Three-dimensional path planning in complex environments

Tijs Leenknegt

Promotor: prof. dr. ir. Rik Van de Walle
Begeleiders: ir. Aljosha Demeulemeester, ir. Jonas El Sayeh Khalil

Masterproef ingediend tot het behalen van de academische graad van
Master in de ingenieurswetenschappen: computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. Jan Van Campenhout
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2012-2013

UNIVERSITEIT
GENT

Three-dimensional path planning in complex environments

Tijs Leenknegt

Promotor: prof. dr. ir. Rik Van de Walle
Begeleiders: ir. Aljosha Demeulemeester, ir. Jonas El Sayeh Khalil

Masterproef ingediend tot het behalen van de academische graad van
Master in de ingenieurswetenschappen: computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. Jan Van Campenhout
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2012-2013

UNIVERSITEIT
GENT

Acknowledgements

I would like to thank my promoter, Professor Rik van de Walle, for the opportunity to do this research. I would also like to thank my supervisors Aljosha Demeulemeester and Jonas El Sayeh Khalil for their guidance and advice. Finally, I wish to thank my family for their support throughout my study.

Tijs Leenknegt, June 2013

Usage permission

"The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use.
In the case of any other use, the limitations of the copyright have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master dissertation."

*"De auteur geeft de toelating deze scriptie voor consultatie beschikbaar te stellen en delen van de scriptie te kopiëren voor persoonlijk gebruik.
Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze scriptie."*

Date and signature
Datum en handtekening

Three-dimensional path planning in complex environments

door
Tijs Leenknegt

Afstudeerwerk ingediend tot het behalen van de graad van
Master in de ingenieurswetenschappen: computerwetenschappen

Academiejaar 2012-2013

Universiteit Gent
Faculteit Ingenieurswetenschappen en Architectuur
Vakgroep Elektronica en informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout

Promotor: prof. dr. ir. R. Van de Walle
Thesisbegeleiders: ir. Aljosha Demeulemeester, ir. Jonas El Sayeh Khalil

Overview

This work is about the path planning for actors that are able to navigate in three dimensions throughout a virtual three-dimensional environment. In other words, these actors fly or swim through the environment; as opposed to moving over a surface, when only two-dimensional navigation is allowed.

First, some common data structures for path planning are discussed. Most path planning structures are for the 2D path planning, but some of them can be adapted for 3D path planning. Both global and local path planning are considered. Global path planning is about finding a good path between all the obstacles of the environment. While local planning is about just moving toward the next point along a path, while avoiding collisions.

A path planning algorithm for 3D navigation in 3D environments is proposed in this works, that is composed of both a local and global path planner.

Keywords: path planning, A*, local avoidance, voxel grids, navigation meshes

Three-dimensional path planning in complex environments

Tijs Leenknegt

Supervisor(s): prof. dr. ir. Rik Van de Walle, ir. Aljosha Demeulemeester, ir. Jonas El Sayeh Khalil

Abstract—**Path planning for 3D navigation in virtual 3D environments is not as straightforward as its 2D navigation counterpart. Two major components can be distinguished in path planning: global and local. First, the main data structures for path planning are discussed, which will lead to a proposal for a simple data structure that represents the navigable space. After that, global path planning will be discussed, where we will consider the shortest path problem from the graph theory, and we show that it is relevant to finding a path through the environment. The most used algorithm used for this is the A* algorithm, which is guaranteed to quickly find the shortest path. Thereafter, techniques are discussed that can be used for local path planning. This will ensure that the actors navigate in an realistic way along their planned path, without having them collide with something.**

Keywords— **path planning, 3D navigation, A*, local avoidance, spatial data structures**

I. Introduction

THE goal of path planning is to control the navigation of different independent actors in an virtual environment. This should provide the best possible path through all obstacles in the environment and it also has to ensure that the actors do not collide with anything. The former is generally referred to as global path planning, while the latter is better known as local path planning.

It is useful to find a good method that is able to efficiently do path planning in three-dimensional navigation. However, this introduces some new difficulties compared to the more studied two-dimensional navigation.

It is desirable that all the operations for path planning can be done quickly when the path planning has to happen in real time, e.g. in computer games and simulations. Another thing of importance is that the global and local path planning have good cohesion. Each actor should reach its destination with minimal effort.

For both global and local path planning we require appropriate data structures. The data structure for global path planning has to be a compact representation of the environment, were we can easily extract a graph structure that we can use to quickly find a shortest path on. In the case of the local path planning, it is necessary that one actor can quickly find all the objects, actors and obstacles, in its direct vicinity.

II. Data structures

There are various data structures that are able to represent the navigable space of a virtual world. Since we wish to use these data structures in real time, we will only take an approximation of the environment, otherwise we would have to deal with too many unnecessary details. Furthermore, it is important that the memory footprint of the data structure is as low as possible. A good trade-off has to be found between the memory usage and the number of necessary operations that an algorithm requires to achieve a desired result.

Normally, a three-dimensional grid is used for path planning 3D. The elements of such a grid are often called voxels, since such grids are also used in image rendering of three-dimensional structures. The main advantage in using voxel grids, in the context of spatial representations, is that not every voxel needs to be explicitly stored. It also allows for quick access to retrieve all objects within a certain range.

A data structure that has a good performance for finding a path in 2D navigation is the navigation mesh. However, there are in the literature different interpretations of what it exactly means. One of the first definitions comes from Snook [1], according to his definition a navigation mesh is a tessellation of the entire navigable space into a non-overlapping contiguous mesh of triangles, with each triangle sharing a single edge with each of its neighbors. More recent game engines, e.g. Unreal Engine [2], use a more general interpretation of navigation meshes, where also other forms can be used. In this thesis, we will generalize this definition so that convex polygons can be used instead of just triangles. This can even be further generalized for the threedimensional case, where we also use convex polyhedra, which now share a full polygon with each of its neighbors, instead of just an edge. The main problem here is that the corners are not longer simple points, but can be complete edges.

III. Global path planning

For global path planning, we consider the used data structure as a graph. This makes finding the shortest path equivalent to the shortest path problem from the graph theory. A solution to this problem was given by Dijkstra [3]. His algorithm is guaranteed to find the shortest path in a graph, but it does not take the direction into account in which an actor navigates through the nodes of the graph. An alternative to Dijkstra's algorithm is to make use of the greedy best-first search method, which relies on a heuristic that takes the node that is closest to its goal at each step. This method is faster than Dijkstra's algorithm, but the guarantee is lost that the resulting path is a shortest path. The solution for this is the A* algorithm, which is a combination of the ideas behind these two methods. The Euclidean metric is the most suitable heuristic for real-time execution of the A* algorithm in a 3D environment.

The obtaining path of such an algorithm contains generally many unnecessary intermediate goals. To solve that, we can smooth the path by modifing or deleting redundant subgoals. An obvious method for doing this is to remove the intermediate goals that are within the line of sight between two others of the path, so we just keep those that are within maximum visibility of each other. One way to check if an object or point lies within the

lines of sight is to make use of the ray-casting algorithm, which searches for the first point of intersection with another object, for a ray (straight line) which is fired in a certain direction from a certain point. When some random shapes of objects are used, this operation can be quite time consuming. But when only simple structures are used, we can greatly simplify this operation. Especially the finding of a point of intersection with a plane in a 3D space is a simple calculation.

A method that works well together with this, is to use a data structure in which the nodes are located along the corners of the obstacles. This is just a simple point for the 2D case, but it can have multiple edges in 3D. It is therefore not as straightforward to make an optimal selection of the nodes through which we wish to do global path planning.

For the 3D navigation mesh that we have proposed, we introduce a new method for smoothing that is able to post-process an obtained path so that better coordinates are selected. It makes use of the method for finding points of intersection with a plane in a 3D environment. The plane considered is the one in which the polygon lies that is shared between the polyhedra of the 3D navigation mesh. The centerpiece of such a polygon is considered here as the node of the graph. The ray cast is done starting from the predecessor of the node in question and goes in the direction to the successor of the considered node. With the knowledge of this intersection we can then choose the closest point on the polygon of the considered node.

IV. Local path planning

For local path planning, it is important that we have a data structure that enables us to quickly retrieve objects in the direct vicinity of an actor. The most appropriate approach seems to us to use a grid structure, and that only the k nearest objects are considered for each actor. The detection of (potential) collisions in 3D environments can be greatly simplified by considering the environment in simple structures, such as planes and spheres. If these objects are still quite complex, we can put simple structure as bounding structures around them so that the average number of required operations can be greatly reduced [4].

But it's better to be safe than sorry, so a good avoidance strategy is desirable. One of the first works on local avoidance for large groups is the boid flocking model of Reynolds [5]. In this work he also makes a distinction between two avoidance methods: one is based on force fields, while the other is velocity-based. The majority that are based on the latter, are build upon the concept of velocity obstacles which was introduced by Fiorini et al [6]. This concept works as follows. Each actor makes a prediction of the location of all the other actors in its direct vicinity. Based on this information the actor will adapt its preferred velocity in such a way that it is guaranteed that no collisions happen along its path during a certain time window.

One problem with this method is that all actors normally use the same technique. Since these predictions are done based on old information of all actors it may happen that this will no longer be valid after all actors have chosen their new velocity. This may lead to undesirable oscillations in the actual path taken.

A solution to this was presented with the ORCA algorithm of Van Den Berg et al [7]. This method also uses these velocity obstacles, but this one splits the navigation plane in two parts (the method was originally for 2D) for each two neighboring actors based on their velocity obstacles. An actor may only use their half-planes, the other half-plane is used by the other actor. These half-planes are then calculated for all nearby actors, based on this we can find the area in which the actor can go with a guarantee that no collisions occur within the time window. The algorithm of Snape et al [8] is an extension of this ORCA algorithm to 3D navigation, in which, instead of half-planes, half-spaces are used. It does not yet take the avoidance of static obstacles into account. We will use this algorithm as our avoidance strategy between moving actors for our algorithm, and we can use our 3D navigation mesh to handle the avoidance of static obstacles.

V. Experiments and conclusion

Tests were done on three different environments, an empty one and two more complex environments: an urban environment and a cave. Our proposed 3D navmesh allows the A* algorithm to quickly find a path for each actor towards its goal. The selection of the allowed convex polyhedra influences the actual number of nodes that can be obtained with such a navmech. The smoothing function that we have proposed can make the acquired paths more optimal, in the sense that the actors require fewer unnecessary actions to reach their goal. This makes the path taken also more realistic.

	Urban	**Cave**
Data structure		
# polyhedra	825	89
# nodes (graph)	2096	174
# edges (graph)	8829	552
Number of steps taken by A*		
min.	9	4
avg.	226.55	20.84
max.	11170	101
Length of resulting path		
min.	7	4
avg.	14.55	7.11
max.	24	11

References

[1] Greg Snook, "Simplified 3d movement and pathfinding using navigation meshes," *Game Programming Gems*, vol. 1, pp. 288–304, 2000.

[2] Epic Games, "Unreal engine," *URL: http://www.unrealtechnology.com/*, 2013.

[3] Edsger W Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[4] Carol O´ Sullivan and John Dingliana, "Real-time collision detection and response using sphere-trees," 1999.

[5] Craig W Reynolds, "Flocks, herds and schools: A distributed behavioral model," *ACM SIGGRAPH Computer Graphics*, vol. 21, pp. 25–34, 1987.

[6] Paolo Fiorini and Zvi Shiller, "Motion planning in dynamic environments using velocity obstacles," *The International Journal of Robotics Research*, vol. 17, no. 7, pp. 760–772, 1998.

[7] Jur Van Den Berg, Stephen J Guy, Ming Lin, and Dinesh Manocha, "Reciprocal n-body collision avoidance," in *Robotics Research*, pp. 3–19. Springer, 2011.

[8] Jamie Snape and Dinesh Manocha, "Navigating multiple simple-airplanes in 3d workspace," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, 2010, pp. 3974–3980.

Driedimensionale padplanning in complexe omgevingen

Tijs Leenknegt

Supervisor(s): prof. dr. ir. Rik Van de Walle, ir. Aljosha Demeulemeester, ir. Jonas El Sayeh Khalil

Abstract— **Padplanning voor 3D-navigatie in virtuele 3D omgevingen is iets minder evident dan zijn 2D-navigatie tegenhanger. Twee grote onderdelen kan men onderscheiden in de padplanning: globaal en lokaal. Eerst worden de voornaamste datastructuren voor padplanning besproken, waarin een voorstel komt voor een eenvoudige datastructuur die de navigeerbare ruimte voorstelt. Daarna wordt de globale padplanning bekeken, we beschouwen het kortstepad-probleem uit de graaftheorie, en hieruit bespreken we hoe dat relevant is voor het vinden van een pad doorheen de omgeving. Het voornaamste algoritme hiervoor is het A*-algoritme, dat in staat is om relatief vlot met zekerheid het kortste pad te vinden. Daarna worden technieken besproken die gebruikt kunnen worden voor de lokale padplanning. Die zal er voor zorgen dat de actoren zich realistisch langs het geplande pad navigeren, zonder dat deze ergens tegen botsen.**

Keywords— **padplanning, 3D navigatie, A*, lokale ontwijking, ruimtelijke datastructuren**

I. Introductie

HET doel van padplanning is om de navigatie van verschillende, onafhankelijke actoren in een virtuele omgeving te controleren. Zo moet dit een zo goed mogelijk pad opleveren doorheen alle obstakels van de omgeving en er verder voor zorgen dat de actoren niet tegen alles aanbotsen. Dat eerste wordt over het algemeen gezien als de globale padplanning, terwijl het laatste vooral gekend is onder de benaming lokale padplanning.

Het is zinvol om een goede methode te vinden die deze padplanning efficiënt kan doen voor driedimensionale navigatie. Dat introduceert wel enkele nieuwe moeilijkheden ten opzichte van de ruim bestudeerde tweedimensionale navigatie.

Vooral als men padplanning in ware tijd wenst te gebruiken, zoals bij computergames en -simulaties, is het wenselijk dat dit allemaal zeer vlot kan gebeuren. Het is ook belangrijk dat de globale en lokale padplanning mooi samenwerken. Het is de bedoeling dat elke actor met minimale inspanningen naar zijn einddoel geraakt.

Voor zowel globale als lokale padplanning zijn er goede datastructuren nodig. Bij globale padplanning is het belangrijk dat de datastructuur een compacte voorstelling is van de omgeving, die men bovendien eenvoudig als graafstructuur kan beschouwen zodat men er vlot de kortste paden op kan vinden. In het geval van lokale padplanning is het nodig dat men voor een actor vlot de objecten, zowel obstakels als andere actoren, in de nabije omgeving kan terugvinden.

II. Datastructuren

Er zijn verscheidene datastructuren die toelaten om de navigeerbare ruimte van een virtuele wereld voor te stellen. Gezien we ook deze datastructuren in ware tijd wensen te gebruiken, zullen we een benadering van de omgeving nemen, anders zou er met te veel overbodige details rekening gehouden moeten worden. Ook is het belangrijk dat de datastructuur weinig geheugen inneemt op de computer. Het is hierbij ook van belang een goede trade-off te vinden tussen enerzijds het geheugengebruik, en anderzijds het aantal nodige bewerkingen dat een algoritme nodig heeft om tot een gewenst resultaat te komen op die datastructuur.

Over het algemeen wordt een driedimensionale rooster gebruikt voor 3D padplanning. De elementen van zo'n rooster noemt men dikwijls voxels; gezien dergelijke roosters ook gebruikt worden bij beeldgeneratie van sommige driedimensionale structuren. Het grote voordeel bij het gebruik van voxel-roosters in de context van ruimtelijke voorstellingen is dat men niet elke voxel expliciet bij hoeft te houden. Ook is het hier eenvoudig om snel alle elementen binnen een bepaald bereik terug te vinden.

Een datastructuur die goede prestaties kan leveren bij het vinden van een pad bij 2D navigatie is de navigatiemesh. Er zijn in de literatuur verschillende interpretaties van wat men er precies onder verstaat. Een van de eerste definities komt van Snook [1]; volgens zijn definitie is een navigatiemesh een onderverdeling van de volledige navigeerbare ruimte bestaande uit een maas van aan elkaar hangende driehoeken die elkaar niet overlappen en die elk een zijde deelt voor ieder van zijn buren. Meer recente game engines, zoals bvb. Unreal Engine [2], gebruiken een meer algemene interpretatie van navigatiemeshes, waarbij ook andere vormen gebruikt worden. De definitie die in deze thesis gebruikt wordt gaat ervan uit dat de navigeerbare ruimte onderverdeeld kan worden in convexe veelhoeken. Dat kunnen we dan uitbreiden naar het driedimensionale geval, waar we convexe veelvlakken gebruiken, waarbij er dan in de plaats van een zijde nu een volledig zijvlak gedeeld wordt. Het grootste probleem hierbij is dat de hoeken nu niet meer slechts één punt zijn, maar volledige zijden kunnen zijn.

III. Globale padplanning

Bij globale padplanning beschouwen we de gebruikte datastructuur als een graaf. Het vinden van een kortste pad is dan equivalent met het kortstepad-probleem uit de graaftheorie. Een oplossing voor dit probleem werd gegeven door Dijkstra [3]. Zijn algoritme vindt steeds een kortste pad in een graaf, maar het houdt totaal geen rekening met de richting waarin we door de knopen van de graaf navigeren. Een alternatief voor Dijkstra's algoritme is om gebruik te maken van een gretig beste-eerst zoekmethode, steunend op een heuristiek die per stap de knoop neemt die het dichtst bij het doel ligt. Deze methode is sneller dan Dijkstra's algoritme, maar hierbij wordt wel de garantie verloren dat het bereikte pad wel degelijk het kortste pad is. De oplossing hiervoor is het A*-algoritme, dit algoritme is in feite een combinatie van beide voorgaande methodes. De Euclidische metriek is meest geschikte heuristiek om het A*-algoritme in ware tijd te kunnen gebruiken binnen een 3D omgeving.

Het bekomen pad van zo'n algoritme bevat over het algemeen veel overbodige tussendoelen. Om dat op te lossen kunnen we het bekomen pad gladder maken door overtollige doelen aan te passen of te verwijderen. Een voor de hand liggende methode hiervoor is om de tussendoelen die langs de zichtlijn liggen tussen twee andere te verwijderen, zodat we enkel diegene overhouden die opeenvolgend volgens maximaal zicht van elkaar liggen. Een manier om dergelijke zichtlijnen te vinden is gebruik te maken van het ray-casting algoritme, die zoekt naar het eerste snijpunt met een ander object voor een straal (rechte lijn) die in een bepaalde richting afgeschoten wordt vanaf een bepaald punt. Wanneer enkel willekeurige vormen van objecten gebruikt worden, kan deze bewerking vrij rekenintensief zijn. Maar wanneer enkel eenvoudige structuren gebruikt worden kunnen we deze bewerking sterk vereenvoudigen. Vooral het vinden van een snijpunt met een vlak in een 3D ruimte is een eenvoudige berekening.

Een methode die hiermee goed samenwerkt is om een datastructuur te gebruiken waarbij de knopen langs de hoeken van de obstakels liggen. In 2D is een hoek een simpel punt, bij 3D kunnen dat ook meerdere zijden zijn. Hierbij is het dus minder evident om een optimale selectie te maken van de knopen waarlangs we globale padplanning wensen te doen.

Voor de 3D navigatiemesh die we voorgesteld hebben stellen we nog een nieuwe afvlakkingsmethode voor die in staat is om een bekomen pad bij te werken om meer geschikte coördinaten te bekomen. Het maakt gebruik van de methode voor het vinden van snijpunten met een vlak in een 3D omgeving. Het beschouwde vlak is datgene waarop de gedeelde veelhoek ligt tussen de veelvlakken van de 3D navigatiemesh. Het middelpunt van zo'n veelhoek wordt hier als de knoop van de graaf beschouwd. We schieten hier dus de straal af startend van de voorganger van de beschouwde knoop en in de richting van de opvolger van die knoop. Met de kennis van dit snijpunt kunnen we dan het dichtste punt op die veelhoek kiezen.

IV. Lokale padplanning

Voor lokale padplanning is het belangrijk dat we een datastructuur hebben waarmee we vlot de objecten in een omgeving van een actor kunnen opvragen. De meest geschikte aanpak lijkt ons om een roosterstructuur te gebruiken waarbij enkel de k dichtste objecten bij een actor beschouwd worden. Het opsporen van (mogelijke) botsingen in 3D omgevingen kan sterk vereenvoudigd worden door de omgeving te beschouwen in eenvoudige structuren, zoals vlakken en sferen. Indien deze objecten toch vrij complex zijn kunnen we er een omhullende eenvoudige structuur rond plaatsen om zo het gemiddeld aantal bewerkingen te verlagen [4].

Maar het is beter te voorkomen dan te genezen, dus een goede ontwijkingsstrategie is wel gewenst. Een van de eerste werken over lokale ontwijking voor grote groepen is het boid model van Reynolds [5]. Daarbij maakt hij een onderscheid tussen twee ontwijkingsmethodes: de ene werkt volgens krachtvelden, terwijl de andere op basis van richtings- en snelheidsvectoren werkt. De meeste die op deze laatste methode gebaseerd zijn bouwen verder op het concept van velocity obstacles dat door Fiorini et al. [6] geïntroduceerd werd. Hierbij doet elke actor een voorspelling van waar de locatie zal zijn van de dichtsbijzijnde andere actoren binnen een bepaald tijdsvenster. De richtings- en snelheidsvector van de actor wordt dus aangepast, zodanig dat deze buiten het voorspelde pad van de omliggers ligt.

Een probleem met deze methode is dat alle actoren normaal dezelfde strategie gebruiken. Gezien deze slechts op de voorgaande bewegingen van de omliggers steunen, maar niet op de toekomstige bewegingen ervan kan dit voor ongewenste schommelingen zorgen in het afgelegde pad. Een oplossing hiervoor werd voorgesteld met het ORCA-algoritme van Van Den Berg et al. [7]. Bij deze methode worden de velocity obstacles beschouwd, maar op basis daarvan wordt het navigatievlak (de methode is voor 2D) voor elke twee dicht bij elkaar liggende actoren onderverdeeld in twee gelijke deelvlakken; waarbij de ene actor het ene mag gebruiken voor toekomstige navigatie, en omgekeerd voor de andere actor. Dat wordt gedaan voor alle actoren in de directe omgeving van een actor, en deze worden dan samengevoegd om het gebied te bepalen waarbinnen de navigatie momenteel toegelaten is. Het algoritme van Snape et al.[8] is een uitbreiding van dit ORCA-algoritme naar 3D omgevingen waarbij in plaats van deelvlakken nu deelruimtes gebruikt worden. Hierbij is wel nog geen rekening gehouden met de ontwijking van statische obstakels. We gebruiken dat laatste algoritme als de ontwijkingsstrategie tussen bewegende actoren voor ons algoritme, en we gebruiken onze 3D navigatiemesh om de ontwijking met obstakels te doen.

V. Experimenten en conclusie

Er werden tests gedaan op zowel een lege omgeving als op twee meer complexe omgevingen: een stadsomgeving en een grot. Onze voorgestelde 3D navmesh laat het A*-algoritme toe om voor elke actor vrij vlot een pad te vinden naar een doel. De keuze van toegelaten veelvlakken van de navmesh hebben invloed op het eigenlijke aantal knopen die men kan bekomen met zo'n navmesh. De methode die de paden gladder maakt kan ervoor zorgen dat de paden meer optimaal worden, in die zin dat de actoren minder onnodige acties hoeven te doen om tot hun einddoel te komen. Hierdoor ziet het gevolgde pad er meer realistisch uit.

	Stad	**Grot**
Datastructuur		
# veelvlakken	825	89
# knopen (graaf)	2096	174
# bogen (graaf)	8829	552
Aantal genomen stappen A*		
min.	9	4
gem.	226.55	20.84
max.	11170	101
Lengte resulterend pad		
min.	7	4
gem.	14.55	7.11
max.	24	11

References

[1] Greg Snook, "Simplified 3d movement and pathfinding using navigation meshes," *Game Programming Gems*, vol. 1, pp. 288–304, 2000.

[2] Epic Games, "Unreal engine," *URL: http://www.unrealtechnology.com/*, 2013.

[3] Edsger W Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[4] Carol O´ Sullivan and John Dingliana, "Real-time collision detection and response using sphere-trees," 1999.
[5] Craig W Reynolds, "Flocks, herds and schools: A distributed behavioral model," *ACM SIGGRAPH Computer Graphics*, vol. 21, pp. 25–34, 1987.
[6] Paolo Fiorini and Zvi Shiller, "Motion planning in dynamic environments using velocity obstacles," *The International Journal of Robotics Research*, vol. 17, no. 7, pp. 760–772, 1998.
[7] Jur Van Den Berg, Stephen J Guy, Ming Lin, and Dinesh Manocha, "Reciprocal n-body collision avoidance," in *Robotics Research*, pp. 3–19. Springer, 2011.
[8] Jamie Snape and Dinesh Manocha, "Navigating multiple simple-airplanes in 3d workspace," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, 2010, pp. 3974–3980.

Contents

List of Figures

Used abbreviations

Θ*	Theta-star
2D	two-dimensional, two dimensions
3D	three-dimensional, three dimensions
A*	A-star
Ch.	chapter
GPU	graphics processing unit
iff	if and only if
navmesh	navigation mesh
ORCA	optimal reciprocal collision avoidance
Sec.	section
VO	velocity obstacle
voxel	volumetric pixel

Chapter 1

Introduction

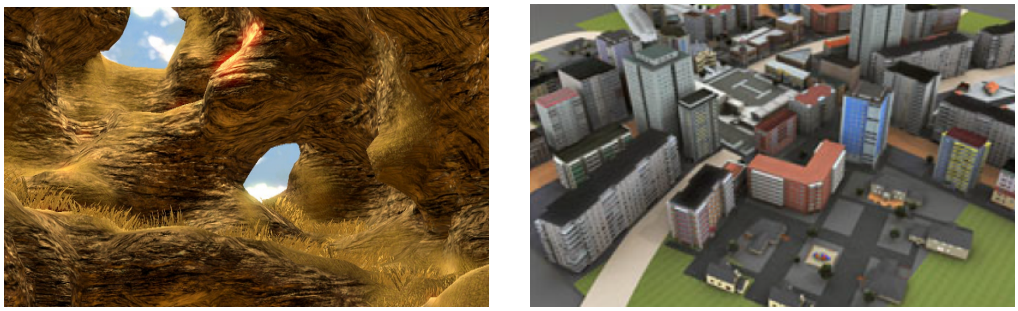
1.1 Motivation

Modern computer systems have the ability to do real-time simulations with large numbers of independent actors that move through a virtual world. Each of those actors will usually have a certain goal. Otherwise, they would just wander around or stand still. Path planning is concerned with moving such actors from their initial location to a goal location. Applications of path planning include: computer games and diverse simulations of traffic, war, evacuations, and various other real-world processes.

Just finding the paths is not sufficient, a path itself should be preferably minimal in length and the actors should try to avoid each other. The movement along the path should also appear as realistic as possible. In addition, everything must happen in real time, this means that there are relatively big time constraints in which the calculations should happen. Especially with very large amounts of actors, e.g. hundreds of thousands, these calculations will take increasing amounts of time.

Most of the research in path planning concerns only two-dimensional navigation, which means that actors can only move on a surface. However, some applications will require path planning for three-dimensional navigation; e.g., when flying or swimming actors are used, such as birds, fishes, airplanes and helicopters. Flocks of birds and schools of fishes are in fact two nice examples of crowd simulation in 3D.

It is also possible that 3D path planning is required for complex environments where there are quite a lot of structures, e.g. caves, forests, and urban environments (see Figure 1.1). In this case there is a need for a specific data structure that defines the navigable space, because only the representation of the terrain here is usually not sufficient for real-time path planning.



(a) canyon (b) urban

Figure 1.1: Two examples of complex environments.

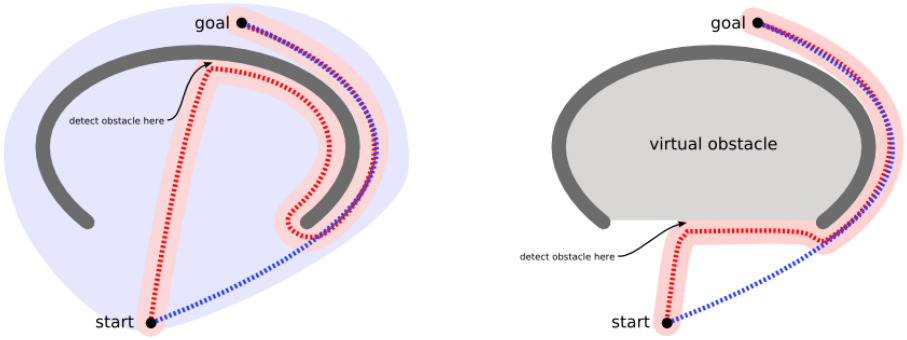
1.2 Problems

Let's consider a virtual environment with various actors in it. Each of those actors has a goal location in the environment. The main problem to solve is moving all these actors from their starting location to their goal location as fast as possible, while also trying to avoid obstacles and other actors along their path.

Path planning algorithms are usually classified in two classes: global path planning and local path planning. Global path planning is finding a path that will allow an actor to move from its initial location to a goal location along an optimal path. This optimal path is normally defined as a shortest path, so that the actor can move to its goal in a minimum amount of time. Global path planning has to consider all the information of the obstacles of the environment towards its location..Local path planning is mainly concerned with avoiding collisions, so only obstacles in a close vicinity of the actor are taken into account.

It is possible that only one type of path planner suffices for a certain application. When moving through an environment with very few or no static obstacles, it might suffice to only use a local path planner, since an actor can usually just move in one straight line towards its goal in that case. However, when there are a lot of static obstacles, we might need a global path planner. An actor with just local path planning will only look at the local environment when making a decision, a problem here might be that a dead end in the environment will be discovered too late , so that the actual path taken is much longer than necessary. The task of a global path planner is to avoid these situations and to avoid obstacles. However, just looking ahead to avoid obstacles is not sufficient to obtain the shortest path. See Figure 1.2 for some examples of these problems with path planning.

The data structures used to model these environments have a huge impact on the performance of the path planning, and have to allow path planning for each actor needs to happen in real-time. So it's desirable to have a data structure that scales well with both the complexity of the environment, and all the actors moving in it. For local path



(a) only local path planning (b) local + simple global path planning

Figure 1.2: Here you can see two examples of problems with path planning. The pink area is the area scanned by the local path planning, the red path is the one taken, the blue path is the optimal path. The first example uses only local planning, and as you can see, the obstacle is detected too late. The second example uses global path planning, but the global path planner is quite simple in this case and the path taken is not yet the shortest one, but clearly better than when no global path planning is used.

planning, easy access to the obstacles and other actors in the neighborhood of each actor is desirable; while for global path planning we desire a minimized representation of the environment, that allows an algorithm to find a path as fast as possible, while still being the shortest path, or as close as possible to it.

Another problem is that most path planning data structures and algorithms are made for 2D navigation. New complications arise when extending this to 3D, especially when the environments becomes more complex.

1.3 Objectives

The main objective of this thesis is to find a good approach of doing real-time path planning in three dimensions within complex, but natural, environments.

Both global and local path planning will have to be taken into account.

- **Global path planning in three dimensions.** A data structure, which is a compact representation of the environment, has to be found for this. The data structure should allow finding shortest paths in the environment, or at least good approximations of them, for all the actors during real time. With the expansion to 3D navigation, this could mean that the environment will be significantly larger than its 2D counterpart.
- **Local path planning in three dimensions.** Only the direct neighborhood of each actor has to be considered for this. This means that a data structure has to be found that can scan the neighborhood quickly in three dimensions, and that the

current movement of each actor along its path needs to be adapted to its direct environment to avoid collisions.

Only static environments will be considered in this work. So, any dynamic obstacle will in fact be another actor. The term obstacle will be mostly used to refer to a static obstacle of the environment.

While parallelism and dynamic environments will not be tested in this thesis, it will be considered for future expansion when choosing a data structure or algorithm.

Chapter 2

Representations of the navigable space

The goal of this chapter is to find the most suitable data structures for both global and local path planning with 3D navigation. There are different possibilities to represent the navigable area used by the path planning. Since most (non-grid) data structures are only suitable for 2D navigation, we can use most of the ideas behind them to create an efficient data structure for the global path planning, namely a volumetric navigation mesh (Sec. 2.2.2). This volumetric navmesh could also be used for the local path planning; but a more evenly structured data structure, like the voxel grid (Sec. 2.2.1) is normally more suitable for this, which on the other hand scales badly for the global path planning. More about global and local path planning in the next two chapters (Ch. 3 and Ch. 4).

These data structures are approximations of the environment that allow for quicker calculations, so that global path planning is possible for real-time applications. However, this also implies that the shortest path in the graph is not always the shortest path in the environment, since information might be lost. Most of these data structures are based on graphs. More information about graphs can be found in the appendix (Sec. A.1).

It should also be noted that some data structures generate nodes along a wall or corner. When using these data structures in practice, it should be noted that the actors themselves are not shapeless. Their distance towards an obstacle should be at least their radius at all times to avoid a collision with it. More about this will be explained in chapter 4. The diameter of the actors should also be taken into account when creating a data structure, since it is pointless to include narrow passages where not even one actor can move through.

Section 2.1 provides an overview of the data structures that are commonly used by path planning. The most frequently used are grids (Sec. 2.1.1), waypoint maps (Sec. 2.1.2), visibility graphs (Sec. 2.1.3), navigation meshes (Sec. 2.1.4) and corridor maps

(Sec. 2.1.5) . Section 2.2 discusses how the ideas for 2D representation can be used to create 3D representations.

2.1 Overview of 2D data structures

Many existing path planning data structures are 2D, since most of the time only navigation over a surface is required. Yet, some of the ideas of 2D data structures are also useful when a data structure for 3D navigation is desired.

2.1.1 Grids

Grids are a spatial distribution that have various uses. Each unit of a grid is called a cell. A grid can be easily indexed to quickly access a cell of the grid. We can even use a coordinate system as spatial indexing when we use a regular grid.

Grids in terms of spatial distribution are usually the geometric regular grids. The geometric unstructured grids will mostly be referred to as meshes.

Each grid can be used as a graph structure (Appendix A). How we interpret a grid as a graph structure depends on how the movement between cells is defined. It will be assumed here that the actors move from one cell to one of its neighbors through a border. So, we can see the grid as a graph in which the cells can be seen as the nodes and the borders between adjacent cells as the edges. There are also different ways to define a neighbor in the grid. But since movement is allowed in all direction, we will also consider the diagonal neighbors.

Because each environment can easily be partitioned into a grid of equally spaced tiles, and since grids can be used as a graph, they form a commonly used data structure to do path planning on. However, the main disadvantage of running a pathfinding algorithm on grids is that their performance is dependent on the amount of cells used by the grid. Both time and space complexity scale linearly with the amounts of cells in the grid. The use of fine grids for path planning leads to a slow performance, while the use of coarse grids may lead to inferior paths.

Bandi et al. [3] describe how static obstacles could be avoided for a grid based data structure. They consider the group of cells that contain obstacles as hole regions. Their neighboring cells, which are still valid locations for the actors, are marked as border cells. These neighbors could be just the direct ones or even all ones, including the diagonal neighbors. The global path planning presented by them avoids these border cells.

The border cells are generated by a flood fill, in which all cells are considered and the neighboring cells of hole cells are then marked as border cells. Unreachable cells are also

marked as border cells during this generation.

While the method presented here is simple to implement, the efficiency of it depends heavily on the grade of discretization used. If it is too coarse it may look very natural. Moving obstacles are also ignored by this method.

Instead of using a discrete occupancy grid where the grids have only empty or occupied state, one could also make use of a more continuous representation. One of such techniques is to use a probabilistic occupancy model. Such model allows to store knowledge (usually just some value between 0.0 and 1.0) about the uncertainty on the obstacles positions and orientation. This takes away the need to impose an arbitrary security margin around obstacles to avoid collisions. A method to extend this approach to 3D voxel grids will be seen in Section 2.2.1.

2.1.2 Waypoint Maps

Waypoint maps are widely used [18, 37]. They can be either generated automatically or placed manually. The latter is of course very time-consuming. A waypoint map is essentially a graph and its waypoints are the nodes of that graph. While waypoints may be good for scripted events, it has no information about its neighbors in the graph and of the surrounding environment. It also has scaling issues with both the amount of actors and the size of the environment.

Visibility graphs (Sec. 2.1.3), navigation meshes (Sec. 2.1.4), and roadmaps (Sec. 2.1.5) are actually extensions of this approach that attempt to remove this disadvantage.

2.1.3 Visibility graphs

Visibility graphs [6] are a special kind of graph that can be used to efficiently find a shortest path. As the name suggests, this is a graph in which each edge represents a visible connection between the nodes. In other words, when one node has a clear line of sight to another, then these two nodes are each others neighbor.

The major drawback of visibility graphs is its heavy memory usage when the environment has large open areas or long corridors. See for example in Figure 2.1, this is a simple environment with only 24 nodes, but some of them have more than half the other nodes as its neighbors.

Theoretically speaking, one node can have all other nodes as neighbor. Take a graph with n nodes, if each node has all other nodes as neighbor, then we have a complete graph with $\frac{n(n-1)}{2}$ edges. This makes the memory complexity $O(n^2)$. Another disadvantage that is a consequence of this high complexity is the cost to add or remove a new node to the graph.

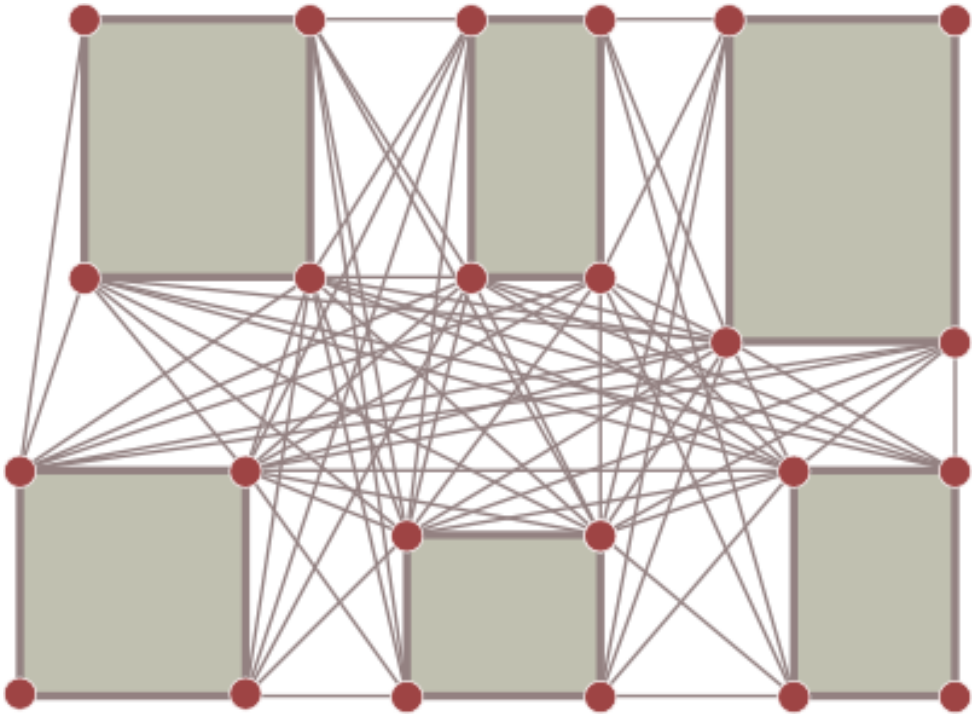


Figure 2.1: A visibility graph on a simple environment with a large open area in the middle. The nodes are placed on the corners of the obstacles.

There are ways to overcome this high memory complexity. One way is to look only at the nearby nodes. However, this can be ineffective in large environments. There are other ways that remove redundant edges from the graph. Careful placement of the nodes will also greatly reduce the amount of edges in the graph. The nodes in Figure 2.1 were chosen on the corners, since that is the shortest way around the corner. Yet, when the nodes are chosen in more central locations, we lose information of the corners and the scope of the navigable area. The next data structure discussed here, navigation meshes, uses similar principles, and it manages to keep more information about the scope of the navigable area in the environment.

2.1.4 Navigation meshes

Navigation meshes, polygon meshes, and polygons maps are often used interchangeably in literature.

Snook [32] describes the use of navigation meshes for pathfinding. A navigation mesh represents an approximation of the open surface area in the environment. The geometry of a navigation mesh needs to adhere to a few simple rules in order to greatly reduce the amount of collision tests required by an object and its static environment:

- the environment has to be divided in triangles;

- the entire mesh must be contiguous, with all adjacent triangles sharing two vertices and a single edge;
- no two triangles should overlap on the same plane.

So, only triangles are used in this definition of navigation meshes. But some of the more recent literature, and some popular game engines (e.g., Unreal Engine [11]), use the name navigation meshes for data structures that are not a mesh of triangles. The principle behind navigation meshes was to reduce the amount of collision tests. Navigation meshes are only defined in the navigable area of the environment. Triangles are convex polygons, this means that every internal angle is less or equal to 180 degrees. Thus, all points that lay inside the triangle are in each others line of sight. This combined with the requirement that all adjacent triangles share a single edge, lets us know for sure that we have a clear line of sight, ignoring any other actors, for any point in one triangle to one of it's neighbors. Since this is due to the convexity of a triangle, one could use any convex polygon instead. Since it is in fact also a mesh of the navigable area, a more general definition for the the term navigation mesh will be given in this work. The navigation mesh as described by Snook [32] can then be seen as a triangular navigation mesh. The term navigable will also be replaced with walkable, since this is still a 2D case.

Definition

A navigation mesh, or navmesh, is a data structure which represents the walkable area of a virtual environment, where:

- the environment has to be divided in convex polygons;
- the entire mesh must be contiguous, with all adjacent polygons sharing two vertices and a single edge;
- no two polygons should overlap on the same plane.

Constructing and updating the structure

Highly detailed navigation meshes might produce the most accurate results, but their overhead could be a limiting factor for real-time games. Both due to the amount of nodes and the memory requirements. The construction itself for detailed navmeshes can also be quite time-consuming, but this happens during an off-line construction. Therefore the navigation mesh should only contain enough detail necessary to facilitate believable movement, not one that necessarily represents every detail.

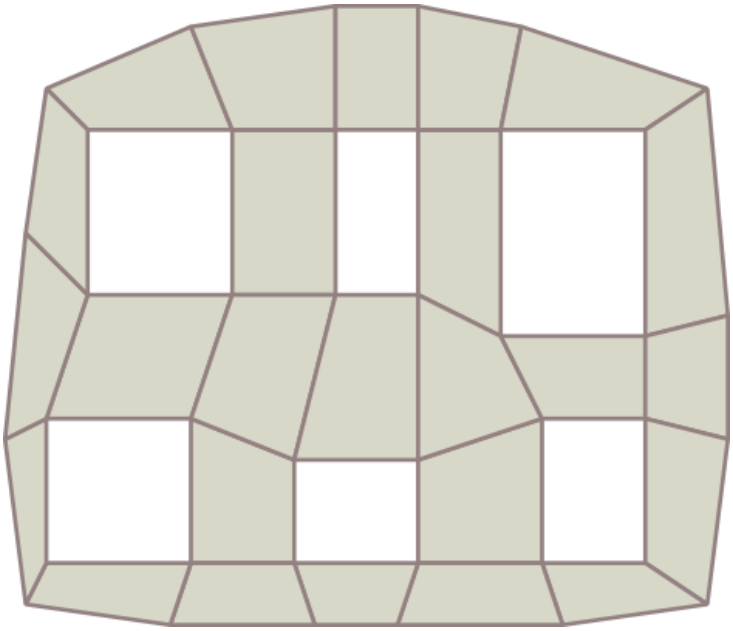
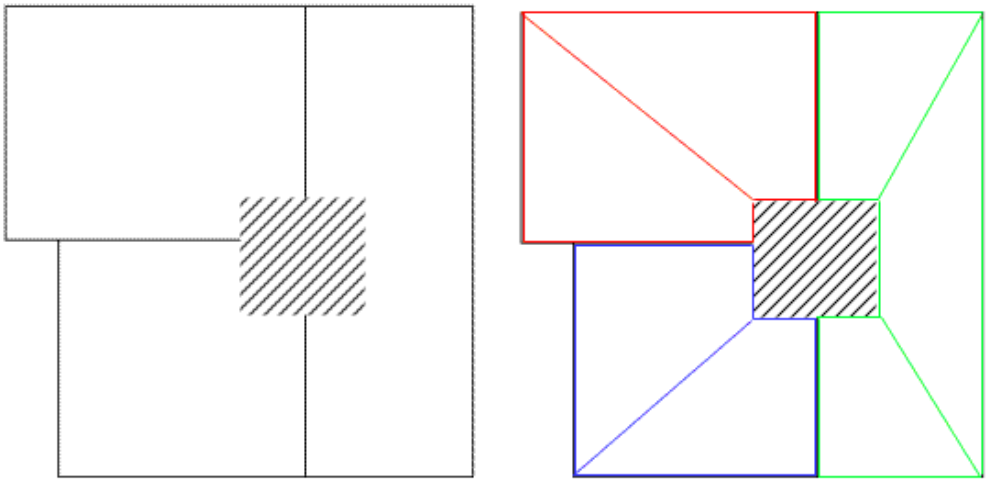


Figure 2.2: A navigation mesh which uses rectangles.



(a) before update (b) after update

Figure 2.3: An example of local updating for navigation meshes

Another disadvantage of navmeshes is that they possibly have to be reconstructed completely when the environment has changed. Yet, this is mostly due to a poor implementation of the navmesh itself or the used tools, since local changes only require local updates. See Figure 2.3 for an example of a local update for a navmesh. An overview of some techniques for local updates in navmeshes are described by van Toll et al. [35].

Using it as navigation graph

There are different ways of using such a navmesh as a navigation graph for a pathfinding algorithm. Similar to grid structures, we can use the center of the polygon as graph node (see Fig. 2.4(a)). The amount of nodes required are minimal with this approach. The neighbors of a node are those in the neighboring nodes. The main problem here is that the path always goes to the center of the polygon. This can become a problem when some of the polygons get quite large.

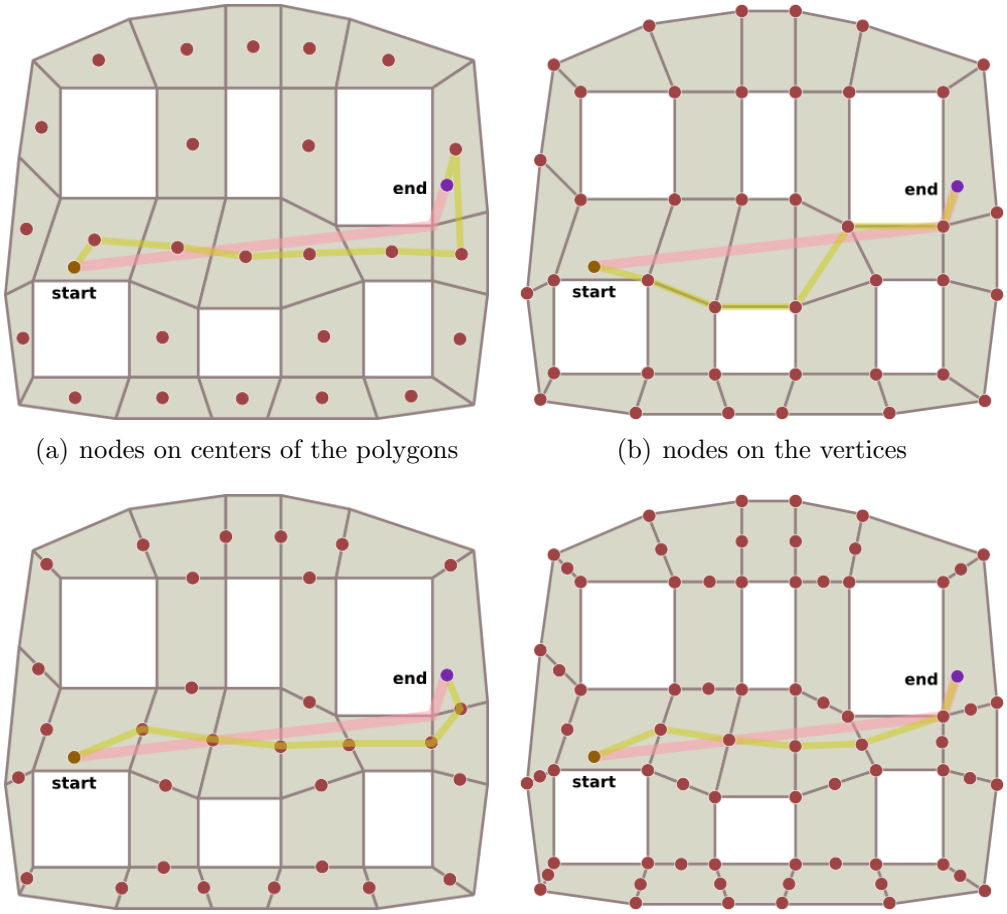
The nodes can also be placed on the vertices, or corners, of the polygons (see Fig. 2.4(b)). The neighbors of a node will be the ones that share a same polygon; in other words, the other corners of the polygons around the node. Yet, the resulting path can stray further from the path than the other approaches, but when the line-of-sight smoothing (Sec. 3.3) is used on it, the optimal path can be obtained, since the list of nodes contains all corners.

Since the movement is between the edges, another approach is to place the nodes in the center of the edges that are shared by two polygons Fig. (2.4(c)). The neighbors of a node here will also be those that share a node with it. The unsmoothed result here is normally quite close to the optimal path. Yet, the resulting path will not contain the corners.

These options can be combined to obtain better results, e.g. the example in Figure 2.4(d). Yet, the amount of nodes will be higher in these cases, which will make the pathfinding slower.

2.1.5 Corridor Maps

The corridor map method was proposed by Geraerts et al. [13]. This method creates a system of collision-free corridors for the static obstacles in an environment. They also present two ways to handle the avoidance of dynamic obstacles: by adding a repulsive force toward the obstacles or by changing the corridor itself. Both techniques are suitable for real-time computation according to their experiments. Local motions are controlled by small potential fields (Sec. 4.3) inside the corridor. A drawback is that the construction of a corridor map can be quite time consuming.



(a) nodes on centers of the polygons

(b) nodes on the vertices

(c) nodes on centers of the shared edges

(d) nodes on both the vertices and the centers of the shared edges

Figure 2.4: Here you can see some different options to obtain a node set from a navigation mesh that can be used by a pathfinding algorithm. The neighbors of the node are those that share a same polygon. The navigation mesh with nodes on both the corners and edge centers. The yellow path is the shortest path on its navigation graph. The red path is the shortest path.

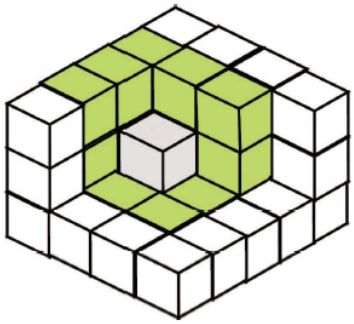


Figure 2.5: A few voxels from a voxel grid. The green voxels are the neighbors of the gray voxel, assuming that diagonal neighbors are also considered.

Geraerts et al. [12] show in a further article how the corridor map method can be used to simultaneously plan the motions of a large number of actors in real-time. So that more than one dynamic object can be avoided, while appearing natural.

They consider two cases of crowd simulation: the first one in which a crowd is navigating toward a common goal and the second one in which the crowd is just wandering around.

This also includes path variation, so that there are alternative paths available for an actor to use within a corridor. It is done by adding a random force to the attractive force. This makes the paths a bit less predictable, while enhancing the realism a bit.

It should be quite possible to extend this method to three dimension. However, due to the fact that this method is based around small corridors, they are mostly suited for urban environments or structures with straight corridors. Therefore, they are not so much suited for environments that include wide open areas that are quite common in the air or underwater.

2.2 Extension to 3D representations

2.2.1 Voxel grid

An extension of 2D regular grids to 3D is usually called a voxel grid, voxel set, or structured volumetric data set [2]. Such data sets are already used in many application that use complex 3D representations of structures, e.g. the data retrieved by some medical scans. Such 3D voxel grids are also used for volume rendering. When the data is taken from scans, it will normally be done on samples taken at equally spaced points in all dimensions (x, y and z). For games and other simulations, these voxels can be used to represent complex terrain features, e.g. caves.

Yet, it should be noted that there are different ways to interpret such data structures. However, we are not directly concerned with rendering, so this will not be pursued further.

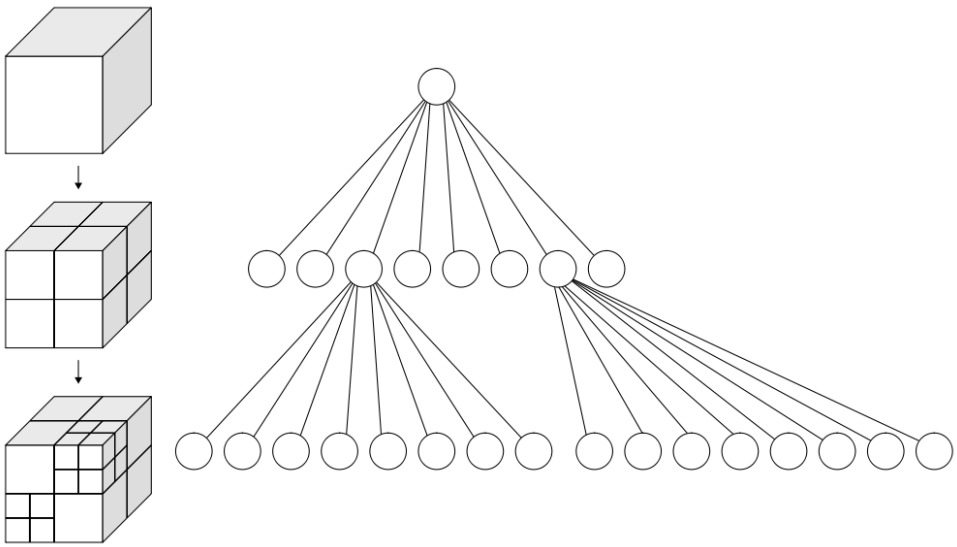


Figure 2.6: Octree

This means that no assumptions will be made about how the environment is exactly represented, so the methods presented here will be usable on both environments represented by voxel structures or by mesh structures.

Basically, we will create new structures which are approximations of these environment representations. These will be either stored as a voxel structure or as a mesh structure.

These voxel structures can be implemented by data structures like octrees or k-d trees. These data structures organize their data in a tree-based way; so that actions like searching, adding or removing objects, can have a good average time complexity: $O(\log n)$ with n the amount of cells. The space complexity can also be greatly reduced when using intelligent data structures like for example the sparse voxel octree as described by Laine et al. [19]. This keeps track of only the surfaces of the environment and not all inaccessible voxels.

An extension of the probabilistic occupancy model discussed in Section 2.1.1 is presented by Payeur [23]. It takes advantage of probabilistic occupancy model characteristics that can be used to build repulsive and attractive potential fields in a three-dimensional voxel grid structure. This approach allows for an enlargement of the space that is used for the planning, especially within narrow areas. His experiments show that the use of a probabilistic model as a multiresolution structure can lead to a reduction of up to 40% of the computation time in general. This strategy appears to be valid when extended into a full 3D space.

The main problem of using voxel grid structures for path planning is the amount of nodes that have to be considered along a path. While the use of multiresolution grids can greatly reduce the amount of work, it usually has still too many nodes to quickly execute a path planning algorithm on them when we have multiple agents. Also, the use

of probabilistic occupancy models is actually a rough estimation. This approach is quite dependent on how fine or coarse a grid of each level of resolution is. The coordinates considered for path planning are usually only the voxel centers or corners. Coarse grid cells may lead to better time performances, but the loss of environmental information might be too big for good results. Furthermore, it is also possible that these voxels are not aligned with the actual environment.

2.2.2 Volumetric navigation mesh

One of the ideas behind navigation meshes is to model a three-dimensional environment as a two-dimensional structure, since it is usually just used for navigation over a surface. However, his definition introduces new problems when more complex environments are used, or when full three-dimensional navigation is required. Different layers of navigation meshes could be used and connected to model multilayered environments [36]. However, this approach is mainly limited to modeling the walkable areas.

We can find a better expansion for 3D for areas with volumetric obstacles. Let's call this new data structure a volumetric navigation mesh. The definition of the navigation mesh used in Section 2.1.4 can be used as a base for this volumetric navigation mesh.

Definition

A volumetric navigation mesh, or volumetric navmesh, is a data structure which represents the navigable area of a virtual environment, where:

- the environment has to be divided in convex 3D polyhedra;
- the entire mesh must be contiguous, with all adjacent polyghedra sharing a single face;
- no two polyhedra should overlap on the same space.

Generating the structure

There are various approaches to generate a volumetric polygonal mesh structure in 3D environments, such as the *Adaptive Space Filling Volumes 3D* algorithm proposed by Hale et al. [14]. It works by seeding the world space with a series of unit cubes. These cubes then grow as big as possible. There is also an automatic subdividing system in it that allows conversion of these cubes into higher-order convex polygons. They show that their method is less complex and provides better results compared to some other approaches like *Space-filling Volumes* and *Automatic Path Node Generation mesh methods*.

Another approach that we propose is to start with a mesh consisting of only one big convex polyhedron, normally a rectangular cuboid, that contains the whole area. Then we can look at each obstacle in the environment at a time, and split up the convex polyhedron(s) it is currently in, according to the characteristics of that obstacle. This can be seen as starting with an empty environment and then keep adding the obstacles, while doing local operations to update the navmesh, quite similar to the 2D example as seen in Figure 2.3. A post-processing step can then merge some neighboring convex polyhedra of the mesh, as long as the properties as given in Section 2.2.2 are enforced.

Chapter 3

Global path planning

Since most data structures that represent environments can be modeled as a graph (see Ch. 2),it is assumed in this chapter that a graph representation is used. The mathematical graph theory is important for this, an introduction to graph theory can be found in Sec. A.1.

It is possible to use any graph search algorithm because the use of a graph-based environment representation is assumed in this chapter. Typically, some variant of the A* search algorithm (Sec. 3.1.3 and 3.1.4) is used to find a shortest path in graphs. This algorithm is guaranteed to find an optimal shortest path to a goal node, while still having a good performance when compared to similar algorithms that find an optimal shortest path. The A* algorithm is based on the ideas behind both Dijkstra's algorithm (Sec. 3.1.1) and the best-first search algorithm (Sec. 3.1.2). Keep in mind that these algorithms are used on discretized structures of the environment. In other words, the structure is an approximation of the environment and some of its information is lost. This means that a shortest path of the structure is not necessarily a shortest path in its continuous counterpart.

After that, the ideas behind path smoothing (Sec. 3.3) are explained. A good method to do this is to remove the redundant subgoals of the graph. This can be done by using line of sight detection. Section 3.4 shows how that can be done efficiently for some simple structures (which can be used as bounding volume for more complex structures). Finally, Section 3.5 explains our method to use the 3D navmesh that was introduced in Sec. 2.2.2 can be used in combination with a pathfinding algorithm, and how the path can be preprocessed to obtain a better result.

3.1 Pathfinding algorithms

We have a graph structure that represents the navigable area of the environment. Our objective is to find the shortest path in this graph. This means that we need a pathfinding algorithm. There are many types of such algorithms, this section gives an overview of some of the most common pathfinding algorithms. First the Dijkstra's algorithm (Sec. 3.1.1) is introduced, then follows the Best-first Search algorithm (Sec. 3.1.2), and finally the A* algorithm and some of its variants are discussed (Sec. 3.1.3 and Sec. 3.1.4).

3.1.1 Dijkstra's algorithm

Dijkstra's algorithm [9] is a graph search algorithm that is able to find the shortest path in a graph with strictly positive weights on its edges. To find a shortest path, we must have a starting node and a goal node.

The nodes are subdivided into three distinct sets:

Set A The visited nodes. For these nodes the path of minimum length from the start node is known. Nodes will be added to this set in order of increasing minimum path length from the starting node.

Set B The unvisited nodes which are a neighbor of at least one visited node. The next visited note will be selected from this set.

Set C The remaining unvisited nodes.

At first all nodes, except the starting node, will be in set C. The starting node will be added to set A.

Each node will have a tentative distance value, which is the distance from the starting node, so the starting node will have distance zero. We now set the distance values of all the other nodes (those in set C) to infinity. We also consider the starting node to be the current node.

Now repeat the following steps until the goal node is added to set A:

Step 1 Move the neighbors from the current node that are in set C to set B.

Step 2 Update all distance values for the nodes in B that are neighbor of the current node. This is done by adding the distance value of the current node to the weight of the vertex that connects the neighbor to the current node. If the calculated value is lower than the current distance value of the neighbor, then the calculated value becomes the new distance value of that neighbor. A simple example of this updating can be seen in Figure 3.1.

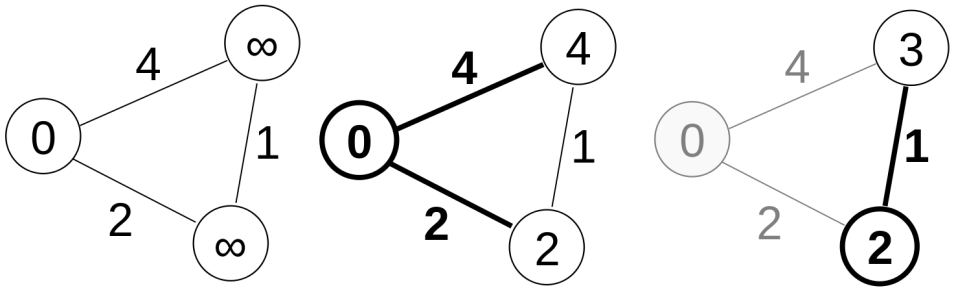
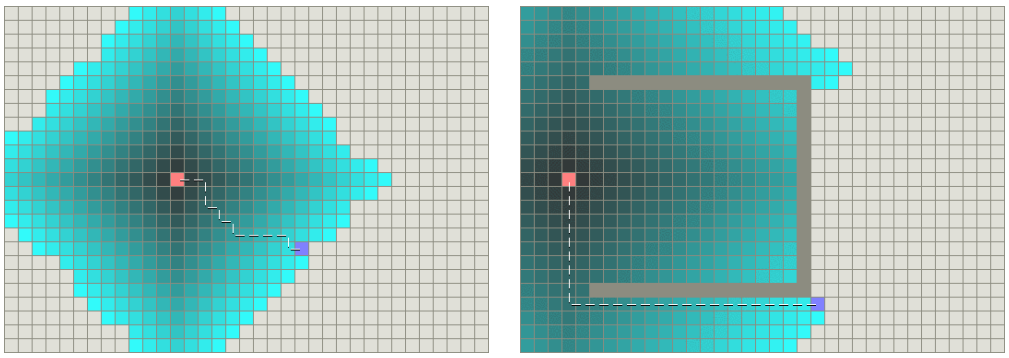


Figure 3.1: A simple example of the updating of the neighbor nodes in Dijkstra's algorithm.



(a) with no obstacles (b) with obstacle

Figure 3.2: Two examples of Dijkstra's algorithm on a grid. The pink square is the starting point, the blue square is the goal, and the teal areas show the areas that were used by Dijkstra's algorithm, the darker ones are closest to the starting point.

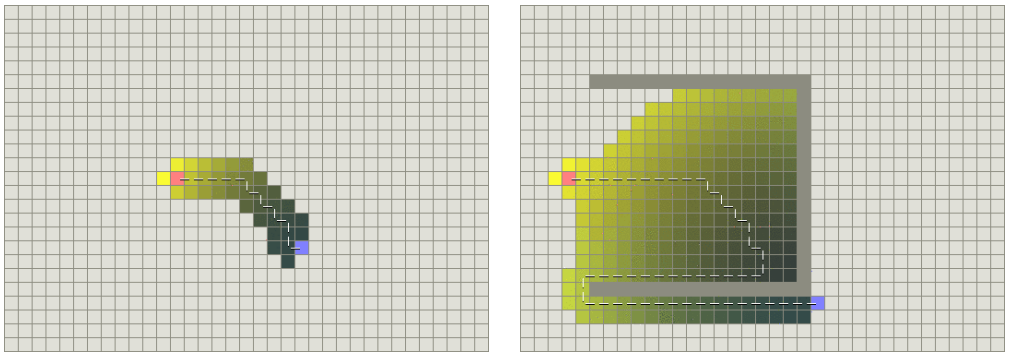
Step 3 When all distance values are updated, we choose the node with the lowest distance value to become the new current node and to be moved to set A.

When the goal node is added to A, we can trace our way back from that goal node to the starting node. The result is a shortest path. However, this algorithm does not always move directly towards the goal, since only the distance towards the starting position is taken into account, and no information of the distance towards the goal is used. See Figure 3.2 for an example of this on a grid (more about grids in Sec. 2.1.1).

3.1.2 Greedy best-first search

The greedy best-first search algorithm works in a similar way, but instead of calculating the distance to the starting point for each node, it estimates the distance to the goal. This estimation is done by using an estimation function called a heuristic (some examples of heuristics can be found in Sec. 3.2).

The basic form of this algorithm is not always guaranteed to find a shortest path when there are obstacles on the path. Because it wants to get to the goal as fast as possible, it might end up in a dead end (see Fig. 3.3(b)).



(a) with no obstacles (b) with obstacle

Figure 3.3: Two examples of the best-first search algorithm on a grid. The pink square is the starting point, the blue square is the goal, and the yellow areas show the areas that were used by the best-first search algorithm, the darkest ones are closest to the goal.

The upside is that it works much faster than Dijkstra's algorithm because it tries to move towards the goal in each step. Figure 3.3 shows that the algorithm has an optimal solution for environments with no obstacles, but a suboptimal path for environments with non-trivial obstacles.

3.1.3 A* algorithm

The A* algorithm [15] (pronounced A-star algorithm) combines the best of both previous algorithms. It uses the idea from Dijkstra's algorithm to keep the distance to the starting node with each considered node, while using a heuristic to estimate the distance to the goal when choosing the next node.

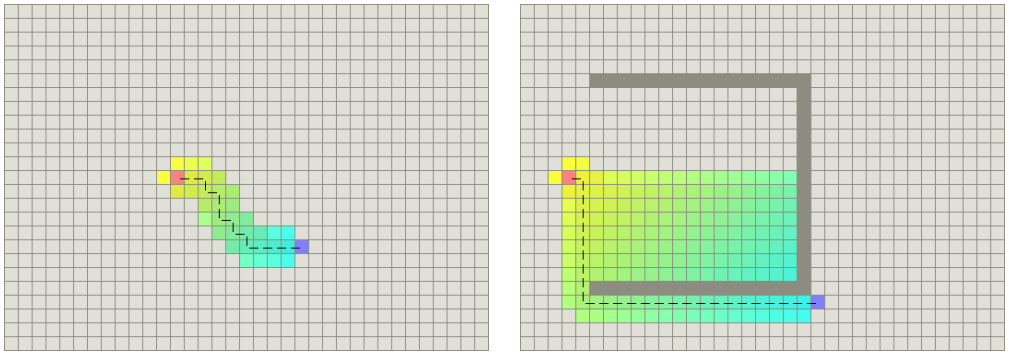
By keeping the distance information in each node, we can guarantee that we will always find the shortest path on the graph with the A* algorithm. Due to the usage of the heuristics, we will also guarantee that we move towards the goal in each step, based on the limited information we know at that point.

The distance to the starting node, which is the distance already traveled, from the node n is denoted with function $g(n)$. The heuristic estimate is denoted with function $h(n)$. The cost function of a node is then denoted as $f(n)$ where:

$$f(n) = g(n) + h(n) \tag{3.1}$$

So $f(n)$ is a combination of the exact cost of the current subpath in that node, $g(n)$, and the estimated cost to complete the path to the goal, $h(n)$. At each step, the next node taken is the one with the lowest value for $f(n)$.

There are many options for the heuristic, $h(n)$. The most important will be discussed in Section 3.2.



(a) with no obstacles (b) with obstacle

Figure 3.4: Two examples of the A* algorithm on a grid. The pink square is the starting point, the blue square is the goal, and the multicolored areas show the areas that where used by the best-first search algorithm, the yellow ones are closest to the starting point and the teal ones are closest to the goal.

Algorithm 1 A* algorithm

$OPEN \leftarrow \{start\}$
$CLOSED \leftarrow \emptyset$
while $OPEN \neq \emptyset$ **do**
  $current \leftarrow$ node in *open* having the lowest value $f = g + h$
  if $current \equiv goal$ **then**
    return path
  end if
  $OPEN \leftarrow OPEN \setminus \{current\}$
  $CLOSED \leftarrow CLOSED \cup \{current\}$
  for all neighbor nodes *neighbor* of *current* **do**
    $cost \leftarrow g(current) + distance(current, neighbor)$
    if $neighbor \in OPEN \wedge cost < g(neighbor)$ **then**
      $OPEN \leftarrow OPEN \setminus \{neighbor\}$
    end if
    if $neighbor \notin OPEN \wedge neighbor \notin CLOSED$ **then**
      $g(neighbor) \leftarrow cost$
      $OPEN \leftarrow OPEN \cup \{neighbor\}$
    end if
  end for
end while

3.1.4 Variants of the A* algorithm

Θ* algorithm

The Θ* algorithm (pronounced Theta-star algorithm) is an extension of the A* algorithm that doesn't only consider visible neighbors, but every other accessible node in the grid. This allows the pathfinding to take more possible directions into consideration at a higher cost. It does have a greater chance than A* to find an optimal path, while it also requires an easier smoothing function [21].

De Filippis et al. [7] consider how the Θ* algorithm can be used on 3D graphs and compare it to the A* algorithm. Their focus is on the flight path generation of UAVs. Their analysis shows that Θ* is a better choice for path planning on graphs with many objects, e.g. an alpine environment cluttered with obstacles.

To implement the Θ* algorithm one needs to either represent the environment through a visibility graph (we will get back to this kind of graph in 2.1.3) or use a more sophisticated neighborhood function that only selects the furthest visible neighbors. These two options have a large complexity and scale badly with the amount of visible nodes, as you will see in the discussion of the visibility graphs.

D* algorithm and variants

Carsten et al. [4] present an interpolation-based algorithm that is optimized for path planning in 3D voxel grids. It is based on the D* algorithm, which is mostly used for robot navigation. The D* algorithm is an extension of the A* algorithm that is able to efficiently repair the paths when changes happen in the graph. Due to this they could be useful for dynamic environments. The Field D* variant is also able to find paths at any angle.

The algorithm uses an efficient approximation technique to provide real-time performance. It produces a nice straight path in a voxel grid. The interpolation methods presented there can be used in voxel grids to smooth the resulting path. Each voxel also has a cost depending on its occupancy (ranging from free to full). It also provides the option to add directional-dependent costs into the planning process.

3.2 Heuristics

There are many options for the heuristics that can be used for A*, or one of its variants.

To guarantee that the shortest path is found with A*, the heuristic $h(x)$ must satisfy the following property for nodes x and y of the graph:

$$h(x) \leq d(x, y) + h(y) \tag{3.2}$$

with $d(x, y)$ the exact distance to the goal. This means that we should never overestimate the actual distance to the goal.

If $h(x) = 0$, then we turn the A* algorithm back into Dijkstra's algorithm.

The heuristic on node n is denoted as $h(n)$.

Some simple heuristics are the Manhattan distance and the Chebyshev distance. However, these two distance metrics are only useful when only tile-based movement of the actors is possible, so they will not be explained in more detail.

Since we work in an Euclidean space, the actual distance between two points in it can be measured with the Euclidean metric. This is also one of the best candidates to use as a heuristic for the A* algorithm.

3.2.1 Euclidean distance

Two dimensions

The distance between two points, $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, in a two-dimensional Euclidean plane is given by:

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{3.3}$$

Three dimensions

The distance between two points, $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$, in a three-dimensional Euclidean space is given by:

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{3.4}$$

Analysis of the heuristic

The algorithm suffices property 3.2 since it can easily be proven with the triangle inequality property (A.5) that:

$$d(x, g) \leq d(x, y) + d(y, g) \tag{3.5}$$

since the same distance metric is used.

3.2.2 Squared Euclidean distance

The square root operation in the calculation of the Euclidean distance is quite expensive. Since we only want to compare distances, we could just ignore it, this means that we use the Euclidean distance squared.

Two dimensions

The squared distance between two points, $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, in a two-dimensional Euclidean plane is given by:

$$d^2(p_1, p_2) = (x_1 - x_2)^2 + (y_1 - y_2)^2 \tag{3.6}$$

Three dimensions

The squared distance between two points, $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$, in a three-dimensional Euclidean space is given by:

$$d^2(p_1, p_2) = (x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 \tag{3.7}$$

Analysis of the heuristic

So we have for point x and goal g the following heuristic h:

$$h(x) = d^2(x, g) \tag{3.8}$$

Let us look if property 3.2 is sufficed for 3.8. It can easily be proven that the equation:

$$d^2(x, g) \leq d(x, y) + d^2(y, g) \tag{3.9}$$

is not valid for all x and y. As a matter of fact, $d^2(p_1, p_2)$ is not a real distance metric as it does not satisfy the triangle inequality property (A.5).

The value of $h(x)$ in this case will be an overestimating heuristic, therefore it will reduce the A* algorithm into a best-first search for large distances.

We could try to scale down $h(x)$ so it suffices Property 3.2, but this will lead to the opposite problem: the value of $h(x)$ will be too small compared to $g(x)$ and the A* algorithm will be reduced into Dijkstra's algorithm for small distances.

3.2.3 Perfect heuristic

The perfect heuristic h^* is the one that returns the true distance from it's current position to the goal through the environment [24]. Although the idea behind heuristics is to make an easy-to-calculate approximation of the true distance, this perfect heuristic still has some useful theoretical and some practical uses. Consider the perfect heuristic h^* for any node n

Let's look at the perfect heuristic $h^*(n)$ for any node n. We know that it's result will be exactly equal the distance along the optimal path to the goal, while the same is always true for the result of $g(n)$ towards the current location of the actor. Consequently will $f(n)$ be exactly equal to the length of the optimal path because $f(n) = g(n) + h^*(n)$ (see Property 3.1). This means that the A* algorithm will never consider nodes with higher values for f than the nodes that are along an optimal path, which will make the time complexity of A* linear with the length of the path.

Theoretically, we can use the perfect heuristic to calculate the error of a heuristic. That error is in fact the distance between the heuristic and the perfect one: $d(h(n), h^*(n))$. The lower that error, the lower the time complexity of A* when using h as heuristic.

We could precompute the perfect heuristic for all nodes, but this is not really feasible since the space complexity of A* is already quite high for graphs with many nodes.

Still, the perfect heuristic could also be precomputed for some key points of the environment, and we can use it to create a new heuristic based on it.

Let's call the nodes we precompute the distance between, waypoints. Assume that we precalculated the exact distances between the waypoints. We can now create a new heuristic h for any node n and g when we add in another heuristic $h'(n)$ [28]:

$$h(n) = h'(n, w_1) + h^*(w_1, w_2) + h'(w_2, g) \tag{3.10}$$

Another data structure can be used to find the nearest waypoint quickly, e.g. a coarse grid on top of a fine grid.

3.3 Path smoothing

Once we have obtained a path, it usually contains many redundant nodes. When the nodes are stored as the path, we can see them as a list of subgoals. Yet, only the final goal is important. The other nodes are there to avoid the static obstacles of the environment. When all the costs for navigation are equal in all directions, we can use line of sight culling to smooth the path. Line of sight detection itself will be discussed in the next section (Sec. 3.4).

Keep in mind that the pathfinding algorithm itself usually had no knowledge of these shorter line-of-sight paths, unless they where specifically used (e.g. with the Θ^* algorithm, Sec. 3.1.4). The nodes chosen by a path planner are not always the most optimal ones, since we use an approximate representation of the environment.

3.3.1 Line of sight path smoothing

The shortest path in an open area with no obstacles is to move in a straight line. So, when moving along multiple successive subgoals, moving to the furthest visible subgoal will be the shortest route in that subpath. When a simple obstacle lays on our path, the shortest path around it is usually around its corner, which is in fact the closest visible point along the obstacle. So, corners will usually be used in some data structures to obtain shortest path. These data structures will be discussed in the next chapter (Ch. 2).

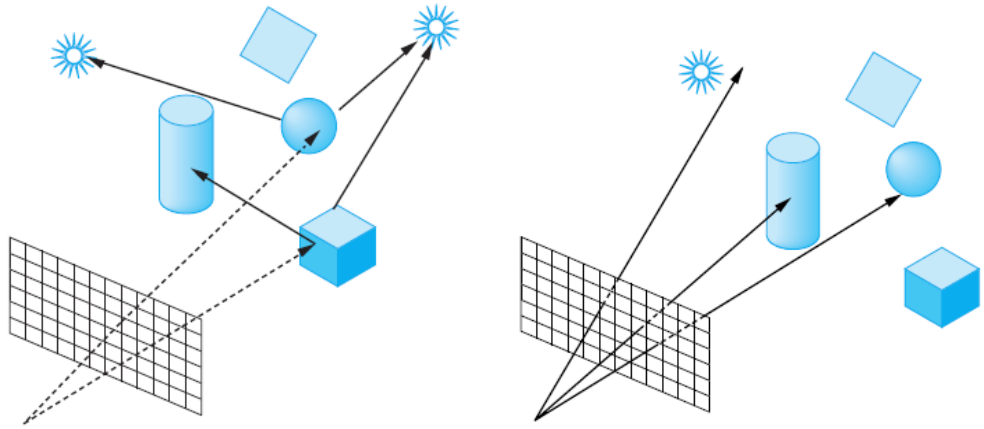
To obtain a more optimal path and since movement to all visible points is allowed, we can remove the redundant subgoals that lay in the line of sight between one goal and its furthest visible subgoal.

This is in fact quite similar to visibility culling used in image rendering, which removes the invisible objects in a scene. Here we remove most visible subgoals, except the furthest one.

The process can go as follows:

1. Node n_i is the final node on the path, which is in fact the goal.
2. Check the previous node n_{i-1} on the path and its predecessor n_{i-2}; in other words, check the two nodes that lay right before the current node on the path.
3. When there is a line of sight between n_{i-2} and n_i, we can remove n_{i-1}. We update the value of i since the current node will move one position earlier on the path, and we continue this step until either $i = 1$ (which means that we are at the front of the list and that there are no other nodes), or
4. If there was no line of sight found, we reduce the value of i by one, so that we move to the previous node on the path. Repeat from step 2.

Other variants on this are of course possible, e.g. when many nodes are in each others line of sight, it can be more efficient to check over larger ranges of nodes.



(a) ray tracing (b) ray casting

Figure 3.5: A comparison between ray tracing and ray casting, as used for visualizing scenes with a light source. The ray tracing (a) in this example is done in order to compute the shadows.

3.4 Line of sight detection

3.4.1 Ray tracing

Ray tracing is a technique which is usually used to aid in visualizing virtual scenes with light sources [2]. The idea behind ray tracing is based on how lighting rays work in nature. Each light source shoots light rays in all directions. These rays go in a straight line until they interact with an obstacle, which will then transform or reflect the interacting ray. Nevertheless, most of the rays do not contribute to the visualizing of a scene. We look at a scene from a particular viewing plane in the virtual world, think of this viewing plane as the lens of a virtual camera that we use to look at the virtual world. Thus, only the rays that intersect with this viewing plane are relevant. Attempting to follow all rays from a light source is a very time-wasting endeavor. However, if we reverse the direction of the rays so that we start from the viewing plane, we know for sure that we have only the relevant rays. So the rays are usually cast from each pixel of the viewing plane into the scene.

There are two main algorithms to implement this technique: the ray casting algorithm and the ray tracing algorithm. The ray casting algorithm stops as soon as an obstacle is intersected by the ray, while the ray tracing algorithm keeps going on with the transformed ray, and is normally used for lighting effects such as reflections, refractions and shadows. Nonetheless, we are are only interested in the ray casting algorithm for doing a line of sight test. Ray casting is also known as ray shooting when it is used for line-of-sight obstacle detection [5].

3.4.2 Calculating intersections in 3D

We can define an object in a 3D space with its surface:

$$f(x, y, z) = f(\mathbf{p}) = 0, \tag{3.11}$$

and a ray from a point $\mathbf{p}_0$ in the direction $\mathbf{d}$ can be represented by its parametric form

$$\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{d}, \tag{3.12}$$

then the intersections are given for the values of t such that

$$f(\mathbf{p}_0 + t\mathbf{d}) = 0, \tag{3.13}$$

which is a scalar equation in t.

Because f is an algebraic surface, f is a sum of polynomial terms of the form $x^i y^j z^k$ and $f(\mathbf{p}_0 + t\mathbf{d})$ is a polynomial in t. Finding an intersection reduces to finding a root of a polynomial. This will require some costly numerical methods in most cases, but there are some special cases which can be solved through quicker methods.

Planes

It is quite easy to find intersections with a plane in a 3D space. The equation of a plane is

$$\mathbf{p} \cdot \mathbf{n} + c = 0, \tag{3.14}$$

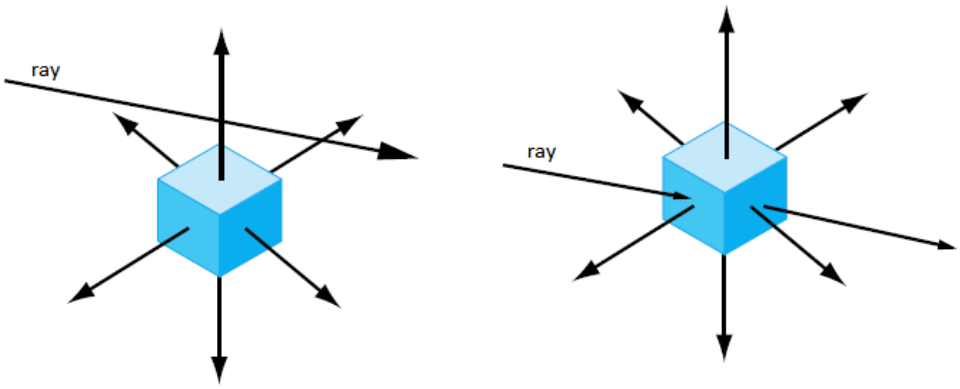
with $\mathbf{n} = (x_n, y_n, z_n)$ the normal vector of the plane. The plane is defined by x_n, y_n, w_n, and c. If we substitute the equation of the ray (3.12) into it, we find

$$t = -\frac{\mathbf{p}_0 \cdot \mathbf{n} + c}{\mathbf{n} \cdot \mathbf{d}}. \tag{3.15}$$

Convex polygons

We can use plane intersection calculations to determine wether a ray intersects with a convex polygon. A polygon is convex when all internal angles are less than or equal to 180 degrees, this means that all corners of the polygon point outwards.

Take an obstacle which is a convex polyhedron with flat faces and straight edges. The ray can enter and leave the obstacle only once. See Figure 3.6(b) for an example. It must enter through a plane that is facing the ray and leave through a plane that faces in the direction of the ray. However, this ray must also intersect all the planes that are extensions of the obstacle's faces, except those that are parallel to the ray. Ignoring the



(a) no intersection (b) intersection

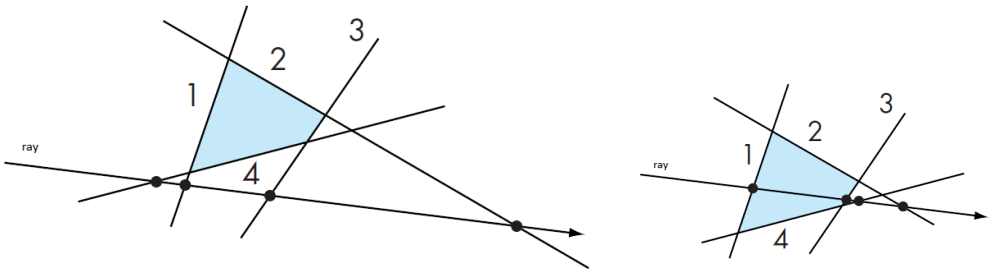
Figure 3.6: Examples of a ray intersecting and missing a cube, which is a convex polygon. The surface normals of each face of the cube are also shown. If these surface normals point towards the starting point of the ray, we have a front-facing plane there. If they point away then there is a back-facing plane at that face.

planes, we can classify these planes into two classes: the front-facing planes and the back-facing planes. The former are all the planes that are extensions of the faces as seen from the origin of the ray, while the latter are those of the faces that are not visible from the origin of the ray.

Consider the intersections of the ray with all the front-facing planes. The entry point must be the intersection furthest along the ray. Likewise, the exit point is the nearest intersection point of all the planes facing away from the origin of the ray. If we consider a ray that misses the same obstacle, we see that the furthest intersection with a front-facing plane is farther from the initial point than the closest intersection with a back-facing plane. So we can try to find these possible entry and exit points by computing the ray-plane intersection points, in any order, and updating the possible entry and exit points as we find the intersections. The test can be halted if we ever find a possible exit point closer than the present entry point or a possible entry point farther than the present exit point. Figure 3.7 shows some examples for this, but for the 2D case. So, lines replace the faces from the 3D case, but the logic remains the same.

Quadratic surfaces

Actors will usually be seen as spheres. Although the global path planner will normally ignore the other actors, it might be useful in some cases of local path planning to do raycasting towards other actors (see Ch. 4). A sphere is a type of quadratic surface. Quadratic surfaces can usually be defined by an equation of second order, their general



(a) no intersection (b) intersection

Figure 3.7: Examples of a ray intersecting and missing a rectangle, which is a convex polygon. The intersection tests are done in the order displayed by the numbers. Line one is the one that always faces the ray origin, which makes it the only one where a candidate entry point can be found. For (a) it will only be found that the ray does not intersect with the polygon after the 4th test, because only then will it find that line 4 yields an exit point closer than the possible entry point. (b) will also require tests on all four lines to determine the intersection, because it still could be possible that the intersection with 4th line was closer than the entry point.

equation can be written in terms of $\mathbf{p} = \begin{bmatrix} x & y & z \end{bmatrix}^T$ as

$$\mathbf{p}^T\mathbf{A}\mathbf{p} + \mathbf{b}^T\mathbf{p} + c = 0, \tag{3.16}$$

where

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{1,2} & a_{2,2} & a_{2,3} \\ a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix}, \ \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{3.17}$$

The 10 independent coefficients in $\mathbf{A}$, $\mathbf{b}$ and c determine a given quadric.

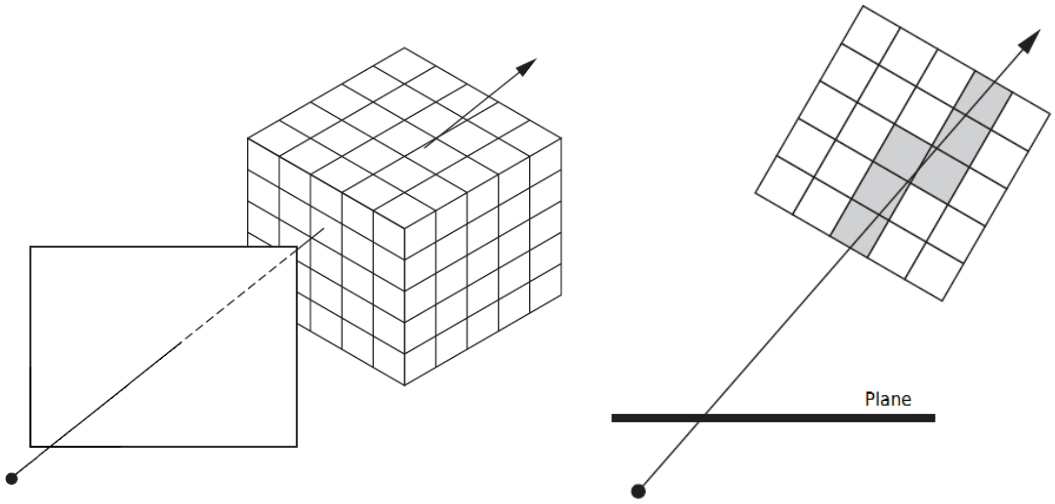
Substituting the equation for a ray leaves us with a scalar quadratic equation to solve for the values of t that yield zero, one, or two intersections. Thus, the solution of the quadratic equation requires only the taking of a single square root. In addition, we can eliminate those rays that miss a quadric object and those that are tangent to it before taking the square root, further simplifying the calculation.

Let's look at a sphere centered at p_c with radius r, which can be written as

$$(\mathbf{p} - \mathbf{p}_c) \cdot (\mathbf{p} - \mathbf{p}_c) - r^2 = 0. \tag{3.18}$$

Substituting in the equation of the ray (3.12) gives us the quadratic equation

$$\mathbf{d} \cdot \mathbf{d}t^2 + 2(\mathbf{p}_0 - \mathbf{p}_c) \cdot \mathbf{d}t + (\mathbf{p}_0 - \mathbf{p}_c) \cdot (\mathbf{p}_0 - \mathbf{p}_c) - r^2 = 0. \tag{3.19}$$



(a) 3D view (b) top view

Figure 3.8: Volume ray casting. This can be used to visualize a volumetric data set, but the principles remains the same for ray casting in terms of path planning. As you can see, only objects that are in, or overlap the gray voxels need to be checked when doing a ray cast.

3.4.3 Intersection culling

Intersection culling means reducing the amount of intersections that have to be calculated during a ray cast. There are two general ways to do this: hierarchical bounding volumes and space partitioning [1].

The idea behind hierarchical bounding volumes is to envelop complicated objects with a simpler object that is much easier to intersect, e.g. spheres (Sec. 3.4.2) or simple convex polygons (Sec. 3.4.2). If there is no intersection, then there is no need to further intersect the complicated object, thus saving time. This can be extended by using a tree to further reduce the amount of intersections.

The amount of ray casts that must be done can also be reduced greatly when using space partitioning to keep track of the obstacles. Take for example a voxel grid. Each voxel keeps track of all obstacles that overlap with it.

3.5 Path planning on the volumetric navmesh

We have seen in Section 2.2.2 how the navigation mesh can be extended from a simple 2D data structure to a volumetric 3D data structure.

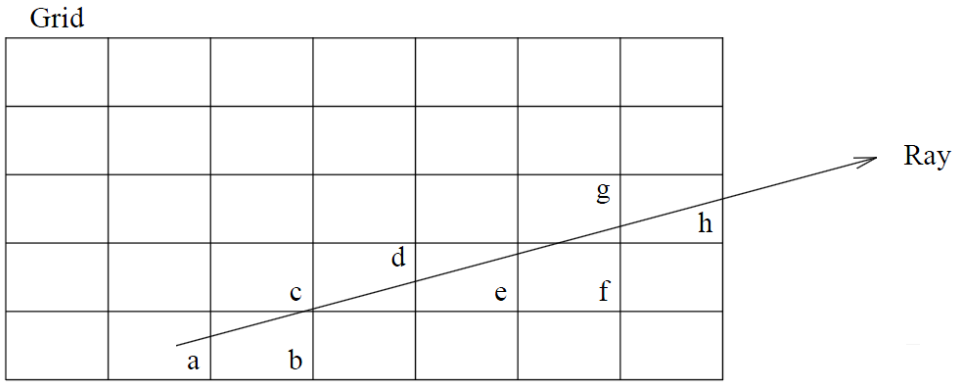


Figure 3.9: Intersection culling by spatial partitioning. In this case we visit the voxels a, b, c, d, e, f, g and h in that order. We only need to check the objects that are referenced by the voxel that we're currently in. This continues until we have an intersected object that is entered by the ray in the current voxel. See the next figure for an example on how to handle obstacles that overlap multiple voxels.

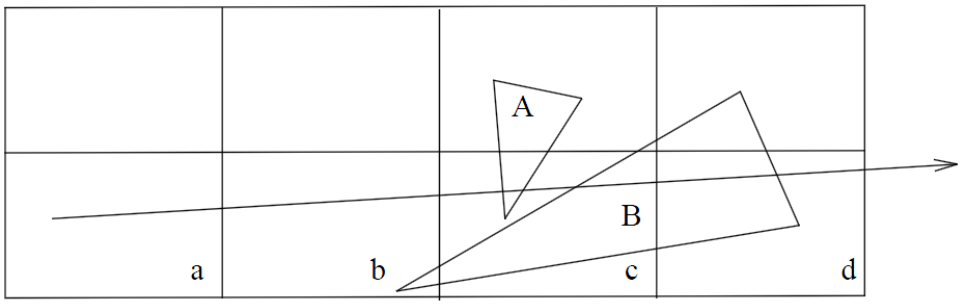


Figure 3.10: Intersection culling by spatial partitioning. Here you can see an example where obstacle B overlaps multiple voxels. When the current voxel is b, we find that B is intersected by the ray. However, the entrance point is not in the current voxel. We need to move to voxel c to look if there are no obstacles that are intersected earlier. As it happens, this is the case with obstacle A. We also don't need to do intersection with B a second time, since we already had done this in the earlier voxel, so we need to keep a list with already tested obstacles during a ray cast.

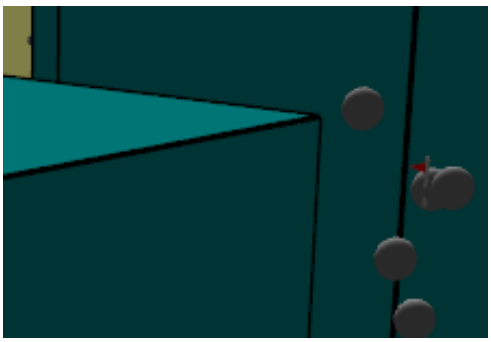


Figure 3.11: A corner in 3D

Using it as navigation graph

We have seen that in the 2D navmesh approach, the best results can be achieved by using the corners as nodes. This idea can be extended to 3D navmesh, but with some modifications. The approach that yields the best results is again attempting to find the best list of corners to aim after doing a line of sight culling of the nodes obtained on the path.

However, while a corner in 2D is a simple point, it is in 3D more than just a simple point. In fact, all the edges and vertices of a geometric face can be viewed as a corner in 3D. See Figure 3.11 for an example of a corner in 3D.

Assume that we used only cuboids to generate the volumetric navmesh. This structure has 6 faces, in which one is a rectangle. So two neighboring cuboids will share one rectangle. An approach to approximate the exact corner taken can be to take all 4 vertices and all 4 edges as the corner nodes. While straight movement between two neighboring cuboids might normally go through near the center of the shared rectangle, we can add the center of the rectangle as another node. This gives us a total of 9 nodes on one rectangle.

Yet, we can obtain the best approximation for the corner nodes when using a slightly modified version of the line-of-sight smoothing algorithm explained in Section 3.3.

It was explained in the previous chapter that the calculation of the intersection of a ray with a plane in a 3D space (Sec. 3.4.2):

$$t = -\frac{\mathbf{p}_0 \cdot \mathbf{n} + c}{\mathbf{n} \cdot \mathbf{d}}. \tag{3.20}$$

We can use that intersection point to obtain a good approximation of the optimal corner point.

Modified line-of-sight smoothing algorithm

We remove the redundant nodes the same way as before, but first we adjust the subgoal coordinates (not the actual nodes on the graph) to let their coordinates better correspond to their optimal corner.

Assume that all nodes of the navigation graph are only on the shared faces of the polygons.

1. Point $\mathbf{g}_i$ is the coordinate of the final goal on the path, which is in fact the goal. Point $\mathbf{g}_0$ the coordinate of the current location.

2. Check the previous goal $\mathbf{g}_{i-1}$ on the path and its predecessor $\mathbf{g}_{i-2}$; in other words, check the two subgoals that lay right before the current goal on the path.

3. A ray cast is done on the plane of the face on which $\mathbf{g}_{i-1}$ lays and which is shared with the polygon that contains $\mathbf{g}_{i-2}$ (this should only be one).

4. Choose the 2D polygon F that contains $\mathbf{g}_{i-1}$ and is the face of the mesh polyhedron which contains $\mathbf{g}_{i-2}$; since it might be possible that $\mathbf{g}_{i-1}$ lies in other faces. Let F^* be the polygon composed of all points taken that are at least r_A, the radius of A, from the boundary ΓF^* towards the center of F. A ray cast is now done on the plane of f, the result is intersection point $\mathbf{p}$.

5. If this intersection point $\mathbf{p}$ is on the boundary of the reduced 2D polygon: ΓF^*, then we only need to adjust the coordinates of $\mathbf{g}_{i-1}$; the visibility with both $\mathbf{g}_{i-2}$ and $\mathbf{g}_i$ is still guaranteed due the characteristics of the 3D navmesh: used mesh polygons are convex; and their faces, with corresponding edges, are shared between successive mesh polygons.

6. If this intersection point $\mathbf{p}$ is not on the boundary of the reduced 2D polygon: ΓF^*, but still inside it, we can remove it.

7. If this intersection point $\mathbf{p}$ is outside the 2D polygon F^*, we have to find the closest point $\mathbf{p}'$ to $\mathbf{p}$ along the edges of f. The coordinates of $\mathbf{g}_{i-1}$ will now be $\mathbf{p}'$.

8. Reduce the value of i by one. Repeat all steps starting from step 2 until $i = 1$, which means that the algorithm went through all subgoals of the path.

The result of doing these steps is a smoothed path with a good approximate of the corners that have to be taken on it as the subgoals. There is no real need of having 9 points on each shared face between successive mesh polygons when we do these smoothing steps afterwards. Yet, it should be noted that the specific knowledge of these optimal corners during the pathfinding on the graph. The influence of the amount of nodes on each shared mesh polygon will be tested in chapter 5.

Algorithm 2 Our proposed smoothing algorithm

$smoothedpath \leftarrow \emptyset$
$g_2 \leftarrow goal$
$g_1 \leftarrow$ subgoal on path before g_2
$g_0 \leftarrow$ subgoal (or start position) on path before g_1
if $||g_0, g_2|| < ||g_0, g_1||$ **then**
  remove g_1 from the path
  $g_1 \leftarrow g_0$
  $g_0 \leftarrow$ subgoal (or start position) on path before g_1
end if
while g_0 is still a node on the path (including the start position) **do**
  $polygon \leftarrow$ the polygon of g_1
  $intersection \leftarrow$ intersection point with plane of $polygon$ for raycast from g_0 to g_2
  $closest \leftarrow$ take the point on $polygon$ that is closest to $intersection$ or just take $intersection$ if it's already inside the polygon
  if $closest$ is near an obstacle **then**
    $closest \leftarrow$ a point closer to the center of the obstacle
  end if
  $smoothedpath \leftarrow smoothedpath \cup closest$
  $g_2 = closest$
  $g_1 = g_0$
  $g_0 \leftarrow$ subgoal (or start position) on path before g_1
end while

Chapter 4

Local Path Planning

The main objective of local path planning is to make sure that the actors do not collide while moving along their planned path. This path, or more specifically the next subgoals of the path, is produced by the global path planner as discussed in Chapter 3. We choose to handle this local path planning with a velocity-based approach. This means that we calculate a preferred velocity before at each interval the engine updates the actors (this normally happens before the generation of a frame, or at each tick of an internal clock). That preferred velocity points towards its next goal on its path, and this velocity is then updated according to the collision avoidance method used. An alternative approach is to use force fields, which will also be discussed briefly in this chapter.

Section 4.1 introduces the concept of neighborhood queries, these are used to retrieve only information about the direct vicinity of an actor, and therefore useful for local path planning. After that, we will look at how collisions can be detected (Sec. 4.2) and how they can be avoided (Sec. 4.3). Finally, an avoidance algorithm that works well in 3D spaces is explained in Section 4.4.

4.1 Neighborhood queries

Neighborhood queries are used to retrieve all other actors and obstacles in the direct vicinity of an agent. If we would consider all other objects in the environment we have a $O(n^2)$ time complexity, with n the amount of objects, to check all pairs of objects and their mutual interactions. Luckily, this complexity can be reduced by a large factor with the use of a good data structure for spatial subdivision. However, the performance of these data structures is usually heavily affected by the maximum density of the spatial subdivision cells, which is usually high.

4.1.1 Neighborhood grid

A neighborhood grid can be constructed as a 3D voxel grid structure. These can have extremely low parallel complexity, like the one proposed by Joselli et al. [16]. Their version is very suitable for simulation of huge flocks.

Yet, the exact implementation of such a neighborhood grid can have huge consequences on the performance of a simulation, e.g. Reynolds [25] describes a technique for running a large flock simulation using parallel processors. It uses a three-dimensional grid structure where each cell has its bucket. So it is in fact a neighborhood grid with voxel buckets. While the results are promising, the method proposed there still has some issues. The buckets have a fixed size, this caused the buckets to overflow when their voxel has a large concentration of actors. The fixed size further creates quite a large memory footprint. Another problem with his model was the collision avoidance. There were also a lot of head-on collisions between groups in the experiment. So a good collision avoidance tactic is desirable for large flock simulations.

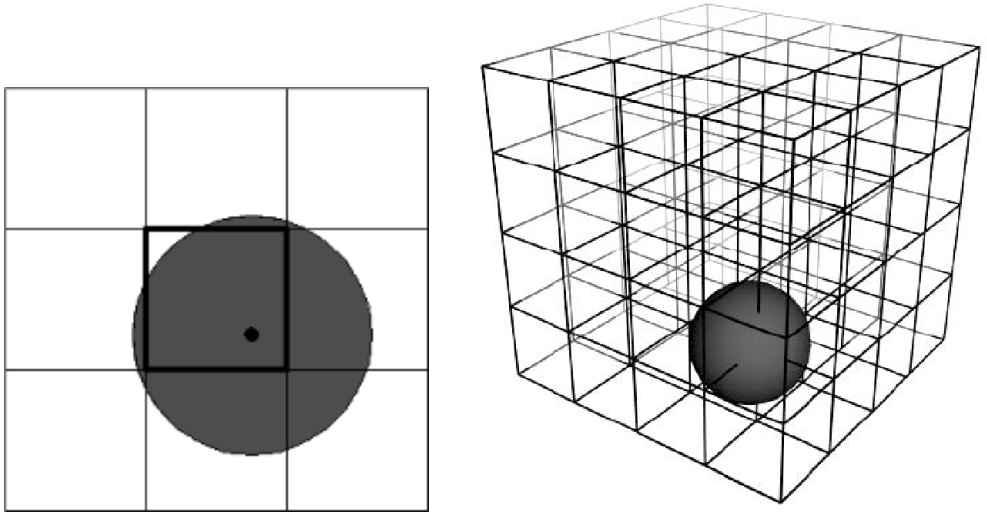
Multiresolution approaches that scale with the amount of actors are also a possibility. Jund et al. [17] present a unified structure for crowd simulation in complex environments. It is based on a generic topological multiresolution model supporting different levels of detail, allowing efficient proximity querying and compatible with real-time rendering and hierarchical path planning. The combinatorial maps model they use is defined in any dimension and may thus be used for 3D path planning. They are convinced that good results can be achieved for flocking simulations.

4.1.2 K-nearest neighbors search

The k-nearest neighbors search is a search method used to find the k nearest neighbors. It is usually used as part of the k-nearest neighbor algorithm for classifying objects. We're not interested in classification, but the search part can be used when we're only interested in the k nearest neighbors for collision detection [25]. A maximum range is normally also specified when using this search in the context of local path planning. Therefore only the k nearest neighbors within that certain range are returned by the search.

Assume we have specified a certain range r and a number k and a point p. For an actor we could use its center point as p. We might also add its radius to the range too, but this depends on how we interpret a neighborhood of an actor. The range r in an Euclidean plane or space is in fact also a radius around p (see Fig. 4.1).

All objects that are in nearby voxel cells are considered, instead of all objects of the environment. The amount of voxels is dependent on the size, the radius of the search, and whether the nearest k neighbors are already found. A reference to all these objects



(a) neighborhood in 2D grid (b) neighborhood in 3D grid

Figure 4.1: Two examples that show the neighborhood of a point in both a 2D plane and 3D space.

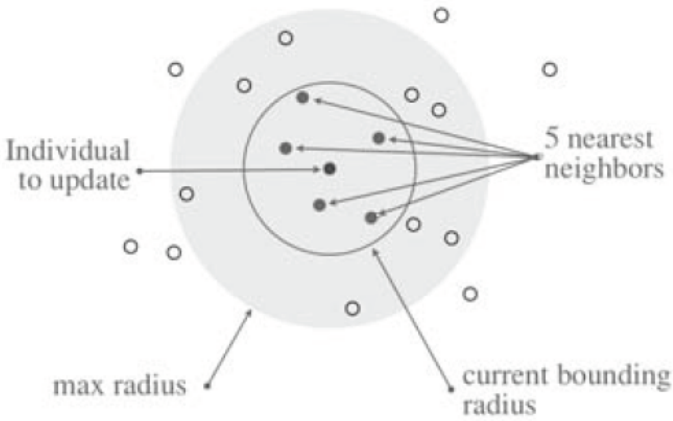


Figure 4.2: This example shows the nearest 5 neighbors of an actor in a 2D plane. The gray area represents the search area within the given maximum range (max radius).

will be stored in a collection and sorted in ascending order according to their distance to p, and an object will be omitted if its distance to p is larger than r. The first k (or less) objects of that collection are the k nearest neighbors of p. See Figure 4.2 for an example.

4.2 Collision detection in 3D

We can say that two objects collide when their shapes intersect with each other. Let's consider two actors A and B in a space $\mathbb{R}^3$, with center points $\mathbf{p}_A$ and $\mathbf{p}_B$, and radii r_A and r_B, respectively. Since we consider two actors as spheres, this calculation can be reduced to calculating the distance between each one's center point and compare it to the

sum of their radii:

$$d^2(\mathbf{p}_A, \mathbf{p}_B) < r_A + r_B. \tag{4.1}$$

Assume that we use a 3D navmesh as defined in 2.2.2, this gives us an approximate of the static environment. The faces of the current polygon the actor (its center point, $\mathbf{p}_A$) that are not connected to another mesh polygon can be seen as the walls. When moving around a corner, we also need to take the vertex or point of it into account. When the distance between $\mathbf{p}_A$ and the closest point of the nearby walls or corners of the current polygon is smaller than r_A, we have a collision.

When actors have more complex forms, we can still use the bounding spheres to simplify the calculation. For example, sphere-trees are a common strategy to do collision detection for complex objects in real-time [22]. This is basically a hierarchical tree of bounding spheres, in which the parent is a bounding sphere of all its children.

4.3 Collision avoidance

One of the first works about large collision avoidance models is the boid flocking model introduced by Reynolds [26]. His model is able to simulate realistic looking flocking behavior, e.g. bird flocks and fishing schools. Each actor in such flock is known as a boid, hence the name boid flocking model.

Two types of collision avoidance are used in his model: one is based on the force field concept, while the other is a steer-to-avoid which is closer to the natural mechanism. The steer-to-avoid is a simulation of an actor guided by vision, which may give a more realistic appearance.

Closely related to this steer-to-avoid tactic are the velocity-based methods. These are based on the principle of adapting the velocity, which is both the speed and direction, of each actor in such a way that collisions are avoided. Some of these velocity-based methods will be discussed in more detail starting from Section 4.3.1. First we will discuss a local path planning method based on the force field concept, namely the potential field method.

Potential Fields

Potential field methods are based on a potential function that is usually discretized in a grid structure. This grid structure is mostly defined in two dimensions, but it should be possible to do with a three-dimensional voxel grid structure.

Methods based on this type of data structure are suitable for large crowd simulations. An example of this is the continuum crowd model of Treuille et al [33]. Global path

planning can be done together with collision avoidance, group coherence conservation and the preservation of other behavior-based features of the moving crowd as needed and defined by the potential function. The potential field can be computed for the entire group. However, there is some loss of individual control, but different crowd behavior like lane forming can be simulated because they are well-known cases in fluid dynamics applications [37].

The potential fields must be computed periodically at a rather high frame rate in order to make the movement seem natural. It must be computed for each actor (or group with common goal) separately. There are multiple studies that try to create a potential field based approach that is suitable to be computed in parallel [20, 8].

4.3.1 Velocity obstacles

Many velocity-based methods are based in the concept of velocity obstacles (abbr.: VO) introduced by Fiorini et al. [10]. The idea behind this is that each actor tries to choose a new movement velocity during each timestep Δt, which is basically the time between each frame, in such a way that this new velocity guarantees no collisions during a certain time window τ. The time window τ should not be confused with the timestep Δt; the former is a distance in the time-space, while the latter is basically the frequency at which this velocity is updated.

Let's consider two actors A and B, with current positions $\mathbf{p}_A$ and $\mathbf{p}_B$, and radii r_A and r_B, respectively. The velocity obstacle $VO^{\tau}_{A|B}$ is the set of all relative velocities of A with respect to B that will result in a collision between A and B at some moment before time τ. It is formally defined as:

$$VO^{\tau}_{A|B} = \{\mathbf{v} | \exists t \in [0, \tau] :: t\mathbf{v} \in D(\mathbf{p}_A - \mathbf{p}_B, r_A + r_B)\} \tag{4.2}$$

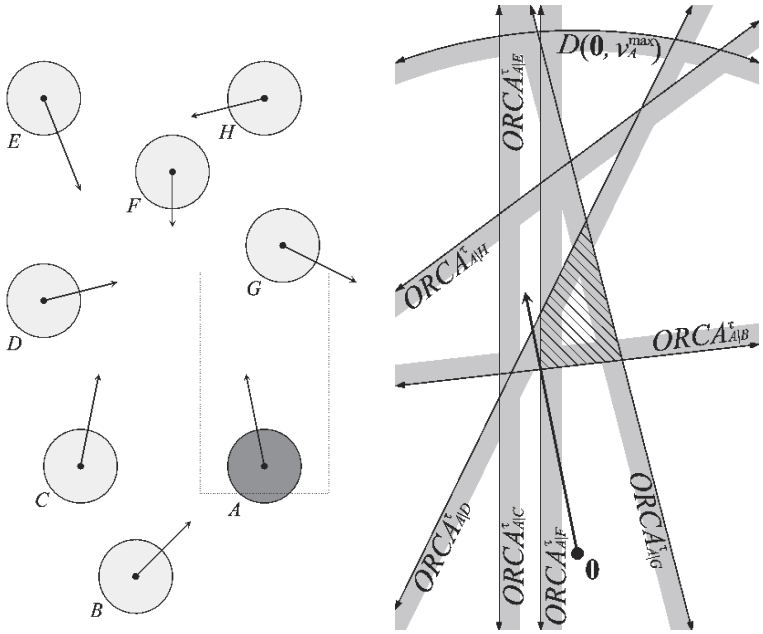
with

$$D(\mathbf{p}, r) = \left\{\mathbf{q} | d^2(\mathbf{q}, \mathbf{p}) < r\right\} \tag{4.3}$$

with D a sphere of radius r centered at $\mathbf{p}$.

4.3.2 Optimal reciprocal collision avoidance

One of the problems with the direct use of velocity obstacles is the fact that undesirable oscillations may occur due to the fact that each actor chooses their new velocity without knowledge of the others. More specifically, it might be possible that the old velocity stays valid with respect to the new VO after all new velocities were taken, but was invalid before they were changed. A solution to this problem was proposed by van den Berg et



(a) velocities

(b) the half-planes of permitted velocities for A

Figure 4.3: An example of the optimal reciprocal collision avoidance approach, (a) shows the actors with their current velocity. (b) shows the half-planes with permitted velocities for actor A induced by each other actor in its vicinity. The time window $\tau = 2$, which means that the $D(\mathbf{0}, v_A^{max})$ models the sphere in which A can move in the time window, and that the half-planes were chosen according to this time window. The half-planes as induced by C and E coincide in this case. The gray-colored sides show the direction of the half-planes. The dashed region is $ORCA_A^{\tau}$ and contains the velocities for A that are permitted with respect to all other actors in its vicinity. In other words, actor A will need to adjust its velocity slightly so that the end point of it falls within that region so that is sure that no collisions will happen within time window τ.

al. [34]. Their method is called optimal reciprocal collision avoidance, or ORCA in short. It provides a sufficient condition for multiple independent actors to avoid each other.

The main idea behind their approach is to define a half-plane (for the 2D case) of velocities $ORCA_{A|B}^{\tau}$ for an actor A with respect to each other actor B. These half-planes are chosen in such a way that $ORCA_{A|B}^{\tau}$ and $ORCA_{B|A}^{\tau}$ contain more velocities close to $\mathbf{v}_A^{opt}$ and $\mathbf{v}_B^{opt}$, respectively, than any other pair of such sets. This also means that the distribution of permitted velocities in these is equal for A and B. The set of velocities that are permitted for A with respect to all nearby actors is the intersection of the half-planes of permitted velocities induced by each other actor; this set is denoted $ORCA_A^{\tau}$ (see Fig. 4.3):

$$ORCA_A^{\tau} = D(\mathbf{0}, v_A^{max}) \cap \bigcap_{B \neq A} ORCA_{A|B}^{\tau}, \tag{4.4}$$

with $D(\mathbf{0}, v_A^{max})$ as the maximum speed constraint on A.

Actor A then choses the velocity that is closest to its preferred velocity $\mathbf{v}_A^{pref}$ as its

new one, v_A^{new}:

$$\mathbf{v}_A^{new} = \underset{v \in ORCA_A^\tau}{\arg\min}\ d^2(\mathbf{v}, \mathbf{v}_A^{pref}). \tag{4.5}$$

The new velocity v_A^{new} is defined by equations (4.4) and (4.5). This computation can be done efficiently as $ORCA_A^\tau$ is a convex region bounded by linear constraints induced by the half-planes of permitted velocities with respect to each of the other nearby actors. It also has only one local minimum.

4.4 Optimal reciprocal collision avoidance in 3D

Snape et al. [31] gives an algorithm for collision-free navigation of multiple flying robots in a 3D space. It is an extension of the 2D version of ORCA (Sec. 4.3.2, [34]). They use a simple-airplane model which can have kinematic and dynamic constraints. This is an extension of the simple-car model used in a lot of works about 2D collision.This simple-airplane model will not be discussed in more detail here, since the actors defined here are simple spheres.

The algorithm itself does not require constant speed nor fixed altitude. It also prevents undesirable oscillations that could otherwise occur due to the avoidance, since it's based on the ORCA scheme. It is capable of handling moving obstacles. However, static obstacles are not yet taken into account in it.

We have chosen to use a simple version of this method as the avoidance technique in our path planning algorithm. This algorithm is velocity-based, which works well with the results from our 3D navmesh global path planning algorithm. The preferred velocity v^{pref} for each actor will be directed to the next subgoal on its current path. To avoid collisions with the static environment, we will also look at the 3D navmesh the center of the actor is currently in. The tactic of collision detection, as discussed in Section 4.2 will be used to check if collisions may occur. When the distance between an actor and the wall corner gets close to the actors radius, the preferred vector will be adjusted to push the actor away from it. The exact range of this boundary zone is specific to the exact requirements of the actors.

4.4.1 Constructing the velocity objects and ORCA spaces

The main steps of the algorithm are the construction of different sets of velocities. These sets are constructed as follows.

Assume that A and B adopt velocities v_A^{opt} and v_B^{opt}, respectively, and let us assume that these causes A and B to be on collision course, i.e. $\mathbf{v}_A^{opt} - \mathbf{v}_A^{opt} \in VO_{A|B}^\tau$. Let $\mathbf{w}$ be

the vector from $\mathbf{v}_A^{opt} - \mathbf{v}_A^{opt}$ to the closest point on the boundary of the velocity obstacle, $\delta VO_{A|B}^{\tau}$:

$$\mathbf{u} = (\underset{\mathbf{v} \in \delta VO_{A|B}^{\tau}}{\arg\min}\, d^2(\mathbf{v}, \mathbf{v}_A^{opt} - \mathbf{v}_B^{opt})) - (\mathbf{v}_A^{opt} - \mathbf{v}_B^{opt}). \tag{4.6}$$

This makes $\mathbf{w}$ the smallest change required to the relative velocity of A and B to avert a collision within τ time.

The reciprocity factor α is used to measure the amount of responsibility an actor has to take. The sum of reciprocity factors α between two actors should equal 1. When all actors share the same responsibility, this factor will always be half for all actors: $\alpha = \frac{1}{2}$. With this value α we can adjust the magnitude of the influence that $\mathbf{w}$ has on the half-space $ORCA_{A|B}^{\tau,\alpha}$:

$$ORCA_{A|B}^{\tau,\alpha} = \{\mathbf{v} | (\mathbf{v} - (\mathbf{v}_A + \alpha \mathbf{w})) \times \mathbf{n}\}, \tag{4.7}$$

with $\mathbf{n}$ the outward normal of $\delta VO_{A|B}^{\tau}$.

So, the set of velocities that are permitted for A with respect to all nearby actors is the intersection of the half-planes of permitted velocities induced by each other actor; this set is denoted $ORCA_A^{\tau}$ (see Fig. 4.3):

$$ORCA_A^{\tau} = D(\mathbf{0}, \mathbf{v}_A^{max}) \cap \bigcap_{B \neq A} ORCA_{A|B}^{\tau}, \tag{4.8}$$

with $D(\mathbf{0}, v_A^{max})$ as the maximum speed constraint on A.

4.4.2 Algorithm

The schematic overview in Figure 4.4 outlines the overall approach.

Each iteration has the following steps for each actor A:

Step 1 The actor acquires its own position and velocity, and those of nearby simple-airplanes based on their prior motion.

Step 2 The velocity obstacles $VO_{A|B}^{\tau}$ for the actor induced by the other actors are constructed for time window τ.

Step 3 The set of velocities $\tilde{V}$, which is the reduced set of constraints, is calculated for each actor.

Step 4 Using the sets $\tilde{V}$, the reciprocity factor α is calculated.

Step 5 The permitted velocities $ORCA_A^{\tau,\alpha}$ for the simple-airplane are calculated as defined by ORCA for a time window τ.

Step 6 The preferred velocity $\mathbf{v}^{pref}$ for each actor is calculated.

Step 7 The velocities $\tilde{\mathbf{v}}^*$ closest to $\mathbf{v}^{pref}$ that lie within $\tilde{V} \cap ORCA$ are selected by sampling $\tilde{V}$. They are then ranked by shortest Euclidean distance, in the velocity space, starting from the *preferred velocity*.

Step 8 The remaining kinematic constraints that are not used in the calculation of $\tilde{V}$ are satisfied by enumerating the set of precomputed curves Γ, a subset of all valid paths defined by those constraints.

Step 9 If the first ranked velocity $\tilde{\mathbf{v}}^*$ (or one within a small threshold) can be attained by taking a path defined by a curve $\gamma \in \Gamma$, then that velocity is valid and the simple-airplane takes that path. Otherwise it tests the next ranked against the curves Γ and so on, until a match is found.

As you can see in the schematic overview (Fig. 4.4), some of these steps can be done concurrently since they don't depend on each other.

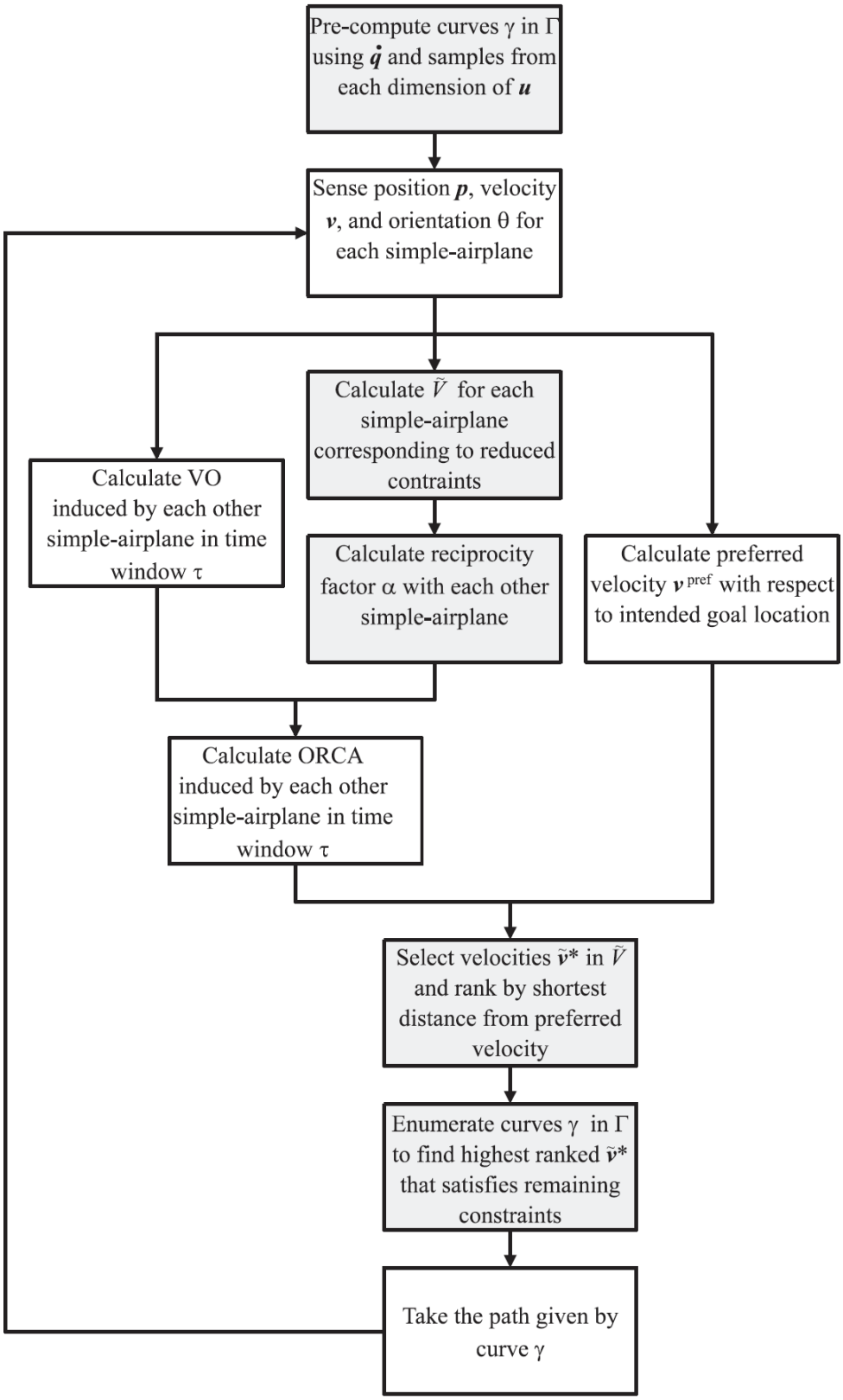


Figure 4.4: This is the schematic overview of the approach for the navigation of a simple-airplane amongst other simple-airplanes, as given by Snape et al.[31]. Shaded actions indicate steps that ensure that the velocities we calculate satisfy the kinematic and dynamic constraints of a simple-airplane. The first stage consisting of the second, third, and fourth shaded actions calculates and samples the reduced set of constraints $\tilde{U}$, with velocities $\tilde{V}$, and the second stage containing the preprocessing and final shaded actions precalculates and enumerates curves in Γ.

Chapter 5

Testing the path planning in 3D

These tests were done using the path planning method based on the 3D navmesh global path planning method we described in Section 2.2.2, 3.5. The local path planning is handled with the avoidance algorithm as described in Section 4.4, which uses neighborhood queries based on a 3D-grid, which is implemented as a kD-tree.

5.1 SteerSuite

SteerSuite [29] is a publicly available open-source framework for visualizing and evaluating steering and path planning algorithms. The framework is made in C++. The benchmarking itself can be done with the SteerBench [30] component of the suite, while the visualizing is done through the SteerSim component.

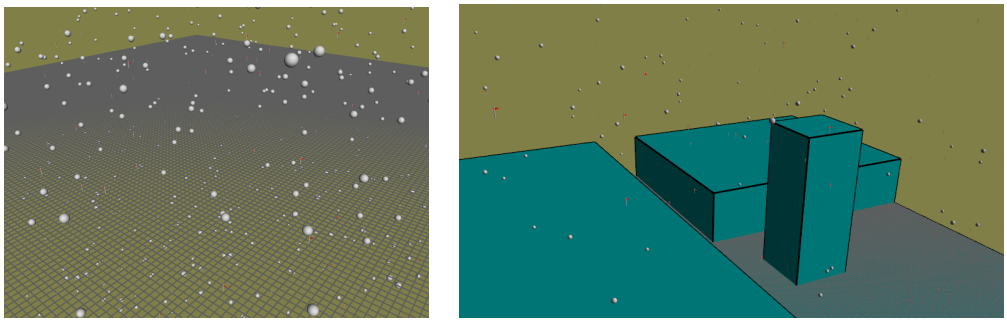
However, this suite was primarily made for navigation across a 2D plane, so some adjustments had to be made to make it possible to evaluate 3D navigation. The main problem was that all metrics, collision detection, and auto-generating is done on the plane with height zero. The height dimension is just ignored.

The graphical component was also slightly adapted to make the objects more pleasing to the eye. Boxes now have a lower plane, and the actors are now spheres instead of discs.

Adjustments were made to the code so that the height dimension was taken into account for all the relevant operations, like the collision checks.

SteerSuite works well with actors up to 4,000 agents. But the SteerBench suite to measure the results, showed some unstable behavior and crashes when more than 1,000 actors were used.

The generation of the test environments was done using external scripts. The initial direction vector of each actor is aimed towards their final goal, this is useful for measuring the average turn rate, which will be explained in Section 5.2.



(a) actors (b) obstacles

Figure 5.1: Steersuite

5.2 Metrics

It is assumed that all the actors are spheres and that they are homogeneous, in other words they all have the same characteristics (radius, maximum speed, and mass). Since the goal is to evaluate the path planning, context-specific constraints on the actors will be ignored. This includes the maximum force magnitude, and therefore the acceleration. In other words, all actors will start the simulation at their maximum speed. This does not mean that all the actors move at their maximum speed at all times, it is entirely possible that an actor might lower its speed to avoid a collision.

We need to evaluate the performance of each actor for a given path planning algorithm in a given environment. This means that some metrics are needed to measure the performance of the actors. The main objective was to get each actor to its goal as fast as possible, while following a realistic looking path, this implies avoiding collisions and keeping the movement as smooth as possible. This means that the three primary variables we want to measure are: travel time, number of collisions, and smoothness of the movement. While the first two variables can be measured directly from the actors, the smoothness is not as straightforward to measure. The number of collisions and smoothness of the movement could also be taken together to get one primary realism variable, but both will be considered separately, since collisions are something we can measure exactly and is something we want to avoid, while the latter is more dependent on interpretation and context.

5.2.1 Travel time

Travel time is the total time each actor spent to reach its goal. The travel is measured as the total time in seconds that an agent spends to reach its goal.

Other useful metrics that are directly related to the travel time are the average speed and the total distance traveled of each agent.

5.2.2 Number of collisions

The number of collisions is the amount of times that two objects collide with each other. This should be as low as possible.

We also keep a log of all collisions with their severity and location in the environment, so that they can be easily be investigated.

5.2.3 Smoothness of the movement

This aspect is not as obvious as the two previous ones, especially since this is one quite dependent on the context. Measuring the smoothness of the movement is to get an idea of the realism of the movement. It is best to first define what can be understood as a *smooth movement*. The following definition is used:

An actor has smooth movement when:

- The amount of changes in the actor's speed is minimal.
- The amount of changes in the direction of the actor is minimal.

These two properties can be measured easily. For realistic behavior, these two should be as minimal as possible, since their use normally requires extra effort of an actor.

Travel effort		
Metric	**Unit**	**Interpretation**
Total time actor spent to reach goal	seconds	Travel time (lower is better)
Total distance traveled per agent	meters	Path length
Average speed per agent	$\frac{\text{meters}}{\text{seconds}}$	
Local avoidance		
Metric	**Unit**	**Interpretation**
Total number of unique collision events		This should be as low as possible
Maximum penetration of collisions	meters	The severity of the collision
Smoothness of the movement		
Metric	**Unit**	**Interpretation**
Total speed change per agent	$\frac{\text{meters}}{\text{seconds}}$	Amount of movement effort, $\|\|velocity_1\|\| - \|\|velocity_2\|\|$
Average speed change per agent	$\frac{\text{meters}}{\text{seconds}^2}$	Time-independent amount of movement effort
Total degrees turned per agent	degrees	Amount of turning effort
Average degrees turned per agent	$\frac{\text{degrees}}{\text{second}}$	Time-independent amount of turning effort

Table 5.1: Overview of metrics

5.3 Experiments

Our tests were done on three different enviroments: an empty environment with no obstacles to test the collision avoidance between agents and to see how the values of some important parameters influence the behavior of each actor. The next one is an urban environment. Buildings are represented by large cuboids in the area. The final environment was a multi-layered cave with a couple of entrances. A 3D navmesh was generated using our proposed method (Sec. 2.2.2). The A* algorithm was then used on the graph of these navmeshes to obtain the shortest paths for each actor. An overview of these results can be seen in table 5.3. We will discuss the urban and cave environment in more detail later.

Only the upper half of the space was used to create the urban and cave environment, so their dimension was $50 \times 100 \times 100$. Thus, the voxel grid structure to model these ones it is $500,000$ voxel cells in size, when the voxel size is the diameter of the actors.

These environments can be represented with far less: 825 and 89. This means that the A* algorithm can find its goal far quicker on these structures, since they also have fewer nodes. It is interesting that the generated navmesh on urban environment was almost 10 times larger than the one generated on the cave environment, since it used lesser obstacles. The cause of this appeared to be the fact that we enforced the usage of cuboids in this algorithm, so the large open areas of the urban area were very tessellated around the tall buildings, since they were not exactly aligned to each other. The cave on the other hand was multilayered and had corridors, so the tessellation did not spread as much.

	Urban	**Cave**
Data structure		
# polyhedra	825	89
# nodes (graph)	2096	174
# edges (graph)	8829	552
Number of steps taken by A*		
min.	9	4
avg.	226.55	20.84
max.	11170	101
Length of resulting path		
min.	7	4
avg.	14.55	7.11
max.	24	11

Table 5.2: Some statistics of the 3D navmeshes

5.3.1 Empty environment

First some tests were run on an empty environment. Only the local path planner is relevant here. The goal of these tests is to see the influence of changes in the values of the following three parameters of the local avoidance algorithm:

- **Neighbor distance:** This is the maximum distance (center point to center point) for the neighborhood query.
- **Maximum number of neighbors:** This is the maximum amount of closest neighbors that are chosen by the neighborhood query.
- **Time horizon:** This is the minimal amount of time for which the velocities of the actors are safe with respect to others.

Tests were run on instances with 1000 actors with random starting locations and random goal locations in an empty 100 by 100 by 100 environment. All the random

points were chosen on a plane in the environment, so that every actor is initially on a collision course with the other actors. Every set of different parameters were tested on the same random seeds, so that the results can be easily compared. You can find an overview of the averages of all the results in Table 5.3.

Parameter	**Values**				
Number of agents	1000	1000	1000	1000	1000
Neighbor distance	15	20	20	20	20
Max. neighbors	10	10	20	20	30
Time horizon	10	10	10	20	10
Metric	**Results**				
Avg. time spent	100.412	100.413	100.441	100.843	100.381
Avg. dist. traveled	99.629	99.629	99.618	99.637	99.585
Avg. speed	0.933	0.933	0.931	0.934	0.933
Avg. number of collisions	0.016	0.016	0.006	0.006	0.004
Avg. total speed change	1.290	1.290	1.285	1.403	1.279
Avg. total degrees turned	172.358	172.363	169.128	190.527	164.945

Table 5.3: Overview of the results in the empty environment

Apparently there were a few collisions between the actors. Closer inspection showed that it were only slight penetrations that can easily be ignored, less than one thousandth of an actors diameter. The root cause appeared to be the precision that was lost due to the linear calculations. The range of the actor was chosen as the one used to calculate the velocity objects, so it was allowed that actors could go straight along each other. This value should be taken greater, when more realistic behavior is required.

The average amount of degrees turned is mainly due the fact that the actors spawned with random starting velocities. Due to these results, all future tests were done with the initial direction vector of each actor targeted towards their final goal so that we can have a better view of the turning effort required to go through the environment. For these tests it was mainly to compare the impact of the parameters on the results.

The changes in neighborhood distance had almost no impact on the result, this should have been expected since only the k nearest neighbors are taken into account. Yet, it should be noted that this value should be preferably large enough to avoid head-on collisions as soon as possible. The maximum neighbors, which are the ones chosen for the k-nearest-neighbors query, had primarily impact on the amount of turning effort required, it was also the only factor which influenced the (near-)collisions. The biggest impact on the effort, however, was the time horizon. If this one becomes too large, the amount of turning effort and the speed changes are increased significantly, this is because the possible areas that can be chosen are the most restricted this way.

The average amount of time spent for all actors was slightly above 100 seconds for all parameter sets. The turning and speed effort of the actors had very low influence on their speed and distance traveled.

The test was now repeated for the best parameters (distance: 20, max neighbors: 30, time horizon: 10) on a case where all 1000 actors were spawned randomly throughout the whole space, with their direction vectors initially aimed towards their goal. See Table 5.4 for an overview of the results.

Parameter	**Value**
Number of agents	1000
Neighbor distance	20
Max. neighbors	30
Time horizon	10
Metric	**Results**
Avg. time spent	131.868
Avg. dist. traveled	131.658
Avg. speed	0.937
Avg. number of collisions	0
Avg. total speed change	0.839
Avg. total degrees turned	32.745

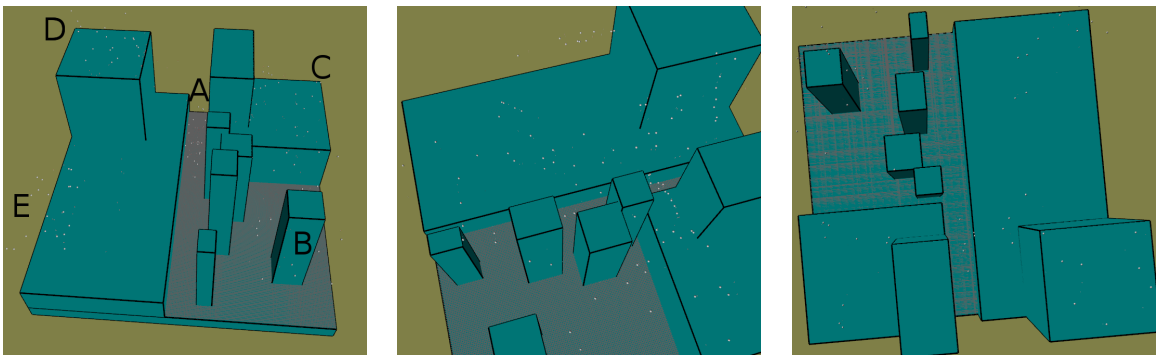
Table 5.4: Overview of the results in the total random test case of the empty environment.

5.3.2 Urban environment

Some screenshots of this environment can be seen in Figure 5.2. Since the generation of random actors throughout the whole area caused uninteresting behavior, we had chosen to designate some spawning areas in the environment. 20 actors were generated in each of them and all of them were given goal locations inside another spawning area.

Parameter	**Values**		
Total number of agents	100	100	100
Smoothing function	none	A	B
Metric	**Results**		
Avg. time spent by one agent	188.874	176.29	165.931
Avg. distance traveled by one agent	184.849	172.229	165.032
Avg. speed of one agent	0.937	0.929	0.937
Avg. number of collisions per agent	0.01	0.03	0.01
Avg. total speed change per agent	14.469	14.133	13.5534
Avg. total degrees turned per agent	678.466	355.25	280.47

Table 5.5: Overview of results for the urban environment



(a) (b) (c)

Figure 5.2: Screenshots of the urban environment. The letters A, B, C, D, E denote the spawn areas.

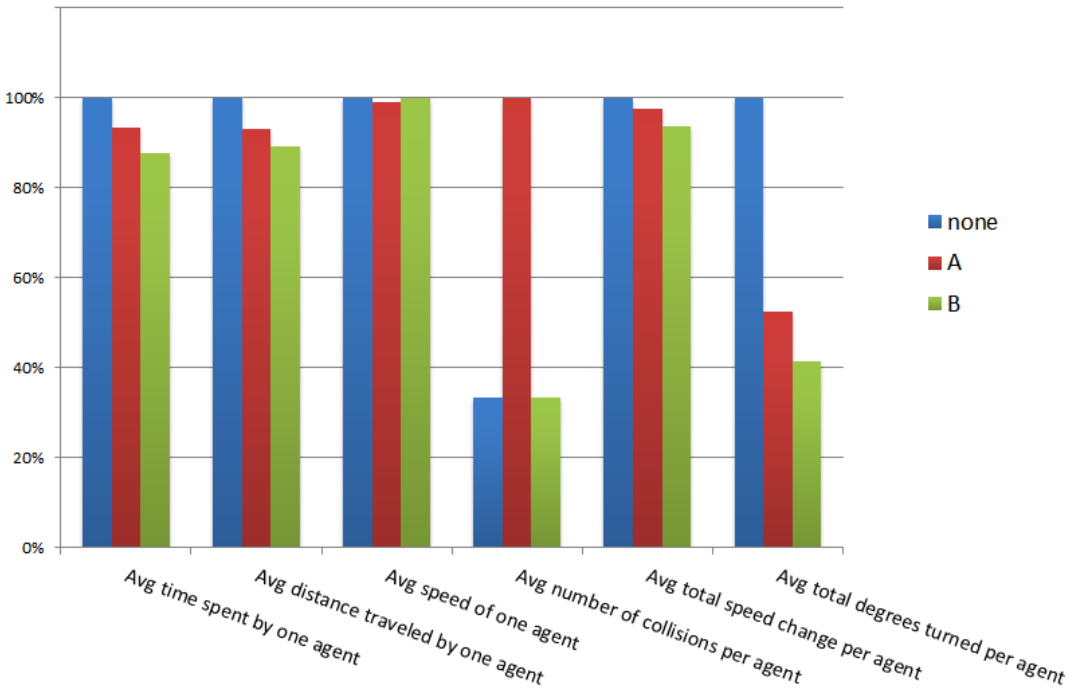
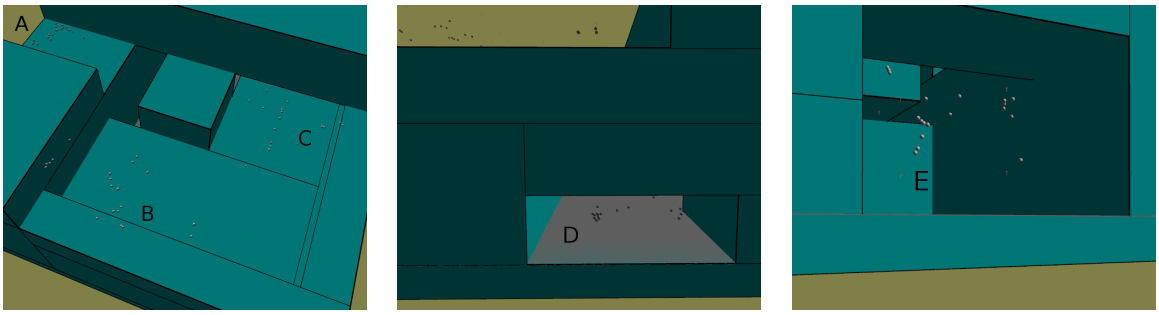


Figure 5.3: Overview of results for the urban environment



(a) (b) (c)

Figure 5.4: Screenshots of the cave environment. The letters A, B, C, D, E denote the spawn areas.

Two versions of our smoothing algorithm were created. The difference between smoothing function A and B was that B placed the resulting coordinates further away towards the center of the polygon when they became to close to corners. The results can be found in Table 5.5 and Figure 5.3.

Only a few collisions per 100 actors is quite low. Yet, it would be better if there were no collisions. Closer inspection into the cause of these collisions showed that it was due to collisions with corners that are close to borders of obstacles, namely when flying upwards along a building to a goal on its roof. The waypoint calculated by the navmesh which was still a bit too close to the wall of the building. Another, somewhat excepted, result of the unsmoothed path was that the turning effort is quite high, this is due to the fact that the last node of the path can be behind the goal itself when seen from its predecessor on the path. See for example Figure 2.3(b) for an example of this problem.

5.3.3 Cave environment

The same tests of the urban environment were repeated on the cave environment, which also received 5 spawning areas. Some screenshots of this environment can be seen in Figure 5.2. The results can be found in Table 5.6 and Figure 5.5. The influence of the difference between smoothing functions A and B is lower here, this is mainly due the fact that the navmesh had fewer polyhedra in this environments. It is interesting that there were no collisions the smoothing function wasn't used. The reason for this is that the actors started moving in queues inside the corridors. This is because they shared the same subgoals and their speed was adapted due to the local avoidance method we use. The smoothed versions had some collisions due to the same reasons as the ones that occurred in the urban environment, some final goals were too close to a corner so the adjusted paths were a bit too close to these.

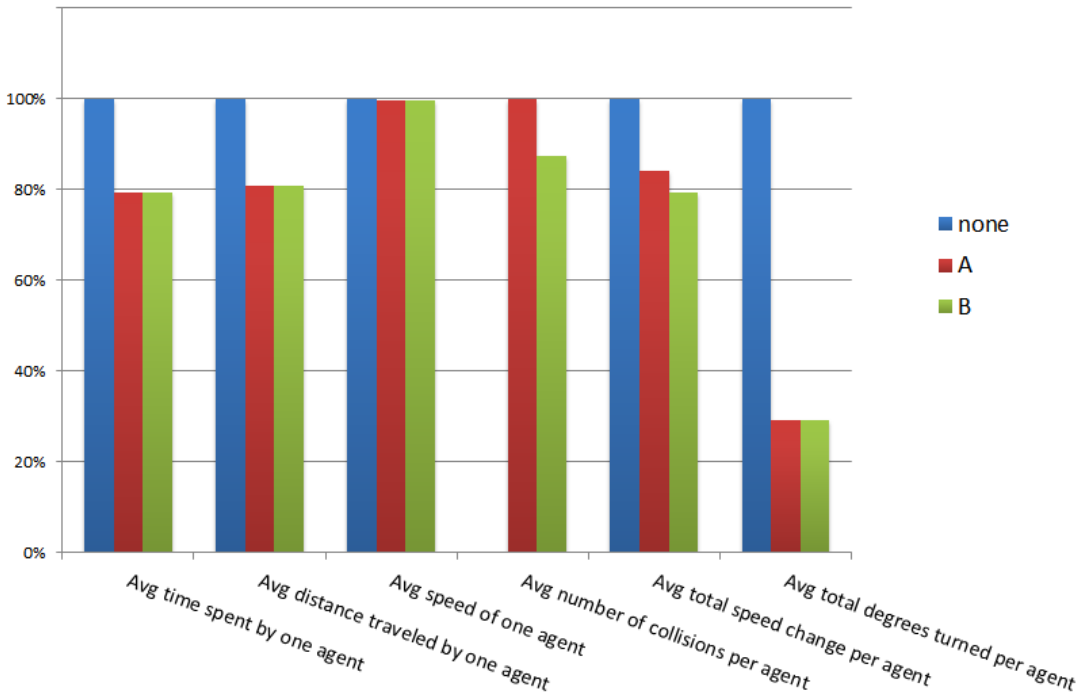


Figure 5.5: Overview of results for the cave environment

Parameter	**Values**		
Total number of agents	100	100	100
Smoothing function	none	A	B
Metric	**Results**		
Avg. time spent by one agent	172.934	137.132	137.235
Avg. distance traveled by one agent	167.982	135.795	135.795
Avg. speed of one agent	0.938	0.936	0.936
Avg. number of collisions per agent	0	0.08	0.07
Avg. total speed change per agent	7.57304	6.363	6.017
Avg. total degrees turned per agent	838.343	245.363	245.084

Table 5.6: Overview of results for the cave environment

Chapter 6

Conclusion

This thesis covered both global and local path planning for 3D navigation in virtual 3D environments. The best results can be achieved by combining these two path planning techniques for each actor.

We have chosen to use the A* algorithm with the Euclidean distance as a heuristic, since this is the most suitable approach for real-time path planning. However, this algorithm scales with the amount of nodes in the graph corresponding to the data structure used. The two candidate data structures were the voxel grid and the 3D navmesh we have defined in this work. The voxel grid achieves the best results when a fine grid is used; but a fine grid has too many nodes for the A* algorithm. Other neighborhood approaches can be used, but these can be quite time-consuming to use the A* algorithm or one of its variants on.

We have opted to use the 3D navmesh to model the navigable area. The idea behind this approach is to represent the navigable space of the environment as a mesh of non-overlapping convex polyhedra, in which each polyhedron shares a single face with each one of its neighbors. This is an extra data structure to use, so it requires some extra memory space and some preprocessing of the environment. The generation of this data structure depends on the method used. There are various ways to do this. The extra memory usage to store this data structure is offset by the significant reduction in the number of steps required to find a shortest path with the A* algorithm.

Our tests show that our proposed smoothing algorithms can be used to obtain useful results, while only requiring a minimal amount of extra operations. This smoothing algorithm can be easily extended to obtain better results. Finding these intersections with other obstacles and actors can be time-consuming when working with complex obstacles. But we can use bounding volumes to represent these actors and obstacles with simpler objects on which quicker calculations can be used to find sufficient (but not the required) conditions to see if an intersection might happen or not.

The local avoidance technique that we have chosen to use has good synergy with the result of our global path planner. It is also based on finding intersections with planes. Each actor adjusts its velocity at each time step in such a way that the actor has sufficient conditions so that the vector has a collision-free path over a certain time window, while still keeping this velocity as close as possible to its preferred one.

The path planning techniques we use are based on the calculations of finding intersections with planes in 3D spaces. This is in fact a simple linear calculation that can be solved quickly by a computer. Our tests show that these calculations may have some loss of precision in the result, but this loss of precision is so small (tinier than 1/100th of a pixel) that it can be ignored under most circumstances.

Our tests show some promising results, but the actual implementation still has room for improvement. The global and local path planning appeared to have good synergy with each other. Even some emerging crowd behavior was observed when examining the results.

Some ideas for future work

- Testing some more variants of our proposed path smoothing function
- Testing environments with more arbitrary forms of obstacles and actors (bounding objects can be used for this)
- Constraints on the behavior of the actors
- Heterogeneous actors
- Combination of flying actors / walking actors
- Dynamic environments

Appendix A

Mathematical background

This appendix provides an overview of some mathematical concepts that are relevant for this thesis.

A.1 Graph theory

This section gives a short overview of some topics within graph theory. The basic definitions of graphs are given in section A.1.1, next some ways to represent graphs are introduced (Sec. A.1.2).

Algorithms that find shortest paths in graphs are covered in chapter 3 (see Sec. 3.1).

A.1.1 Definitions

A *graph* is an ordered pair $G = (V, E)$ with a node set V and an edge set $E \subseteq V \times V$. Each edge $e \in E$ represents a connection between two nodes, these are the *end nodes* of the edge e. Two nodes that are connected by an edge are called *neighbors*. We name an edge a *loop* when both its end nodes are actually the same node. It is also possible that two nodes x and y are the end nodes of different edges, in this case we call them *parallel edges*.

A *directed graph* is a graph $G = (V, A)$ where the order of the end nodes are of importance. These edges, which are in this case usually called arcs, represent a one-way connection from its first end node to the other one. So an arc is an ordered pair $a_1 = (x, y)$ that models a connection from node x to node y, but not the other way around, we need another arc $a_2 = (y, x)$ for that. When the order of the end nodes is important, we have a *directed graph*. The edge is just a set in this case, so two edges $e_1 = x, y$ and $e_2 = y, x$ are in fact the same edge.

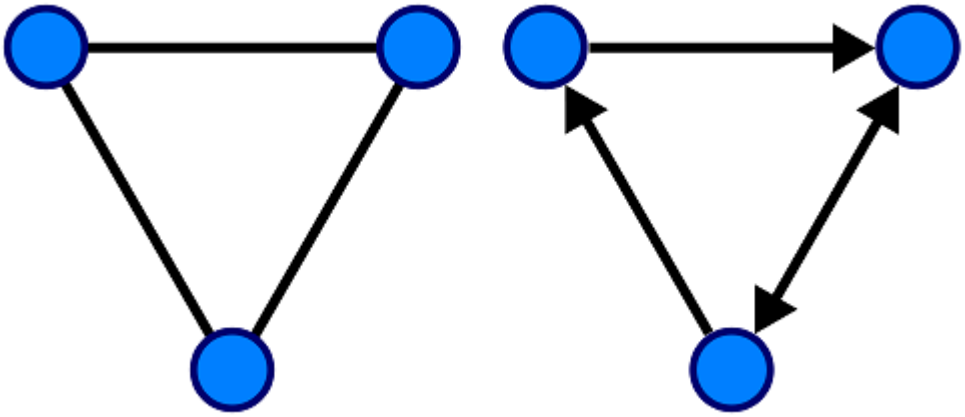


Figure A.1: left: *an undirected graph*, right: *a directed graph*

The edges of a graph could also have a number assigned to them, which can be used to model the cost of traversing its corresponding edge, we call that number the *weight* of the edge. These weights are often strict positive values and can be modeled through a weight function $w : E \to \mathbb{R}_0^+$.

A *path* P in G from a node u_1 to u_k is a sequence of arcs $(u_1, u_2), (u_2, u_3), ..., (u_{k-1}, u_k)$. The sequence of nodes $[u_1, u_2, ..., u_k]$ is a shorter notation for the same path when there are no parallel edges in the graph. The *length* $w(P)$ of a path P is the sum of all weights of the edges that belong to P. We call $P = [u_s, ..., u_g]$ a shortest path between u_s and u_g if there no other path P' from u_s to u_g such that $w(P') < w(P)$. When no weights are given, we assume that all edges have the default weight 1.

Within the context of pathfinding, the undirected graph is usually approached as a directed graph. All edges of the graph are approached as two arcs in opposite direction, with the same weights.

A.1.2 Representations

There are many ways to represent graphs. One of the most common ways is to visualize the nodes as circles and the edges as lines, or arrows for arcs, that connect those circles. Two edges in opposite directions between the same two nodes are usually combined as a bidirectional arrow. See figure A.1 for some examples.

Multiple options exist to store these graphs in a computer system: lists, polygons, grids, ...

A.2 Distance metrics

A *distance metric* on a set X is a function

$$d : X \times X \to \mathbb{R} \tag{A.1}$$

that satisfies the following four properties for all x, y, z in X:

Non-negativity

$$d(x, y) \geq 0 \tag{A.2}$$

Identity

$$d(x, y) = 0 \equiv x = y \tag{A.3}$$

Symmetry

$$d(x, y) = d(y, x) \tag{A.4}$$

Triangle inequality

$$d(x, z) \leq d(x, y) + d(y, z) \tag{A.5}$$

A.3 Complexity analysis

Complexity analysis provides useful theoretical estimates for the performances of algorithms and data structures.

A.3.1 Asymptotic dominance

Consider the two functions $f : \mathbb{N} \to \mathbb{R}$ and $f : \mathbb{N} \to \mathbb{R}$. We say that the function $g(n)$ asymptotically dominates $f(n)$ when there exists a positive integer k and a positive constant c such that:

$$n \geq k(f(n) \leq cg(n)), \forall n \in \mathbb{N} \tag{A.6}$$

In other words, ignoring some number of small cases and ignoring some constant factor c, $f(n)$ is bounded from above by $g(n)$. The symbol O is normally used to denote the asymptotic dominance relation, so $O(g(n))$ is the set of all functions that are asymptotically dominated by $g(n)$. Thus, if $g(n)$ asymptotically dominates $f(n)$, we can write:

$$f(n) \in O(g(n)). \tag{A.7}$$

This is usually read as: *"f is big-O of g"*, and it is often written $f(n) = O(g(n))$, although this statement is not literally correct since $O(g(n))$ is a set of functions, not a function [27].

Time Complexity

The complexity of the execution time for an algorithm A can be described easily using asymptotic dominance. We can describe the time required to execute A as a function of n, with n a measure for the size of P's input. We can say that A runs in time $O(g(n))$ iff $f(n) \in O(g(n))$.

Space Complexity

The complexity of the memory space used for a data structure D can be described easily using asymptotic dominance. We can describe the space required to store that the data in D as a function of n, with n a measure for the size of D's input. We can say that D runs in time $O(g(n))$ iff $f(n) \in O(g(n))$.

Bibliography

[1] John Amanatides, Andrew Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Proceedings of EUROGRAPHICS*, volume 87, pages 3–10, 1987.

[2] Edward Angel and Dave Shreiner. *Interactive computer graphics: A Top-Down Approach with Shader-Based OpenGL*. Pearson Education Limited, 2012.

[3] Srikanth Bandi and Daniel Thalmann. Space discretization for efficient human navigation. In *Computer Graphics Forum*, volume 17, pages 195–206. Wiley Online Library, 1998.

[4] Joseph Carsten, Dave Ferguson, and Anthony Stentz. 3d field d*: Improved path planning and replanning in three dimensions. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 3381–3386. IEEE, 2006.

[5] Mark De Berg. *Ray shooting, depth orders and hidden surface removal*, volume 703. Springer, 1993.

[6] Mark De Berg, Otfried Cheong, and Marc Van Kreveld. *Computational geometry: algorithms and applications*. Springer, 2008.

[7] Luca De Filippis, Giorgio Guglieri, and Fulvia Quagliotti. Path planning strategies for uavs in 3d environments. *Journal of Intelligent & Robotic Systems*, 65(1-4):247–264, 2012.

[8] Aljosha Demeulemeester, Charles-Frederik Hollemeersch, Pieter Mees, Bart Pieters, Peter Lambert, and Rik Van de Walle. Hybrid path planning for massive crowd simulation on the gpu. In *Motion in Games*, pages 304–315. Springer, 2011.

[9] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[10] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. *The International Journal of Robotics Research*, 17(7):760–772, 1998.

[11] Epic Games. Unreal engine. *URL: http://www.unrealtechnology.com/*, 2013.

[12] Roland Geraerts, Arno Kamphuis, Ioannis Karamouzas, and Mark Overmars. Using the corridor map method for path planning for a large number of characters. In *Motion in Games*, pages 11–22. Springer, 2008.

[13] Roland Geraerts and Mark H Overmars. The corridor map method: Real-time high-quality path planning. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 1023–1028. IEEE, 2007.

[14] D Hunter Hale and G Michael Youngblood. Full 3d spatial decomposition for the generation of navigation meshes. *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2009.

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

[16] Mark Joselli, Erick Baptista Passos, Marcelo Zamith, Esteban Clua, Anselmo Montenegro, and Bruno Feijó. A neighborhood grid data structure for massive 3d crowd simulation on gpu. In *Games and Digital Entertainment (SBGAMES), 2009 VIII Brazilian Symposium on*, pages 121–131. IEEE, 2009.

[17] Thomas Jund, Pierre Kraemer, and David Cazier. A unified structure for crowd simulation. *Computer Animation and Virtual Worlds*, 23(3-4):311–320, 2012.

[18] C. Kalani. Fixing pathfinding once and for all. *URL: http://www.ai-blog.net/archives/000152.html*, 2013.

[19] Samuli Laine and Tero Karras. Efficient sparse voxel octrees–analysis, extensions, and implementation. *NVIDIA Corporation*, 2, 2010.

[20] Tianlu Mao, Hao Jiang, Jian Li, Yanfeng Zhang, Shihong Xia, and Zhaoqi Wang. Parallelizing continuum crowds. In *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology*, pages 231–234. ACM, 2010.

[21] Alex Nash. Theta*: Any-angle path planning for smoother trajectories in continuous environments, 2010.

[22] Carol O Sullivan and John Dingliana. Real-time collision detection and response using sphere-trees. 1999.

[23] P Payeur. Improving robot path planning efficiency with probabilistic virtual environment models. In *Virtual Environments, Human-Computer Interfaces and Measurement Systems, 2004.(VECIMS). 2004 IEEE Symposium on*, pages 13–18. IEEE, 2004.

[24] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Pub. Co., Inc., Reading, MA, 1984.

[25] Craig Reynolds. Big fast crowds on ps3. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121. ACM, 2006.

[26] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21:25–34, 1987.

[27] Elaine Rich. *Automata, computability and complexity: Theory and applications*. Pearson Prentice Hall, 2008.

[28] Amit Shets. Amits game programming information. *URL: http://www-cs-students.stanford.edu/ amitp/gameprog.html*, 2013.

[29] Shawn Singh, Mubbasir Kapadia, Petros Faloutsos, and Glenn Reinman. An open framework for developing, evaluating, and sharing steering algorithms. In *Motion in Games*, pages 158–169. Springer, 2009.

[30] Shawn Singh, Mubbasir Kapadia, Petros Faloutsos, and Glenn Reinman. Steerbench: a benchmark suite for evaluating steering behaviors. *Computer Animation and Virtual Worlds*, 20(5-6):533–548, 2009.

[31] Jamie Snape and Dinesh Manocha. Navigating multiple simple-airplanes in 3d workspace. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3974–3980. IEEE, 2010.

[32] Greg Snook. Simplified 3d movement and pathfinding using navigation meshes. *Game Programming Gems*, 1:288–304, 2000.

[33] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 1160–1168. ACM, 2006.

[34] Jur Van Den Berg, Stephen J Guy, Ming Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. In *Robotics Research*, pages 3–19. Springer, 2011.

[35] Wouter G van Toll, Atlas F Cook, and Roland Geraerts. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds*, 23(6):535–546, 2012.

[36] Wouter G van Toll, Atlas F Cook IV, and Roland Geraerts. Multi-layered navigation meshes, 2011.

[37] Tomàs Vomàcka. Terrain representation for artificial human navigation. 2010.