*:

RPL += CollectNeighboringSubPolygonsHelper(*CSP*, *TSP*).

Return *RPL*.

Else we need to examine if there are any edges of sub-polygon *SP* overlapping somehow with those of target sub-polygon *TSP*.

If there is such an overlap **then return** *SP*.

Else return {}.

Algorithm 12: How to find all neighboring sub-polygons for a given polygon in a possible subdivided NavMesh.

originate from the limited footprint representations. Also, because we apply weighting in relation to the distance towards the center of the footprint volume, we could even use slightly larger footprint volumes than strictly necessary if we want to really make sure there will be enough clearances between all travellers. There is room for plenty more enhancements also; for a start we could try to only subdivide at the boundary of the footprint volume so that we do not enhance the resolution on the inside of the volume unnecessarily.

The hardest part of the NavMesh resolution enhancement is by far the updates that are needed on the graph. As we will see during the next discussions, there are a number of problems that need to be addressed in order to successfully utilize NavMesh modifications. First is how to link up polygons after they have been subdivided; we never know how the neighbors of a polygon will turn out to be (they might be original NavMesh polygons, or newly created subdivided polygons that have been subdivided further, etc.). For this we need to write functions that can determine which polygons have edges that somehow share a line-segment (or just parts of them). During the subdivision we can generate a lot of useful information in the form of 'inheritance information' that can be used during the search to speed them up (also see Figure 52). Finding all neighboring sub-polygons for a possible subdivided NavMesh is defined in Algorithm 12.

6.2.3 Graph detailing

Another problem with the footprinting scheme is that we are making modifications to the graph while other travellers might be using it to guide them through the world. We can not just remove edges and nodes that have been replaced by new ones to represent the higher resolutions; we only need to mark them so that the path-finder will ignore them and use the newest instead. This means that the NavMesh will be in a constant 'state of flux' and that it can potentially have many paths between certain areas ('low' and 'high' detail graph edges and nodes will coexist). A solution to this is that all travellers must link themselves to the NavMesh and acquire locks on nodes and edges that are part of their current path solutions (in the form of simple lock counters, for example). Once a traveller is finished moving, it will notify the NavMesh system that all the edges and nodes in its path solution are no longer needed, on which the system will decrease all relevant counters. If counters return back to 0 again, we know that the local enhancements are no longer needed and we can safely remove them. The use of counters will also enable us to handle multiple travellers in the same area in an economic way. If they bump into each other and need to route paths around them, the locking system will save us some work because parts of the resolution increments might already be there (so in a sense, travellers will be able to share each others enhancements). This means that, in general, the system scales relatively well because many more collisions between travellers will only marginally increase the amount of enhancements that need to be made. As a final optimization, I added a delay timer that would temporarily suspend the decrease in resolution after the system has identified edges and or nodes that are no longer in use. This proved to be beneficial because travellers sometimes repeatedly bump into each other when they move in unanticipated directions and/or speeds.

The NavMesh's caching system thus needs to be designed in such a way that it is possible to incrementally increase the resolution of the NavMesh at any point and any given time. Only when previously enhanced segments are no longer in use can we purge all modifications in one go. The

Find a path between world-space start point *WS* and goal point *WG*:

Notify the NavMesh system that all edges and nodes in the previous path solution can be unlocked (if needed).

Start node *S* = FindHighDetailNode(*WS*).

Goal node *G* = FindHighDetailNode(*WG*).

Find all travellers **that** are 'near enough' and **store** them **in** list *TL*.

BeginFootprintSession(*TL*): Apply NavMesh resolution enhancements in the traveller's footprint volumes.

Perform a normal path-find session, but treat any edge that is leading towards a node that belongs to a (sub-)polygon that has been subdivided as 'inaccessible'.

If a path has been found **then** notify the NavMesh system that all edges and nodes in the path resolution need to be locked.

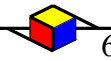
EndFootprintSession(*TL*): Clean up, remove all the cost penalty weights.

Algorithm 13: Procedure for travellers to find a path through the world using a NavMesh whereby avoiding dynamic obstacles. The support function FindHighDetailNode() is used to map a world-space point into a NavMesh node/polygon of the highest detail level. BeginFootprintSession() is further defined in Algorithm 14 and is amongst others responsible for applying node weighting penalties. The corresponding EndFootprintSession(traveller list TL) function basically does nothing more than removing these imposed penalties as part of a cleanup phase.

procedure for a traveller that wants to find a path through the world using the NavMesh whereby dynamic obstacles should be avoided is defined in Algorithm 13 and Algorithm 14. The entire procedure basically entails that before the path-finder can do its job, we need to be sure that all footprints have been 'drawn' correctly for all the other travellers that are near the actual traveller.

The procedure uses a number of supporting functions such as: BeginFootprintSession() (which is further detailed in Algorithm 14), EndFootprintSession() and FindHighDetailNode(). The EndFootprintSession() function will simply remove all the cost penalty weights that were applied earlier and doesn't need any further detailed explanation. The FindHighDetailNode() function is used to map a world-space point into a NavMesh node/polygon of the highest detail level. This function will first need to find the original NavMesh polygon in which the world-space point is located and, if needed, recursively walk through its inheritance tree of divided (sub-)polygons until it finds the one that still contains the world-space point. It is very similar to the CollectNeighboringSubPolygons() function. In practice it will help to cache this information somehow, so that the 'expensive' look-ups are not required. We can obtain this information easily while traversing a previously found path; in some cases we will only need to do recursive searching when a node from a path has been further subdivided.

There are many optimizations that can be applied that are not shown in the previous pseudo-code. One of the more obvious ones is to cache all the original NavMesh polygons that are currently occupied by a traveller's footprint volume, so that we do not need to look them up all the time.



Before starting a footprint session we first construct a list of neighboring travellers *TL* that are close enough to the actual traveller to warrant avoidance.

BeginFootprintSession(traveller list *TL*):

For each of list *TL*'s traveller's footprint volume *VOL*:

Find all the original NavMesh polygons that are intersecting with the footprint volume *VOL* and **store** them **in** polygon list *PL*.

For each polygon *P* **in** *PL*:

SubdividePolygon(*P*, *VOL*).

If polygon *P* was further subdivided **then**:

Mark polygon *P* **as** 'internally dirty': the NavMesh system needs to update its graph nodes and edges inside the boundaries of polygon *P*.

Mark all neighboring NavMesh polygons around *P* **as** 'externally dirty': the NavMesh system needs to update its graph edges that are linking polygon *P* with all its neighboring polygons.

After all subdivision has been performed we need to update the NavMesh graph to represent the new enhancements:

Handle internal restructuring of polygons: BFSRestructureInternalPolygons() (see Algorithm 15).

Handle external restructuring of polygons: BFSRestructureExternalPolygons()

(seeAlgorithm 16).

Reset the 'internally dirty' and 'externally dirty' flags of all the original NavMesh polygons that were processed.

Now we can finally apply the actual footprint 'weighting':

For all travellers **in** *TL*:

Apply cost penalty weights **onto** all the 'deepest' sub-polygons in their footprint volume (these weights are applied cumulatively).

Algorithm 14: Before the actual path-finder can be run we need to start a footprint session that will subdivide NavMesh polygons where needed and apply penalty weights on all relevant nodes.

When the traveller moves we can obtain the new set of polygons it is standing on and determine which polygons it no longer needs. For these polygons left in its 'wake', we can tell the NavMesh system we no longer require any resolution enhancements by decreasing their lock counters.

At regular intervals the NavMesh system will perform clean-up operations to remove resolution enhancements that are no longer needed (this will free up memory again). The process for updating the NavMesh and making it consistent again is very similar to how a footprint session is run. Instead of creating new nodes, we now remove all of them and their corresponding edges for all original NavMesh polygons that are no longer locked. It is not always possible to fully clear an area of any local enhancements that are required; we must not forget that any neighboring polygon might still have some remaining enhancements left that we need to remain connected to. Therefore we

Continuation of BeginFootprintSession():

BFSRestructureInternalPolygons():

Sub-polygon list *SPL* = {}.

For each original NavMesh polygon *P* **that** was marked as 'internally dirty':

Recursively create graph nodes for each new subdivided polygon *SP* that has not got one yet (and link it to them) and add *SP* to list *SPL*.

For each sub-polygon *SP* **in** list *SPL*:

Find all the neighboring sub-polygons: *NSPL* = CollectNeighboringSubPolygons(*P*, *SP*).

For each neighboring sub-polygon *NSP* **in** list *NSPL*:

Create (bidirectional) graph edges between *SP* and *NSP* **if** none exists already.

Algorithm 15: Helper function for BeginFootprintSession() that restructures polygons that need to be internally modified to be able to accurately represent footprints of neighbouring travellers that need to be avoided.

Continuation of BeginFootprintSession():

BFSRestructureExternalPolygons():

For each original NavMesh polygon *P* **that** was marked as 'externally dirty':

For each neighboring polygon *NP* **of** polygon *P*:

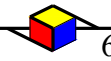
Find each 'deepest' sub-polygons by recursively traveling down the 'hierarchy tree'; for each such sub-polygon *SP*:

Determine which neighboring (sub-)polygons are overlapping sub-polygon *SP* and store them in a (sub-) polygon list: *NSPL* = CollectNeighboringSubPolygons(*NP*, *SP*).

For each neighboring (sub-)polygon *NSP* **in** list *NSPL*:

Create (bidirectional) graph edges between the nodes of sub-polygon *SP* and neighboring (sub-) polygon *NSP* **if** non exist already.

Algorithm 16: Helper function for BeginFootprintSession() that restructures all the polygons that are neighbors of polygons that were internally restructured to be able to represent traveller's footprints.



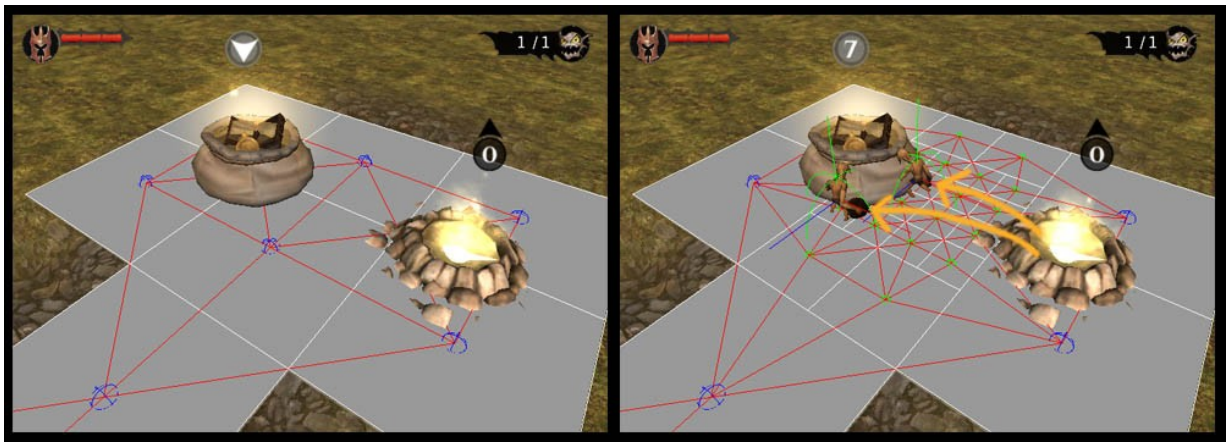


Figure 53: A simple example of how the NavMesh's resolution has been increased to allow two travellers to navigate around each other. Notice that the granularity increases towards the 'hot-spot'.

must apply the 'externally dirty' flag mechanism again as shown in BeginFootprintSession() on both the unlocked polygons and all their neighbors. That way, any mix of original NavMesh polygons and subdivided ones will be properly linked again in the NavMesh graph. Any original NavMesh polygon that is not locked but still has edges leading towards enhancements in neighboring polygons will automatically be cleared by the algorithm when these neighbors get cleaned themselves.

6.2.4 Results

In Figure 53 we see how an area of the NavMesh has been subdivided because two travellers moved between similar goals whereby one was forced to 'walk around' the other. In Figure 54 we see an example of a more complicated scene whereby multiple travellers traversed an area in opposite directions (or are about to). It is clear that for irregular polygon shapes in the NavMesh we will always obtain fairly irregular subdivided areas; however because we strive towards sub-polygons in 'quad shapes', the finer grained resolution still remains acceptable for our purposes.

From my experiments, I concluded that the movement of the travellers is already quite acceptable but clearly needs a bit more polish. Part of this polish is the tweaking of some of the footprinting scheme's parameters, such as cost weighting and subdivision thresholds. However, the absence of the Boids repulsor and whiskers system (with physics) is mostly to blame for failure of hiding some of the more 'rougher anomalies'. With such a system in place, I think many of the irregularities we currently see will be smoothed away quite naturally.

All these enhancements to the NavMesh system come at a price. The main problem is that subdividing and rebuilding of the NavMesh graph locally takes up a lot of time due to the many memory allocations needed. I think it should be possible to optimize this a lot, though. If we look back at Figure 52, we can see that any subdivided node automatically becomes a quad; we should be able to take advantage of this by using data structures of a pre-computed size that need less

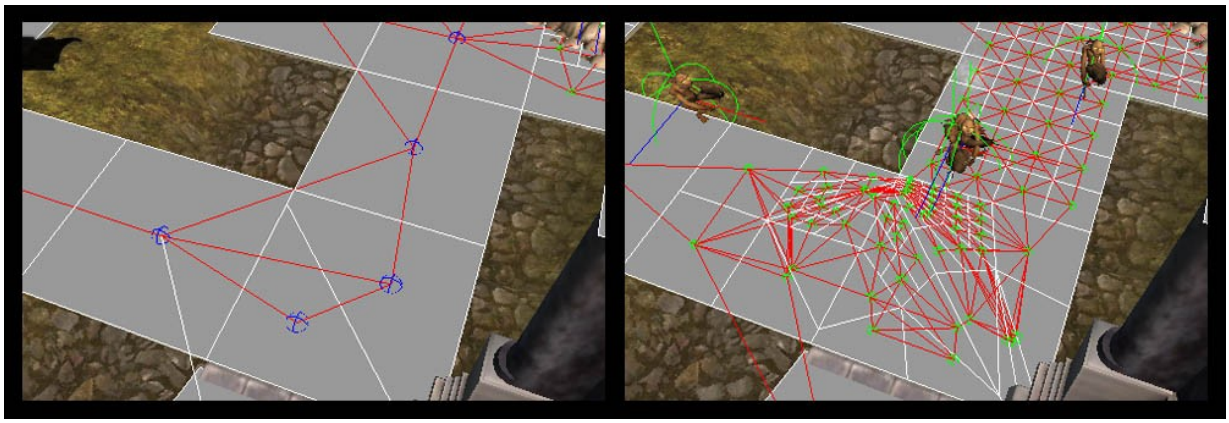


Figure 54: A more complex situation where NavMesh polygon subdivision was applied to solve collisions between the dynamically moving travellers. Irregular shaped polygons cause more irregular sub-polygons, but the results are still acceptable for our purposes. The enhanced NavMesh areas will not be removed immediately so that subsequent collisions in the area can be dealt with much quicker by re-using 'work' that has been done previously.

allocations. Still, all the additional meta data that needs to be maintained, and the searching for neighboring (sub-polygons) still make this a costly operation (more than I had expected). Also, additional memory allocation is required to create new edges and nodes in the NavMesh graph, but this will be a lot harder to circumvent, I think.

The system turned out to take a lot of work to implement and to debug; many pieces of additional visualization code needed to be added just to figure out why sometimes things tended to go wrong. I think that the performance of the entire system will automatically be improved if the quality of the NavMesh can be increased using some of the restructuring methods as discussed in subsection 4.4.1 (notably the polygon reduction schemes). If we can make the NavMesh polygons cover larger areas then the amount of subdivision per polygon will become cheaper (subdivided polygons become quads, whereas if we have a lot of smaller neighboring polygons we have to constantly deal with unknown amount of edges per polygon).

Also note that the increment in NavMesh resolution has (small) negative impact on the path-smoother that was described earlier in section 5.2. It is vitally important that it uses the subdivided polygons instead of those originally from the NavMesh because here we are not allowed to 'cut-corners'; that might make the traveller walk through other dynamic obstacles which is just what we were trying to avoid from the outset. Because there will be a lot more sub-polygons in hot spots, this will automatically take more time for the smoother to process. A small bonus is that because the subdivided polygons are a lot smaller the path smoother will have 'less room' to squeeze its path through. As a result the smoother will accurately follow the flow of nodes in the path solution, which is exactly what we need.

The downside of this system compared to Triumph Studios' system used in the Overlord games - which will be discussed in greater detail later in section 8.1 - is that drawing footprints is much harder to optimize. The Triumph Studios' engine utilizes a uniform dense grid of nodes to represent

the static world, which generally means there will be a lot more graph nodes than with a NavMesh. However, processing the nodes in order to determine what is inside a footprint volume is very straightforward (point vs sphere test, for example). Also, no extra memory allocations are required if the nodes themselves have meta data entries reserved for footprint cost weights. This, of course, comes at far greater memory costs; the NavMesh resolution enhancement approach does not have this problem. Also, if needed, the NavMesh system will allow us to directly control the accuracy of the collision system by regulating the granularity of the subdivisions. More details on the pros and cons of the NavMesh system will be discussed in section 8.2.

7 A* parallelization

In this chapter we will start with having a more detailed look at the fundamentals behind the classical A* path-find algorithm and some of its variations. The goal is to devise some form of parallelization that will make the A* algorithm run faster on currently available consumer multi-core architectures which may (even) have up to 4 cores. Some of these parallelization attempts turn out to be very beneficial as we will see during the subsequent discussions in this chapter.

7.1 A algorithm*

The task of any path-finding algorithm is to find a path between a given start point *S* and goal point *G* across a given map or world. These points can basically be of any form or data structure, but a generic representation is that of an undirected graph consisting of nodes/vertices that are interconnected by any number of edges. Often, additional data is stored in the nodes and edges that somehow represent various attributes of the map or world such as ‘height’ and ‘terrain type’.

The quality of the path that a path-finding algorithm must generate can be any combination of demands and restrictions. For example, the algorithm must not generate paths across parts of the terrain that are too steep or even impassible. We would rather have it be biased towards using roads or other easy/quick forms of transport, etc. All these attributes are often modeled into a single ‘cost-function’ that we can use to derive an (estimated) cost for traveling from a node A to a node B via an edge. An ideal path combines all the terrain properties into a most ‘efficient’ or ‘economic’ path that has the lowest overall cost to travel. By far the most popular choice that combines all these properties is the so called 'A*' algorithm, which we will discuss next.

7.1.1 Flooding

The A* algorithm, conceived in 1968, is basically a refinement of an older algorithm by Dijkstra using a so-called ‘heuristic’ in order to control the search direction more effectively. Dijkstra's original algorithm is a breadth-first search algorithm that is said to be 'greedy' (see [16]) because it recursively floods large areas unnecessarily during its search. A*'s improvements make it a best-first search by searching in the direction that is most promising at any given time. This is done by introducing a total cost computing function named *f(n)* that provides a rough cost estimation for any given node *n*. Let *g(n)* be the actual cost of the shortest path from start node *S* to node *n* (the path traveled so far). And let *h(n)* be the estimated cost of the shortest path from node *n* to the goal node *G* (the path ahead of us). A* then relies on the following heuristic function to determine the best candidate nodes while flooding its search space:

$$f(n) = g(n) + h(n)$$

So function *f(n)* represents an estimation of the **total** cost of the shortest path through node *n* based on the explored lowest cost to node *n* (= *g(n)*) and an estimated cost for the remaining unexplored path (= *h(n)*, also called the ‘heuristic estimation function’). Function *h(n)* is often implemented by using a cheap 'guesstimate' of the remaining travel distance, such as a Manhattan or euclidean distance between node *n* and the goal node *G*.

AStarSearch(start node *S*, goal node *G*):

Initialize:

Add node *S* to the Open List.

$G_{meta}[S] = 0$.

$H_{meta}[S] = h(S)$.

$F_{meta}[S] = G_{meta}[S] + H_{meta}[S]$.

Persistence: **while** Open List not empty and node *G* not in Closed List **do**:

Select the node **from** the 'Open List' **that** has currently the lowest $F_{meta}[N]$ value and make it our current node *N* (in case of ties we can select one at random).

Remove this node *N* **from** the 'Open List' and add it to the 'Closed List'.

For each outgoing edge *E* **of** node *N*:

Examine destination node *M*: **if** it is not already in the Closed List **then**:

Assume no (cheaper) local path found yet: *ImprovedFlag* = false.

Compute the travel cost leading up to node *M*: *Cost* = $G_{meta}[N]$ + EdgeTravelCost(*E*).

If node *M* is not in Open List **then**:

Add node *M* **to** Open List.

Estimate remaining cost: $H_{meta}[M] = h(M)$.

ImprovedFlag = true.

Else if *Cost* < $G_{meta}[M]$ **then**:

Correction: *ImprovedFlag* = true.

If *ImprovedFlag* **then**:

$G_{meta}[M] = Cost$.

$F_{meta}[M] = G_{meta}[M] + H_{meta}[M]$.

For path reconstruction we need to walk back from node *M* to its parent node *N*: LinkBackTo(*N*, *M*).

Algorithm 17: The core of the A algorithm for finding paths through a given directed graph. Note that its persistence and correctional properties ensure us that the most economic path is found. The EdgeTravelCost() function computes the actual travel cost of an edge, where the heuristic function h(n) only provides a total cost 'guestimation' for a given node, guiding the algorithm towards the goal. The LinkBackTo() function stores information that will tell us how to reconstruct the actual path when the goal node has been found. It simply links each node with its corresponding 'predecessor' node so that we can find our way from the goal node back to the start node.*

The algorithm uses 2 lists in order to control its search behavior:

- 'Open List': this is a list of nodes that are currently under examination (so this is our current 'search boundary').
- 'Closed List': this is a list of nodes that have already been visited. For these nodes we know the exact cost of the shortest path leading up to them. This list is often implemented as an additional meta data flag in the graph nodes, if enough memory is available.

Initially the Closed List is empty and the Open List contains only start node *S*. During the flooding process we keep track of a number of search attributes for each node that is currently in the Open List. This search meta data is often stored in a separate list that runs parallel with the Open List, for readability we will just denote them as follows:

G_{meta}[node n] = All the $g(n)$ values.
H_{meta}[node n] = All the $h(n)$ values.
F_{meta}[node n] = All the $f(n)$ values (which means always: $F_{meta}[n] = G_{meta}[n] + H_{meta}[n]$).

And finally we keep track of how we can reach the cheapest 'parent' node from a given 'child' node that has been flooded in direction of goal node *G*. This information is used to reconstruct the final path whereby we walk backwards through the 'inheritance tree' starting at the youngest sibling (= end node) up to the oldest grand-parent (= start node). The linking information can be stored in a separate buffer but it can also be stored as meta data in the graph itself depending on availability of additional memory. In this discussion we will use the function LinkBackTo() to denote the child-parent relation of a set of nodes:

LinkBackTo(node *P*, node *C*): mark node *C* that its parent is node *P*.

The cost function $g(n)$ is expressed by successively summing the cost of traveling across an edge *E* between its local begin to end node:

EdgeTravelCost(edge *E*): the actual cost to travel across edge *E*. This is often an euclidean distance with a weight modifier (>= 0).

Using these helper functions the A* algorithm proceeds as described in Algorithm 17. It is important to note that we really have to run the algorithm until goal node *G* has been added to the Closed List because it might still be possible that we find slightly better solutions by selecting different neighbor nodes around *G*. This '**persistence**' of the algorithm is important as we will see in later discussions of parallelization attempts. Only when the Open List runs empty prematurely will the algorithms abort early, in such cases there is no possible path between nodes *S* and *G*.

An important issue that is often neglected in discussions about A* star is its 'self-**correctiveness**' component. The problem is that A* does not actually look ahead to which node it walks, it only guesses. So when A* decides to walk from a node A to node B, it will **not** look at the **real** cost, but only at the **rough estimate** of that cost (that is the $h(n)$ part). So it might well turn out that the transition was way more expensive than initially guessed and thus other paths via neighboring nodes might be cheaper. A* will only figure this out after the Open List has been sorted again, only then will the actually most promising node have been located (so the resorting needs to be done each time F_{meta} is updated).

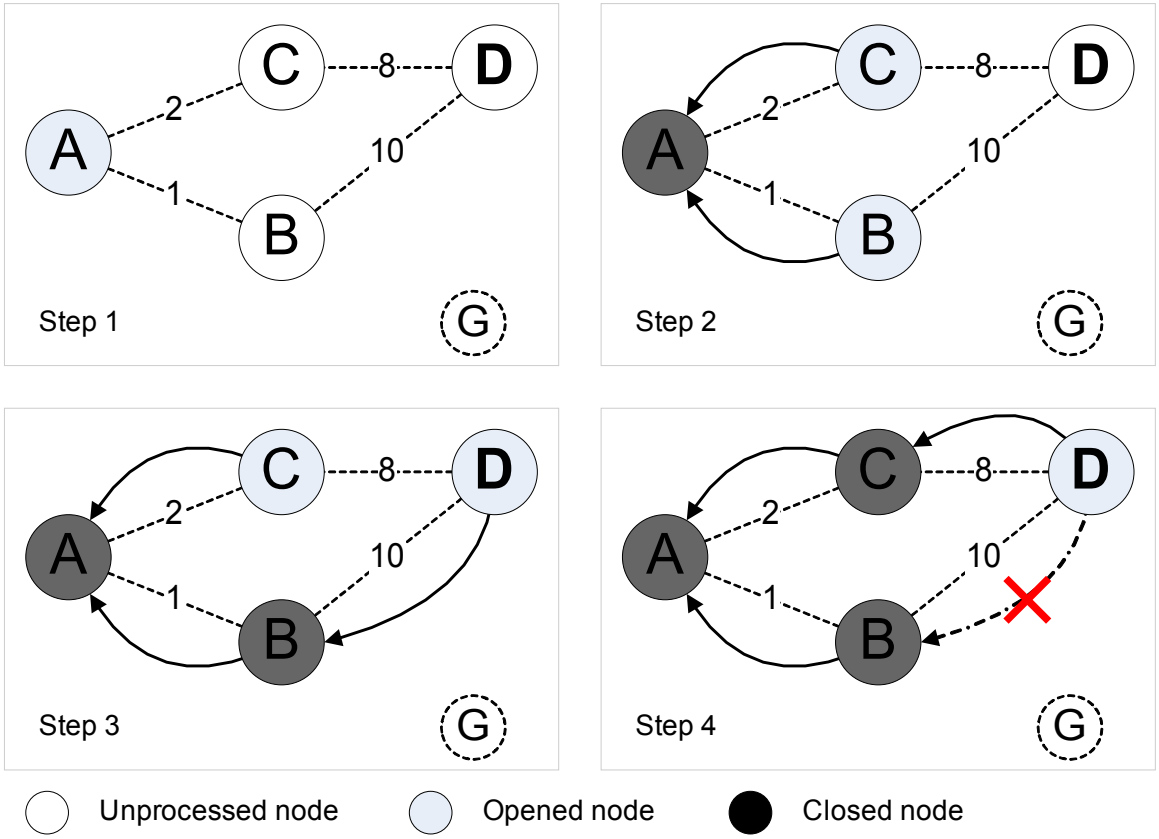


Figure 55: An example of how A floods through graphs. Each time a node is closed we need to determine if better paths have become available in order to later reconstruct it properly (= correction).*

We can demonstrate the importance of this by taking a naive approach whereby we would just store a link to the parent node each time we open new nodes (so without implementing the correctional component). In Figure 55 we see a very simple graph that is being flooded in the direction from node *A* to node *D* with the actual goal being a goal node *G* somewhere far off. Beginning at step 1 we see that node *A* has been opened at some point in time. In step 2 the path-finder decides to close it and open its neighboring nodes *B* and *C*. When these nodes are placed on the Open List we connect them with their parent node *A* because currently these both suggest to be the shortest path available. Then, in step 3, it turns out that node *B* has the lowest *f(n)* value (we are being pulled in the direction of goal node *G*) and it is closed. By doing this, node *D* will be opened and linked back to its parent node B again. However, it turns out that traveling from *B* to *D* is actually very costly, but because A* relies on the *h(n)* function to do its look-ahead guessing this will not be noticed. With *d(n)* being the true cost between nodes *n* and *G,* we only need to guarantee that $h(n) \leq d(n)$ for A* to be admissible. If we take *h(n)* to be the euclidean distance for example then we might quite often underestimate the true travel cost by a lot (especially when all sorts of artificial penalties have been applied). So, when in step 4 node *C* is finally closed we actually need to check if there are any nodes in the Open and Closed list that we should link to if that would result in a lower total travel cost. In this case we need to appoint node *C* as the parent of node *D*. If the graph is non-directed then we do not need to check the nodes in the Closed List. For a directed graph we need to check both lists because when a node is closed it will only check nodes for outgoing edges, not incoming.

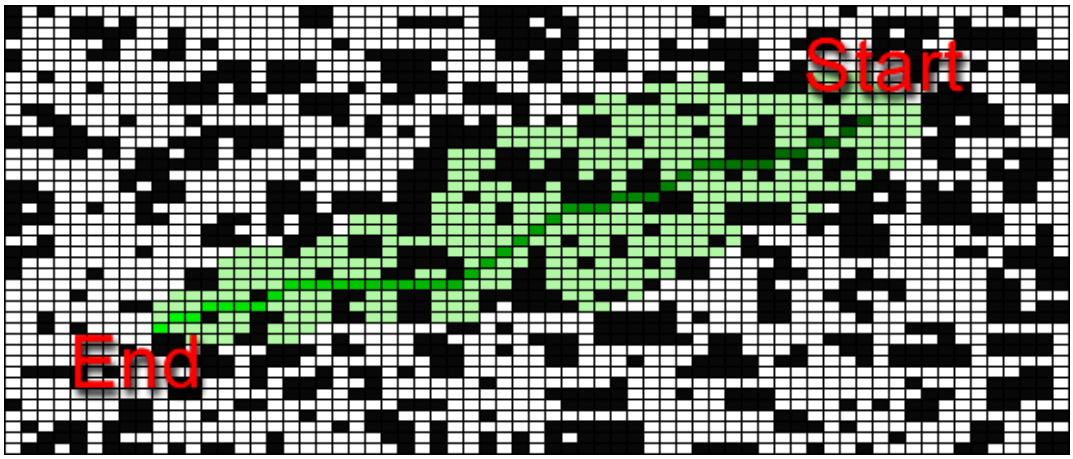


Figure 56: Example of how A finds a path by flooding only a limited area in the entire map.*

When we look at the description of the A* algorithm, we see that once a correction is applied this will automatically influence the next node that is selected to be processed (the Open List will be resorted). It could well be that the corrected node suddenly becomes the cheapest and should thus be processed next. During the discussion on parallelized versions of the A* algorithm, we will encounter examples whereby the corrective component has been removed as part of a trade-off between 'accuracy' and performance.

7.1.2 Performance

There are several reasons why A* is a good algorithm. To begin, the heuristics function *f(n)* literally 'pulls' the flood boundary towards the goal node *G* (also see *Figure 56)*. This will not only save us many CPU cycles but it will also reduce the memory footprint of the algorithm because we are no longer flooding in all directions as Dijkstra's original algorithm would. Especially so, that in many cases where goal node *G* is directly or almost directly 'visible' from start node *S* the algorithm will find *G* with only a minimal amount of work (which makes A* very fast indeed).

From an academic point of view we can state that as long as the travel costs between the nodes is non-negative, A* is both:

- **Complete:** if a path exists it will find it.
- **Optimal:** it will find the cheapest path.

We cannot state that A* will always be the fastest search algorithm; it is easy to see that trying to move in the direction of the goal node is not always favorable (just imagine a maze with lots of dead ends leading up to the goal). However, in most practical cases it is definitely a very appropriate algorithm.

Note that A* can only be classified as **admissible** if the used heuristics function *h(n)* is itself 'admissible', that is: it should never overestimate the true cost for the optimal path between current node *n* and goal node *G* (also denoted as *d(n)* or sometimes *d(n,E)* or *h*(n)*). If always $h(n) \leq d(n)$ then A* will never ignore nodes that would actually have given shorter paths.

The popularity of A* can also largely be attributed to its flexibility: we can very easily control how the algorithm finds optimal paths. By increasing the cost of certain edges between nodes, for example, we can force the algorithm to avoid them. Similarly, reducing the cost will make it more likely to be part of the final path. If we would do this for mountains and roads for example then the resulting solution will automatically look very natural. We have already seen this similar technique in section 6.2 whereby we 'drew' footprints in order to make the path-finder steer clear of dynamic obstacles.

7.1.3 g(n) vs h(n)

Let us have a closer look at the heuristics function that A* relies so heavily on to perform its duties: $f(n) = g(n) + h(n)$. As mentioned before $g(n)$ is the actual cost of the shortest path between start node S and the current node n, and $h(n)$ is the function that computes an estimated cost of the shortest path from the current node n to goal node G. In order for A* to be able to find the optimal path, the estimation function must be 'admissible', which means it may never overestimate the true cost of a possible shortest path between node n and node G. The estimation function $h(n)$ is of course chosen in such a way that it is much faster to compute than the true cost $d(n)$ to travel from node n to node G via a path. The relation between estimation $h(n)$ and true cost $d(n)$ is very important, we can discern the following cases:

- $h(n) = 0$. At this extreme we no longer have a heuristics behavior and the true cost function $g(n)$ controls the search pattern. This is exactly what Dijkstra's algorithm does: it is guaranteed to find the shortest path but the search will be quite exhaustive.
- If $h(n) <= d(n)$ for every node n then A* is guaranteed to find the shortest path. The lower $h(n)$ is in relation to $d(n)$ the more nodes A* will open, making it behave more and more like Dijkstra's algorithm.
- If $h(n) = d(n)$ for every node n then A* will directly find the shortest path and never examine any other nodes outside the shortest path. In this ideal situation A* will run extremely fast, which is often the case for strictly horizontal or vertical shortest paths for example (this is always good to know). For all other situations, however, whereby the environment we are searching through is much more unpredictable, we will not achieve this because it would require some sort of 'oracle' to get all the 'guesstimates' perfectly accurate.
- If $h(n) > d(n)$ for some of the nodes then A* will sometimes wrongly discard nodes and will thus not always return the most optimal path, though it might be able to do so in less time (e.g.: if $h(n)$ is much cheaper to compute). The algorithm is then however no longer admissible.
- If $h(n) >> d(n)$ or always $h(n) > d(n)$ then the influence of $g(n)$ is suppressed and the behavior turns into a search algorithm that completely disregards actual path costs and is only blindly focused on getting nearer to goal node G.

So clearly the way we compute $h(n)$ is very important. If we are careful, we can fine-tune the algorithm's performance by balancing between precision and performance. For $g(n)$ there are most often no serious cost issues as it is easily computed by successively adding the travel costs of the edges that have been traversed. Often we can precompute these (static) edge costs and store them as meta data.

In principle we can choose any metric for *f(n)* as long as it adheres to the above mentioned restrictions. For NavMeshes this will most often be euclidean distances because its polygons are located arbitrarily in 3D space. If we know for certain that changes in height are irrelevant then we could opt to drop the 3rd dimension in order to speed things up a bit.

We must also realize that both *g(n)* and *h(n)* should have similar 'scales'/'dimensions' or else one of them will bias the algorithm. It is tempting for example to drop the expensive square-root when computing the length of the vector between node *n* and goal node *G*, and just use the squared distance instead. This will however degrade the behavior into the *h(n) >> d(n)* case, which will make the algorithm flood many more nodes than necessary.

7.2 Fringe Search

Central to my parallelization attempt has been a recent addition to the family of A* derivatives called 'Fringe Search' (see [2]). This variant caught my attention because its authors rather boldly claimed speed ups of 10% to 30% or more (using their highly optimized versions) and because it utilizes an approach which would make it easy to parallelize.

7.2.1 Flooding

One of the most costly parts of classic A* is keeping the Open List sorted so that we can determine which is the most promising node to process next. Fringe Search does not do this, and instead just utilizes a 'Now List' and 'Later List'. At its core it processes the Now List sequentially and opens any node which *f(n)* estimate is under a certain cost threshold; all the other nodes that were deemed too expensive are directly moved to the Later List. After each pass the cost threshold is relaxed a bit and the process starts anew. This is basically identical to the Iterative Deepening A* (or 'IDA*' for short) strategy (see [12]). IDA* also forgoes the use of an expensive sorted Open List by repeatedly flooding nodes from the start node and relaxing the cost threshold for each iteration until the goal node is finally found. The major drawback of this approach is that IDA* excessively refloods nodes during each new attempt to find goal node *G*. This makes it an unfavorable choice, especially if long paths need to be found often. In these cases the speed-up gained by not having to maintain a sorted Open List is offset by the amount of nodes that needlessly keep being re-opened.

Fringe Search, however, has managed to reshape the idea of IDA* and made it much more viable to run through large search spaces. As stated, Fringe Search does not employ a sorted Open List but just processes the entire Now List from start to finish. The Now List basically defines the outer boundaries of the flood area (hence the name 'fringe'), which can be substantially large. Processing the Now List is however extremely cheap (especially when using linked lists, for example); so even though Fringe Search does a lot more visitations, the end result is still faster than that of classic A* which continuously needs to apply sorting. And because Fringe Search explicitly keeps track of its flood boundary, it will not reflood nodes needlessly such as IDA* would.

Another advantage that the authors of Fringe Search claim is essential to its success is its simplicity. They note that implementing A* efficiently requires a lot more work because of the need for good caching and sorting systems; Fringe Search requires none of these and is thus inherently much easier to optimize. Although from an academic viewpoint this is not any proof that Fringe Search is algorithmically better than A*, from a practical viewpoint this certainly makes a lot of sense.

```
FringeSsearch(start node S, goal node G):
  Initialize:
      Make sure that all nodes are marked as 'Open'.
      Now List = { start node S }.
      Later List = {}.
      G_meta[S] = 0.
      H_meta[S] = h(S).
      F_meta[S] = G_meta[S] + H_meta[S].
      We do not have a sensible cost threshold yet: f_threshold = ∞.
  Persistence: while Now List not empty:
      Compute new cost threshold (that should never become less than the old one):
          f_min = smallest f(n) of all the nodes in the Now List.
          f_threshold = min(f_min + CostRelaxation, f_threshold).
      For each node N in the Now List:
          Remove node N from the Now List.
          If f(N) > f_threshold then:
              Add node N to the back of the Later List.
          Else we are going to close this node and process its neighboring nodes:
              Persistence: If node N = goal node G then:
                  An optimal path has been found, just stop and reconstruct
                  it!: Return.
              Mark node N as 'Closed'.
              For each outgoing edge E of node N:
                  Examine destination node M: if it is not marked as 'Closed'
                  then:
                      Assume no (cheaper) local path found yet:
                      ImprovedFlag = false.
                      Compute the travel cost leading up to node M: Cost
                      = G_meta[N] + EdgeTravelCost(E).
                      If node M is in Now List or Later List then:
                          If Cost < G_meta[M] then:
                              Correction: ImprovedFlag = true.
                      Else we have opened a new node:
                          H_meta[M] = h(M).
                          ImprovedFlag = true
                      If ImprovedFlag then
                          G_meta[M] = Cost.
                          F_meta[M] = G_meta[M] + H_meta[M].
                          For path reconstruction we need to walk
                          back from node M to its parent node N:
                          LinkBackTo(N, M).
      Swap Now List and Later list.
```

Algorithm 18: The Fringe Search path finding algorithm for a directed graph. It uses the same helper functions as those of the classic A implementation (see Algorithm 17 on page 72). The CostRelexation constant basically controls how many nodes are opened during each pass through the Now List.*

During the next bit I will describe the Fringe Search algorithm in more detail using some pseudo-code. I have taken the liberty to restructure it a bit so that it will be easier to discuss parallelization attempts later on, the actual algorithm is more optimized and, for example, does not need to swap the Now and Later lists explicitly. I have also added a *CostRelaxation* constant which controls with how much we are going to increase the upper limit on *f(n)* estimates that should be processed during single pass through the Now List. Again, we employ a number of helper tables such as F_{meta}[node *n*], G_{meta}[node *n*] and H_{meta}[node *n*] that can be implemented efficiently in various ways. We are also using the same functions EdgeTravelCost () and LinkBackTo() as were earlier seen in the classic A* discussion to be able to reconstruct the path afterwards. The implementation of the Fringe
Search algorithm for a given start node *S,* goal node *G* and a given *CostRelaxation* constant is described in Algorithm 18. The path reconstruction afterwards is exactly the same as was discussed for classic A*. If the Now List becomes empty prematurely then no path exists between the start and goal node.

7.2.2 Performance

We see that Fringe Search also has the **persistence** and **corrective** elements such as classic A* does. This guarantees us that it also inherits the same A* properties: so Fringe Search is both **complete**, **optimal** and **admissible** for *h(n)* functions in identical conditions. In fact, when we select the *CostRelaxation* constant to be 0 we force Fringe Search to open nodes in exactly the same order as A* would (and IDA* for that matter). This means both have the same bias and will generate the exact same path solutions.

Although my A* and Fringe Search implementations were not highly optimized (I had to defer this due to time constraints), I confirmed that Fringe Search is indeed in general faster. If we take *CostRelaxation* to be 0 then A* will only still beat Fringe Search once in a while, but never by much. In all other cases, Fringe Search is significantly faster. It most noticeably seems to excel at finding long but simple paths (where the heuristic function tends to pull the algorithm straight towards the goal), or when no paths can be found at all. It is in these cases where the reduced cost in node visitations really comes into play.

During my experiments I noticed a considerable speed up when using values of *CostRelaxation* larger than 0. The original paper does not explicitly treat this property (the *CostRelaxation* constant is an addition of my own), it only illustrates how to find a 'proper' cost threshold value by deriving it directly from *f(n)* values in the Now List. The reason for this is that it influences the order in which nodes are flooded. If we set it to >0 then more than one node could be closed per traversal through the Now List. This means that if normally node X is opened, and later corrected via opening node Y we might not always be able to apply the correction depending on the order of the nodes in the Now List (this was basically illustrated in subsection 7.1.1 and Figure 55). Normally it would take a minimal of three iterations to process these nodes: first node X is opened, doing so made node Y suddenly cheapest so that node is opened and thus the correction on node X is applied. If node Y does not provide any links with as yet unprocessed nodes, then node X is closed again. However, with the threshold relaxation we might do this in less iterations whereby we run the risk of failing to apply cost estimation corrections in some cases. When we, for example, manage to open node X and Y right after each other during the first iteration, then we run the risk of closing node X before node Y during the next iteration. As a result, the impact of the correction from closing node Y will be too late because we have already flooded onwards from node X with a

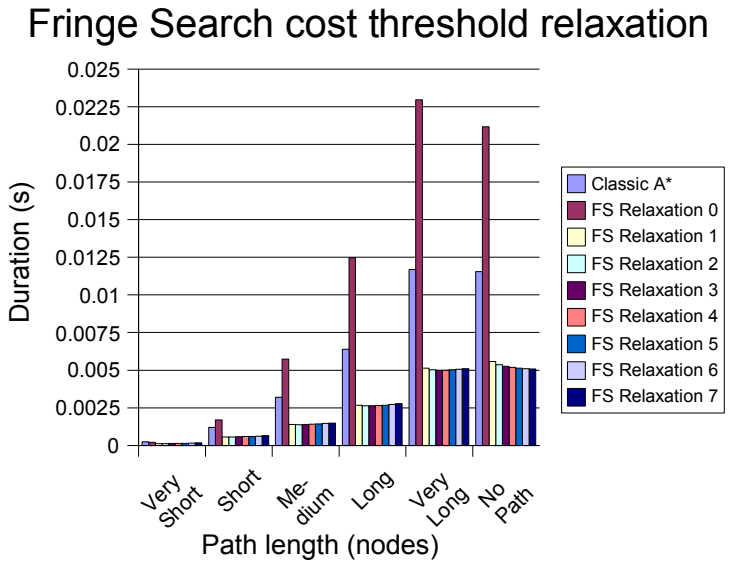


Figure 57: Measurements show that a cost relaxation value of 5 yields the lowest average path-finding session durations (when using a uniform grid as search space).

higher path cost estimation than is strictly the case. Because node X was not corrected in time, all subsequent nodes flooded from that node will now have incorrect path cost estimations as well in a cascading effect. So in short: by picking *CostThreshold* > 0, we will no longer always have correction and thus the algorithm is no longer **optimal**.

However, by artificially increasing a found cost threshold we can force Fringe Search to 'do more work' during a single pass through the Now List, so that it floods spaces 'faster' because less iterations are needed in total. By keeping the *CostRelaxation* constant 'within the margins' we can make sure that the heuristic element in the *f(n)* function still remains the strongest driving force and not run too high a risk of sabotaging the corrective aspect of the algorithm (we already noted that only under certain conditions will it fail). Also, if we would choose *CostRelaxation* too large, Fringe Search would mindlessly flood in all directions which would basically degrade it back into Dijkstra's original breadth-first algorithm (see subsection 7.1.3 with case *h(n) = 0*). Luckily, experiments have shown me that the effect of the loss of correction has turned out to be negligible (more on that in section 7.4).

For my experiments, whereby the travel cost between nodes is basically uniform, a *CostRelaxation* value of 5 proved to be the 'sweet-spot'. This is summarized in Figure 57 where we can see that higher values tend to slow down the process more (the flooding then becomes apparently too arbitrary). With this setting, Fringe Search will beat classic A* significantly in virtual all attempts, with the speed-up being a factor of 1.5 up to 2.5 times as fast. We can see an example of this in Figure 58; even though Fringe Search flooded more nodes it still managed to beat classic A* because the node visitations themselves are so cheap. Note that for a *CostRelaxation* of 0 my Fringe Search implementation structurally performed worse because I use an explicit Now and Later List (the original version as described in [2] is much more efficient). The reason for this is that it will provide better comparisons for parallelized implementations (these are discussed in section 7.3 and onwards).

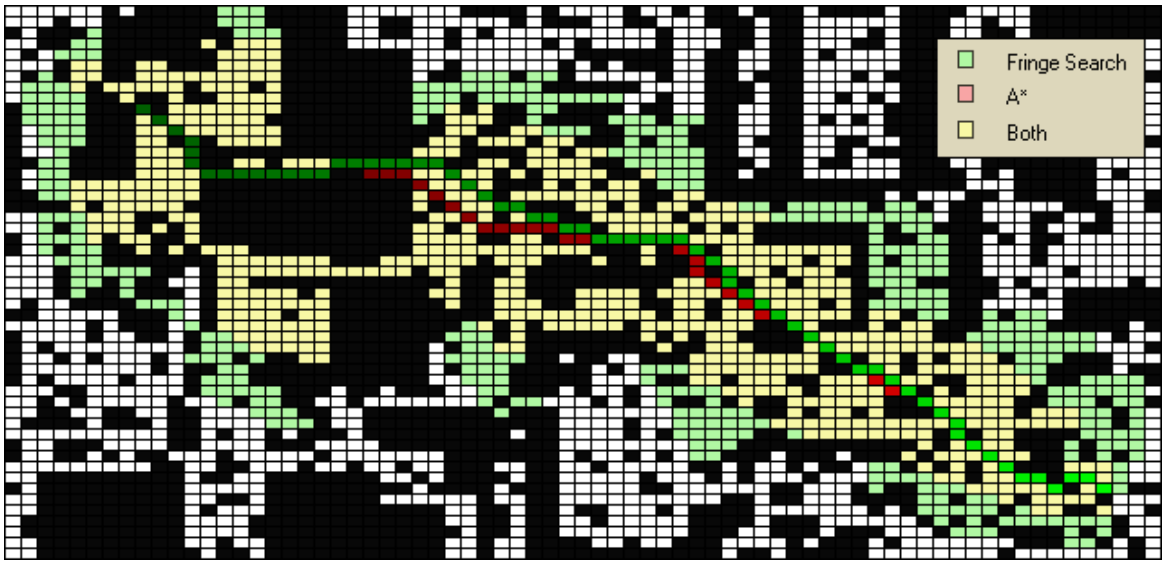


Figure 58: An example of how Fringe Search with a CostRelaxation value of 5 was 1.8 times faster than classic A. We can clearly see that Fringe Search flooded a lot more nodes (but at far less computational costs), and that because of the relaxation both algorithms no longer have the same node bias. The resulting paths are still optimal with both having identical travel costs.*

The reason I chose Fringe Search as the basis for parallelizing path-finding algorithms is because it has two interesting traits that are really helpful. First, there is the Now and Later lists; because Fringe Search does no longer use a central data structure to make its decisions, it will be possible to distribute the Now and Later lists over multiple CPUs. Second, we can directly control how much 'work' Fringe Search performs using the *CostRelaxation* constant. If we parallelize the algorithm, we should be able to boost all CPU cores' performance simultaneously. Also, as we will see, if we use Fringe Search in a 'bidirectional search' approach we can coerce it to flood larger areas which will help us in finding overlaps between flood boundaries of individual CPU cores sooner. These properties make Fringe Search an ideal candidate, as we will see during the next discussions.

7.3 Parallelized alternatives

In the short amount of time that remained, I investigated some path-finding implementation variations that were specially tailored to multi-core CPU architectures. Such CPUs are becoming readily available on the consumer market but have not yet seen widespread usage. A multi-core CPU is basically a unification of a number of separate CPUs that are sharing a single main memory pool. Each CPU is equipped with its own cache which means that these architectures can easily be utilized via threads. The Quad Core CPU that I have been using has roughly the layout as depicted in Figure 59.

An important aspect of parallelization is how data is separated. Because multi-core architectures share the same memory pool, it is vital to keep memory access non-overlapping as much as possible, so that each core can fully rely on its own high-level cache. Each time we write in memory, we run the risk of causing a cache refresh for all the remaining cores. An important aspect to note here is that data is stored 'contiguously' in main memory. So ideally we should not just separate data on a 'logical' level, it should also be spaced apart on a 'physical' level (see Figure 60).

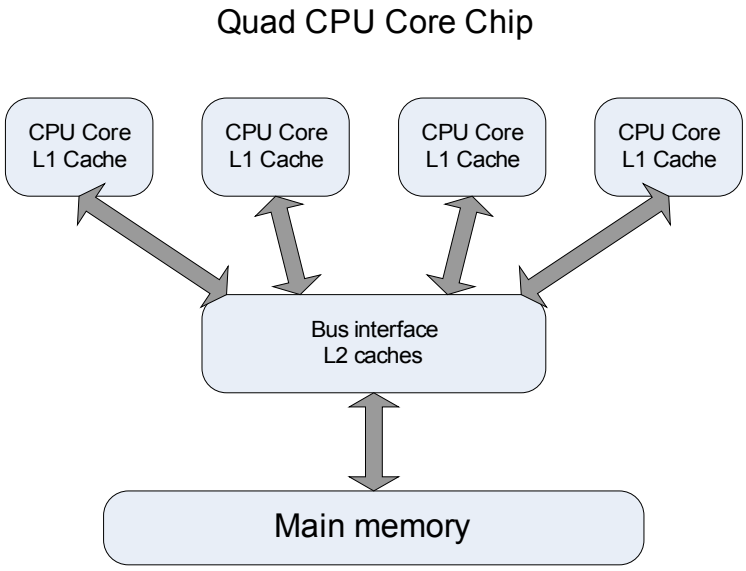


Figure 59: Each of the individual cores has its own level 1 cache. All cores access the main memory via a shared level 2 cache.

```
Struct CoreData { ….. };

struct GridData
{
    CoreData m_data[2];
};

GridData grid[100][100];
```

Bad data separation

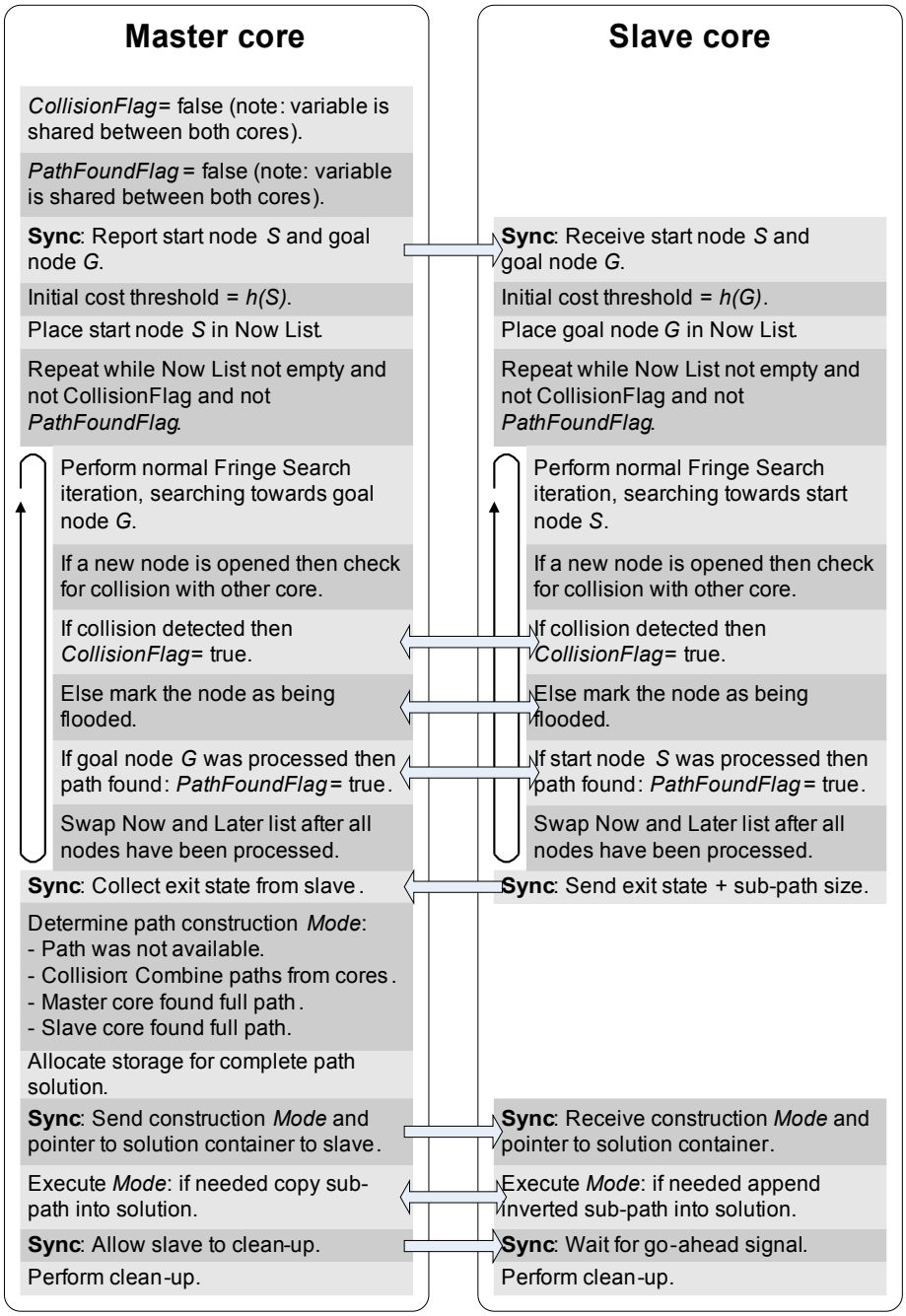
```
Struct CoreData { ….. };

GridData grid[2][100][100];

or even better:

GridData grid1[100][100];
int      dummy_padding;
GridData grid2[100][100];
```

Good data separation

Figure 60: Example of how to separate a 100 by 100 data grid for 2 cores. The one on the left will put many items for both cores on the same cache page because the data is intertwined. The example on the right is much better because each core now has its own contiguous set of items, with only a minor possible cache page overlap. We can even use some 'dummy padding' to make sure that both grids are located on completely separate cache pages.

7.3.1 Parallel Bidirectional Search

An obvious strategy to utilize multiple CPU cores is to take 2 cores and have them each search towards each other and 'meet in the middle'. As the main strategy of A* is to keep opening the most promising node we can consider all nodes at the boundary of the flood area 'most' optimal (this is not strictly true, of course). Whenever we hit a node flooded by an opposite core, we can immediately complete the path using the alternate core's path-finding meta data. We can also be reasonably sure that the resulting path will be 'cheap enough' even though we have not exhausted all possible candidates. The algorithm behind this strategy, that I have dubbed Parallel Bidirectional Search (or PBS for short), is illustrated in Algorithm 19.



Algorithm 19: The Parallel Bidirectional Search path finding algorithm for a direct graph in pseudo-code. The arrows indicate data being exchanged implicitly or explicitly as part of synchronization between CPU cores.

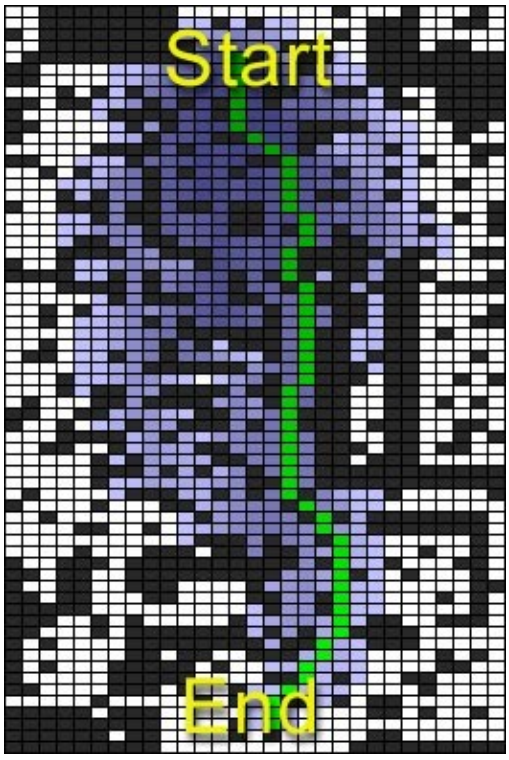


Figure 61: The area that A floods often has a leaf shape. Here we have rendered all non-path nodes in shades of purple to denote how much time passed before they were closed. As expected, the nodes become brighter outwards from the start node as time has progressed.*

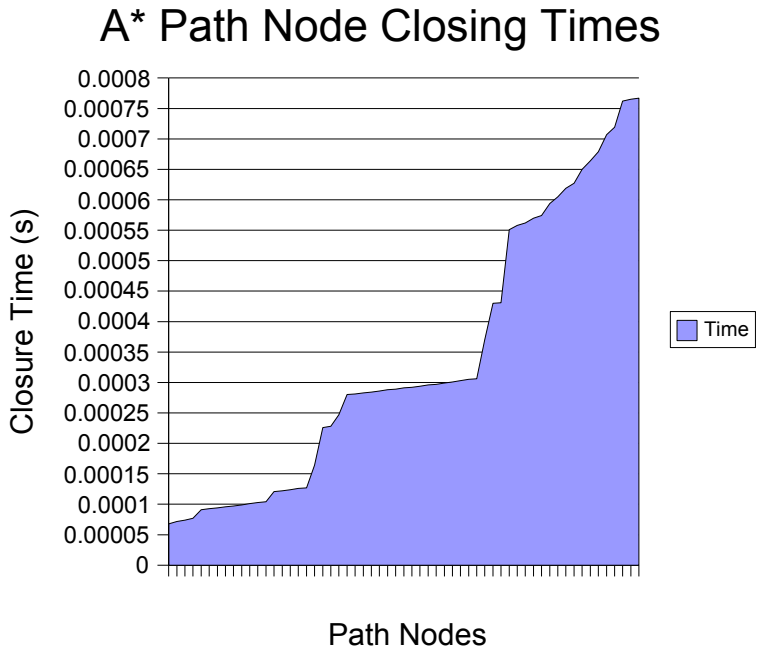


Figure 62: A plot of how long it takes before each successive node that is part of the path in Figure 61 has been closed. We can clearly see that as more nodes are flooded the further we are from the start node, the longer it takes before the algorithm decides on which are the best nodes for the final path.

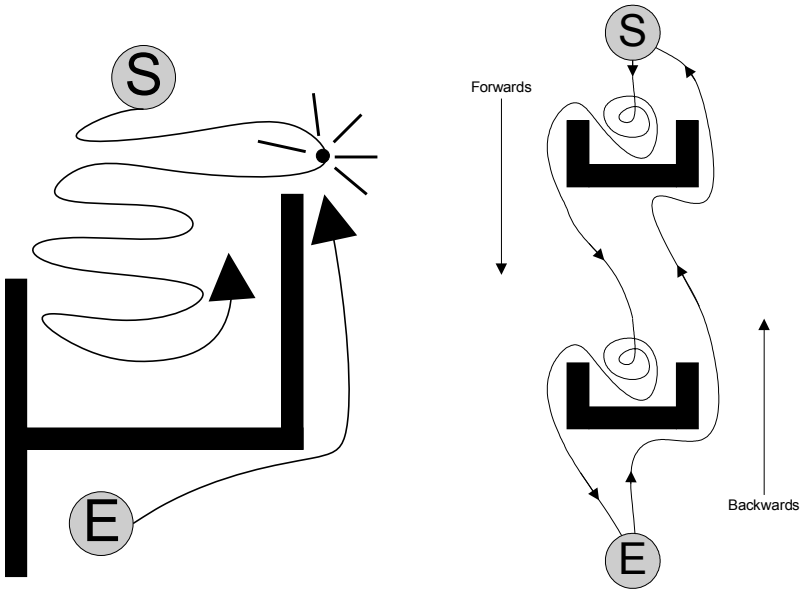


Figure 63: On the left we see an example how the core starting at the end node will hit the other core somewhere in its 'flank', finding a path very quickly. On the right we see that even if both cores have managed to evade each other until the very end we potentially still have a speed up.

If both cores report a collision at the exact same moment (which is very rare), I chose to pick the core with the collision node that is nearest to half the guessed travel cost. This way we can be a bit more certain that each core needs to do roughly half the work while reconstructing its own part of the total path.

It is easy to see that this 'meet in the middle' approach will speed up the path-finding process because each core would ideally now only need to examine half the amount of nodes before a solution can be obtained (a reduction of 50%). If we examine the flooding behavior of a classical A* implementation more closely, we will see that it takes considerably longer to advance the flood front the further we are from the start node (an example of this is shown in Figure 61). Generally, A* floods a 'leaf' shaped area that has the center of mass at the start node and its 'stem' leading up to the end node. The reason for this is that the *h(n)* heuristic function often grossly underestimates the travel cost towards the goal node because it uses straight lines. As the true travel costs for nodes at the flood boundary start to rise, the nodes nearer to the start node will suddenly appear to be cheaper and thus get investigated first.

In Figure 62 we see how long it takes before the algorithm closes each successive node in the resulting path for the given example. We can clearly see that as the path grows longer, it takes increasingly more time before A* decides on which is the best node. More importantly, we can see that the node which is halfway down the resulting path is closed at roughly a third of the total time it will take to complete the final path. From this we can deduce that a 'meet in the middle' approach could potentially reduce the time to find a complete path by far more than just 50% (more like 70% or so).

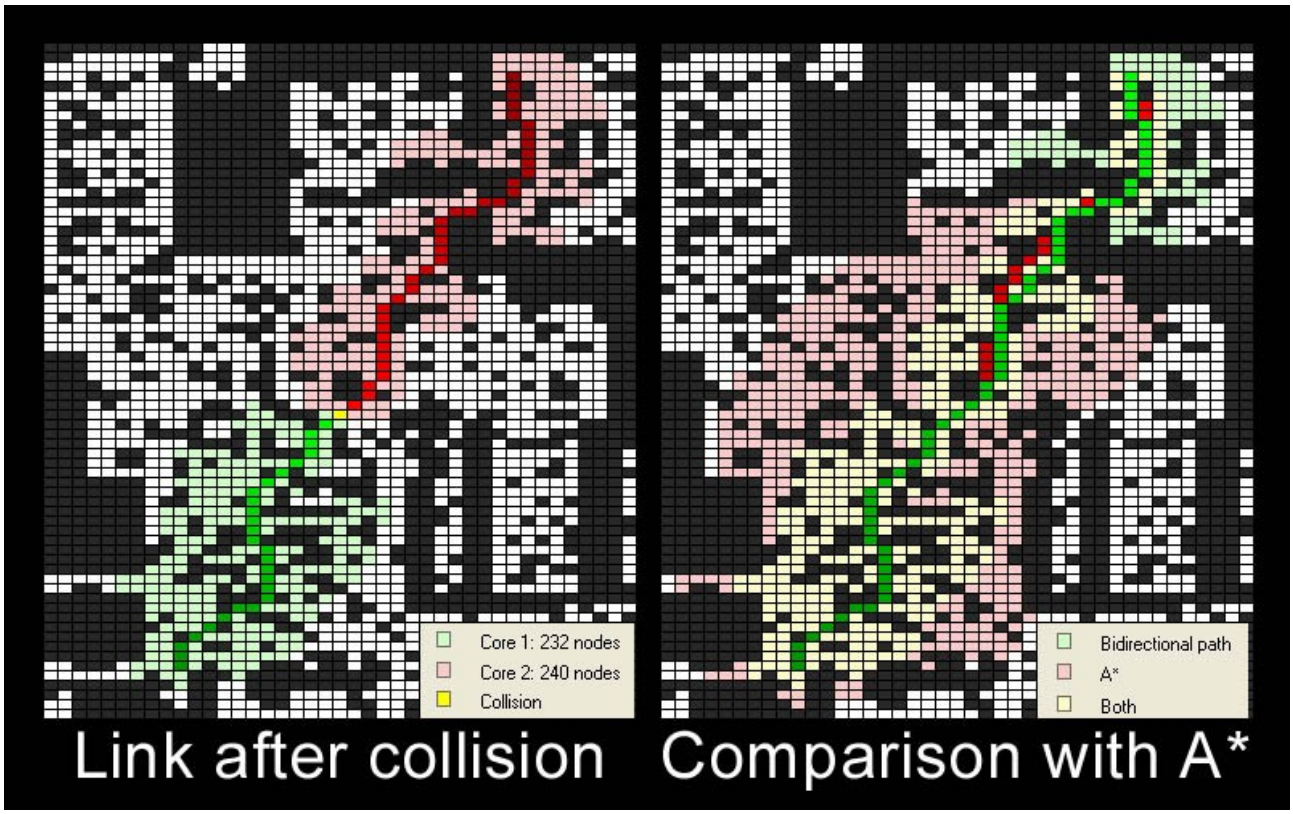


Figure 64: On the left we see how 2 cores flood towards each other until a collision is detected. The full path is then constructed by linking both 'halves' together at the collision node. On the right, we see an overlay with the path that a classical A has produced, whereby the red cells denote path deviations. On closer inspection, these are merely the result of a different bias because of the reversed search direction, and we concluded that they are not erroneous when taking the flow of the full path into account. It is immediately clear that much less nodes have been flooded than with classic A*.*

In practice, we can expect additional speed-ups because there are a lot of cases whereby a bidirectional search can 'short-circuit' and find paths much faster. In Figure 63 we see some examples of why searching in both directions can prove to be very beneficial. The cores will, for example, never hit each other 'head-on' but collide somewhere in each others flank. Because the A* heuristic will try to pull the flood front towards the target node, it can accidentally run into a local sink that first needs to be 'filled' up before alternative paths around it will be considered. When searching from the opposite direction, however, this can often cause a quick short-circuit when the alternate core finds a node that has been flooded by its rival core some time ago. Also, if no path exists, then one of the cores will often detect this sooner and can force the other core to terminate early.

Strictly speaking we can state that this approach is still **complete** (it will find a path if it exists), but no longer **optimal**. The latter has already been discussed, and the former is demonstrated by the example on the right in Figure 63: in worst-case scenarios where there is no collision the search is basically reduced to a normal A*search, for which we know it to be complete.

For my implementation each core used its own copy of the search graph so that meta data could be stored locally without interference with the other core (the parent-child links, for example). Both cores shared a single buffer of flags for each graph node which simply indicate if either core has visited it yet, the so-called 'collision buffer'.

An example path that has been found using bidirectional search is shown in Figure 64. On the left we see that a collision was detected somewhere half-way, so both cores have virtually done the same amount of work. When we connect the two 'half-paths' together we obtain a path that is only slightly more expensive than the most optimal path (just 1% more expensive in this case). The actual optimal path is shown in the figure on the right, with a classic A* overlay. The cells marked in bright red are nodes that the bidirectional search has deviated from. Upon closer inspection we notice that most of these are the result of a different 'bias' for equally expensive nodes. In those cases, classic A* decided to step horizontal a bit sooner than the bidirectional search, but overall the chosen path has the same cost. Note that we do not see any discrepancies in the lower half of the path which makes perfect sense: that part has been flooded in the same way as classic A* would. It is only the top half that has been flooded in the opposite direction, hence the changes in node bias. In the figure on the right we can clearly see that much less nodes have been flooded with PBS than with classic A* (denoted with the light red nodes).

The significant loss of optimality is not caused by the change in bias, it is due to the fact that we stop the search just a bit too soon when we detect a collision. The area around the collision node often does not get fully flooded, which means we have lost a bit of the **persistency** in the search process. Potentially, there might be cheaper nodes in this area, which will now no longer be discovered. Through my experiments, though, I concluded that the amount of error we introduce here is generally low. This is probably due to how the flood front expands with 'small' increments in cost. That and the fact that both cores are continuously searching in the most promising direction with a different bias seems to dampen any serious decision errors. It must be said, though, that travel costs in my experiments are all 'uniform' euclidean distances, without any additional weighting. The results will most likely be significantly less optimal if this is no longer the case (think of different terrain types, or footprinting schemes such as discussed in section 6.2). However, this will only be actually relevant if such 'aberrations' are extreme and/or close to the collision node. For the majority of nodes that are nowhere near the collision node, we can state with certainty that they have been processed correctly, which still makes bidirectional search a valid alternative in my opinion.

I experimented to try and optimize the access to the collision buffer a bit in the hope that I could reduce the amount of cache updates between the cores. An intuitive idea is for example to not test the cache until either core has opened a node that is further away than half the direct distance between start and end points. It became quickly clear, though, that this would require a lot more experimentation to get right, and that it often has an averse effect because the algorithm then has to handle more decision logic inside its tight processing loop. In practice it turns out that the majority of the algorithm's processing occurs on separate cache pages; only when they get near enough for a collision to occur will there actually be any noticeable cache penalties (this is later confirmed in section 7.4).

To summarize, the pros and cons of Parallel Bidirectional Search:

Advantages:
- Intuitive and easy to implement.
- Many A* variations can be used (not just Fringe Search).
- Virtually no 'overflooding'.
- Does not require much more memory (only the 'collision buffer').
- Is inherently 'cache-friendly'.
- Does not require explicit synchronizations of the cores: both cores always run at full speed.

Disadvantages:
- Not scalable (just 2 cores).
- (Slightly) less than optimal paths (could be worse for non uniform cost distributions).

7.3.2 Distributed Fringe Search

My next attempt to gain parallelization speed-up was to use the Fringe Search's Now and Later lists to literally distribute 'work' among multiple CPU cores. As discussed, during each pass through the Now list, Fringe Search performs some very simple tests in order to determine if new nodes should be processed. The processing of nodes itself mostly involves adding new nodes to the Later List. The main idea behind the algorithm I have come to call 'Distributed Fringe Search' (or DFS for short) is thus to distribute the Now List over all available cores, have them process their parts, merge the individual Later Lists, swap them and start all over again. The nice thing about Fringe Search is that the Now and Later lists do not need any sorting which makes distributing them very easy. Also, each core can compute the smallest *f(n)* value it has found locally, so that the master core can collect them and only needs to do a few comparisons to determine the next cost threshold that should be used.

To speed up the distributions of Now and Later lists, I devised a scheme I dubbed 'Shared Balanced Lists'. The idea is that 2 cores share a single memory buffer that is 'large enough' for worst-case scenarios. The memory buffer is basically a 'node array' that starts growing from the center of the memory buffer. The first core will make the node array grow 'forwards' whereas the other will grow it 'backwards'. After all cores have converted their Now Lists into Later Lists it is very easy to rebalance the work for pairs of cores that are using the same Shared Balanced List: we only need to locate the 'node' in the middle. In Figure 65 we can see a simple example of this for 2 cores that both utilize the same Now and Later Shared Balanced Lists. Step 1 resembles the state of the cores after they have completely processed all the nodes in their parts of the shared Now Lists. For whatever reason Core 2 was able to open more nodes than Core 1, which means we need to somehow balance their upcoming work-load. In step 2 we begin with switching the Now and Later lists, whereby each core aims its 'growth pointer' back to the center nodes of the shared list. We then distribute all the nodes in the Now List by setting the 'read pointer' of each core to the entry that represents the middle of the used part of that list. By doing so, Core 1 has effectively taken over some of the work from Core 2. The Fringe Search algorithm can now enter its next iteration. Note that we have achieved load-balancing very cheaply because we only needed to do some pointer manipulations, we did not need to physically distribute any data to other cores.

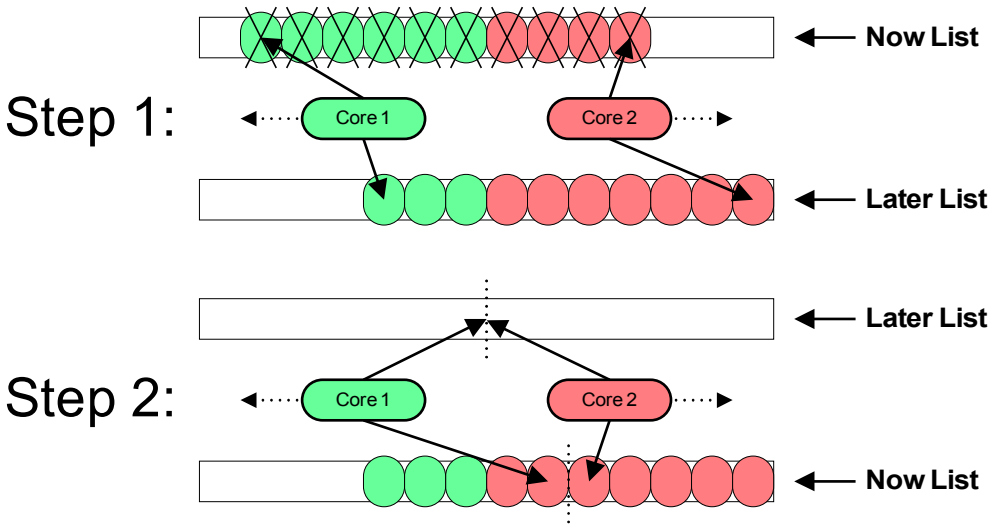
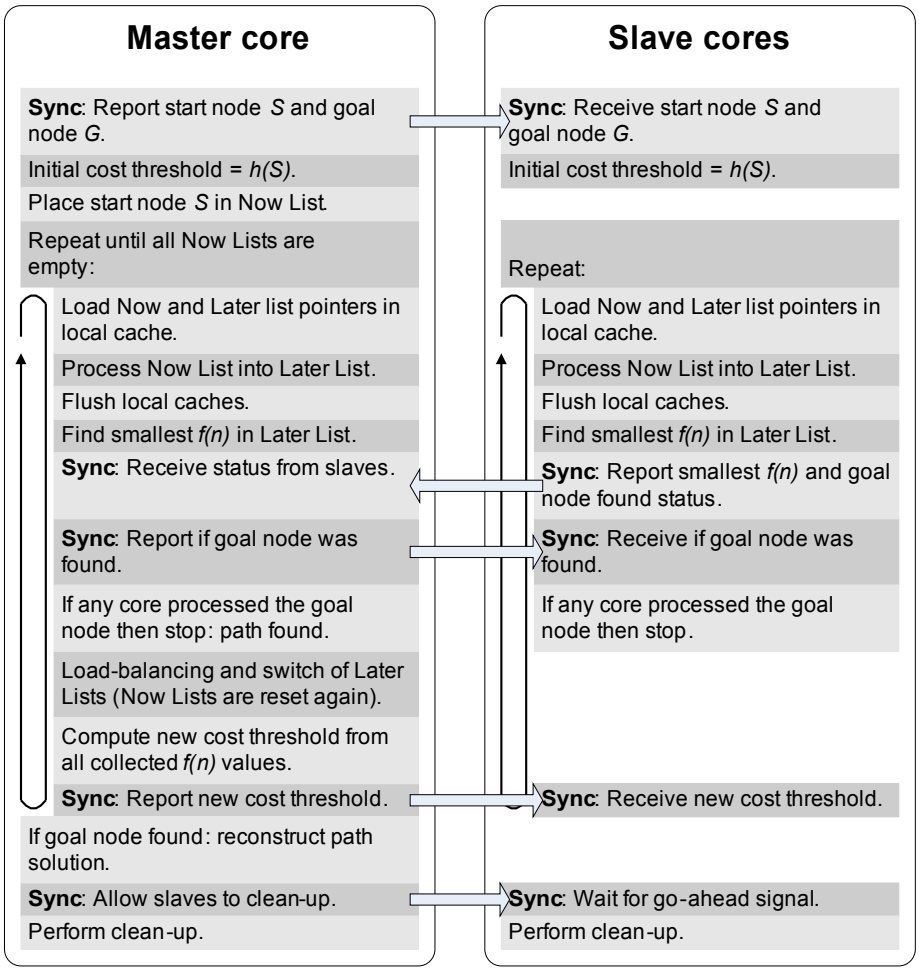


Figure 65: An example of how load-balancing is achieved by using shared lists that grow in 2 directions. Core 1 will always read an grow leftwards, and Core 2 will read and grow rightwards.



Algorithm 20: The Distributed Fringe Search path finding algorithm for a direct graph in pseudo-code. The arrows indicate data being exchanged implicitly or explicitly as part of synchronization between CPU cores.

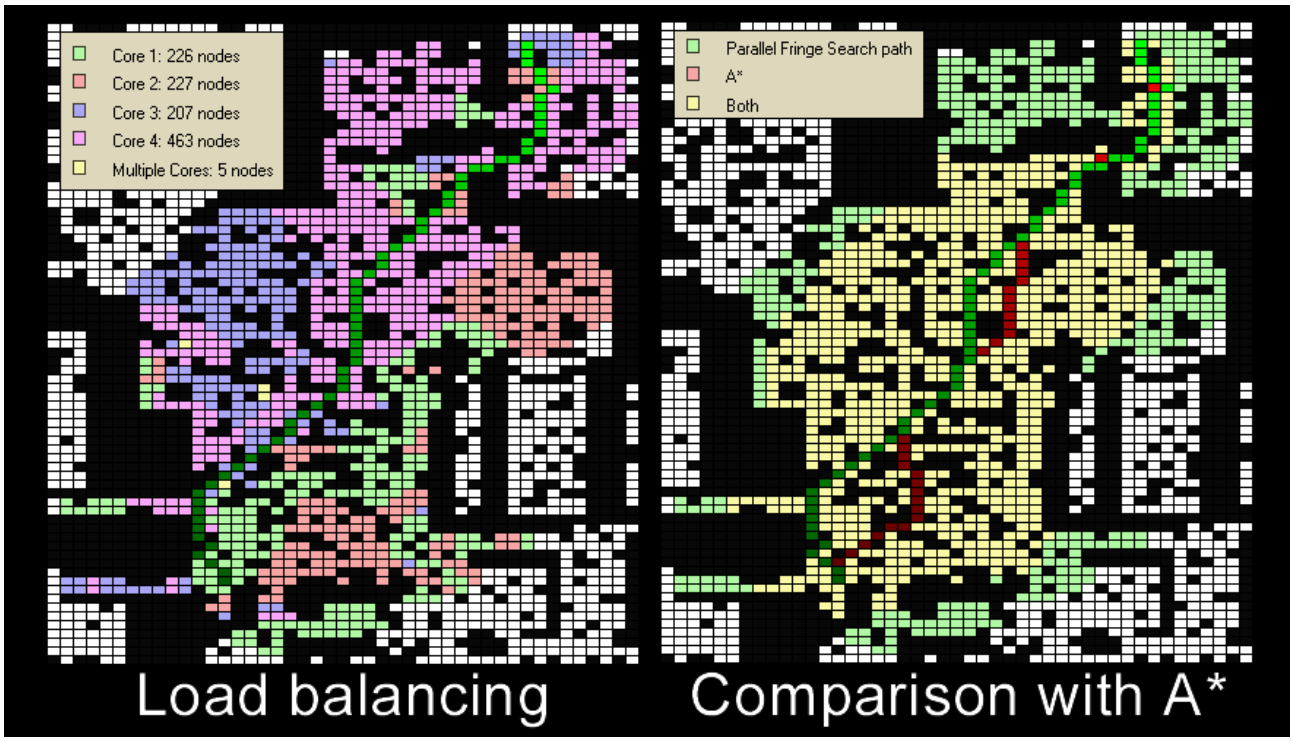


Figure 66: On the left an example of how load-balancing was achieved by Distributed Fringe Search, using a CostRelaxation constant of 5. On the right we see that with the loss of the corrective property we end up with less than optimal paths. Note that the deviations with the optimal A path are also just a matter of different bias, the actual additional path cost was only roughly 2%.*

This strategy of course only works for pairs of cores; if we have more of them there is no other alternative but to physically distribute nodes across multiple Shared Balanced Lists. Because my test hardware is a Quad Core architecture I only needed to load-balance between 2 pairs of cores which is very easy. After each iteration, the master core collects the status of all the Shared Balanced Lists and determines which one contains the least amount of nodes. If the difference between both lists exceeds a certain threshold (which in my case was an experimentally obtained '8'; we only want to load-balance when it is really necessary), it will select a source slot that contains a node and move it to an empty destination slot. The source slot is selected from the fullest list by picking the one that is furthest from the dead center of the list, whereas the destination slot is picked as the one that is nearest the center of the alternate list. This will ensure that the lists will not become 'skewed' but will always 'gravitate towards the centers'. Note that if we manage to position these list centers right between cache page memory boundaries we will remove almost all the cache penalties (writing is where it hurts most, not reading). The entire flow of the algorithm is roughly defined in Algorithm 20

The main advantage of the Distributed Fringe Search approach is that it offers us a strategy that can utilize larger number of cores effectively (we are no longer limited to 2 cores as with Parallel Bidirectional Search). Although this is a great strength, it is also its Achilles' heel. By distributing nodes 'arbitrarily' over multiple cores, we have lost the ability to perform **correction** as a normal Fringe Search implementation would do. This means that it is no longer guaranteed that DFS will find the cheapest path and it is thus no longer **optimal** (as we can clearly see in Figure 66).

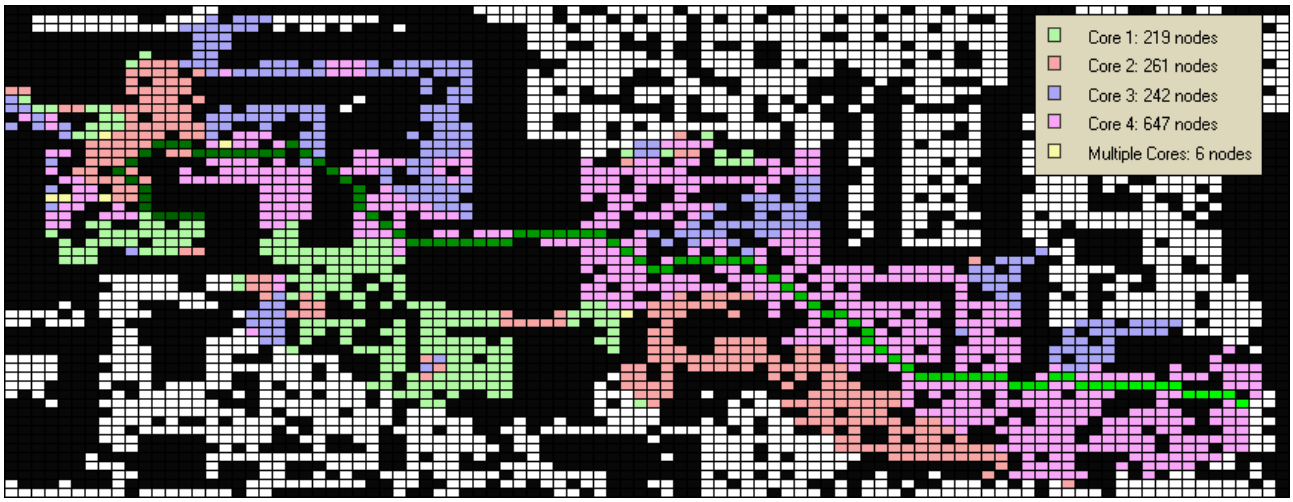


Figure 67: In this DFS example, Core 4 seems to be doing most of the work; it is this core that has flooded most nodes near the solution path that will be found. What we witness here is the influence of the heuristic function pulling the core towards the goal node. All the other cores are inadvertently 'delegated' to searching through spaces that turn out to be dead-ends.

If we were to try to re-implement **correction** again in the algorithm when processing a particular node then we would first need to figure out which core currently owns the corresponding child node and then somehow interrupt it to allow us to 'interfere'. The problem is that all cores are processing their local Now Lists at the same time so there is no way to guarantee that we will be 'in time' to apply the correction; the specific node might have already been processed. The loss of correction is regrettable; but it is all part of a trade-off between 'accuracy' and speed. Another drawback, however, is that it is no longer possible to keep track of the 'parent-child' relation of nodes during flooding. This means that the solution path needs to be reconstructed by starting at the goal node *G* and search our way back to the start node *S* by repeatedly traversing towards the neighboring node with the lowest *g(n)* value. This search is made a bit more expensive because each core needs to store these *g(n)* values in a local container so that we do not have any race conditions (a floating point is already 4 bytes so it is not safe to assume that writing it to memory is 'atomic' enough). So for each node we need to search through all cores' flood buffers and determine the lowest value. Although this sounds discouraging it is actually not: later we will see that there are surprisingly few cases of 'overflooding'. This means that each node only has a single *g(n)* computed for it so we do not need to do any expensive floating-point comparisons to find the smallest one. The whole path construction process, however, is more expensive than for other parallelized A* variants (and we must not forget to 'freeze' all slave cores during its construction; only afterwards should all cores perform their clean-up).

Initially, the measured speed-up was actually quite disappointing; classic A* was often still notably faster. It was only after I increased the values for the *CostRelaxation* constant that DFS started to win regularly (but then still only on longer paths). This makes sense because without the cost relaxation we are very limited to the amount of nodes that can be processed (often just the one with the lowest *h(n)* value). The overhead of controlling multiple cores is then greater than that of a single-core Fringe Search implementation doing all the work by itself. Using a larger cost relaxation also has the advantage that, because nodes are opened faster, we obtain a better load-balance sooner in the process; we open more nodes so more cores get work thrown at them faster. The overall load-

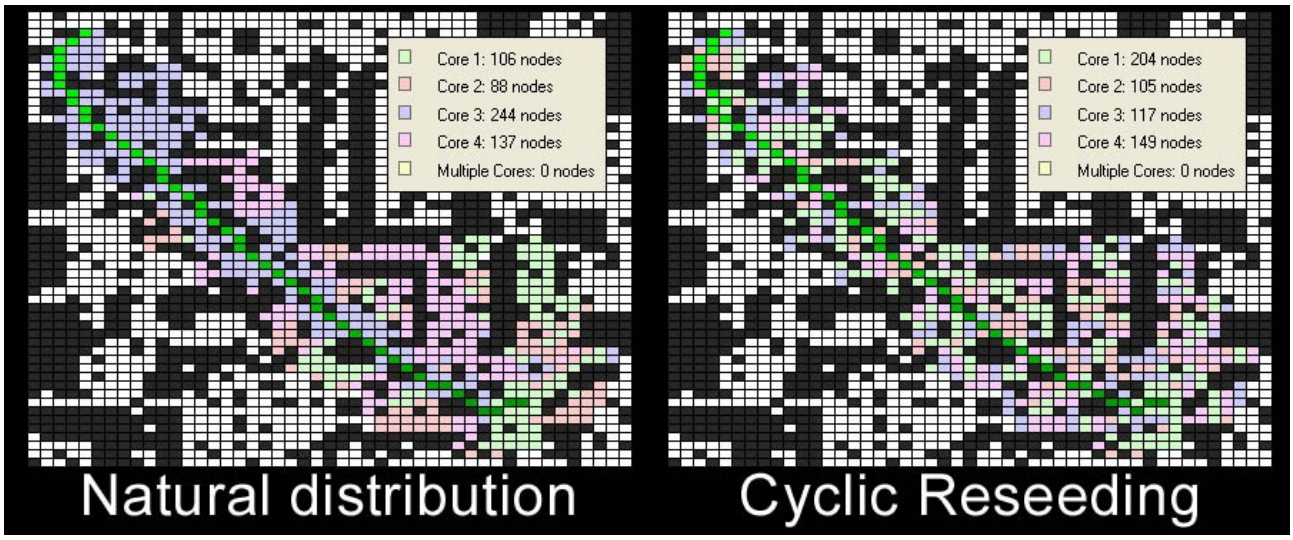


Figure 68: In an attempt to improve the load-balancing of the cores for the Distributed Fringe Search algorithm we re-assign the most promising node of each core to the next in a continuous cycle. This adaptation clearly helps to 'break the monopoly' that one core often seems to have (as seen for Core 3 in the image on the left). Instead, the flooding patterns of all the cores have become completely intertwined (see the figure on the right).

balance is fairly good, as we can see in Figure 66. I have noticed that there is always one core that seems to be doing the brunt of the work. That core has often flooded the areas in which the final path was found which leads me to believe that this is the influence of the A* heuristics function. That function is designed to pull the algorithm towards the goal node and as long as this goes on 'unhindered' it will always favor nodes for that particular core (a good example of this can be seen in Figure 67). The other cores will often be searching through 'branches' elsewhere that turn out to be dead-ends later on.

Note that with the DFS algorithm we have basically foregone any attempt to prevent race-conditions in order to get the least amount of overhead (so that no additional synchronization mechanisms need to be in place). The race conditions can occur in the separate buffer that I used to mark nodes as being 'Opened'; all cores are reading and writing this buffer simultaneously. That is why I also allocated a buffer to store *g(n)* values for each of the cores separately; that way they never write in the same memory area. From my experiments I have found that the race-conditions rarely caused nodes to be opened by more than one core at the same time, the chances of this are virtual negligible. Such 'overflooded' nodes are depicted by yellow grid cells in Figure 66 and Figure 67, and as you can see these are scarce. Such nodes rarely instigate more 'overflooding' because there is almost always one core faster than the other when opening neighboring nodes, the slower core will then automatically discard the node without doing anything else.

On paper, the DFS approach sounded really promising because it allows us to distribute the work-load quite naturally on all the available cores. In practice however, the obtained results are less spectacular (more on this in section 7.4). After some profiling, it turned out that a significant amount of time is still lost on cores waiting for each other. This suggested that the load-balancing is still far from optimal. In an attempt to improve this I modified the algorithm a bit so that, when 4 cores actually have nodes to process, after each iteration we re-assign the most promising node for

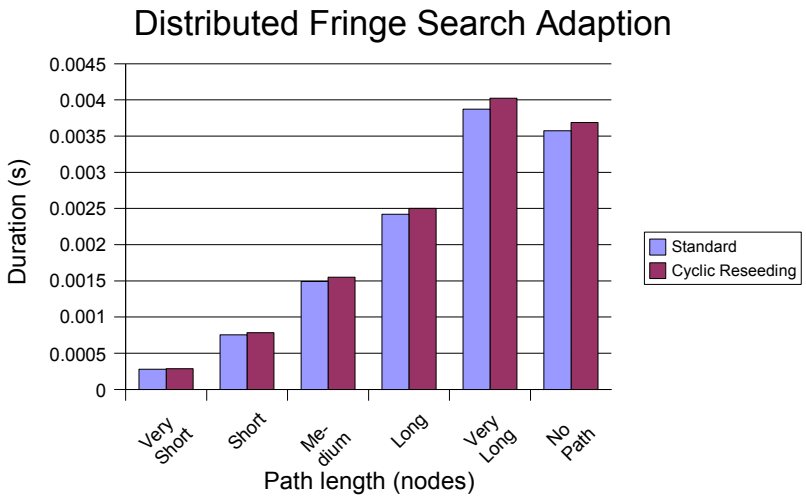


Figure 69: Applying cyclic re-seeding turns out to have an averse effect on the average performance of the Distributed Fringe Search algorithm.

each core to the next core (in a cycle). This 'Cyclic Reseeding' scheme certainly results in a more fine-grained distribution of nodes as we can see in Figure 68.

Again on paper this sounded like a good idea, but in practice the results have actually become worse. This is demonstrated in Figure 69; we can clearly see that the overall performance of the algorithm has actually deteriorated slightly. The only reason I can find for this is that Cyclic Reseeding is very 'cache-unfriendly'. With the modification in place, each core will constantly need cache page updates because other cores have written into the same area in main memory. So even though the load-balance has improved, this has been completely negated by penalties imparted by the cache system. I therefore decide to drop Cyclic Reseeding and just use the standard implementation instead.

The DFS algorithm seems to perform best for long paths whereby a lot of flooding needs to be done; in such cases we can expect a speed-up between 1.5 and 2.5 times that of a classic A* search (provided we apply cost threshold relaxation). On small to medium paths, though, there is not much to be gained using DFS. I think this is also in part due the nature of the heuristic function: it is designed to restrict the flood boundary as much as possible and only flood in the most promising direction, which can really only be followed by a single core at a time. Clearly, 'load-balancing' is the main issue with this algorithm. The remaining challenge is, though, finding a way to do this without running into more cache penalties, this will require considerable further testing and experimentation.

To summarize, the pros and cons of Distributed Fringe Search:

Advantages:
- Scalable (2 or more cores).
- Virtually no 'overflooding'.

Disadvantages:
- Harder to implement: requires special techniques to optimize, more synchronization moments.
- Uses a lot more memory in order to circumvent race-conditions.
- Less than optimal paths.
- Harder to get effective load-balancing.
- Can be 'cache unfriendly'.
- Is only useful on longer paths.

7.3.3 Parallel Hierarchy Search

Another obvious way to successfully utilize multi-core architectures is to have each core find small segments of the total path. Small searches whereby the segment's goal node is relatively close to its start node are significantly faster because far less nodes will become flooded (think of the 'leaf' shape again as shown back in Figure 61 on page 84). In order to be able to do this we will need to somehow guess where way-points will be located in the search space, so that we can 'connect the dots' between them as it were. This can only be done properly if we have access to a high-level graph representation of the actual graph to search through. With this high-level graph we can roughly guess how the full path will traverse the search space and obtain way-points from it (hence I came up with the name 'Parallel Hierarchy Search', or PHS for short).

There are many techniques to obtain such high-level graphs, ranging from manually adding way-points to automated schemes such as ‘Probabilistic Roadmap Method’. For a more detailed discussion I refer back to subsection 4.4.4 about NavMesh graph hierarchies. For my experiments, I created a grid randomization algorithm with a top-down approach that generates 'chambers' that are linked with smaller corridors, which are then filled with randomly placed obstacles. This enabled me to easily generate lots of correct high-level hierarchies so that I could run large test batches. An example of this can be seen in Figure 70.

Each node in the high-level graph is connected with a corresponding anchor node in the actual graph that needs to be searched. The first step in the PHS algorithm is to find a path through the high-level graph so that we know which anchor nodes we will need to connect. Doing this will however never result in an immediately natural looking path because the anchor nodes might be needlessly off course. So a second 'beautification' step is applied by constructing new way-points halfway at the found path-segments (by just picking the middle node of the path-segments sequence of solution nodes). The idea behind this is that it will help us find 'short-cuts' between the high-level way-point anchor nodes, as demonstrated in Figure 71. In a sense this has some analogies to 'Simulated Annealing' methods whereby the quality of a path is improved upon in successive refinement iterations. From my experiments, I found that just a single additional beautification iteration already yields acceptable results.

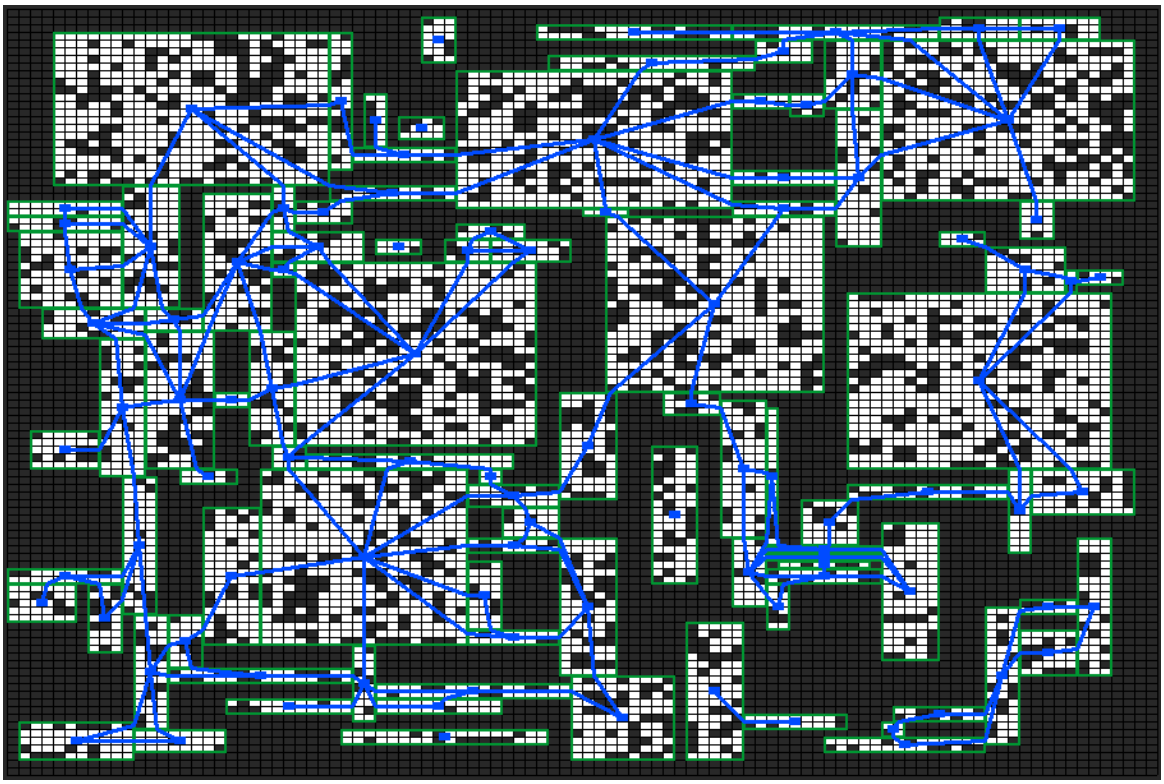


Figure 70: An example of a randomly generated 'game map' consisting of interconnected rooms. The high-level graph is represented by the connections in blue.

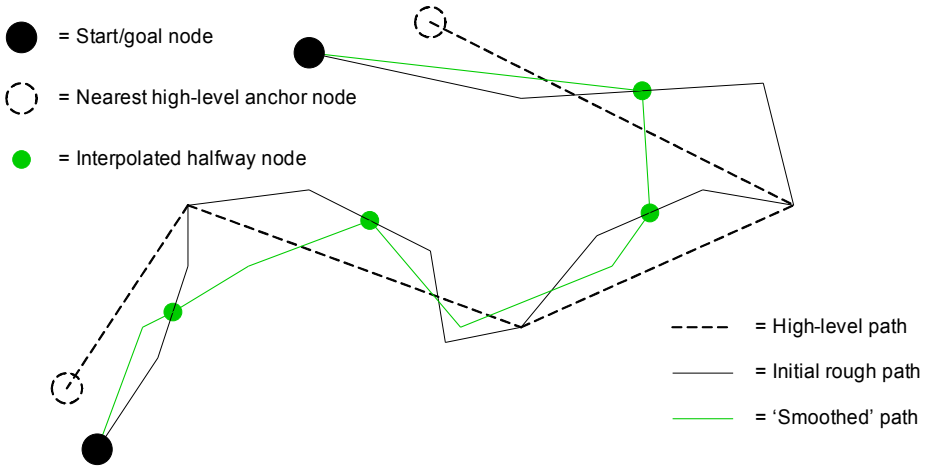


Figure 71: An impression of how a smoothed path is created from an initial high-level path.

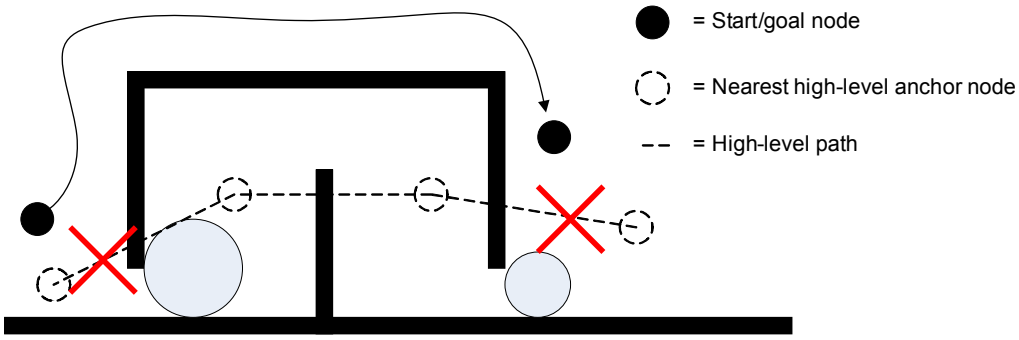


Figure 72: When parts of the high-level path are actually blocked we need to find ways around the obstacles. This means that we can no longer rely on path-finding short segments, now a long path is required.

An added bonus to using high-level graphs is, of course, that if there is no path available we can terminate very quickly and save us a lot of work. The downside to this is that a high-level path will only tell us something about the general static topology of the map. In a sense it can only tell us if it might be possible to move from room A to the adjacent room B, but it cannot tell us if the door connecting them is actually accessible. The door itself might be temporarily blocked by other travellers moving through it or there might be other obstructions which actually yield room B inaccessible. In my experiments this possibility is introduced by randomly solidified cells: there is always a chance that adjacent rooms are not directly accessible.

For the PHS algorithm, this has some serious impacts. We basically generate a set of way-points twice (once from the high-level graph anchor nodes and then from the halfway nodes) that we will try to connect with 'sub-path segments'. Each time we fail to construct such a sub-path we cannot immediately assume that no path exists between the specified start and goal node for the whole path. It could well be that the high-level graph suggested a path initially through an area that is currently blocked and that there actually exists a path around it (see Figure 72).

The basic steps that are required for Parallel Hierarchy Search are defined in Algorithm 21. Note that the algorithm can be optimized quite a bit. The sub-path segments list does not need to contain sub-path segments that, for example, got canceled, and it could be linked directly to way-point node list *WPNL*. The algorithm will keep on discarding way-point nodes until it has managed to construct sub-path segments that can all be concatenated 'head-to-tail' into a single path. The expression "SizeOf(*WPNL*) not equal SizeOf(*WPNL'*)" is only there to make sure that the algorithm stops when there is actually no path we could find directly between start node *S* and goal node *G*. This extreme situation can occur if all way-points nodes got discarded. After a while, *WPNL* will only contain nodes *S* and *G* and we only need one more iteration in a final attempt to find a path between them. If *WPNL'* still contains nodes *S* and *G* after that, then the algorithm will finally terminate with a 'path not found' state. During my experiments, I noticed that it is actually quite rare for the algorithm to need more than 2 'beautification' iterations in order to discard way-points that it could not reach, which is a good thing (although never a solid guarantee, of course). There is always the possibility however that the algorithm might need a lot more of such iterations if the situation is unfavorable; I would therefore suggest that if the algorithm fails to construct a path after the second iteration, it is better to just perform a normal path-find attempt between the start and goal directly. This will give a 'guaranteed' upper-limit to the worst-case performance of the algorithm.

In order to handle any unforeseen obstructions we need to apply the following steps to construct a sensible path between start node *S* and goal node *G*:

Initialize:
- No high-level path found yet: *HLL* = {}.
- No anchor nodes found yet: *ANL* = {}.
- No way-point nodes found yet: *WPNL* = {} and *WPNL'* = {}.
- No sub-paths found yet: *SPL* = {}.
- *PathFoundFlag* = false.

Find a high-level path and store the segments **in** list *HLL* (if it does not exist then fail immediately).

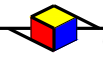
Construct the 'initial guess' list of anchor nodes *ANL*:
- **Add** start node *S* **to** list *ANL*.
- **Add** the successive anchor nodes of the high-level path *HLL* (but not the very first and last one).
- **Add** goal node *G* **to** list *ANL*.

Construct the first way-point node list *WPNL*:
- **Add** start node *S* **to** *WPNL*.
- **For each** 'line-segment' between successive anchor nodes ANL_i and ANL_{i+1}:
  - Find path between nodes ANL_i and ANL_{i+1}.
  - **If** such a path exists **then** take the halfway node and **add** it to way-point node list *WPNL*.
- **Add** goal node *G* **to** *WPNL*.

Repeat while not *PathFoundFlag* and SizeOf(*WPNL*) not equal SizeOf(*WPNL'*).
- Optimism: *PathFoundFlag* = true.
- Start tracking alternate way-points: *WPNL'* = { start node *S* }.
- **For each** 'line-segment' between successive nodes $WPNL_i$ and $WPNL_{i+1}$:
  - **Search** list *SPL* to see if we already have a sub-path between nodes $WPNL_i$ and $WPNL_{i+1}$:
    - **If** sub-path exists **then**:
      - Add $WPNL_{i+1}$ to *WPNL'*.
    - **Else**:
      - **Find** path between nodes $WPNL_i$ and $WPNL_{i+1}$.
      - **If** path exists **then**:
        - **Add** its solution **to** sub-paths list *SPL*.
        - **Add** $WPNL_{i+1}$ **to** *WPNL'*.
      - **Else**:
        - *PathFoundFlag* = false.
        - We want to implicitly discard the node to which we cannot find a path (but never the goal node), so **if** $WPNL_{i+1}$ equals goal node *G* **then**:
          - **Add** goal node *G* **to** *WPNL'*.
- *WPNL = WPNL'*.

If *PathFoundFlag* **then** we can construct the final path by concatenating all the sub-paths segment stored in *SPL* for successive pairs of nodes in *WPNL*.

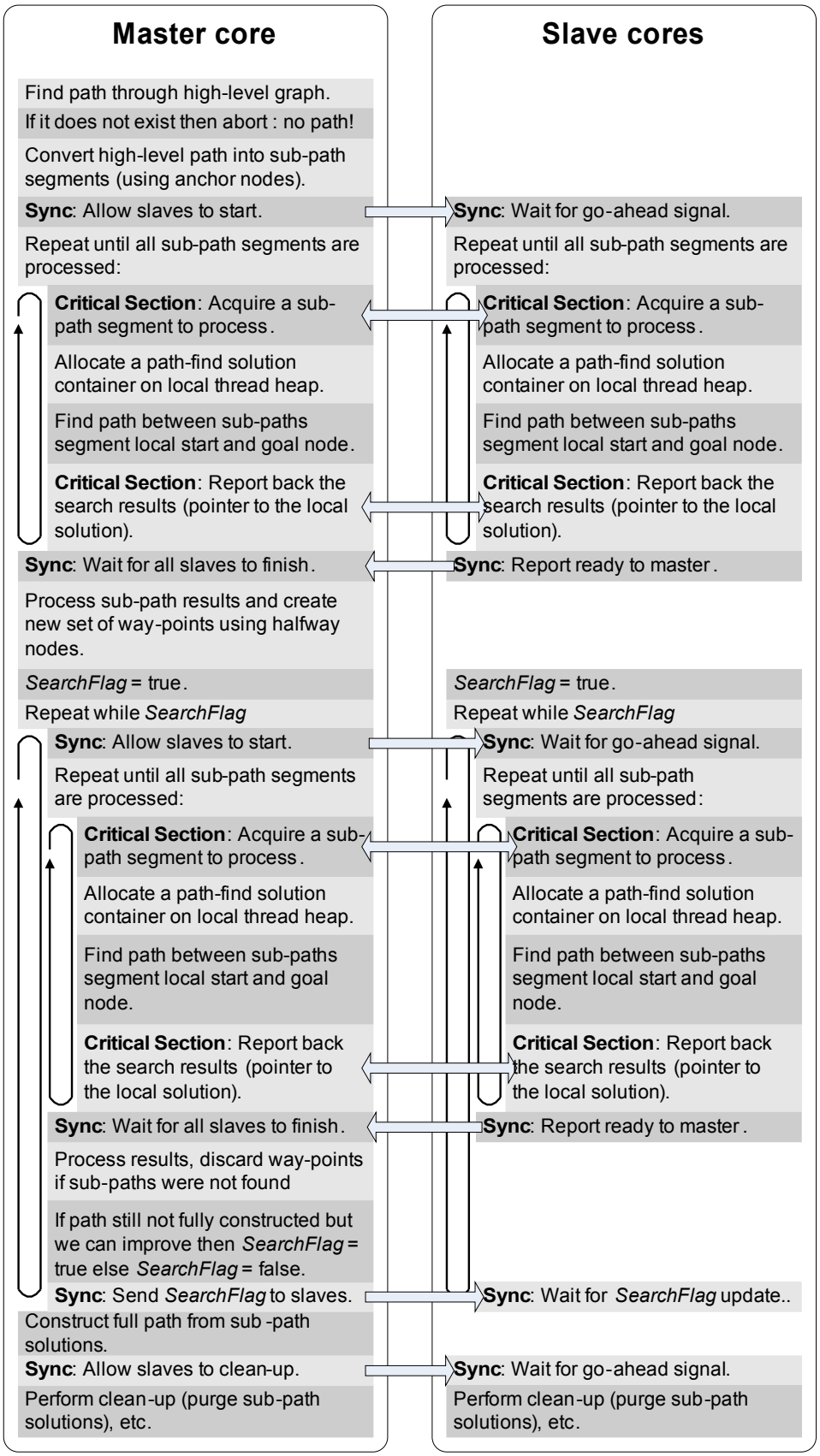
Algorithm 21: The basis of the Parallel Hierarchy Search path finding algorithm.



Parallelizing the algorithm is basically a matter of having the master core generate the way-points of sub-path segments and then letting all the cores try to construct them. To enable the cores to gain access to the sub-path segments information buffer, we are forced to employ a more expensive 'critical section' which is provided by the operating system. This will allow safe access to selecting a sub-path segment and returning a pointer to found sub-path solutions (cores are not allowed to clear these solutions until all processing has been completed, that way the master core can safely access them). We can give each core its own copy of the graph so that there will be no cache collisions during the searches themselves. The entire flow of the algorithm is roughly defined in Algorithm 22.

In Figure 73 we have an example of a path that has been generated using the PHS algorithm. We can clearly see though that PHS does only flood nodes in the near vicinity of the final path and does not fan out into a 'leaf' shaped flood space as the classic A* implementation did. The algorithm was about 1.7 times faster than a classical A* approach, but this came with a penalty. The resulting path has a noticeably higher cost (roughly 5%), and has a less 'smooth' appearance. In the middle of the figure we can see that the high-level path is distanced quite far away from the optimal path (as could be expected). With that in mind, I think it is quite safe to conclude that the 2-phased smoothing does actually work quite well; most of the initial high-level anchor nodes are no longer part of the resulting path, which is exactly what we intended. Finally, on the right we see that load balancing is fairly acceptable. We see a lot of nodes that have been flooded by multiple cores, but this is again due to the 2-phased smoothing. Because cores will process new sub-path segments when they are done with the previous one, we will see some nodes being flooded by different cores in different phases. We could potentially try to re-assign sub-path segments so that these will be processed by the same core again, but with such small path segments I doubt it would make much difference. Also note that we can never have a 'perfect' distribution because, depending on the amount of path segments and their lengths, each core may process a different amount of them.

The results that I obtained with PHS in general are rather divergent. In some cases we can obtain very good speed-ups such as shown in Figure 74, and in other cases we do not. The overall path quality leaves a lot to be desired; often the paths will stray quite a bit from the optimal path (which also lengthens the duration of the search). The causes of some of these problems are the same as we already saw earlier during the discussions of finding paths through NavMeshes, in the beginning of section 5.1. There we saw that in situations such as shown in Figure 37 (on page 44) that using anchor nodes can pull the path-finder way off course from the intuitive path. As discussed before, there are relatively cheap solutions for this that could be implemented whereby we locally enhance the resolution of the graph where needed (see Figure 38 on page 44).



Algorithm 22: The Distributed Fringe Search path finding algorithm for a direct graph in pseudo-code. The arrows indicate data being exchanged implicitly or explicitly as part of synchronization between CPU cores.

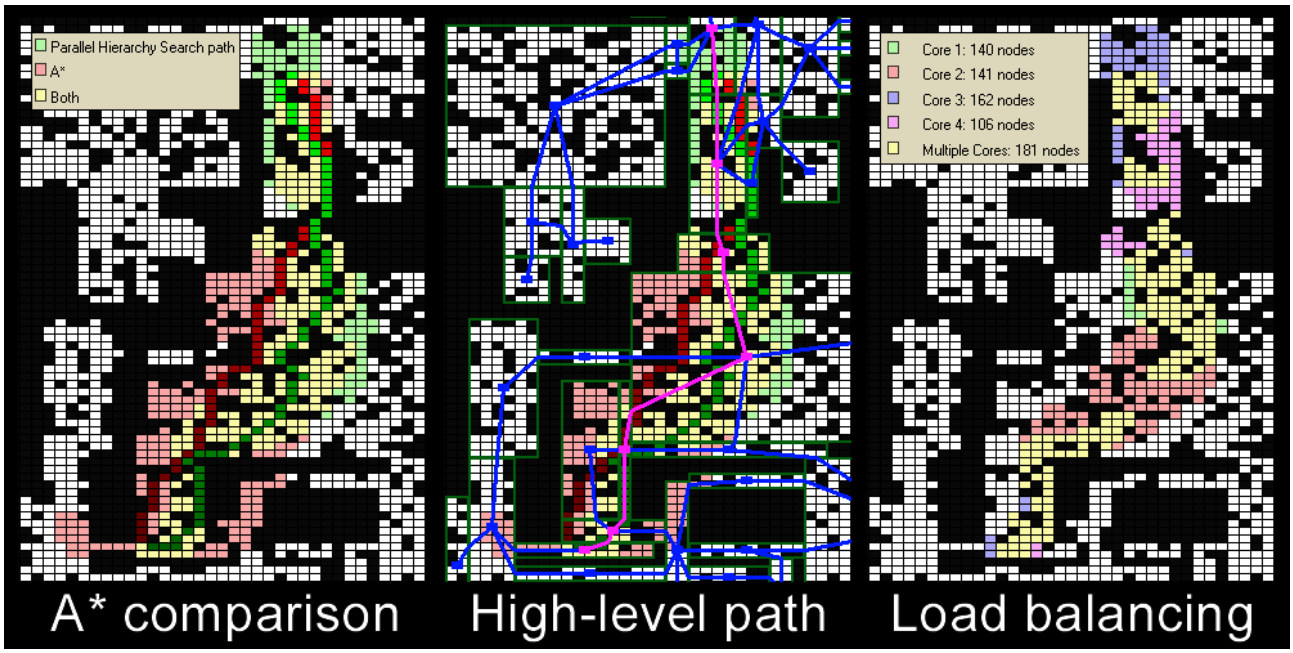


Figure 73: An example of a path generated by the PHS algorithm. On the left we see that substantially less nodes have been flooded than with classical A, but the resulting path is not as optimal and smooth. In the middle we see the high-level path that was used to form the resulting path (notice how high-level way-points have successfully been 'smoothed out'). And finally on the right we see how the flooding has progressed per core. We see a lot of overlap because of the 2-phased smoothing process whereby sub-path segments are assigned arbitrarily to any available core.*

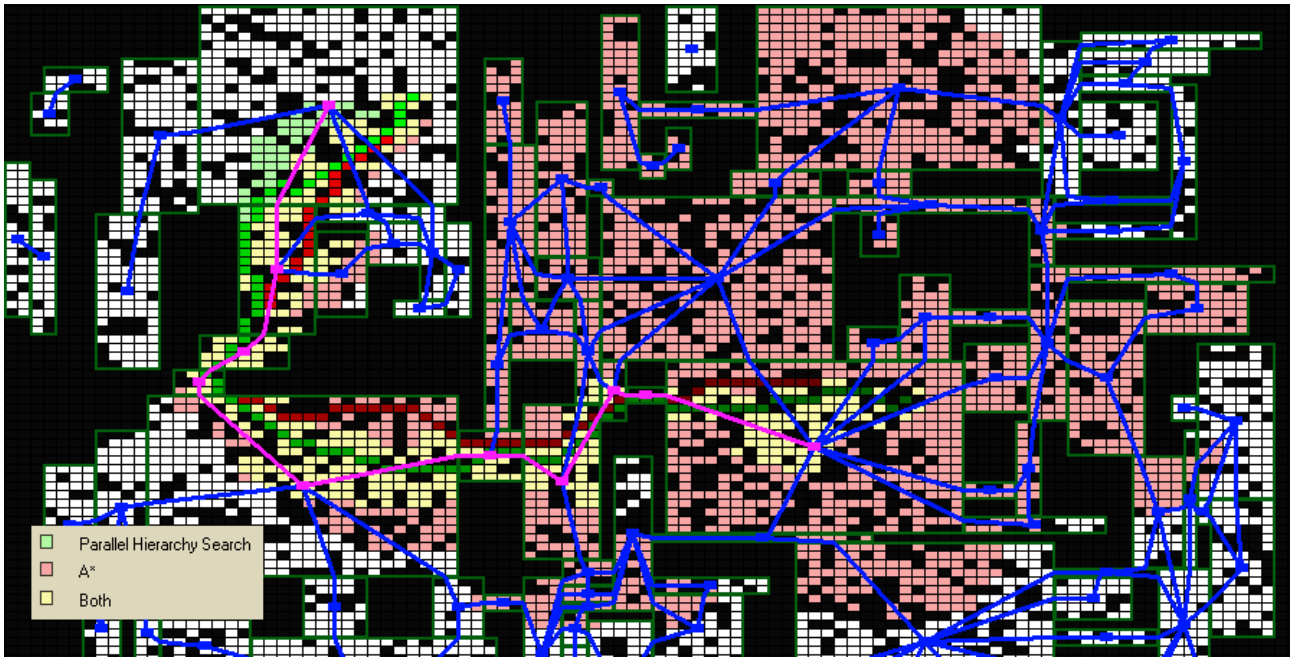


Figure 74: An example whereby PHS runs 6.5 times faster than a classic A implementation. The usage of a high-level path has enabled PHS to restrict its flood area considerably. Where classic A* needs to search extensively for a way through the walls (the heuristic function will initially pull it into a dead-end), PHS already immediately knows where to go.*

Figure 75: An example whereby PHS generates poor quality paths. In this case a simulated 'dynamic obstacle' is blocking an edge in the high-level path and thus PHS will try to find a way around it. However this means that parts of the resulting path are still pulled towards the way-point nodes, which now do not make sense any more compared to the optimal and more intuitive path.

There are however situations that cause serious problems for the PHS algorithm. In Figure 75, for example, we can see that a high-level path was actually partially blocked by some imaginary dynamic obstacles. As a result, part of the final path is skewed and makes a strange detour. These problems are hard to solve without some sort of additional (but probably expensive) look-ahead logic that will be able to detect troublesome high-level anchor nodes. This problem is not limited to pathways that get blocked by dynamic obstacles. If a dynamic obstacle would be standing on an actual anchor node then this will also be immediately problematic. We could circumvent this partially by not using a single anchor node but use a number of nodes instead that are clustered around the anchor. The path-finder should then not force itself to find the actual center node but will also be happy to find one of its close neighbors. This will however introduce much more overhead and will still not give us any solid guarantees.

It turns out that applying any threshold relaxations for this variation does not prove to be beneficial. The reason for this is that we often only need to find short paths, so flooding larger areas will not be a significant advantage.

Potentially, PHS should scale much better on more cores than the DFS variant. DFS needs some time before all cores get work assigned to them and if there are many of them then the work will be highly fractured. This will result in many 'race-conditions' for which we already concluded that these cause significant overhead due to cache penalties. PHS does not suffer from this because we can immediately assign work to all the cores, and the work itself can all be done locally without any cache problems.

To summarize, the pros and cons of Parallel Hierarchy Search:

Advantages:
- Intuitive.
- Scalable (2 or more cores), much more effective than DFS.
- Exhibits 'cache-friendly' load-balancing.
- Can use any A* variation.

Disadvantages:
- The algorithm self is easy to implement, but generating the high-level graph that is required is not (automating this requires a lot more work).
- Paths quality varies greatly (often significantly less than optimal).
- 'Overflooding' (but can potentially be kept to a minimum).

7.4 Results

My parallelized implementations have tried to use dynamic memory allocations when possible in an attempt to get a decent physical separation (see the discussion back in section 7.3). All the measurements that have been performed were repeated 5 times for each implementation variant so that cache content would 'stabilize'. The best result of all the taken samples was then taken as the ultimate measurement result (this so that we can more or less exclude any 'noise' induced by the operating system's task switches and such). Also, all threads and processes were running on highest priorities. All samples have been taken on a 2.4 GHz Intel Core2 Quad CPU running Windows XP Pro SP2.

A single sample measurement spans the time it takes for the implementation of the algorithm to get its context as input and it giving the corresponding output. This includes the time it takes to perform necessary clean-up such as reseting the 'visited' flags in graphs and such (this is important because with less nodes flooded per core, we also need to do less clean-up). Other non-relevant programmed 'side-effects' such as gathering data to visualize the flooded areas were of course excluded. Where needed, pre-allocated buffers were used that all have the same size for each of the implemented variant: 4K of logical items. The setup time for creating, configuring and destroying threads on the cores has been excluded from the measurements because these are dependent on all sorts of operating system specific behaviors. Also, any real-life applications will most likely have such threads already primed to run large batches of path-find requests. Similarly, I have used software 'spin-locks' wherever possible to synchronize the threads. The reason for this is that it will give us the best guarantees that the operating system will not interfere with thread scheduling. Also, it will give us more accurate measurements because there are no expensive operating system mutex and semaphore related mechanisms that need to be 'kicked into gear'; all threads will immediately respond when a spin-lock is released by the master core.

The graph that has been searched for taking the samples is an 8-way connected uniform grid (= octiles). Each cell in the grid is represented by a single graph node and neighboring cells are connected by directed edges. The reason for this is that it more or less resembles your average NavMesh topology and connectivity but the grid makes it a lot easier to visualize and interpret the results. For similar reasons, the euclidean distance between grid positions has been used for both cost estimation and as an A* heuristic function. Each individual sample takes 2 randomly selected nodes from a 200 by 200 graph grid which must both be accessible (it is pretty pointless to test how

Path classifications	
Very Short	Less than 50 nodes
Short	Between 50 and 100 nodes
Medium	Between 100 and 150 nodes
Long	Between 150 and 200 nodes
Very Long	More than 200 nodes

Abbreviations	
A*	Classic A*
FS	Fringe Search
PBS	Parallel Bidirectional Search
DFS	Distributed Fringe Search
PHS	Parallel Hierarchy Search

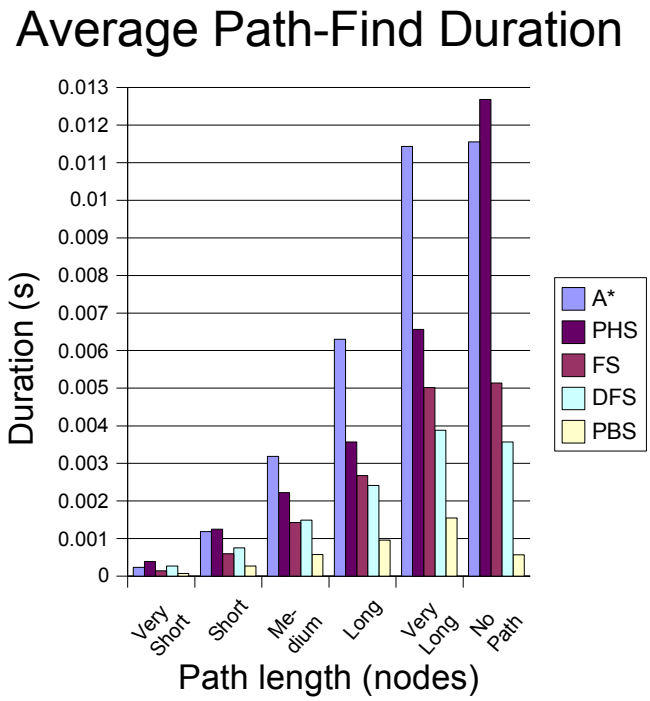


Figure 76: Overview of average path-find durations for a set of 2000 random samples with threshold relaxation set to 5. Note how PBS can outperform a classical A implementation by up to 7 times.*

fast the algorithms abort when specifying the inaccessible nodes that are depicted as black cells in the examples). Because Fringe Search plays such a prevalent role in my experiments, I have used this algorithm as an A* alternative wherever possible. I have also included a normal Fringe Search in all the measurements so that we can clearly tell if the speed-up is due to the parallelization, and not just the fact that Fringe Search was used instead of a classic A* implementation. Each batch run consists of 2000 samples by trying 100 random paths in 20 random maps.

The paths have been classified into a length category based on how many nodes the classic A* found for its path, the results of which are shown in Figure 76. We can see that for very short paths the results are mixed which probably means that parallelization has too much overhead in comparison with the actual amount of work that has to be done. When the work-load increases we can clearly see that the parallelized algorithms start to outperform the classic A* implementation

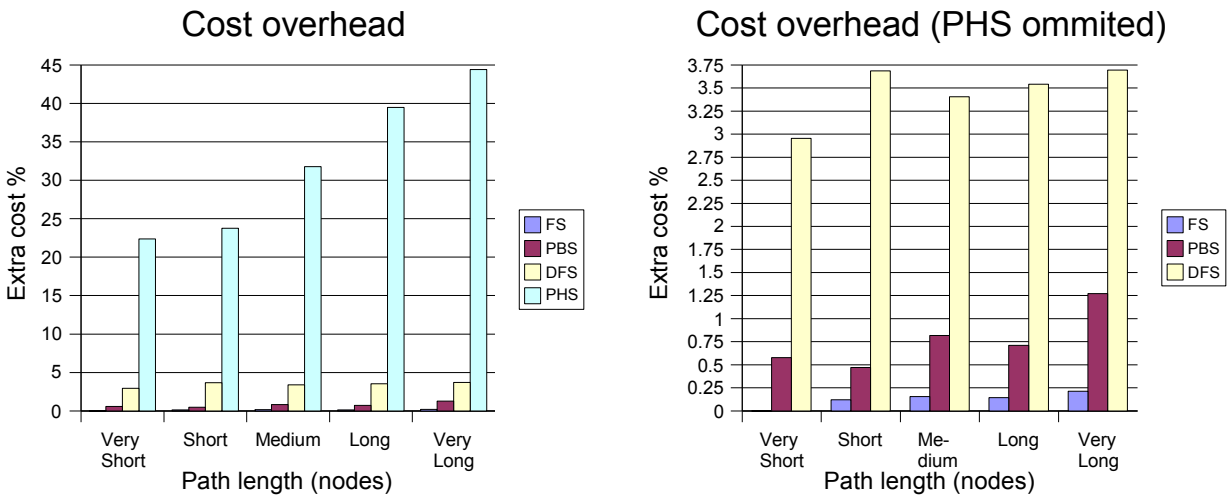


Figure 77: Average cost overhead for the the A variants for 2000 random samples with threshold relaxation set to 5. Parallel Hierarchy Search has been omitted from the figure on the right so that we can visualize Fringe Search's penalty better.*

extensively. It is also demonstrated that Fringe Search is significantly faster than classic A* on medium paths or longer. It even manages to beat PHS consistently which is quite surprising. I think that the use of the critical section to control access to shared lists between the cores is largely responsible for this. That, and the fact that we have to perform 2 'beautification' iterations means that on average PHS has a lot of extra work to execute. We should not expect it to run 15% to 30% faster than a classic A* implementation, and certainly not faster than a normal Fringe Search implementation. We also see that PHS actually performs far worse in cases when no paths could be found. This sounds very counter-intuitive because the high-level graphs it uses should give us early outs that save a lot in performance. Although this is true, we should not forget that PHS performs rather badly if the high-level graph indicates that a path should be available while in reality there isn't one because of dynamic obstacles (it will basically run the full 2 iterations followed by a normal path-find attempt). Apparently such cases are considerably more arduous to solve for PHS, so much so that on average it completely negates the early out effect of the high-level graph. For DFS we can conclude that it is only worthwhile on longer paths; otherwise the normal Fringe Search implementation still outperforms it (though not by much). It is apparently only for the longer paths that cores manage to get work done without interfering too much with each other's caches (which makes sense because the flooded areas will be much larger and further spaced apart). We can expect a 40% to 55% speed-up from DFS in relation to a classic A* implementation. And finally, on to the winner, which clearly is PBS on all fronts: an impressive 60% to 85% speed-up. As expected, it even performs well when no paths could be found; in which case the duration is bound by whichever core terminates first.

From these results I think it is safe to conclude that cache penalties have by far the largest impact on the performance of these algorithms. The longer we can prevent the cores to flood nodes in each other's areas, the better the results will be. PBS clearly does this best because it starts its search on both cores at the maximum distance apart from each other. It also explains the poor performance of the DFS algorithm: all the cores are continuously in each other's way and cause many 'race-

conditions' while trying to open the same nodes. I also think that PBS performs so well because there is virtually no idle time on any of the cores: both of them are fully busy until the collision is detected. This is not the case for the other parallelized variants and thus they suffer from it. For DFS, each iteration is bound by the slowest core (for which we have seen that it is not trivial to get the load-balancing right). This is the same for PHS whereby the work-load per core can be even more 'skewed' because there is no active load-balancing at all. This is made worse by the fact that PHS performs rather badly when the initial guesstimate of the high-level graph turned out to be off: dynamic obstacles can force a core to do a lot of extensive flooding in order route around the problematic areas.

In Figure 77 we see the path cost overhead that is incurred by all the A* variants. Not surprisingly, PHS has by far the worst path cost overhead which can range between anything from 20% up to 45%. Combined with the sometimes very awkward paths that PHS generates I think it suffices to say that this implementation variant in its current form is just not practical. The difference between PBS and DFS is also quit considerable. For DFS we already had expected that its performance would be hampered by the fact that it has lost its corrective property in a far more significant degree than that of PBS. The reason that DFS's cost overhead is fluctuating a bit is because it is partially influenced by race-conditions. In some cases, nodes will get accidentally flooded from what will turn out to be the most optimal node and hence the resulting path quality will be improved a bit. For PBS, the loss of its corrective property is only accumulated near the area of the collision node, which is generally very small. This means that we can expect that the introduced error will be roughly the same for any length of path. Even for very long paths it still only generates paths that are less than 0.25% more expensive than the most ideal path, which is negligible.

The largest penalty of any parallel implementation seems to be the additional memory that is required. For all my implementations, memory consumption increased linearly with the amount of cores I used. The main reason for this is that it is often best to store process data in the graph so that look-ups are virtually free. PBS requires the least amount of extra memory because it already needs to share the 'visited flags' between both cores anyway (so no need to duplicate those). PHS needs a bit more memory because it needs to store sub-path solutions for all cores until the master core can process them all. But, because the sub-paths are often very short, these additional memory segments are often relatively small. DFS takes up the most amount of memory because it needs to store all of its process data in separate memory blocks so that cores do not overwrite each other; only the visited flags of the nodes can be safely shared. Added to this come the Shared Balance Lists which need to be large enough for worst-case scenarios (otherwise trying to scale these up dynamically during path-finding will induce a very large penalty).

Clearly it is very worthwhile to attempt to use parallelization in order to speed up the path-finding process as long as we are willing to accept that the results will no longer be fully optimal paths. From all the previous measurements and discussions, it is clear that PBS is the most viable alternative. Not only is it easy to implement, it also performs very well (both in terms of speed and path quality) and requires less additional memory than the other variants. Its main strengths are the optimal use of each core (they are never idle) and the fact that it does not need any explicit synchronization (collisions are detected 'automatically' via the core's caches). Because PBS exhibits a cache-friendly signature, the cores can go about their business largely uninterrupted. It seems that the Intel Core2 Quad CPU I have been using has a really good cache look-ahead prediction as long as shared pages do not get written into too often.

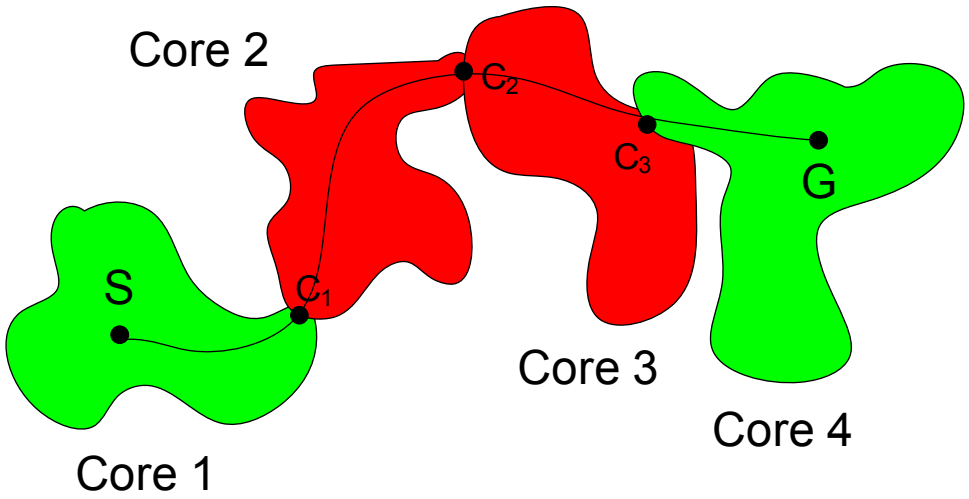


Figure 78: An impression of 'Parallel Ripple Search' whereby a path is constructed by first flooding the search-space between start node S and goal node G at 'equidistant' positions. As soon as enough collisions have been detected we can construct the full path.

7.5 Future improvements

The algorithms that have been introduced and discussed previously certainly leave some room for improvements. As the new algorithms already demonstrated that they can obtain modest to good speed-ups (which was the main focus of the experiments), we are primarily concerned with the suboptimal paths that come as a penalty.

The past experiments have been performed on a uniform grid with euclidean distances as travel costs. In real-life scenarios, we might have all sorts of additional travel penalties applied which will deteriorate the quality of the paths. The problem here is that if the corrective aspect of an A* variant is lost, it will in these cases severely underestimate the travel costs and have 'bad' paths as a result. An obvious solution would be to somehow improve the accuracy of the heuristic estimation function (= *h(n)*), but this is no easy feat. Perhaps it would already help if we include an estimation of the true travel cost between node *n* and its neighbor, but this will come at a severe performance penalty. I think it will prove more fruitful to try to fully regain optimality again or, at least, as much as possible.

The easiest algorithm to turn optimal again is the PBS variation. This can be achieved by continuing the search on the master core after a collision has been detected (the slave core must stop in order to prevent race-conditions). From the collision node, we can directly compute the full cost of the path through this node and use that as an upper bound for the rest of the search. While the master core floods the remaining nodes it should keep track of nodes that have also been flooded by the slave core in order to determine which of the nodes should ultimately be chosen to plot a path through. Note that there is always the off chance that the master core might find a cheaper path that does not cross any of the nodes that was flooded by the slave core. Still, from the experiments we concluded that the average 'overcosting' of the paths was very low which would indicate that PBS is never far from the optimal path. The additional amount of time spent in searching through the remaining options should thus be small.

Regaining optimality for the DFS variant is going to be impossible (or at least not without completely redesigning the algorithm). Reasons for this were already discussed: we do not want cores to interrupt each other continuously because this will severely degrade the overall performance. My current implementation has thus completely omitted the corrective aspect. We could partially re-introduce it again if we are willing to sacrifice a lot more memory. If we mark, for each node, on which core it is located, we could at least try to perform corrections if both nodes are currently owned by the same core. The downside of this is that it will be more work to perform load-balancing between the cores; each time a node is reassigned we also need to update its core index accordingly (thus also if we rebalance a Shared Balanced List). I think this will introduce quite a bit of overhead, maybe even too much.

Finally, I think that the PHS is a bit of a dead-end. Although it will scale a lot better for larger amounts of cores and for very long paths, its overall performance has been rather disappointing. The main problem is its very poor path quality, which can only be improved upon by adding more expensive look-ahead logic, able to generate more sensible way-points.

I think we should better take a new direction and combine some of the results we have seen into a single new algorithm. From my experiments, I concluded that, as long as cores do not need to write in each other's cache pages, performance will be excellent, so we should try to capitalize on that. The PBS algorithm already does this but it cannot scale beyond two cores. My suggestion thus is to use a high-level graph to find half-way points for other cores to start flooding areas in advance; as if we are generating ripples in a pond. I have dubbed this new approach the 'Parallel Ripple Search', or PRS for short. It is illustrated in Figure 78.

The big potential of PRS is that when we find enough collisions, we can 'short-circuit' and find paths through previously flooded areas (for which we know that they must exist). With a bit of luck these areas might still be (partially) lingering in a core's cache so access should be fast. This algorithm will also be able to deal with dynamic obstacles a lot better than PHS could. If way-points turn out to be blocked, then this will just mean that adjacent ripples will not collide. So although it might take longer before a collision occurs with a ripple located further away, we will no longer run the risk of pulling the path in weird directions. The cores that flood from the start node S and goal node G compute what we could call the 'essential' ripples. In a worst-case scenario whereby none of the non-essential ripples ever collide with the essential ones we will basically have degraded to a normal bidirectional search, for which we now know that it performs very well. The advantage of this new approach is that we can utilize many more cores; basically one for each segment of the high-level path. For the non-essential ripples we could use heuristics cost estimation functions that are either biased to grow towards nodes S and G, or towards the way-points of the adjacent ripples:

$$h(n) = \min(\text{Estimate}(S, n), \text{Estimate}(G, n))$$

With Estimate() being a cost estimation function such as the euclidean distance.

How this algorithm performs in practice remains to be seen, regretfully there was not enough time to include it in my experiments.

8 Evaluation

This chapter looks at the differences between Triumph Studios' navigation system and the new NavMesh system, and gives an overview of the advantages and disadvantages of both.

8.1 Triumph Studios' system

Triumph Studios' path-finding approach for Overlord 1 and 2 has been rather straightforward in its design and consists of a uniform grid of nodes, spaced apart at 0.5 meter intervals in the horizontal plane. The entire game world is divided into a uniform 3D grid using blocks of 4 by 4 by 3 meters. Each block contains a local ‘height-map’ using a uniform 2D grid of nodes (see Figure 79). These nodes actually represent accessible spaces in the virtual world and can have various attributes such as terrain types and an accessibility mask which is used to prevent certain type of units from crossing them.

This use of multiple layers of blocks will enable us to model complicated worlds containing buildings, caves, bridges, etc. Because the individual blocks span only a small space in the world we can suffice by using single byte values to encode the height of each node. The large 3D grid of blocks is generally only sparsely populated so it is possible to obtain a good deal of compression using bit-masks and run-length encoding techniques. Static scenery such as buildings are all constructed using blocks. Each such block not only contains rendering information but also the local node height-field which is then referred to as a ‘walk-plane’. It is up to the graphics designers to manually add these walk-planes in special meshes and also to limit their designs so that their complexity can still be reasonably represented by the limited amount of nodes in the walk-planes. When a static scenery block is placed in a map by a level designer, the level editor will automatically merge the walk-planes with those of the neighboring blocks and the terrain, if needed.

The main advantage of this system is its simplicity. We do not require any additional software modules that try and 'parse' the map in order to obtain abstract representations of it; the path-finder can directly perform its search on the map data. This simplicity makes the system a lot easier to implement and visualize (which also helps the level designers in tracking down problems). Because there is no need for a post-processing step, the map editing turnaround time is excellent: a level designer can immediately test his modifications on the map by loading it into the game application (this is a very important aspect!) Dynamic obstacle avoidance is implemented by using a 'footprinting' system as described earlier in section 6.2. Because the granularity of the grid is already 'high', there is however no need for an extensive subdivision algorithm. The spacing distance of 0.5 meters between nodes proved to be adequate for getting travellers to route around each other without any problems.

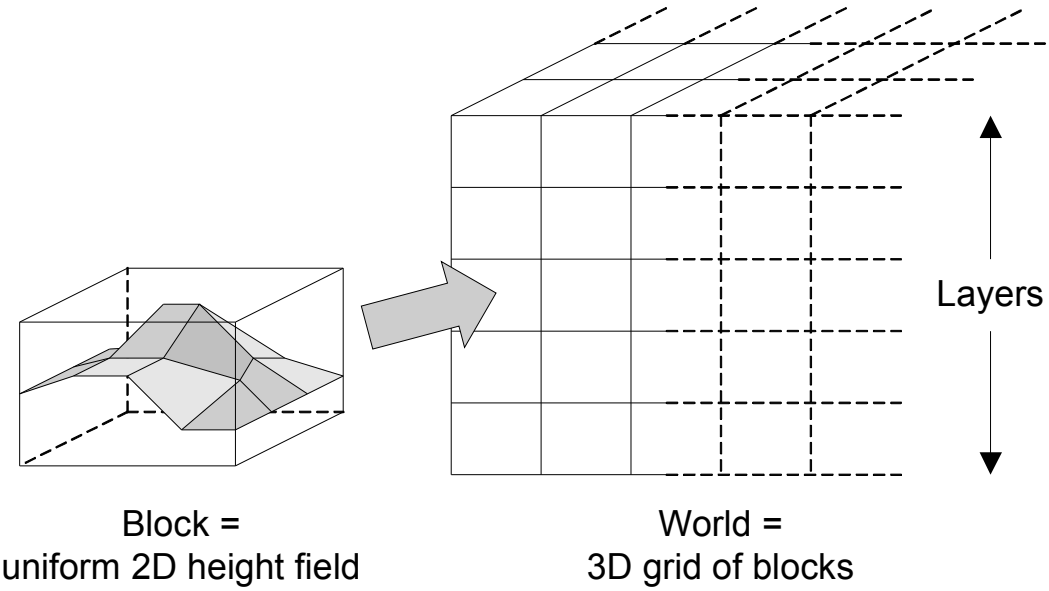


Figure 79: In Overlord, the game world is represented by small blocks containing uniform 2D height fields that are part of horizontal layers. By stacking multiple layers it is possible to obtain a fairly accurate mapping of path-finding nodes onto buildings and terrain.

The disadvantage of this system is that, even with compression, there is still an enormous amount of nodes. This not only makes the memory-footprint of the game application high, it will automatically mean a lot of nodes need to be flooded for any path-finder implementation: there is no way to efficiently represent large areas. And even though we have a large amount of nodes, it is still hard to represent hit-volumes or any other geometry accurately due to the uniform nature of the grid. Level designers are also handicapped because blocks can only be rotated in 90 degree angles; they will always be snapped onto the grid. There is also a lot of manual labor involved: all walk-planes need to be added manually by the modelers. As a result we see that some considerable time is spent between level designer and artists to tweak blocks for specific scenes in the game.

8.2 NavMesh comparison

The advantages of NavMeshes have already been widely discussed in previous chapters. They are polygon based, which means they accurately represent any walkable geometry and do this efficiently for large open areas. As a result, graph sizes are reduced which will free up precious memory and will take less time for a path-finder to flood through.

As a downside, though, NavMeshes are **much** harder to build and manage. In this thesis I have discussed an extensive number of techniques that are needed to generate them automatically from hit-volumes. Many of these techniques are not quite so trivial to implement correctly and efficiently, and suffer from all sorts of problems due to limited numerical precision. The effort needed to implement them will be considerably larger because of the complexity of the software (debugging is much more involved). As a result, I have only just been able to get the system working on a purely functional level. Optimizing it further, by improving the NavMesh quality for example, would take up more time and was thus beyond the scope of my research. The code base of all this system is already very large; roughly 75K lines of well documented code and debug utilities. From this we

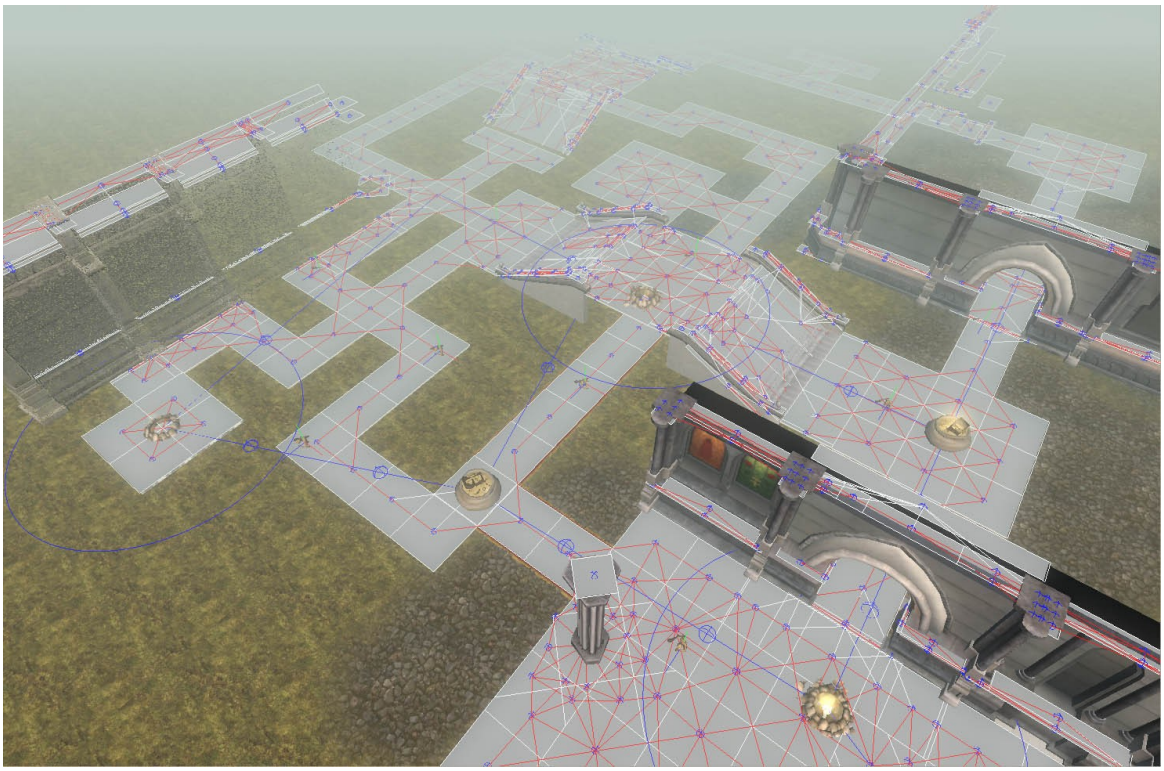


Figure 80: A performance test map with a fully autonomously generated NavMesh consisting of roughly 1000 polygons. Game characters randomly carry gold back and forth between the big bags back to the pits using an A path-finder to navigate through the map.*

can immediately conclude that maintenance and support will also be considerably more expensive in the long run when such a system would 'go live'. From my experience as a game developer the last few years, it has become clear that simpler systems are often preferred because less things tend to go (subtly) wrong. Deadlines are often very tight during a project, and with more complex systems it is easy to loose days of productivity while hunting down a complicated bug (and often at the worst possible moment).

To get a better idea of the obtained results some measurements have been taken using a test map as shown in Figure 80. As a comparison, Figure 81 and Figure 82 show the differences between the old uniform node grid and the NavMesh. Both systems cover the same areas but the NavMesh does this far more efficiently with the use of its polygons. Because my current implementation does not yet utilize the optimization techniques as described in subsection 4.4.1, we can clearly see the tiling of the map block hit-volumes reflected back in the NavMesh. If these techniques would have been implemented then the amount of nodes the NavMesh would cover would be considerably less (I would expect a 70% to 80% reduction).

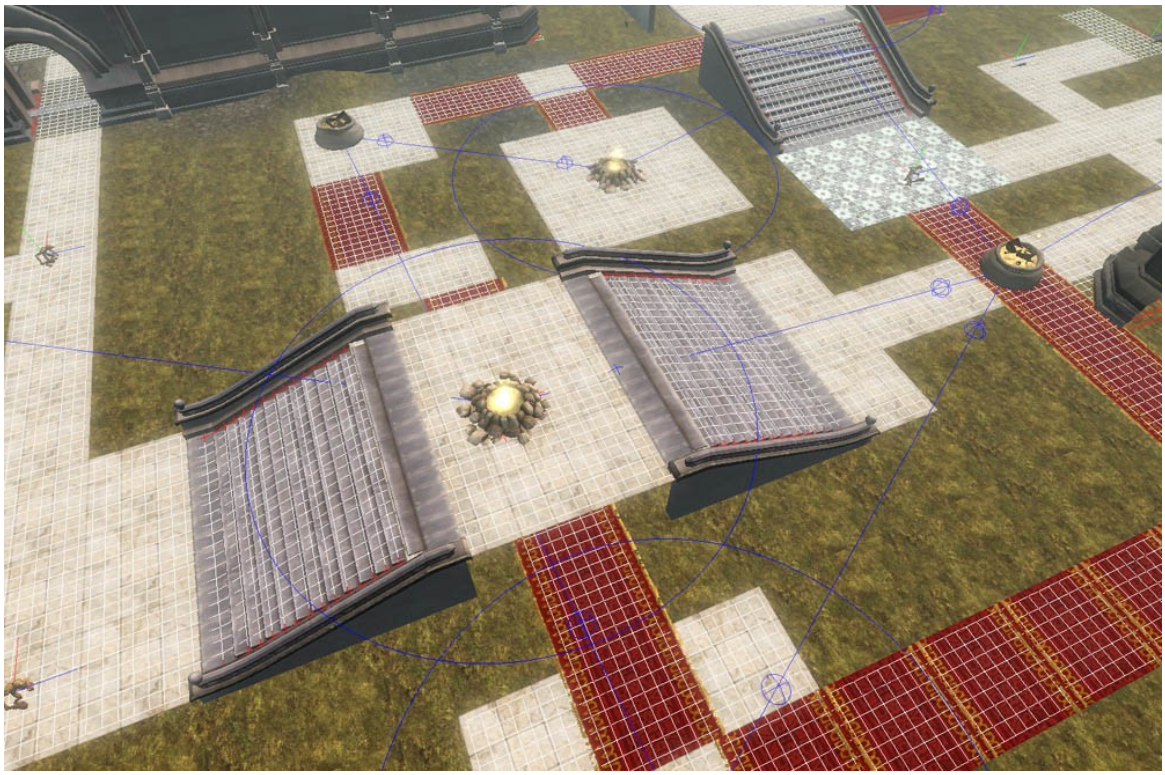


Figure 81: Example of how the uniform node grid needs many nodes to cover larger areas in the game map.

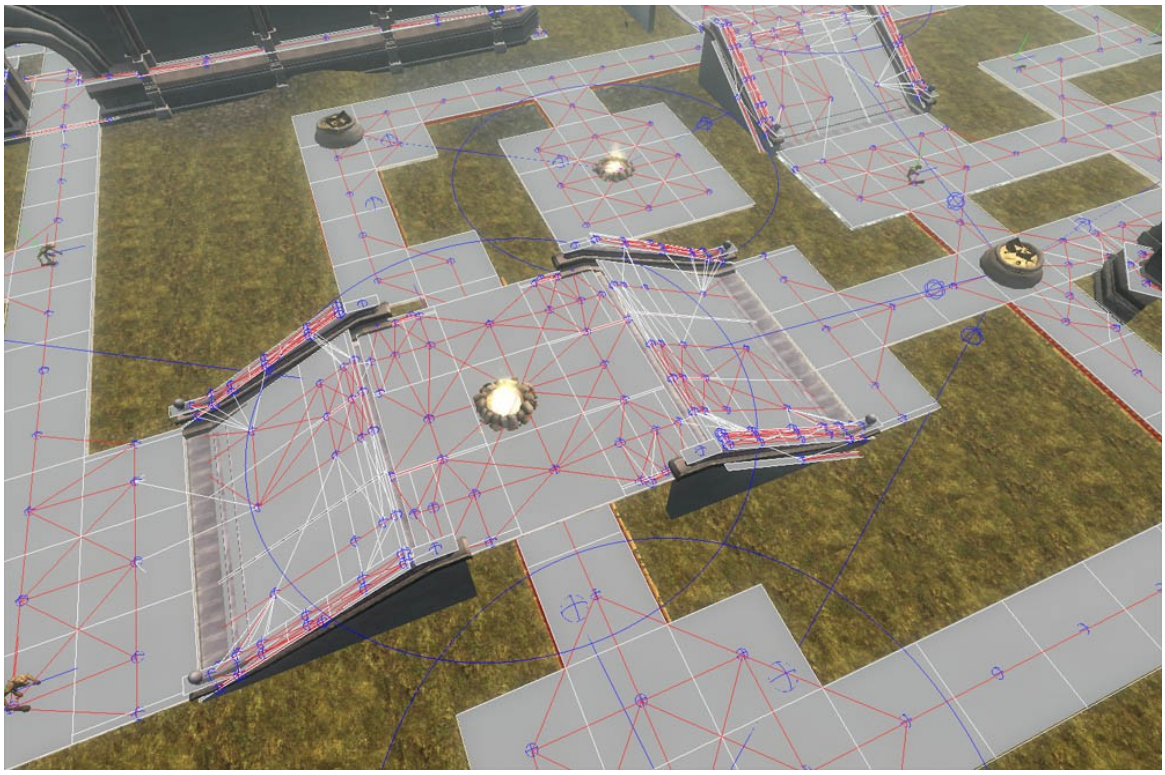


Figure 82: The corresponding NavMesh covers the same areas but does it far more efficiently with its polygons (and there is still more room for improvement).

Considering purely the nodes that are actually used by the travellers, we come to the following estimations for required storage space:

Uniform Node Grid	*NavMesh*
Roughly 18750 nodes	Roughly 990 nodes
Storage space: 21k	Storage: 49k

This amounts to an almost 95% reduction in nodes. The actual node count for Triumph Studios' uniform node grid is in reality even much higher because a terrain layer is always present. The maps are 500 by 500 meters so this translates into an additional million nodes that are always present. I have not taken this into account however because my NavMesh system is not completely finished and does not parse the terrain (also for performance reasons). So for the test map, I am only comparing those parts of the map that are actually used for the travellers in order to walk around.

Now, if we look at the required storage space required for both systems, we see that the NavMesh requires more space even though it has far less nodes. This is because Triumph Studios' system has been greatly optimized, using just 1 byte per node (a height value combined with very little meta data). For NavMeshes there is a lot more 'expensive' data to be stored such as 3D vertex coordinates, shared polygon edges, centroids, etc. These can be compressed quite a lot (using 16 bit vertex and edge indices, for example), but still leaves quite a bit of data in the end (we could potentially use 16 bit floating-point formats for the vertices, but this has not been taken into account now). Once again it shows that NavMeshes really do require polygon merging techniques before they can become fully viable. In its defense though, once loaded, Triumph Studios' system will require 4 bytes to leave room for linking information and temporary flooding meta data. For the NavMesh system we do not need nearly as much memory because there are far less nodes, so in-game it will actually take up less memory in all. Also, NavMeshes are far more flexible and will in general take up less space if levels are relatively spacious. For the uniform node grid we will always have a large amount of nodes by default, regardless of map content (the terrain nodes are always present, for example).

Generating the NavMesh for the test map seen in Figure 80 takes about 78 seconds, which is clearly prohibitive for quick cycles of testing. My NavMesh generator has a lot of room for optimization though, and perhaps we can think of schemes whereby maps are divided into sectors that only need local NavMesh regeneration (which could be potentially done as part of a background process). So although the NavMesh generator is unsuitable in its current form, it might be viable in the near future with better hardware and/or proper optimizations.

It is clear that the NavMesh system provided a much better representation of the static world. The use of polygons makes it possible to represent any part of the geometry; something which is much harder to do with the uniform node grid system. Another advantage is that we are more flexible with the radius of the travellers; for the old uniform grid system these always had to be multiples of 0.5 meter because of the nodes' spacing.

For further comparison, some performance measurements were taken by having 15 traveling game entities walk between random way-points on the map, the results of these are shown in Figure 83. It must be said that it is hard to get a good comparison between both systems because Triumph Studios' system has a hard-cap on the amount of nodes it will flood, forcing the path-finder to give

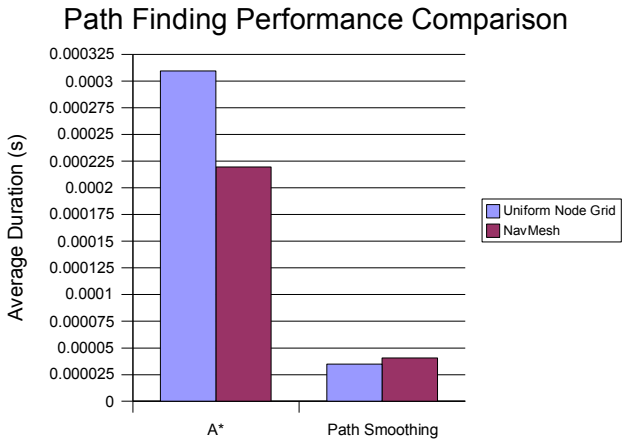


Figure 83: Overview of the average performance of an A path-finder on Triumph Studios' uniform node grid and the NavMesh.*

up 'early'. This prevents the system from bogging down when attempting to find longer paths, and is thus roughly limited to a maximum range of 25 to 50 meters (depending on how many corners the path-finder needs to flood around). The NavMesh system however does not have such limitations and can find paths through the entire map cheaply.

Figure 83 shows us that utilizing the NavMesh has already reduced the cost of path-finding by roughly 25%. This might be less than expected, especially when we consider the 95% node reduction. Again, however, this is due to the fact that Triumph Studios' A* implementation is highly optimized by tuning it for this particular uniform node grid. Their A* implementation is basically a Manhattan distance heuristic Beam Search variation that places a hard-limit on the size of the Open List (see [12]). It also uses a clever hashing mechanism that forgoes the need of a traditionally sorted Open List. It processes nodes during the path-finding session by directly accessing the node grid in a matrix form, so flooding to a neighboring node is basically moving a pointer to the next few bytes in memory. This means that it is very cache efficient: a node entry is only 4 bytes so we can load a lot of them into a single cache page. As a downside however, all these modifications make the A* implementation no longer optimal; it is no longer guaranteed to find the most economic path. We will hardly notice this in practice, however, due to the granularity of the uniform node grid, an effect that was also verified back in section 7.4. In turn, my A* implementation is a very generic one that has no significant optimizations and utilizes an explicit graph with pointer logic. Such an implementation is far less cache-friendly because the memory pointers can make the path-finder criss-cross unpredictably through memory during its searches. Also, the NavMesh quality is still low because none of the polygon merging techniques has been implemented. This means that we can expect a far better performance of the NavMesh path-finder in the future with all the proper optimizations in place.

From Figure 83 we can also conclude that smoothing the raw paths that are generated by A* takes just about the same amount of time for both systems. Those done for the NavMesh are slightly more expensive because quite a lot of floating-point vector computations are needed, whereas the uniform node grid paths can be processed with a lot of discrete math. Again, we can expect a reduction of the smoothing process duration when the NavMesh has been properly optimized and consists of less polygons.

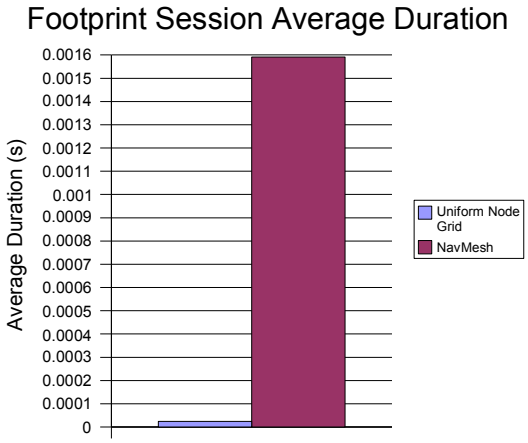


Figure 84: Performance measurement for the footprinting system for Triumph Studios' uniform node grid and the NavMesh's.

The dynamic footprinting mechanism that was implemented for the NavMeshes has proven to bring about a dramatic performance penalty (see Figure 84). This can be attributed to the large amount of memory allocations that the NavMesh systems requires to perform when increasing its resolution. Not only does it need to allocate memory for the sub-polygon representations themselves, it also needs to allocate new nodes and edges for the NavMesh graph in order to represent them. Triumph Studios' implementation has virtually no extra costs attached to it because it has chosen to sacrifice a lot of memory to have a 'high' node granularity available by default.

Clearly, my dynamic NavMesh resolution management systems needs to be improved significantly before it is of practical use. The problem is that with my current implementation we never know how many neighboring polygons and nodes you will end up with for any given subdivided node polygon in the graph. We could partially improve on this by not supporting incremental resolution enhancements but only support an 'on/off' system that will immediately highly enhance the resolution of larger sections. This means that we can precompute how many new sub-polygons and nodes will need to be created, so that these can be allocated all at once (more on this approach in subsection 8.3.2). Note that we should still largely depend on the Boids and whiskers systems to handle all the normal collision avoidance for us; only when a traveller fails to find its way around dynamic obstacles it will need to fall-back onto the dynamic resolution system of the NavMesh (which ideally should not be very often).

8.3 Alternatives

It has become clear from the previous discussions that some concessions have to be made in order to improve upon performance and stability of some of the NavMesh's systems.

8.3.1 NavMesh generator

The main advantage of the NavMesh builder that has been presented in this thesis is that it is fully automated. This makes it guaranteed to generate correct output and that no traveller will ever be 'mislead' and get stuck. The system can fully autonomously parse the input-data and extract all the

information we need for routing travellers safely through the world. This is a very powerful feature that can save a lot of work for level designers. But, as the system is currently too cumbersome to run, we need to ask ourselves if there are other practical approaches that can solve some of the performance (and stability) problems.

Other game developers have taken different approaches for generating NavMeshes. The Unreal engine, for example, relies a lot on manual definition of NavMesh polygons by allowing a designer to 'draw' them with a special tool (see [5]). This tool also allows them to add all sorts of meta data that can, for example, be used to define good positions for snipers, places to defend from, etc. This approach is completely opposite from my fully autonomously generated NavMeshes and requires a strict level design process in order to prevent having to make extensive updates when geometry needs to be modified. The Unreal game engine also relies for a large degree on physics and whiskers systems to be able to move entities through the world. As it is primarily designed for First-Person Shooters, the amount of entities it has to handle in a single scene is in general quite low (this in contrast to the Overlord games). With a lower amount of entities there will, in general, be less problems navigating them through worlds, so apparently a 'low' quality NavMesh that is manually created will suffice in most practical cases.

A similar sort of approach used by Valve and their Source engine is to generate 'rough' NavMeshes from 'rectangular navigation areas' (see [19]). These areas are generated by flooding outwards from certain way-points and implicitly limit level design to a grid (as long as the rectangles are able to fit). A level designer can then 'tweak' the NavMesh to solve troublesome spots or to add links where the generator failed to find them. This approach also eliminates the need for a 'full blown' B-rep engine such as mine, but still manages to releave some of the work-load from the level designers. They can still do short test-cycles during the initial stages of map design when a lot of the geometry is modified; the 'quick-and-dirty' NavMesh generator will then supply rough versions only.

Apparently other game developers have struggled with the same issues I have concerning the precision of floating-point numbers (see [18]). Another often used approach is to generate the NavMesh using a 'voxelization' of the world as input, as seen in Figure 85. This approach has some analogies to the Probabilistic Road Method concepts (see [1] and [2]) whereby we measure if the static world is 'solid' or not at regular intervals (see [11] and [15]). With such a uniform matrix of voxels, we can reverse engineer polygons that represent walkable surfaces by scanning along the outer edges of voxel clusters. Handling various traveller radii is then a question of 'growing' additional layers of solid voxels before the actual polygon extraction is attempted. With adequately dimensioned voxels, we can still represent virtually any geometry, so models no longer need to be snapped to a grid with a limited set of orientations. Although the approach is 'rough' depending on the size of the voxels, it is at least a lot more robust and does not suffer from all the stability problems I described back in subsection 4.3. It is also 'easier' to implement, which makes it a far more practical solution than mine. The voxelization itself can be stored in octree-like data structures which reduces the memory footprint considerably during the process. These could potentially be generated in a background process while a level designer is editing his maps, and stored on file. So when local changes are made in the map, only a small portion needs to be 're-voxelized'. Maybe this could keep the turnaround times for editing and testing low enough.

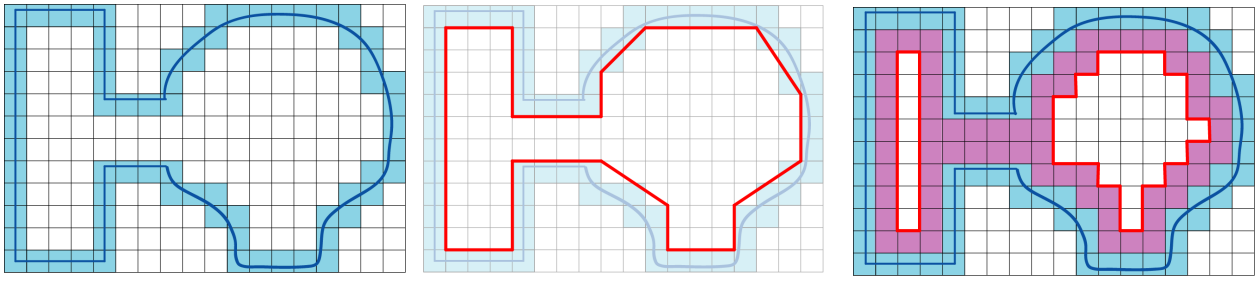


Figure 85: An example of how 'voxelization' is used to obtain simplified NavMesh representations of static worlds (images taken from [11]). On the figure on the left we see how polygon meshes are first converted into a uniform voxel representation. In the figure on the middle we see how approximating polygons are obtained by walking across the voxel edges. On the figure on the right we see how different traveller shapes can be incorporated by growing additional layers of voxels.

Based on the experience of other successful game developers, I think it is therefore safe to conclude the following:

- It is certainly worthwhile for Triumph Studios to adopt the use of NavMeshes.
- For this to be successful though, my current NavMesh generation strategy should be revised and use some of the voxel techniques.
- The successful introduction of such NavMesh-based systems will require quite a bit of effort and should always be accompanied by the introduction of decent visualization systems (this will help track down problems more easily during maintenance).

The uniform node grid based approach currently in use by Triumph Studios has already proven its practical benefits. It might not have been the most 'optimal' solution but it has certainly enabled them to release two competitive game titles on time and within budget (which is no small feat!) So, as there is already experience with such uniform grid based representational systems, I feel that a switch to a voxel-based system might actually come quite naturally. This way we can still have automatically generated NavMeshes but in a more stable and flexible form. Such a choice will clearly be beneficial for the productivity and creativity of the level designers.

8.3.2 NavMesh dynamic resolution enhancement

The main problem with the system that enhances the dynamic resolution of the NavMesh is that it needs to allocate a lot of memory on the fly, as we do not know in advance how many extra node and edge connections need to be made. A solution to this might include borrowing some ideas from the voxel system discussed in the previous chapter and from Triumph Studios' current approach. So say that we have a uniform grid definition in the game world, and arbitrarily segment it in sectors of 8 by 8. With this we have a reference grid for which we can determine which grid positions are close to NavMesh polygon edges (see Figure 86).

Using a system of sectors we can virtually cut large polygons into manageable blocks, so that we can enhance the resolution of larger segments all at once. Because of this reference grid, we can now precompute exactly how much memory is required to represent the resolution enhancements in

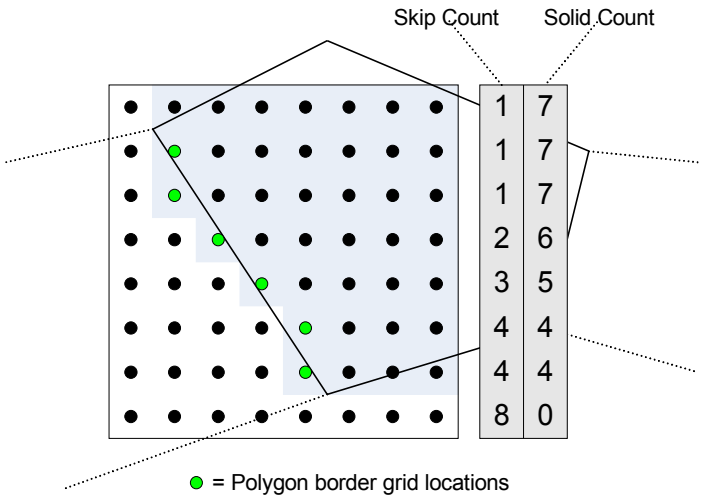


Figure 86: Example of how a uniform reference grid can be used to define shared grid points between NavMesh polygons. The polygons themselves can be described in this coordinate grid using a very simple run-length compression. The green grid locations are shared at the border of the 2 polygons, so when both polygons are represented by a local high resolution sub-graph they will be connected via these grid locations.

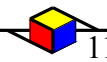
real-time. We could even opt to link these blocks with special nodes in the NavMesh so that the path-finder can process them as being matrices that need to be flooded (similar to what Triumph Studios' engine does now). These 'node matrices' can be very quickly reconstructed using a run-length encoding as shown in Figure 86; the amount of additional memory needed will still be low enough (this is a small sacrifice we need to make in order to boost the overall performance). This way we do not need to allocate much more memory for creating new NavMesh edges and nodes, only at the borders of the polygons do we need to provide links with the base NavMesh graph. So basically we just sacrifice the gradual NavMesh resolution increments using subdivision, and instead directly switch between 'low' and 'high' node densities (which is far more manageable).

I think this approach is a promising strategy because it utilizes components from both Triumph Studios' old system and that of a NavMesh. We would, however, require extensive further testing to see if it provides the benefits we are aiming for and to ascertain if its performance will be acceptable.

8.4 Parallelization

From what has been discussed in chapter 7, it is clear that it is very worthwhile to invest in parallel versions of a path-finder. Especially the Parallel Bidirectional Search algorithm that is easy to implement can significantly improve the overall performance. My experiments showed, that even without any serious optimization, we can expect a 60% to 85% speed-up which is certainly worth the extra time it will take to implement (if you can spare the extra required memory, that is). The other variants presented in this thesis, Distributed Fringe Search and Parallel Hierarchy Search, have proven to be less than spectacular. Not only are they more involved to implement, their performance gain is less than that of Parallel Bidirectional Search. The only real drawback of Parallel Bidirectional Search is that it is limited to a maximum of two cores and does not 'scale' well in that respect. Because it is expected that multi-core architectures will evolve greatly during the

following years it will certainly pay off to find ways to utilize more of them at once per session. Personally, I think the Parallel Ripple Search adaption that was sketched in section 7.5 is a promising step in the right direction. It can utilize many more cores and still promises to have a decent upper-bound on its running time similar to that of a Parallel Bidirectional Search. Unfortunately, I was not able to implement and experiment with this path-finder variant due to time restrictions.



9 Conclusion

This last chapter consists of a recapitulation of the entire discussion presented in this thesis and formalizes conclusions and recommendations for Triumph Studios. A brief overview of potential future work is also left as a final thought.

9.1 Results and recommendations

Based on the previous discussions, I would certainly recommend Triumph Studios to adopt NavMeshes in one form or other. NavMeshes can easily fulfill the originally set requirement of a flexible and memory efficient storage format because they are polygon based. This will enable us to represent any area accurately (preferably large ones) and we can easily store additional meta data if and where desired. It has been demonstrated that fully automatic NavMesh generation is feasible, although subject to some restrictions in its current form. Clearly a lot more work has to be invested to make NavMeshes a viable alternative, a crucial part being their optimization using polygon merging techniques. Only with such techniques in place can we expect significant path-finding speed-ups and reduced memory footprints. Without these, a NavMesh will only roughly perform similar to Triumph Studios' uniform node grid system, for example. Currently my 'non-optimized' implementation achieved a modest 25% path-finding speed-up with a slightly worse memory footprint. So with the proper optimizations in place, we can certainly expect dramatic improvements that are well within the limits of the original goals (less than 10MB). As a developer, however, one must expect that the overall NavMesh generator and matching path-finding implementations will be more complex by at least one order of magnitude. A lot more time will therefore be spent on streamlining, debugging and visualizing problematic cases.

Using a B-rep engine for generating the NavMeshes has been proven to be problematic. Not only are there many problems with the numerical stability of such a system; it is also prohibitively slow on current day hardware. With this in mind, we must conclude that the original goal of fully autonomous generation of navigation data has only been partially met. Other mainstream approaches have either relied on complete manual labor or special voxelization techniques that sacrifice a bit of accuracy for far greater stability and performance, and above all they are easier to successfully implement. Potentially, the use of a B-rep engine promises a very high degree of accuracy combined with full automation. This would make it the preferred choice for artists and level designers as indicated by the original goals. From my experiments, however, I have concluded that having a slightly 'less accurate' geometry interpretation is not problematic as long as game entities do not get stuck or clip too much. Such results can be obtained using these voxelization techniques, so I think that this approach would be more beneficial for Triumph Studios to adopt. With it we are still able to provide a high degree of automation but keep the test cycles for maps fast and short enough (we can easily scale down the accuracy, for example to generate temporary NavMeshes).

Although I have not been able to experiment with Boids and whiskers based systems for handling dynamic obstacle avoidance, I can say that these are well known practices and have proven to result in very natural movement of game entities (I have some limited experience with them from hobby projects of mine). These systems are also easily incorporated into flocking algorithms and other ways of forcing natural group coherency such as have been discussed in previous chapters (most notably for guiding groups of entities through corridors). These properties match the criteria of the

original goals and I would expect that, had I had enough time, they could have been successfully implemented and proven their worth. Combined with the fact that such systems will reduce the reliance on an 'expensive' footprinting scheme, I feel that it would prove beneficial to adopt it if only for that, especially for games with a high density of moving travellers.

The footprinting system for NavMeshes using dynamic subdivision has turned out to be more expensive than expected. The main cause of this is the extensive memory allocations that need to be performed. The subdivision system in its current form is so slow that is it unsuitable for commercial real-time applications, and thus fails to meet our initial requirements. For games with a rather low density of travellers, we might be able to cope without it and entirely rely on the Boids and whiskers system. But in general, though, these systems are just too 'fuzzy' and we need to be able to fall back onto systems that can provide us with deterministic routing solutions. So, as an alternative, I have suggested a scheme that will increase resolutions in larger areas all at once and do this in such a way that we can precompute how much memory is required in advance. This can be achieved by overlaying a uniform high-resolution 'reference grid' that can identify positions of shared sub-nodes on all the NavMesh polygon edges that need to be connected with the existing graph. The NavMesh polygons themselves can use this reference grid to be represented in a discrete matrix form that could be efficiently flooded by a path-finder when properly attuned. Future experiments should be performed to see if this new scheme will solve our performance problems.

Finally, the recent introduction of multi-core CPU architectures on the consumer market provides us with interesting opportunities for speeding up the path-finding process in general. These are especially rewarding for the genre of Real-Time Strategy games, whereby large quantities of entities need to be routed across large maps. This thesis has introduced a small number of A* path-finder variations, each of them with a unique approach to exploit parallelization. The most basic variation called 'Parallel Bidirectional Search' has proven itself to be the most beneficial. Its basic concept is to perform two searches at once from the two extreme points and in opposite directions and then 'meet-in-the-middle'. It is very intuitive and easy to implement, and can be expected to obtain speed-ups from 60% up to 85% on average (which more than matches the 20% target which we originally set out to achieve). It has managed to do so because its natural tendency is to avoid having different CPU cores writing in the same location of memory for as long as possible. Both CPU cores will then be able to run unhampered at full speed until a collision is detected; no other expensive synchronization schemes are required. The only drawbacks of this algorithm are the fact that it is only nearly optimal and can only utilize two cores at once. Finally, a new approach that can remove this last limitation has also been introduced for future research and experimentation: the 'Parallel Ripple Search'.

9.2 Future extensions

Having a fully autonomic and automatic NavMesh generator offers us opportunities for experimenting with new technologies and game-play. The most obvious one is that it will enable us to deal with much larger worlds that would otherwise require a prohibitively amount of manual labor to process. The current generation of MMORPGs already features such large worlds, but in general tend to have fairly weak AI because of the amount of work it is to generate routing meta data. Single-player games such as Grand Theft Auto 4 feature large an highly detailed city scapes through which a player can freely navigate. With this new breed of games, procedural content generation will play an increasingly important role, and it is to be expected that an autonomous NavMesh generator will subsequently become a necessity. The NavMesh generator that has been

presented in this thesis is currently still too cumbersome, but a future generation of (multi-core) hardware might prove powerful enough to generate high quality NavMeshes in real-time (either directly or as a background process).

A particularly interesting application of a NavMesh generator based on B-rep techniques in in-game simulations is whereby the 'static' world itself might be subject to significant and dramatic changes. Imagine, for example, a game whereby buildings can be collapsed or all sorts of large pieces of debris can land anywhere. By parsing the scenes in real-time we can construct new NavMeshes and enable game entities to navigate around and/or over dynamically introduced obstacles of considerable size. This would, for example, enable AI soldiers to maneuver through recently wrecked buildings or through other kinds of disaster areas and/or combat zones. Such applications of enhanced NavMeshes system can promise more varied and realistic game experiences, and potentially give an 'edge' over competing titles.

References

[1] Amato, N. (2004) Randomized Motion Planning. University of Padova, Texas, USA, http://parasol.tamu.edu/~amato/Courses/padova04/lectures/L8.prms.pdf

[2] Björnsson, Y., Enzenberger, M., Holte, R. and Schaeffer, J. (2005) Fringe Search: Beating A* at Pathfinding on Game Maps. *IEEE Symposium on Computational Intelligence and Games*, Essex, University of Alberta, Canada, http://www.cs.ualberta.ca/~games/pathfind/publications/cig2005.pdf

[3] Brand, S. (2007) Real-time path-finding in video games, Research Assignment Report, Delft University of Technology, Delft, The Netherlands

[4] Buckland, M. (2005) Practical Path Planning. In: *Programming Game AI by Example*, isbn 1-55622-078-2, chap. 3, Obstacle Avoidance, pp. 99

[5] Burkart, B. Setting up NavMesh. http://www.evilmrfrank.com/NavMeshInfo.htm

[6] Girvan, M. and Newman M.E.J., (2002) Community structure in social and biological networks. Department of Physics, Cornell University, USA 99, 7821-7826, http://www.pnas.org/content/99/12/7821.full.pdf

[7] Goldberg, D. (1991) What Every Computer-Scientist Should Know About Floating-Point Arithmetic. In: *Computing Surveys*, March 1991, ACM, http://www.validlab.com/goldberg/paper.pdf

[8] Hertel, S. and Mehlhorn, K. (1983) Fast Triangulation of Simple Polygons. In: *FCT (Fundamentals of Computation Theory),* pp. 207-218

[9] Kamphuis, A. and Overmars, M. H. (2004) Finding paths for Coherent Groups using Clearance. *Eurographics/ACM SIGGRAPH Symposium on Computer Animation 2004*, Utrecht University, The Netherlands, pp. 19 – 28

[10] Mika, M. and Charla, C. (2002) Simple, Cheap Pathfinding. In: *AI Game Programming Wisdom*, isbn 1-58450-077-8, chap. 4.1, pp. 155

[11] Miles, D. (2006) Crowds in a Polygon Soup: Next-Gen Path Planning. *Game Developers Conference 2006*, http://www.navpower.com/gdc2006_miles_david_pathplanning.ppt

[12] Patel, A.J., Variations of A*. *Amit's Game Programming Site*, http://www.dc.fi.udc.es/lidia/mariano/demos/amits/theory.stanford.edu/~amitp/gameprogramming/variations.html

[13] Reynolds, C. (1987) Boids - Background and Update. *SIGGRAPH '87*, http://www.red3d.com/cwr/boids/

[14] Reynolds, C. (1988) Not Bumping Into Things - Notes on 'obstacle avoidance'. *SIGGRAPH '88*, http://www.red3d.com/cwr/nobump/nobump.html

[15] Smith, P. (2002) Polygon Soup for the Programmer's Soul: 3D Pathfinding. *Game Developers Conference 2002*, http://www.gamasutra.com/features/20020405/smith_01.htm

[16] Sniedovich, M. Dijkstra's Algorithm. The University of Melbourne, Australia, http://www.ms.unimelb.edu.au/~moshe/620-261/dijkstra/dijkstra.html

[17] Tozour, P. (2002) Building a Near-Optimal Navigation Mesh. In: *AI Game Programming Wisdom*, isbn 1-58450-077-8, chap. 4.3, pp. 171

[18] Tozour, P. (2008) Fixing Pathfinding Once and For All, In: *Game/AI Blog*, http://www.ai-blog.net/archives/000152.html

[19] Valve Developer Community, The (2009) Navigation Meshes. http://developer.valvesoftware.com/wiki/Navigation_Meshes

A Mesh manipulations

In this appendix we go into more detail on the 'PM_Space' library that was introduced in chapter 4.2. This library is used extensively during the NavMesh generation process and is basically a 'light-weight' B-rep or Boundary representation engine. With it, we can perform all the construction and manipulations of polygon meshes that are required. These complex operations will be discussed in greater length here.

A.1 Vertex and edge manipulations

We start at the lowest level and discuss some manipulations we can perform on vertices and edges. The simplest of these is creating a new vertex and is defined in Algorithm 23. The next step, creating edges, is already quite a bit more involved. The problem is that we first need to determine if there are already existing polygon edges that will somehow be collinear with the new edge. So before we determine how to create new edges, we first define how to split existing ones. Splitting an edge at some parametric point t within the range of [0.0 .. 1.0] is defined in Algorithm 24. Most of the algorithm is involved with book-keeping in order to keep the 'PM_Space' container continuum consistent; the splitting itself is very easy.

Creating a new edge is much more involved. A good example of this is given in Figure 87 where we see that we may need to do any combination of cutting and creating of new 'sub-edges' to be able to express the intended line-segment (we keep the discussion limited to 2D for a moment and ignore the fact that the line-segment might intersect polygons in other ways). In the given example we want to create a new edge between points P_s and P_e for a new polygon N that is going to be instantiated. The procedure for this is defined in Algorithm 25. It is clear that when we create a new edge we might actually end up with a whole set of smaller sub-edges that span the intended line segment. This implies that the convex polygons we create in the 'PM_Space' container can contain one or more collinear edges. Later on we will notice a similarity when creating a new polygon for a mesh face: we might end up with a collection of 'sub-polygons' whom together cover the intended area. We will also see that when dealing with polygons we need to solve some harder cases in 3D in order to retain a consistent continuum.

A.2 Polygon manipulations

Before we discus how to create new polygons we should first determine how to cut existing ones. This process is actually quite straightforward; most of the work is book-keeping. The procedure for splitting an existing polygon and given a split-plane in 3D is defined in Algorithm 26. With all these solid ways to manipulate vertices, edges and splitting polygons we can continue with how to create new polygons. This will often involve cutting neighboring polygons to generate intersections and such. The problem is now that already existing polygons need to be taken into account, and because these are in 3D the complexity of how they can be intersecting with the new polygon increases correspondingly (also see Figure 88). The whole procedure for creating a new polygon N (or rather, the area that it encompasses) is roughly formalized in Algorithm 27 and Algorithm 28. The procedure is largely involved with detecting existing edges and polygons in order to determine how to 'fill' the designated area. It is thus not unlikely that a whole set of new and existing 'sub-polygons' is returned instead of just a single new polygon.

CreateVertex(point *P*):

Determine if a vertex already exists for point *P* (or within the minimal thickness ε range). This is basically an octree look-up which can be done really fast.

Otherwise create a new vertex instance for point *P* and return it.

Algorithm 23: Creating a vertex in the 'PM_Space' container.

SplitEdge(edge *E*, parametric point *t*):

Early out: **if** $t \leq 0$ or $t \geq 1$ **then** no splitting is needed, just **return** edge *E*.

Interpolate point *P* at *t*, if *P* is within the minimal thickness ε of either one of edge *E*'s end vertices then the edge does not need to be split, just return edge *E*.

Vertex *V* = CreateVertex(point *P*).

Create 2 new 'sub-edges', E_1 and E_2, that together span the original edge *E* and share the new vertex *V* in the middle.

If needed: inherited and/or distribute some of edge *E*'s meta information onto the 2 new 'sub-edges'.

Book-keeping: process all the polygons that shared the original edge and update their edge lists by removing the reference to the original edge and replacing them with 2 new references to the new sub-edges (we also have to make sure the winding order remains intact).

Remove edge *E* **from** the 'PM_Space' container (it is no longer a valid edge).

Return the resulting edges E_1 and E_2.

Algorithm 24: Splitting an existing edge E in the 'PM_Space' container, with parameter t in the range of [0.0 .. 1.0]

CreateEdge(point P_s, point P_e):

Find all vertices and edges that are crossing the space **between** points P_s and P_e (again taking the minimal thickness ε into account).

Split edges if needed at P_s and P_e as described earlier using SplitEdge().

Filter all vertices and edges that are actually collinear with the intended edge, discard everything else

Sort all remaining edges and vertices from P_s to P_e.

Step through the sorted list and **create** new sub-edges between the gaps.

Return the sorted list of all existing sub-edges between P_s and P_e, sequentially these form a single line-segment interconnected with the shared vertices.

Algorithm 25: Creating a new edge in the 'PM_Space' container between points P_s and P_e.

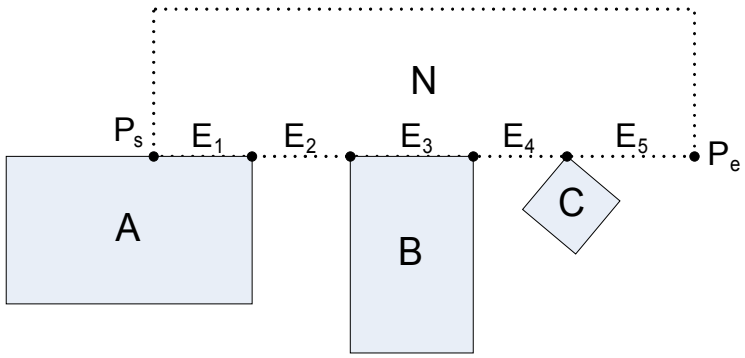


Figure 87: A non-exhaustive example of creating new edges in 2D. Creating a new edge from P_s to P_e for a new polygon N can be quite involved as is illustrated here. Not only do we need to create new vertices on an edge for polygons A, we also need to 'fill in the gaps' by creating new edges E_2, E_4 and E_5.

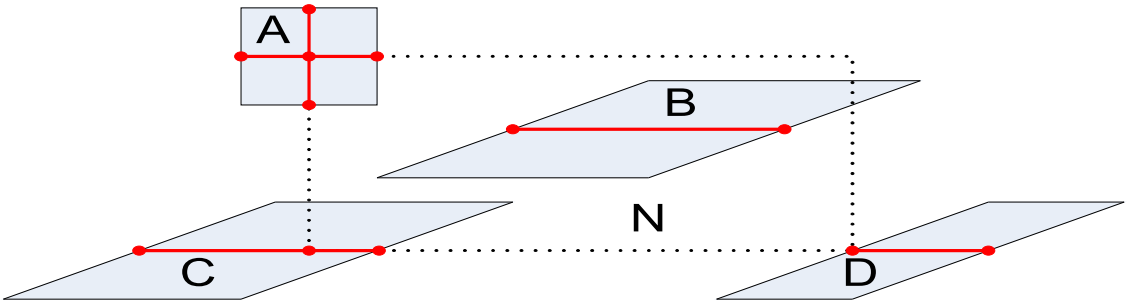


Figure 88: A non-exhaustive 3D example of which vertices and edges we might need to create before we can actually 'fill-up' a new polygon area N. Note that coplanar polygons need different processing than polygons that are not. Also note that during this process some polygons such as A, C and D might become needlessly fractured unless we also apply a sort of 'clean-up' phase.

In Figure 89 it is illustrated that neighboring geometry can intersect areas in many different configurations. There are many ways to implement an area filler for FillConvexArea(), such as a Delaunay tessellation or Advancing Front algorithm. I have opted to use a recursive mechanism that keeps on subdividing the convex area that needs to be spanned until we have generated enough 'sub-polygons' to fill it entirely. The sub-division is guided by all the items we have found to be coplanar with the newly to be constructed polygon. If such an item is, for example, an edge then we will construct a plane through this edge and use it to recursively fill the area in front and behind it. Such an approach will generally result in larger and more 'square'-ish polygons, which will make it easier for us to apply various polygon reduction schemes (these were discussed in subsection 4.4.1 and further).

The filling algorithm will be demonstrated using a simple example, as shown in Figure 90. In step 1, we start off with the boundary of the area that we intend to fill and all coplanar items touching the boundary, such as polygon *A*, and items that are inside it, such as polygon *B* and edge *C*. The initial convex area boundary will take the form of a circular linked set of edges in clockwise order. All these edges have been found and created in the previous steps and are thus consistent. Note that polygon *A* is not actually in the list of coplanar items that need to be processed, only its edge is present in the cyclic edge list. It is only added in the example to see how adjacent polygons will be treated.

SplitPolygon(polygon *P*, split plane *S*):

If polygon *P* is coplanar with split-plane *S* **then** simply abort (no cutting is needed).

Run through the edges in clockwise order and **split** them where they are actually intersected by the plane (using the SplitEdge() procedure). Each time we end up with a vertex in the plane - existing ones or new ones due to splitting - we store a link to those in a temporary list *L*.

If we assume that we have infinite numerical precision, list *L* should only contain up to a maximum of 2 vertices:

If 0 vertices **then**: the polygon is entirely behind or in front of the split-plane, **return** polygon *P*.

Else if 1 vertex **then**: only a single vertex of the polygon is coplanar with the split-plane. So no split is needed, **return** polygon *P*.

else 2 vertices **then**: this could mean one of the following situations:

Both vertices are part of an already existing edge: the polygon has a single edge that is coplanar with the split-plane. So no split is actually needed, **return** polygon *P*.

Otherwise the split-plane actually runs through the polygon and we need to **split** it into a front and back part. Note that the SplitEdge() procedure might have already increased the amount of edges in polygon *P* when it replaced edges with new 'sub-edges'.

Create 2 new 'sub-polygons' P_f and P_b.

Create a new edge *E* between the 2 found intersection vertices (note that it is safe to create such an edge now because we know polygon *P* is not intersecting anything else).

Distribute the edges of polygon *P* onto 'sub-polygons' P_f and P_b, and link them up with new edge *E* so that we again have circular chains of edges that span a convex space (note that edge *E* is now shared between both 'sub-polygons').

Again we must not forget that we might need to inherit and/or distribute some of the original polygon's meta information onto the 2 new 'sub-polygons'.

Book-keeping: update all the meshes that shared the original polygon *P* by replacing the link to *P* with 2 new links to P_f and P_b.

Remove polygon *P* **from** the 'PM_Space' container (it is no longer a valid polygon).

Return 'sub-polygons' P_f and P_b.

Algorithm 26: Splitting an existing polygon P in the 'PM_Space' container with a given split- plane S in 3D (the plane itself is perfectly flat, so no need to incorporate the thickness threshold ε here).

CreatePolygon(convex coplanar point list *QL*):

Find all polygons that are in close neighborhood of the area that polygon *N* will span (so this includes the minimal thickness ε). Note that we assume that there can never be any vertices and/or edges 'floating' around without being part of a polygon somehow.

Roughly **filter** all the polygons that will actually be intersected by the new polygon *N*. We can do this reasonably fast by checking if polygons are completely in front or behind the plane through polygon *N*, and by creating perpendicular boundary planes at the edges of polygon *N*'s boundaries.

Create an empty list, *CPL*, that will contain references to vertices, edges and polygons that are coplanar with the intended polygon *N*.

For all non-coplanar polygons:

Use SplitPolygon() to **cut** the polygon using the plane through the new polygon *N* as a split plane.

During the cutting we need to track all the vertices and edges that are coplanar, also the newly created ones. All these are appended to list *CPL*.

For all remaining coplanar polygons: these all get **added** to the *CPL* list directly.

Early out: if *CPL* remained empty then we can immediately create the new polygon *N*, there are no complicated overlaps and/or intersections.

Next, for each edges of polygon *N* we will construct a temporary cutting plane that is perpendicular to polygon *N* and aligned to an edge. Together in a list *BPL*, these cutting planes define the 'physical' boundary of polygon *N* whereby everything behind or inside the planes is considered to be 'inside' the new polygon *N*.

For each item *I* **in** the *CPL* list (while it is not empty):

For each cutting plane *BP* **in** the *BPL* list:

We are going to determine which part of item *I* is inside or behind plane *BP*, initially we assume that there is none and thus item I_{behind} = {}.

If type item *I* = vertex **then**:

Determine if the vertex is inside or behind plane *BP*, when this is the case then set $I_{behind} = I$.

Else if type item *I* = edge **then**:

SplitEdge(item *I*, *t* = intersection with plane *BP*), and I_{behind} = new 'sub-edge' behind the plane.

If SplitEdge() determined that the entire edge is coplanar with plane *BP* then $I_{behind} = I$ (we do not want to loose any edges of polygons that should become part of new polygon *N*'s boundaries).

Algorithm 27: How to create a polygon in the 'PM_Space' container for a given set of coplanar points QL that form a convex area (continued in Algorithm 28).

Continuation of CreatePolygon():

Else type item = polygon:

SplitPolygon(item *I*, plane *BP*), and I_{behind} = new 'sub-polygon' behind the plane. Note: it is impossible for polygon *I* to be coplanar with plane *BP* because this was filtered out in the earlier steps.

Item *I* is **removed** from the *CPL* list and **if** I_{behind} is not still empty, **then** it is appended to the back of the *CPL* list so that it will be processed further later on.

Early out: **if** list *CPL* turns out to be empty now **then** we can immediately **create** the new polygon *N*, there are no complicated overlaps and/or intersections.

Otherwise we have to 'fill up' the area spanned by intended polygon *N*, 'around' all the items in list *CPL*. For this, we will be using a recursive helper function 'FillConvexArea()' that will be discussed shortly:

First, **construct** all edges for the boundary of the new polygon *N* using the CreateEdge() function; the result will be a circular 'edge-chain' stored in list *ECL*. Note that because of the cutting we have done previously, we are now guaranteed to find edges through all the intersection points around polygon *N*.

Recursively fill the area spanned by polygon *N* and return the resulting list of polygons: return FillConvexArea(*ECL*, *CPL*).

Algorithm 28: Continued from Algorithm 27, how to create a polygon in the 'PM_Space' container. The implementation of the FillConvexArea() function is defined in Algorithm 29).

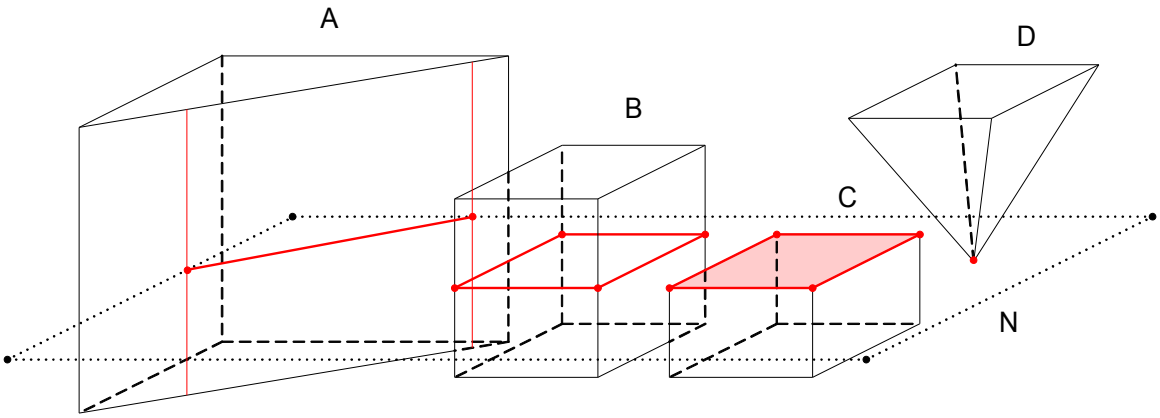


Figure 89: Yet another example of some of the ways that meshes can intersect the area that we want to span with a new polygon N. Cutting through mesh A will have generated an edge that touches the boundaries of the polygon N for example. In turn, processing mesh B will have resulted in an 'island' of edges that need to be filled in on either side. Mesh C however just happens to have a face that is coplanar with polygon N and thus we will have found an entire polygon that somehow needs to be incorporated into the result. In some rare cases we might even see that a single vertex just happens to 'poke' into polygon N's surface.

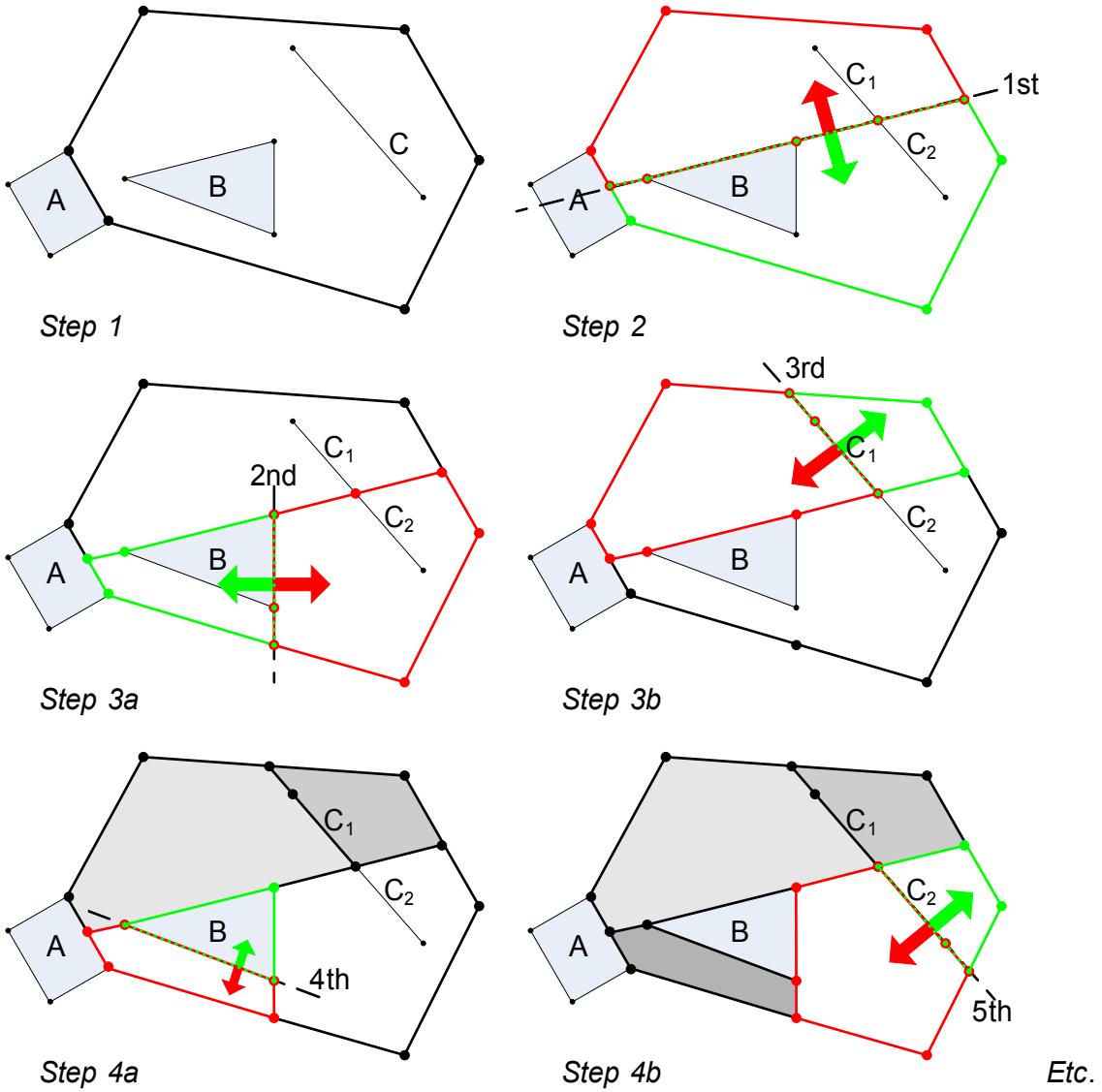


Figure 90: An example of how the polygon area filling algorithm uses recursion to generate all the 'sub-polygons' that cover the intended area. Note how polygon B is neatly integrated into the result.

We now have to select a 'pivot' item that will be used to generate split-planes from. In theory, we could use any of the items but I preferred to give highest priority to polygons and lowest to vertices (the latter are not included in the example though). Ideally we should use split-planes that cause the least amount of splits to be made. I opted however to leave this as a future exercise because it would involve more work in the form of prediction logic. In the given example, we initially choose polygon *B* as pivot and we start generating split-planes for all its edges in clockwise order. We then begin with its first split-plane and use it to create 2 'half-spaces' by cutting the cyclic edge chain and all items stored within it. This is demonstrated in step 2. Notice that the cutting process will generate new vertices and edges that we are going to need to link into the 2 halves of the cyclic edge chain. We do the same for all the items that are within the initial area; these also get divided into a front and behind set, apart from the pivot item. The pivot item will automatically be part of the output result if it is a polygon only, otherwise it will just be discarded. We then enter recursion using the front/behind areas and item sets to try an fill them up further. This is demonstrated in step

3a and 3b. In step 3a we will continue with the next edge of polygon *B* and use it as a new split-plane. For the initial front area we select the 'sub-edge' C_1 in step 3b to form a new split-plane and repeat the entire process. By the time we arrive at step 4a we will have ended up with 2 smaller cyclic edge chains at the top half of the intended fill area that no longer contain any items. For these we can finally generate new polygons and return these as results. By that time, we have also reached the last edge of polygon *B* to be used as split-plane in step 4a. Using recursion we will find that we have again ended up with 2 'empty' areas: one which is truly empty for which we need to create a polygon, and another that is already represented by polygon *B* and thus requires no further actions to be taken.

The entire process continues with step 4b and so on until all recursion paths have returned all the individual 'mosaic facets' which as a whole cover the intended polygon area. Notice that the use of recursion is very powerful and will enable us to fill areas with highly complex intersections. Dealing with coplanar vertices is also very easy, we can basically use any split-plane as long as it passes through them. In my implementation I just create a split-plane that runs though the vertex in question and the closest vertex that is part of the cyclic edge chain that encloses it.

This entire filling procedure is formalized in function FillConvexArea() that is defined in Algorithm 29, Algorithm 30 and Algorithm 31. It uses a helper function, called SplitConvexChainAndItems(), that is in charge of splitting a convex chain of edges and all the items that are located inside it.

Note that the algorithm that has just been described is very 'rough' and will create unnecessary cuts. We might for example figure out 'too late' that a polygon that was split by a cut-plane *n* is actually entirely behind a later cut plane *n* + i which makes the initial cuttings redundant. Also, in cases whereby existing polygons are coplanar with new polygon *N* and are only touching its boundary with a single edge, we split the polygon whereby it could suffice with just splitting the edge in question. Such cases, can of course, be 'mended' as part of a post processing step or by detecting them early on and handle them as special cases. The actual implementation of the entire algorithm was already very laborious and thus I primarily focused on ensuring the robustness and 'correctness' of it. I opted to mend errors as a post-processing step for only the most obvious cases. This explains for example why way back in Figure 9 we see that the polygon meshes consist of more polygons than would be strictly necessary. This has however no serious impact on the quality of the final NavMeshes that we extract from them. Also, as we saw in subsection 4.4.1, we can improve overall quality via a sort of post-processing step in the NavMesh generation process, by merging smaller polygons into larger polygons again. Additionally, it turns out that the input data that we feed into the NavMesh generator is already 'fractured' because of how maps are created using smaller models as building blocks. The walkable areas of these separate blocks can often be merged into larger ones (think of a set of separate tiles that cover an entire hallway). So this post processing step will automatically polish some of the noise we might have introduced during the NavMesh generation process.

A.3 Meshes

With all the splitting and creation functions that have just been presented, it is now relatively easy to move on to polygon meshes. In my case, the input data consists of bounding box hit-volumes that are part of models that can be placed in the maps. We can convert these boxes easily into polygon meshes by creating polygons for each side of the boxes. A polygon mesh in 'PM_Space' is thus nothing more than a data structure that contains links to all its boundary polygons. Because

'Helper' algorithm for filling a convex area defined by a list of edges that is called from Algorithm 28. Coplanar item list and *CPL* has been constructed in a preprocessing phase whereby all surrounding edges and polygons have already been found and split if needed. The previous pivot item *PPI* is initially undefined.

FillConvexArea(convex edge list *ECL*, coplanar item list *CPL*, previous pivot item *PPI*):

It is possible that under the influence of recursive cutting, some of the coplanar vertices and/or edges have become collinear with the convex edge list, we will have to remove those first: **For all** items *I* **in** *CPL*:

If type item *I* = vertex **then**: **if** any edge in list *ECL* shares this vertex **then** just **remove** *I* from list *CPL*.

Else if type item *I* = edge **then**: **if** this edge is also in the *ECL* list **then** just **remove** it from list *CPL*.

Otherwise no further processing is needed for the item.

Early out: **if** list *CPL* is empty **then** there are no more coplanar items and we can immediately **return** a new polygon that is spanned by the edges in *ECL*.

Early out: **if** the pivot item *PPI* is not empty but is actually a polygon that is exactly matched by the edges in *ECL* **then** we are done: **return** {}.

Otherwise we will have to subdivide the area somehow and fill it in using recursion. We start with an empty polygon list to which we will continuously append 'sub-results': *RESL* = {}.

Then we will have to pick a 'pivot' item *PI* from list *CPL*. I found that best results were obtained by prioritizing polygons the highest and vertices the lowest.

Remove item *PI* **from** list *CPL*.

If type item *PI* = polygon **then**:

The polygon *PI* already covers a bit of the target area so we are not going to subdivide it but immediately **add** it to the result container: *RESL += PI*.

Next we will **create** a list of temporary split-planes *SPL* for all the edges in polygon *PI* so that the entire polygon is behind them (the planes are also perpendicular with polygon *PI* itself). Note that polygons might have sequences of collinear edges, in those cases we only need one split-plane that runs though them altogether. For each temporary split-plane *SP* in *SPL*:

SplitConvexChainAndItems(*ECL*, *CPL*, *SP*, *&ECL*$_{front}$, *&CPL*$_{front}$, &*ECL*$_{back}$, *CPL*$_{back}$). Note: the & is used here to denote that the function will also return new edge and item lists.

Fill in the front: *RESL* += FillConvexArea(*ECL*$_{front}$, *CPL*$_{front}$).

The filling area will now be reduced to the remaining back area:

ECL = ECL$_{back}$.

CPL = CPL$_{back}$.

Algorithm 29: Helper algorithm that recursively fills a convex area with existing and/or new polygons in the 'PM_Space' container (continued in Algorithm 30).

Continuation of FillConvexArea():

Else:

If type item *PI* = edge **then**:

Construct split plane *SP* through edge *PI* and perpendicular to the original polygon area we are trying to fill.

Else type item *PI* = vertex:

Construct split plane *SP* perpendicular to the original polygon area we are trying to fill and through vertex *PI* and any vertex from one of the edges of list *ECL*.

SplitConvexChainAndItems(*ECL*, *CPL*, *SP*, *&ECL*$_{front}$, *&CPL*$_{front}$, &*ECL*$_{back}$, *CPL*$_{back}$). Note: the & is used here to denote that the function will also return new edge and item lists.

Fill in the front: *RESL* += FillConvexArea(*ECL*$_{front}$, *CPL*$_{front}$).

Fill in the back: *RESL* += FillConvexArea(*ECL*$_{back}$, *CPL*$_{back}$).

Return all resulting polygons in *RESL*.

Algorithm 30: Continued from Algorithm 29, how to fill a convex area in the 'PM_Space' container. The SplitConvexChainAndItems() helper function is defined in Algorithm 31

SplitConvexChainAndItems(

convex edge chain list *ECL*, coplanar item list *CPL*,

split-plane *PL*,

[by reference] front convex edge chain *ECL*$_{front}$,

[by reference] front coplanar item list *CPL*$_{front}$,

[by reference] back convex edge chain *ECL*$_{back}$,

[by reference] back coplanar item list *CPL*$_{back}$):

Clear the result containers: *ECL*$_{front}$ = {}, *CPL*$_{front}$ = {}, *ECL*$_{back}$ = {}, *CPL*$_{back}$ = {}.

Determine where split-plane *PL* intersects the edges in list *ECL*.

Split the intersected edges using SplitEdge() and store the resulting 'sub-edges' in *ECL*$_{front}$ and *ECL*$_{back}$ respectively.

The remaining edges that are not intersecting or only have a single coplanar vertex in the plane are distributed to *ECL*$_{front}$ and *ECL*$_{back}$ respectively.

Make sure *ECL*$_{front}$ and *ECL*$_{back}$ are sorted in clockwise order again.

If there are 2 intersection points **then** we need to **create** a list of 'linking' edges *LEL* between them using CreateEdge().

Add list *LEL* **to** both *ECL*$_{front}$ and *ECL*$_{back}$ in such a way that we will again have 2 circular clockwise and convex edges chains.

Repeat this splitting process but now for all the items in list *CPL* and **store** the results in *CPL*$_{front}$ and *CPL*$_{back}$ respectively. For this we will be using SplitEdge() and SplitPolygon(); if we find a vertex item that is coplanar with split-plane *PL* then we can safely discard it (we will have already encountered and processed it earlier when we created the set of linking edges *LEL*).

Algorithm 31: Helper function for FillConvexArea() that splits a convex chain of edges and a set of items inside the area spanned by the chain, using a given split-plane.

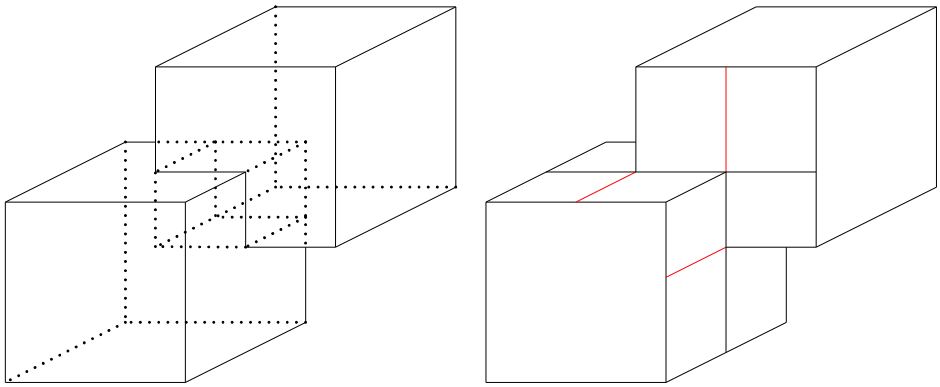


Figure 91: The 2 bounding box volumes on the left will be converted into polygon meshes and have their intersections removed. The result is a boundary representation as seen on the right. The red edges are superfluous and could be removed again by either a post-processing polygon merging step or better prediction logic during polygon cutting.

polygons can be shared between meshes, for example when these just barely touch each other, we also locally store flags in the mesh containers that define for each of its polygons which side is facing outwards. Again we store the meshes in an octree to speed up access to the 'PM_Space' container class.

Polygon mesh manipulation is also very straightforward because we now have solid polygon splitting algorithms. We only need to add more book-keeping rules that update the mesh containers when one or more of its polygons gets split: in these cases the links to the original polygons need to be replaced by links to all of their corresponding 'sub-polygons'. Also, when mesh polygons get cut we may need to apply the cutting plane on the entire mesh. This is needed when 'holes' are cut into the entire mesh or when for some reason the mesh will become 'hollow' due to some Boolean operation. In such cases the original mesh will be fractured into more 'sub-meshes' that individually are still convex.

Intersection removal is a fairly straightforward process. We can basically check if any of the polygons of the meshes involved are inside more than one of the original bounding box volumes. If this is the case then these polygons can just be removed. If none of the polygons will 'survive' then one mesh is completely inside another and can just be removed in its entirety. Usage of the CreatePolygon() function will automatically result in a boundary definition of the meshes, as we can see in Figure 91. There will again be more fracturing than strictly needed as we have seen with polygons. In theory we could correct this by merging separate mesh containers into a single new one as long as they remain convex, but this was currently not needed for my NavMesh generator.

Although the NavMesh generator currently only supports bounding boxes, it should not be hard to implement support for other shapes. We could even support adding entire model meshes, as long as these span convex spaces (and if they do not we could just first cut them up into smaller convex pieces).