3D Pathfinding
– Simulating Intelligent Navigation for Flying Creatures in Computer Games

Johan Stigsson

TRITA-NA-E03099
3D Pathfinding
– Simulating Intelligent Navigation for Flying Creatures in Computer Games

Johan Stigsson

TRITA-NA-E03099
3D Pathfinding — Simulating Intelligent Behaviour for Flying Creatures in Computer Games

Abstract

Pathfinding is a central part of the artificial intelligence in computer games and robotics. It deals with the concept of navigating and finding the best path between two points in some kind of environment, either virtual or real. The core of the pathfinding problem is to find a proper representation of the environment. For 2D, there are a number of strategies to solve this problem. For flying creatures moving in 3D, it gets a lot more complicated. This thesis mainly investigates two different representations for 3D pathfinding — the octtree and the visibility graph.

Due to the high demands in the game industry, the focus of the evaluation of these two representations has been on computation time and memory demands. To add realism, restrictions have been introduced in the pathfinding, such as penalty for making sharp turns.

3D Pathfinding — Simulering av intelligent navigering för flygande varelser i dataspel

Sammanfattning

Pathfinding är en central del av artificiell intelligens i dataspel och inom robotiken. Den behandlar hur en datorstyrad karaktär kan hitta den bästa vägen mellan två olika punkter i en värld som antingen kan vara virtuell eller verklig. Kärnan i hela problemet är att hitta en bra representation för denna värld. Det finns en rad olika strategier för att lösa problemet i två dimensioner, men för varelser som kan röra sig fritt i tre dimensioner (till exempel flygande varelser) blir det mycket svårare. I detta arbete behandlas i huvudsak två olika representationer för pathfinding i tre dimensioner — ”oktalträdet” (octtrees) och ”synlighetsgrafer” (visibility graphs).

För att ett spel ska vara ekonomiskt lönsamt och skilja bra krävs att det kan köras på så många olika typer av datorer som möjligt, från det absolut senaste till äldre modeller. Beräkningseffektivitet och minnesåtgärdning har därför varit nyckelfaktorer vid utvärderingarna av de olika lösningarna. I arbetet presenteras dessutom en del idéer för att få en mer realistisk navigering genom att till exempel ta hänsyn till svängradier.
Acknowledgements

In October 2002, I started my Master’s project at Starbreeze AB in Uppsala. Prior to that, I had spent a couple of weeks reading through some books and articles about pathfinding. In March 2003, I decided to call the quits and put together what I had accomplished. The 3D pathfinding problem is certainly not an easy task to solve and the six months of work had left me with two major prototypes of which none was optimal. There is certainly much more to be done in the search for the solution. Some ideas that I never had the opportunity to investigate are presented at the end of this thesis.

First of all, I would like to thank the great guys at Starbreeze AB, especially Anders Olsson (supervisor and AI mentor), Anders Backman (roommate and fountain of knowledge), Samuel Ranta-Eskola (for giving me a chance to prove myself) and Marco Ahlgren (for general support).

I would also like to thank Andreas Bodén for letting me borrow his laptop which gave me the opportunity to work on my thesis during the three hour trip back and forth to Uppsala every day.

Last, but not least, I would like to thank Lars Kjell Dahl, my supervisor at The Royal Institute of Technology, Stockholm, for his input on the development of the thesis.
# Contents

1 Introduction .................................................. 1

2 Artificial Intelligence ........................................ 4
   2.1 What is Artificial Intelligence? ........................... 4
   2.2 A Brief History of AI .................................... 5
   2.3 Artificial Intelligence in games .......................... 5
       2.3.1 Pathfinding ....................................... 6

3 Different Representations of the Scene ....................... 7
   3.1 Potential Fields ........................................ 9
   3.2 Cell Decomposition ..................................... 9
       3.2.1 Uniform Cells .................................... 10
       3.2.2 The Octtree ...................................... 10
   3.3 Skeletonisation ......................................... 11
       3.3.1 The Visibility Graph ............................. 12

4 The Search Algorithm ......................................... 13
   4.1 The A* Algorithm ....................................... 13

5 Implementation ................................................. 15
   5.1 The Environment ....................................... 15
   5.2 Implementing the Different Representations ............. 16
       5.2.1 Uniform Cells .................................... 17
       5.2.2 The Octtree ...................................... 17
       5.2.3 The Visibility Graph ............................. 19
   5.3 The Search Algorithm .................................. 25
       5.3.1 Modifying the A* Algorithm ....................... 25
       5.3.2 Adding Direction .................................. 25
       5.3.3 Adding Turn Cost ................................. 26
       5.3.4 Other Search Restrictions ......................... 26
       5.3.5 Searching in the Detailed Grid ................... 26
       5.3.6 Searching in the Octtree ......................... 28
       5.3.7 Searching in the Visibility Graph ................. 29

6 Results .......................................................... 30
   6.1 Uniform Cells .......................................... 30
   6.2 The Octtree ............................................. 30
   6.3 The Visibility Graph .................................... 32
   6.4 Searching ................................................ 37

7 Conclusions and Recommendations ............................ 42
   7.1 Further Improvements ................................... 43

References ......................................................... 45

A Appendix .......................................................... 47
1 Introduction

In the last couple of years the computer game industry has grown exponentially. The fast progress has indeed also affected the PC industry, demanding better graphics resources and faster machines. Better resources have given the game developers the opportunity not only to improve the quality of the graphics, but also to make the games appear more realistic overall by introducing rather complex physics and artificial intelligence. A central part of the artificial intelligence in games is pathfinding. Pathfinding deals with intelligent navigation in an environment — how a computer controlled character may get from point A to point B in a reasonably “intelligent” and efficient way, without colliding with any obstacles. This can be achieved in a number of different ways, all of which have some advantages and some disadvantages. For pathfinding in computer games, the key elements are speed and memory. To maintain a smooth and flowing gameplay the game constantly has to render at least 30 frames per second. For slower rendering speeds, the human eye starts to detect individual frames. This rendering criteria leaves a limited capacity left for other processes such as pathfinding. Also, with a limited amount of RAM (Random Access Memory) available, there is also a restriction to the amount of memory the pathfinding may use.

By only allowing ground based movement the pathfinding is (more or less) restricted into 2D. This type of pathfinding is a problem the game developers have tried to solve for years. Due to the complex nature of the problem with all the different cases that may appear, the different methods used in games today usually works but still have some flaws. In 3D, the problem gets even harder. Characters moving without restrictions (levitating or flying) may move in all three directions and thus expands the pathfinding problem into 3D. The search space easily grows beyond acceptable sizes which is devastating for the memory requirements and the search times. Therefore, there are no standard ways of solving this problem.

The purpose of this thesis has been to investigate, implement and evaluate some different ways of solving the 3D pathfinding problem to try to find a proper method that may be used for 3D pathfinding for any type of flying creature. Search times and memory demands have been the key factors for this evaluation. I have also made some attempts to add realism by the addition of different restrictions to the search. The project has been done at Starbreeze AB in Uppsala, Sweden and the results of my work may be used in any upcoming game from Starbreeze AB.

The core of the pathfinding problem is to choose a proper representation for the environment, the scene. I will briefly describe some of the most common methods used in computer games and robotics and explain some of these a bit more in detail. The ones that I have considered to be the most interesting are the uniform cell, the octree and the visibility graph representations. With all of these representations the major work are done offline (prior to rendering), at the compiling stage, leaving only the search to be performed online. In this work I have focused on getting the representations by an automated process. Of course, there are also ways to do this manually. The most common way to do this is to let the level designers complement their scenes with predetermined
paths, e.g., rails. Of course, this certainly affects the level of realism in the game.

In the uniform cell representation, the entire scene is divided into cells of equal size. Each one of these cells are evaluated and labelled as traversable or non-traversable for a character of a specified size. A cell is considered traversable if a character of this size may occupy any point inside the cell without interfering with any obstacles in the scene. The size of the cells determines how much information about the scene that is stored for the pathfinding - the smaller the cells, the more detailed the information about the scene. With large cells, detailed information such as small and narrow passages may be lost.

The octree is based on the same idea as the uniform cell representation. In this case, however, the scene gets divided into cells that are not uniform in size. Instead, the appearance of the octree depends on the structure of the scene. Large, open spaces are represented by large cells while more complex structures are handled by smaller cells. A predefined smallest cell size determines the level of detail for the octree. The octree is stored in a tree structure where it is actually the leaves that contain the information about the scene. As in the case of uniform cells, these leaf node cells are evaluated and labelled traversable or non-traversable.

The visibility graph differs from the previous cell representations. Instead, the scene is represented by a graph, a skeleton. This skeleton is based on the corners of the solid obstacles in the scene. A problem with this approach is that even the simplest object may consist of hundreds of polygons (triangles) and generate hundreds of corners. Therefore, the structure of the obstacles needs to be simplified. This can be done by using simplified bounding volumes or, as in my case, by letting the obstacles be represented by an octree. When the corners have been found, they are connected into a graph. As the name suggests, two corners are connected by an edge if they are visible from one another (a ray can pass through both of them without intersecting any obstacles).

In the end, by using any of these different representations, the pathfinding problem comes down to finding the shortest path from vertex A to vertex B in a weighted search graph. There are a number of different algorithms dealing with this problem. Some are simple and easy to implement and understand but not very good at finding optimal paths. Others are more complicated and might in some cases be slower but, in the end, always finds the best path. The only one I have decided to implement is the A* (A-Star) algorithm, the star of all search algorithms. The A* algorithm is the standard algorithm for dealing with the shortest path problem since it always finds the optimal path whenever there is one.

In chapter 2 the history of artificial intelligence and pathfinding is briefly summarized. There is also a short section that deals with AI and pathfinding in computer games. Chapter 3 describes the essence of the pathfinding problem and explains the different basic strategies and methods of solving this. Chapter 4 is devoted to the A* algorithm. My implementation of the different methods that I have considered to be the most interesting is described in chapter 5 and the results of my work is presented in chapter 6. In chapter 7 I present my con-
conclusions and talk about what may be done to refine and optimise the different solutions. For the average reader there is hardly any point in reading through the entire thesis. If you are uninitiated but interested in knowing more about pathfinding, I recommend reading through chapter 2 and 3. If you also intend to use pathfinding for an application of your own, you should read through chapter 4, 5 and 7 as well, though bear in mind that chapter 5 might be rather technical. If you are familiar with the pathfinding problem and only interested in the conclusions, you could stick to chapter 5 and 7. Those interested in the data that I base my conclusions on should also read chapter 6.
2 Artificial Intelligence

2.1 What is Artificial Intelligence?

Ever since the introduction of the first computers back in the early 1940s, the thought of thinking machines has been intriguing to mankind. There have been numerous books and movies in the science fiction area exploring this fascinating topic. Artificial Intelligence (AI) is a term that is loaded with subjective judgements. Because of this, a distinct definition of AI has been debated since the term’s inception and does not exist. It is a science closely associated with Computer Science but it also has links to Mathematics, Psychology, Cognition, Biology and Philosophy. The various definitions of AI can be divided into two main categories: those emphasizing on the reasoning aspect and those dealing with performance. Each one of these can be further divided into two subcategories: striving to behave like humans or focusing on rationality [3, 15, 17].

In the field of Artificial Intelligence you study the design of intelligent agents. An intelligent agent is a system that acts intelligent in an environment and makes decisions appropriate to achieve its goals depending on the circumstances. The agent is also flexible to change these goals due to changes in circumstances, it learns from experience and makes appropriate choices given perceptual limitations and finite computation. This may involve imitating characteristics from human intelligence and applying them as algorithms in a computer friendly way. An agent can either be a robot, a machine whose inputs are physical from some kind of perception or sensors, or a infobot (bot), an agent that acts in a computer generated environment [3, 15].

There is a number of different strategies to create intelligent agents. The most common one is rules-based techniques. This could either be Finite State Machines or Fuzzy Logic. Finite State Machines consists of a set of states that is usually represented by binary boolean logic (TRUE or FALSE). Based on the initial state and a set of inputs, a state transition function calculates a new state and a set of outputs. The term finite comes from the fact that there is only a finite number of states. Fuzzy Logic is a superset of traditional boolean logic and is extended beyond the binary true/false to include partially true expressions. Other ways to simulate intelligent behaviour is by using techniques that actually allows the agent to learn from previous experience. This can be achieved by either Neural Networks, Evolutionary Algorithms or Artificial Life (A-Life). Neural Networks consists of a number of simple components connected in a system that can produce output based on identification patterns in data. This technique is inspired by the anatomy of the nervous system. Evolutionary Algorithms are search procedures that incorporate the natural selection and natural genetics. The most common evolutionary algorithm is the Genetic Algorithm. These algorithms gradually evolves the agent to become smarter by either selection and recombination operators or by creating new individuals in the population. Artificial Life studies systems that show behaviour characteristics of natural living organisms. The higher level functionality is a result of interactions between lower level mechanisms often implemented by a variety of rules-based techniques or by techniques that allows learning [6, 24].
2.2 A Brief History of AI

The relatively short history of Artificial Intelligence starts in 1950, when A.M. Turing published the famous "Computing Machinery and Intelligence". However, the term Artificial Intelligence did not appear until six years later, in 1956, when John McCarthy organized a conference called "The Dartmouth Summer Research Project on Artificial Intelligence". The same year the trio Allen Newell, J.C. Shaw and Herbert Simon demonstrated the first running AI program The Logic Theorist at Carnegie Mellon University. The fast evolution of the computer machines made it possible for McCarthy to invent the LISP (LISt Processing) language at Massachusetts Institute of Technology (MIT) in 1958, a major breakthrough in the AI field. In 1964, Danny Bobrow showed that computers could learn to understand natural language well enough to solve algebra word problems, an effort that helped Joseph Weizenbaum to build ELIZA, an interactive program that could carry on a dialogue in English in any topic, in 1965. In 1969 researchers at Stanford Research Institute (SRI) built SHAKEY, a robot that combined locomotion, perception and problem solving and in 1972 another major breakthrough was made when Alain Colmerauer invented the logic programming language PROLOG. The researchers at the Stanford AI Lab, led by Hans Moravec, presented the Stanford Cart in 1979, the first computer-controlled, autonomous vehicle. In 1985 Aaron, a drawing program created by Harold Cohen, was demonstrated at the AAAI National Conference. During the 90s there were major advances in all areas of AI; machine learning, intelligent tutoring, case-based reasoning, multi-agent planning, scheduling, uncertain reasoning, data mining, natural landscape understanding and translation, vision, virtual reality and computer games. In 1997, IBM computer Deep Blue beat world champion Gary Kasparov in an informal chess match. Due to the information technology explosion in the late 90s, web crawlers and other AI-based information extraction programs became essential. In 2000, interactive toy robot pets became common and the Nomad robot explored remote regions of Antarctica, looking for meteorite samples [2, 22].

2.3 Artificial Intelligence in Games

Since the dawn of the video games in the 1970s, the computer game industry has gradually grown. In the last couple of years it has grown exponentially, with sales now surpassing the film industry’s gross revenue. The fast progress of the entertainment industry has pushed the hardware development in the PC industry, demanding better graphics resources and faster machines. As a result of this, the market has increased its demand for realism and complexity and, therefore, also better AI. As a matter of fact, today the quality of the AI is crucial in the success or failure of a game [6, 8]. Due to this fact and the fast development in the graphics hardware industry, game AI gets about five times as much CPU resources today, as it did five years ago. The main AI criteria of former years "as long as it does not effect the frame-rate", is certainly not valid any more and today every major game project has at least one person totally dedicated to AI. As a matter of fact, many of the recent years bestselling titles (e.g. Black and White and The Sims) are games where the artificial intelligence is the central part of the game [16, 23, 24].
The turn-based strategy games were the earliest pioneers in game AI. Due to its structure, the turn-based game can take whatever time it needs to figure out its next move before the actual execution. The amount of time can vary depending on computer speed and the game state. In a turn based game the player expect a gap in the gameplay and uses this time to plan his next move. However, the turn based games are almost history and have been replaced with real-time strategy games (for players who still prefer the turn based system this is sometimes given as an option). With the help of faster machines and better graphics hardware, more complicated AI have made its entrance in the first person shooter games.

After some experiments with nontraditional AI techniques during the last couple of years, such as neural networks and genetic algorithms, there is for the moment a strong tendency to stick to the more traditional approaches with rule-based techniques such as Finite State Machines (FSM) or Fuzzy State Machines (FuSM). The main reasons for this are simplicity and efficiency. The traditional approaches are well-known, works well enough and can be optimized for a cheap share of CPU cycles. There is, however, one exception to this. A-Life techniques has become more common and is widely used in flocking algorithms and for desire/satisfaction approaches to get more lifelike and unpredictable behaviours [24].

2.3.1 Pathfinding

A central part of the artificial intelligence in games is pathfinding, the way a bot navigates in its environment. Bad pathfinding can ruin any good game. With the level of realism in the games of today, the player expects that the computer controlled characters (either friendly or enemies) should behave somewhat "intelligent". The easiest way to ruin this is by implementing poor pathfinding. You do not have to be a devoted gamer to be familiar with the symptoms of poor pathfinding - computer controlled friendly followers get stuck behind corners and forces the player to go back to pick them up or the sight of charging enemies running head first into walls. In real life, intelligent creatures usually checks the environment, looking for an optimized way (due to the circumstances) prior to movement. A living creature does not aim straight at the target point and later on suddenly realizes that there suddenly is a obstacle in its way, forcing it to chose another way. This should also be the case for every computer controlled character (agent). In a strategy game, the player would probably raise an eyebrow if the computer sends its enemy forces straight through a nearly impenetrable swamp rather than choosing the faster and easier (however somewhat longer) way around it. Most of the pathfinding in today's games is done in 2D. There have been some attempts for 3D pathfinding but the topic is rather unexplored.
3 Different Representations of the Scene

In computer graphics, objects are rendered as polygons (triangles). The rendered polygons are handled as single entities (primitives). To represent objects in a game, the programmer has to group the polygons in some kind of way to be able to use commands like DrawBox(). This is usually done by an object oriented approach, having each class implement a specific render function. This approach also helps when it comes to collision detection between objects. Since an object consists of polygons and every polygon can be represented by a small section of a plane, collision between objects can be determined by checking intersections between polygons (the intersections of planes). This can be extremely tedious for complex objects and to simplify this process it is common to use bounding volumes. A bounding volume is an outlining volume that, to a certain extent, can be used as an approximation for the space taken by the object. Instead of checking intersection between all the polygons of the complicated objects, it may be enough to check for intersections between the simplified bounding volumes. Common bounding volumes are spheres, boxes and cylinders. Some examples of different bounding volumes can be seen in figure 1. The collisions test can be further optimized by the use of a hierarchical structure [1].

![Bounding Volumes](image)

Figure 1: Three different types of bounding volumes (bounding areas) in 2D.

To formulate the pathfinding problem in a more mathematical way we will need some basic definitions. Since the problem with pathfinding also is a common problem in robotics we can use the definitions from this field [9, 10].

If the state of the agent, \( A \), can be described by \( k \) real numbers \( q_1, \ldots, q_k \) (e.g., center space coordinates, rotational angles) it has \( k \) degrees of freedom. These \( k \) values can be considered a point in a \( k \)-dimensional space, the configuration state space, \( C \). If the scene contains any other objects, \( B_j \), the agent may collide with these. A position inside \( C \) for which \( A \cap B_j = \emptyset \), for all \( B_j \), is called a safe position. The set of states in \( C \) for which this does not happen is called the configuration space obstacle, \( O \). The set difference \( C - O \) is called free space, \( F \subseteq C \). The problem with this definition is that \( A \) is considered to be a point, an assumption that usually is not true. An easy way to reduce \( A \) down to the point representation is expand all other objects \( B_j \) with the volume taken by \( A \), which gives expanded geometries. If the scene contains agents of different sizes (bounding volumes) this is done by using the largest of these volumes. The less complicated the structure of this object's bounding volume is, the easier this operation will be. Rotational symmetric bounding volumes are preferred, but if that is not the case, one has to make sure that the bounding box's largest distance from its center point is used during this operation. \( O \) is now calculated
as the dilation (Minkowski sum) [5] of the space occupied by the objects, \( B_i \), by \( A \):

\[
B_i \oplus A = \{ z \in C : (A)_z \cap B_i \neq \emptyset \}
\]

where \((A)_z = \{ c \in C : c = a + z, a \in A, z \in C \}\)

This can be visualized as dragging the center of \( A \) along the perimeter of \( B_i \) (figure 2). The space enclosed by this operation is the expanded geometry [9, 11, 25].

![Figure 2: Expanded geometry constructed by dilation.](image)

Based on these definitions we are ready to formulate the pathfinding problem:

Let \( S \) be a scene containing the objects \( B_i \). Find a path for \( A \) from position \( s \) to position \( g \) such that \( A \) always belongs to \( F \). This is called a safe path. If there is more than one safe path between \( s \) and \( g \), the shortest of these is called the shortest safe path.

The essence of pathfinding is thus finding the “right” way from a starting point to a goal point. What is “right” and what is “wrong” depends on the circumstances. It can be the fastest, shortest or perhaps the safest path. Another problem that might occur is if that the representation may change due to changes in the environment (e.g., a door has opened). If this is the case, the environment is said to be dynamic, otherwise it is static. For agents in computer games, the environment is usually considered to be known, thus the agent has information about the entire scene. There are, however, approaches to deal with only partly known environments [19]. For real-time applications, like computer games, these approaches are not really suitable since they are way to slow.

There are four main approaches for pathfinding:

- **Online Algorithms** - This is the simplest way of pathfinding, the naive approach. The entire pathfinding procedure is done in real-time, online, with no prior calculations. The agent aims straight at its target and moves in that direction until it reaches the goal or collides with an obstacle. If it collides with an obstacle it has to choose another travelling direction, either from some kind of rule (e.g., tracing around the obstacle) or completely by random. After some movement in this new direction it once more aims for the goal and the process is repeated [20]. The major strength of this approach is that it is fast and also can be used in an unknown environment, but since this is a non realistic approach for simulating intelligent navigation, I will not go any further into it in this thesis.
• **Potential Fields** - The free space, $F$, is represented by a scalar function. Every object acts as the source of a potential field that is weakened by the distance from the object. The goal point has a very low potential and acts as a sink in the field. The negated gradient of the field exposes the agent to a force, pulling it towards the goal point.

• **Cell Decomposition** - $F$ is decomposed into cells and each cell is labelled traversable or non-traversable. When searching for a path, each cell may be represented as a node in a graph. If two cells are adjacent and traversable they will be connected by an edge in the graph, hence a node has edges to all adjacent cells. There are two main approaches for this kind of representation: **Uniform Cells (Uniform Grid in 2D)** or **Non-uniform cells (e.g., Octree (Quadtree in 2D)).**

• **Skeletonisation** - The free space is represented by a network of 1D curves, a skeleton. This skeleton can be directly transferred into a graph that can be searched by a search algorithm. Three examples of skeletonisation are **Voronoi diagrams, Roadmaps and Visibility Graphs.**

### 3.1 Potential Fields

The essence of the potential field method is to describe the free space with a scalar field. The objects in the scene, $B_i$, are sources for the potential field and the goal point, $g$, acts as a field sink. The fields are ideally smooth without any local minima and grows to infinity near the objects. The attractive potential from the sink is typically depending on the square of the distance. The agent is considered to be a particle (without any extension) moving around in the scene influenced by the forces due to the field. These forces are equal to the negated gradient of the field. In each step the agent is moved according to the net force acting on it. With simple bounding volumes, potential fields are rather easy to implement and can be stored in a fairly compact way by just representing each object by a potential field (can be stored in the objects class). The insertion of the starting point and the goal point is easy and fast. The disadvantages with this representation is that it tends to generate a lot of calculations (vector additions) for large and complex environments. Even if the objects potential fields do not have any local minima, the total field of all of these may have one. This may result in a trapped agent somewhere in the middle of nowhere. Also, when moving through the potential field, the path will have a tendency to avoid moving close to objects, a behaviour that sometimes is wanted (e.g., moving in cover, out of the enemy's sight) [7, 9].

### 3.2 Cell Decomposition

Cell Decomposition is a reliable way of pathfinding. The major strength of cell decomposition is that the representation can be generated offline when compiling the scene [9]. With the expanded geometries discussed earlier, collision detection becomes redundant. This is due to the fact that a cell is only considered traversable if all its enclosing points are traversable. The drawback is that this representation has a tendency to become rather big and demands a lot of memory, especially for large environments.
3.2.1 Uniform Cells

The most common type of cell decomposition in 2D is the uniform sized grid (figure 3). This can either be a rectangular shaped grid or a hexagonal grid. Extending the rectangular shaped grid to three dimensions gives us uniform cells. The hexagonal grid is not possible to extend to a suitable 3D representation without having some void in between the cells. For simplicity, I will consider the rectangular grid to be squares and the cells to be cubes. This, however, is not a restriction in any way.

The scene is divided into equal cells. The movement from one cell to another can be restricted in two ways. Either diagonal movement is allowed (26 possible directions for movement, in 2D this narrows down to 8) or not (6 possible directions, 4 in 2D). Diagonal movement may cause intersection with obstacles. If a cell contains part of an objects volume, it is considered to be a non-traversable cell and is removed from the search space. If a small cell is used, the representation will contain detailed information about the scene, but if a larger cell size is used some of the fine detailed information will be lost. Thus, the detail of the representation is depending on the cell size.

![Figure 3: Uniform Cells.](image)

The major drawback with the uniform cell representation is the large amount of memory it uses, especially in large scenes with small cell sizes. The pathfinding also becomes very inefficient since there is a huge amount of search nodes to expand, even during short searches. Also, the resulting path from a graph search may appear edgy and robotic and often needs some kind of post-search processing for a smoother appearance.

3.2.2 The Octree

A more compact representation of the scene in 2D is the quadtree (figure 4). Expanding this representation to 3D gives an octree (figure 5). The octree also consists of cells. These, however, are not uniform in size. The octree is generated by recursively subdivision of the scene, starting with the entire S. In
the first step, the entire scene is divided into eight equally sized cells \((2 \times 2 \times 2)\). Each one of these are further explored. If the cell contains part of an object it is divided into eight new equally sized cells. If a cell is entirely enclosed in an object or does not contain any part of an object, it is left as it is. This step is repeated until a predefined minimum size has been reached. As in the case of the uniform cells, each of these cells are considered traversable or non-traversable. The reason why this approach should be preferred instead of the uniform cell representation is the more compact representation. This becomes especially evident in large, vast scenes.

![Quadtree](image)

**Figure 4: Quadtree.**

![Octree](image)

**Figure 5: Octree.**

### 3.3 Skeletonisation

Instead of dividing the scene into discrete entities of space, skeletonisation techniques represents the space with a one-dimensional network, a skeleton. As in the case of cell decomposition, most of the job is done offline and with the use of expanded geometries collision detection becomes unnecessary. This structure is generally simpler and more compressed than the cell decomposition representations [9].
3.3.1 The Visibility Graph

A visibility graph is a very compact representation that evolves from the corners of the objects (figure 6). Every corner is represented with a vertex in a graph. If a corner, \( c_i \), is visible from another corner, \( c_j \), their corresponding vertices are connected with an edge in the graph. A corner is considered visible from another corner if the straight line between them does not intersect with any objects. The graph obtained after this process is a static graph that can be reused for many searches. Although, prior to the graph search, it has to be modified by the insertion of a start and end node. This operation must be done online which can be very demanding in terms of computational power.

An optimal path always makes its turns around edges, never outward from an edge. This fact implies that visibility graphs are certain to give the shortest path in 2D if a proper search algorithm is used (see section 5.2.3). Since the visibility graph is based upon corners and not edges, this is usually not the case in 3D.

![Visibility graph](image)

Figure 6: Visibility graph. Partly optimized with concave corners removed.
4 The Search Algorithm

In the previous chapter we saw that the search for an optimal path, if we use the cell decomposition or a skeletonisation method, all comes down to finding the shortest path in a weighted graph. Given a graph \( G = (V, E) \) with a weight function \( w(e) \), the problem is to find the shortest path from a starting vertex \( v_{\text{start}} \) to a goal vertex \( v_{\text{goal}} \). To solve this problem we need a search algorithm. The algorithm will start on the initial state, \( v_{\text{start}} \), investigate what states (vertices) that can be reached from this and further explore these according to some specific strategy. It may be helpful to visualize the search process as if the search algorithms builds a search tree. The initial state corresponds to the root node and the states that can be reached from this state becomes its children in the search tree. In this starting phase all the search algorithms are the same. The differences becomes evident when it comes to further exploration of the graph (expanding the search tree) [17].

4.1 The A* Algorithm

There is a number of different search algorithms. Some are simpler, and easy to implement and often accurate enough for their purpose while others are rather complicated but guarantees to find an optimal way whenever there is one. Examples of simpler algorithms are the Depth-First Search and the Breadth-First Search [4]. The most effective that is guaranteed to find an optimal path whenever there is one is the A* algorithm [12, 13].

The classical algorithm to use for finding a minimum-cost path in a graph is the A* algorithm (A-Star). The A* algorithm is guaranteed to find a path of minimum cost whenever there is one, otherwise it will return failure. A detailed pseudo C-style code for the A* is presented below (Algorithm 1). The algorithm iteratively explores \( G \) by following paths originating from \( v_{\text{start}} \). Whenever a node \( v_{\text{current}} \) is visited, the previous iterations has stored a path from \( v_{\text{start}} \) to \( v_{\text{current}} \), but the algorithm only stores the path of minimum cost of the paths found so far. For each node a cost function is assigned. This function is an estimate of the cost of the minimum path from \( v_{\text{start}} \) to \( v_{\text{goal}} \), that passes through \( v_{\text{current}} \). This cost function is:

\[
f(v_{\text{current}}) = g(v_{\text{current}}) + h(v_{\text{current}})
\]

where

- \( g(v_{\text{current}}) \) is the cost of the path from \( v_{\text{start}} \) to \( v_{\text{current}} \)
- \( h(v_{\text{current}}) \) is the heuristic estimate of the minimum cost path from \( v_{\text{current}} \) to \( v_{\text{goal}} \)

These costs could for example be the euclidian distance between the points.

The heuristic function, \( h(v_{\text{current}}) \), is admissible if it satisfies the condition:

\[
0 \leq h(v_i) \leq h^*(v_i), \forall v_i
\]

where \( h^*(v_i) \) is the minimum-cost path between \( v_i \) and \( v_{\text{end}} \).
Under the condition that the heuristic function is admissible, the algorithm is
guaranteed to find the shortest path whenever there is one.

**Algorithm 1: A*Search**

```
LET S = the start node
LET E = the end node
ASSIGN g and h values to S
WHILE the Open list contains any nodes
  LET B = the best node from the Open list
  IF B = E
    RETURN PATH FOUND
  FOR EACH adjacent node N to B
    ASSIGN g and h values to N
    IF N is in the Open or the Closed list
      IF the new path has a lower f-value
        UPDATE the path and the g-value
    ELSE
      ADD N to the Open list
    ADD B to the Closed list
  IF the Open list is empty
    RETURN FAILURE
```

If the graph contains \( n \) nodes the length of the open list is \( O(n) \). With a proper
heuristic function, each node can only be inserted into the open list once. If the
open list is not sorted, insertion in the list takes \( O(1) \) operations and removal of
the element with the lowest cost function value requires \( O(n) \). For every node
popped from the open list there is an expansion of its children. In the worst case
there are \( O(n) \) children. Thus, the time complexity of the A* with an unsorted
list is \( O(n^3) \). On the other hand, if the open list is sorted, insertion and removal
are done with \( O(\log n) \) operations. The complexity for the A* algorithm with a
sorted list is therefore \( O(n^2 \log n) \).

A variant of the A* algorithm, the dynamic D*, may be useful for pathfinding
when the scene only is partly known and the agent gradually receives information
about it [19].
5 Implementation

All of my implementations have been integrated into the Starbreeze engine (SBZ), as featured in the game Enclave released on XBOX and PC by Starbreeze Studios.

5.1 The Environment

The small test scene is a rather typical environment from a computer game, however somewhat small, created with the intention of covering as many cases as possible that may appear during pathfinding. The scene contains some ordinary obstacles such as trees, narrow passages, corners and walls that (in the case of poor pathfinding) possibly can trap the agent. The test scene has been developed in OGIER, the level editor designed to work with the Starbreeze engine. Modified screenshots of the test scene can be seen in figures 7–9. Some tests have also been performed on one of the levels from the Starbreeze game Enclave. Due to long generation times when creating the octtree and the visibility graph, I have only used this to validate my methods in a more realistic game environment.

Figure 7: XY-view of the test scene.
5.2 Implementing the Different Representations

The different representations mentioned in chapter 3 are only some of the different techniques that have been used in the fields of robotics and computer games. In this thesis, I have focused on the uniform cell representation, the octree representation and the visibility graph representation. Another interesting technique that would be worth further investigation is the convex polygon/polyhedron representation discussed in [18, 21]. This could probably be adapted into 3D by using some sort of convex volumes. The general consideration to keep in mind when creating the search graph is that more nodes usually gives a better (shorter) and smoother path but also a more complicated and slower search.
5.2.1 Uniform Cells

A crucial decision for the uniform cell implementation is to choose a proper cell size. As mentioned earlier, smaller cells are better if small details in the scene are important. Small cells give a smoother path and a better approximation of the shortest path, but also result in a larger search space and therefore, longer searches. With large cells the detailed information about the scene is lost. The search path becomes “edgy” and tends to go in zigzag instead of going in straight lines. In any case it might be a good idea to smooth the path by doing some post search path manipulation. To maintain detailed information about the scene it is wise to choose a cell size smaller than the width of a typical character (agent). In my implementation the cell size is about one third of a typical agents (human sized) width, thus a humanoid creature can be fitted into $3 \times 3 \times 8$ cells. This cell size has been used since the SBZ engine already stores information about the world in a uniform grid of this size. All of the cells are labelled as traversable or non traversable.

5.2.2 The Octtree

As in the case of uniform cells, the minimum octtree cell size determines the level of detail for the pathfinding. A smaller minimum cell size also gives a larger octtree. When generating the octtree, one also has to consider that a octtree cell only is considered traversable for a specific agents size if all its enclosing points are considered traversable for this agent. All of the octtree cells are $2^n$ times larger than the minimum octtree cell size ($n$ is an integer), so the first problem is to decide the size of the root node. This can be done by checking the size of the scene’s bounding box. The root node has to be $2^n$ times the minimum cell size. At first, for each dimension, we check how many times the minimum octtree cells can be fitted into the scenes bounding box. This gives three integers. We choose the largest of these and find the smallest integer $n$ such that $2^n$ is larger than this value. This gives the size of the root. The root cell is then recursively divided into cells with the minimum size. These cells are evaluated and labelled as OPEN or BLOKED. If a node’s all eight children are of the same type, the children are deleted and the parent node becomes a leaf node with the same label as its children. Non leaf cells are labelled MIXED. As mentioned before, the SBZ engine already stores a uniform grid representation of the world and this fact has been very useful in terms of generating the octtree. The algorithm for this procedure is described in pseudo code below (Algorithm 2).
Algorithm 2: DivideAndEvaluate

LET C = THE CURRENT OCTTREE CELL
LET S = THE MINIMUM OCTTREE CELL SIZE
IF THE SIZE OF C > S
    CREATE 8 NEW CELLS
    FOR EACH OF THESE CELLS N
        DivideAndEvaluate(N)
    IF ALL OF THE NEW CELLS ARE CLOSED
        SET THE STATUS OF C TO CLOSED
        DELETE THE NEW CELLS
    ELSE IF ALL OF THE NEW CELLS ARE OPEN
        SET THE STATUS OF C TO OPEN
        DELETE THE NEW CELLS
    ELSE
        SET THE STATUS OF C TO MIXED
    END IF
    Evaluate the status of C
END IF

The octtree is then generated by calling the function with the root node as parameter.

When the octtree has been generated, each cell may be linked to its neighbours. Two octtree cells, c_i and c_j, are considered neighbours if and only if one of c_i's wall is a subset of one of c_j's walls or vice versa.

This can be achieved by the following algorithm:

Algorithm 3: CheckPotentialNeighbour

LET C = THE CURRENT OCTTREE CELL
LET P = THE POTENTIAL NEIGHBOUR
IF P IS A LEAF NODE
    ADD N TO C'S NEIGHBOUR LIST AND VICE VERSA
ELSE IF P'S STATUS = MIXED
    FOR EACH OF P'S CHILD NODES N
        IF N AND C SHARE A WALL SUBSET
            CheckPotentialNeighbour(C, N)
    END IF
END IF

The neighbours are then added to an octtree cell by calling the function above with the current cell and the root node as parameters. The algorithm can of course also be used for online determination of an octtree cell's neighbours.

Insertion of Start and End Position

When inserting the start and end position, there might be the case when one (or both) of these is part of a CLOSED octtree cell. To execute the A* search in the octtree, we first need to find a path out of the CLOSED octtree cells to OPEN cells. This can be achieved by first searching in the detailed grid (towards the corresponding start or end point) until the current search position is part of an OPEN cell. Algorithm 4 describes this process.
Algorithm 4: InsertOctreeStartAndEndNode

\[
\begin{align*}
&\text{LET } s = \text{THE START POSITION} \\
&\text{LET } e = \text{THE END POSITION} \\
&\text{LET } \text{S} = \text{THE OCTREE LEAF THAT CONTAINS } s \\
&\text{LET } \text{E} = \text{THE OCTREE LEAF THAT CONTAINS } e \\
&\text{IF E'S STATUS = BLOCKED} \\
&\quad \text{GRID SEARCH FROM } e \text{ TO } s \text{ UNTIL AN OPEN NODE IS FOUND} \\
&\quad \text{IF THE SEARCH ENDS IN FAILURE} \\
&\quad \quad \text{return FAILURE} \\
&\quad \text{ELSE IF THE SEARCH ENDS IN } s \\
&\quad \quad \text{return PATHFOUND} \\
&\quad \text{ELSE} \\
&\quad \quad \text{LET } \text{E} = \text{THE FOUND OPEN OCTREE CELL} \\
&\quad \text{DO THE SAME FOR } S \\
\end{align*}
\]

Note: It is better to start with the end node since it is more likely that the end node will be situated in a position that is impossible for the character to reach.

5.2.3 The Visibility Graph

In computer games of today, objects and obstacles almost always tend to have very complicated structures. Even the simplest stone may contain hundreds of polygons. This fact makes it rather tedious to create the visibility graph based on the polygon structure of the objects and some simplification is necessary. Complex outer structures (e.g., walls in a cave), also need to be simplified. This could be achieved by storing a simple geometry for each type of object (e.g., a bounding box) but this would be a very tedious and memory inefficient way of doing it. In my case I have decided to generate an octree to represent the objects and obstacles in the scene. This octree is also stored for further use to simplify and speed up the insertion of the start and end points.

Getting the Corners

Not all corners are relevant for creating the visibility graph. There are some corners through which a shortest path will never pass. In a 2D visibility graph there are only two kinds of corners: concave ("inner") corners and convex ("outer") corners. Here, the concave corners, are redundant. The case of ending a move in a corner like this will be taken care of by the endpoint. In fact, there will never be a shortest path from one point to the other passing through a concave corner. In 3D it gets a lot more complicated.

For each closed octree cell we have to check every corner (8 corners) to see if this might be an interesting corner. If this is the case, we store the corner for further processing, otherwise we leave it. So how can we check if the corner is interesting or not? Well, if we know all the interesting types possible and then it is just a simple case of checking if the current corner is one of these cases. To classify a corner we can let the current corner be a corner point in a small "evaluation cube" pointing outwards from the current octree cell (figure 10). With the help of this cube, the corner can be evaluated by checking each of the evaluation cubes corners to see if they are contained in a open or closed octree
cell. If this evaluation is done systematically, the evaluation can be represented by a 7-digit binary number (one bit for each corner of the evaluation cube). This binary numbers corresponding base-10-integer can then be used in a lookup table to see if the current corner should be further processed.

![Octree cell with evaluation cube](image)

**Figure 10**: Octree cell with evaluation cube.

In my work I have considered exactly the half of these 128 cases ($2^7$) to be interesting. Each one of these 64 cases have a simple geometric property in common — they have at least three evaluation corners situated in open octree cells and these should be forming a "V-formation" (figure 11). This "V-formation" corresponds to a possible path around a corner. Of course, this formation also appears with ordinary edges, but these have to be sorted out as irrelevant cases. For a continuous space a visibility graph node can be placed in the current corner, but in the case of a discrete space (grid space) the current corner may have to be split into as many as six visibility graph nodes. This split can be done at the same time as the evaluation.

![Various configurations with V-formation](image)

**Figure 11**: Some different configurations containing V-formations. The top left figure shows a basic V-formation. The darker boxes are situated in closed octree cells.
Connecting the Nodes

When the interesting corners have been found and each one of these are represented by one or more visibility graph nodes, the nodes need to be connected into a visibility graph. The process is simple, although rather computational (see Algorithm 5). For each node a ray is sent to every other node. If the ray reaches its destination node, the nodes are connected with an edge. If the ray intersects with a closed octree cell (or some other expanded geometry if that is used), the nodes are considered not visible from each other and are left as they are. This pass is done for all the nodes.

```
Algorithm 5: CreateEdges

FOR each node A
  FOR each node B
    IF no edge connects A and B
    IF B is visible from A
      Connect A and B with an edge
```

Culling Redundant Edges

Depending on the scene for which the visibility graph has been generated, the procedure described above usually results in a lot of edges. However, it contains a lot of redundant edges that can be removed without losing any important information. If we consider the shortest path between two points in 2D, the changes in direction will always be done around corners. There will never be a case where there is a shortest path with a turn directed outwards from a corner (see figure 12). This is also the case in 3D.

![Figure 12: The shortest path in 2D.](image)

Consider a corner that has been evaluated as an interesting corner. There is a number of the obstacle’s planes intersecting in this corner. The planes that intersect in the corner span a zone in space, the redundant zone (figure 13). According to the criteria stated above, all edges in this zone are redundant since they will never be part of a shortest path. Therefore, they may be removed from the graph without losing any important information. If an edge is considered redundant from one point of view but important from another, it can be removed anyway [9, 26].
In the general case, the calculation of the redundant zone might be a computational process since it is calculated from the obstacles planes. However, in the case of cubic geometries (such as the octree cells) this can be accomplished in a simpler way by temporary storing a "normal" for each corner. Pseudocode for this process is presented in algorithm 6.

```
Algorithm 6: CULLZONEEDGES

FOR each node A
  FOR each node B connected to A
    LET v = the vector from A to B
    LET O = the obstacle associated with A
    LET P = the set of planes of O intersecting A
    IF the scalar product of all of P's normals with v < 0
      DELETE the edge between A and B
```

Alternate Path Culling

The redundant edge culling described above usually removes a lot of edges but sometimes this may not be enough for the purpose. One way to remove even more edges is to check for alternate paths between nodes. An edge between two nodes may be removed if there exists an alternate path between these that fulfill some kind of criteria (see figure 14). For example, that the alternate path may not be longer than a certain factor times the direct path length. A extension to this could be that the change in direction for any pair of the included edges should not be larger than a certain angle. These two parameters can be set to control the amount of edge removal. Prior to this edge removal the edges need to be sorted according to their length, shortest first. By doing this, we are sure that an alternate path always fulfill our predefined criteria. If we had done the opposite, longest edges first, edges in an alternate path may be culled and replaced by other paths, thus no longer fulfilling the length criteria. The search for alternate paths is done with the A* algorithm (without allowing the direct path between the two nodes). In contrast to the previous way of culling edges, by using the redundant zone, this method actually removes important information about the scene and should only be used if there are too many edges after the zone culling.
Figure 14: An alternate path in 2D. Since both the angles are less than a specified critical angle and the total length of the alternate path is less than a factor times the current edge (dashed), the edge may be removed.

Other Culling Options

Depending on the specific situation, there may be other relevant culling techniques that can be used. One example is if no strict vertical movement is allowed. In this case, there might be a good idea to remove all of these edges prior to the alternate way culling.

Insertion of Start and End Node

As mentioned before, the different procedures mentioned above can be done offline. The only thing that has to be done online is the insertion of the start and end node. The faster this is done, the faster the search can start. A fast insertion is therefore crucial for the efficiency of the pathfinding. The naive way of doing the insertion is to use the algorithm described earlier for connecting the nodes. However, this gets rather computational for scenes containing many nodes. Therefore, we need a faster way. Perhaps it is not so important to try connecting the start and end node to all other nodes. If we accept some minor deviation from an optimal path we can restrict this test and only try to connect the start and end node to some of the nodes. This is not a huge setback since we have already agreed that the use of a visibility graph only can guaranty optimal paths in 2D and not in 3D. This gives us an idea to concentrate on the nodes in a close vicinity to the start and end node. But how can we find these nodes? If we store the octree which the visibility graph was built upon, we can use this to do a fast check and find these nodes. By doing this, the memory advantage of using the visibility graph is lost, but since its structure is so compact it may still be an alternative if we gain large improvements in search times. However, the insertion is not a simple task since there is a lot of different cases possible. Algorithm 7 describes the insertion.
**Algorithm 7:** InsertVisibilityGraphStartAndEndNode

Let s = the start position
Let e = the end position
Let S = the octtree leaf that contains s
Let E = the octtree leaf that contains e

If E's status = BLOCKED
    Grid search from e to s until an OPEN node is found
    If the search ends in FAILURE
        Return FAILURE
    Else if the search ends in s
        Return PathFound
    Else
        Let E = the found OPEN octtree cell
        Let e = the latest help node
        Do the same for s
        If e is visible from s
            Connect s and e with an edge
            Return PathFound
        Else
            If S contains any other visibility graph node besides s
                Connect s with these
            Else
                Octtree search from s to E until a cell with VC-nodes is found
                If the octtree cell returned by the search contains any VC-nodes
                    Let S = this cell
                    Make a path from s to the closest point in this cell p
                    Let s = p
                Else if the octtree cell returned by the search = E
                    Return PathFound
                Else
                    Return FAILURE
            End If
        End If
    End If
End If

The algorithm described above sometimes results in paths that contain help nodes that link the start and end node to the rest of the graph (see figure 15). Since this help node path is single linked, a search from the start node to the end node is redundant and the search needs only to be done between the nodes that connect the help node paths to the visibility graph.

![Figure 15: Generated path with added help node (unfilled).](image-url)
5.3 The Search Algorithm

The search algorithm is based on the A* algorithm, where the nodes can be either grid cells, octree cells or visibility graph nodes. The algorithm is described in detail in chapter 4. In my implementation I have used sorted lists for the open and closed list. When doing a search during gameplay, the search process needs to be divided to not affect the frame rate too much. Otherwise, there would be cases when the frame rate would drop way below the accepted minimum of 30 fps (frames per second). For example, if the desired lowest frame rate is 30 fps, each game tick (cycle) is about 33 ms long. Besides the pathfinding, the game engine has to do all the rendering, handle inputs and take care of the physics as well. If the AI is given about 15 % of the CPU power, the search time during each game tick may not exceed 5 ms. One way to do this is to put a limit to the number of search nodes the algorithm may expand each frame.

5.3.1 Modifying the A* Algorithm

When nodes are expanded during the A* search their children are inserted in a sorted list. Nodes with a low total cost are inserted in the beginning of the list while elements with high total cost are inserted at the end. As explained in section 4.1, the cost function, $f(v_{current})$ assigned to each search node $v_{current}$ is the sum of the minimum cost of the path from $v_{start}$ to $v_{current}$, $g(v_{current})$, and the heuristic estimate of the minimum cost of the path from $v_{current}$ to $v_{goals}$, $h(v_{current})$. The cost function can be modified to speed up the search. This is done by adding weights to the different terms.

$$f(v_{current}) = w_g \cdot g(v_{current}) + w_h \cdot h(v_{current})$$

(4)

For the shortest path, both of these weight factors should be equal to one, but depending on the situation the weight can be shifted towards either one of the terms. If the cost weight, $w_g$, is set higher than the heuristic weight, $w_h$, the algorithm tends to be more of a Depth-First-Search. With the heuristic weight is set to zero it becomes Dijkstra’s algorithm. If the heuristic weight is set higher than the cost weight the algorithm gets greedier. When the cost weight is set to zero the algorithm becomes a Best-First algorithm. Thus, a higher heuristic weight usually tends to speed up the search but may not return the shortest possible path.

Another way of speeding up the algorithm is by not allow previously expanded nodes to be reinserted in the open list, even if a shorter path to the node than the current one is found. This may also result in a somewhat less optimal path, but usually this can be overlooked.

5.3.2 Adding Direction

With the addition of direction, each position in space loses its uniqueness. Thus, each node in the scene can be contained in many different search nodes depending on from which direction it was reached and two search nodes are only considered to be the same if the have the same position and evolves from the same parent node.
5.3.3 Adding Turn Cost

A central part of adding realism to the pathfinding is the addition of some turn cost. This is an additional cost that introduces a penalty cost for each turn taken, thus making the algorithm to prioritise more straight paths. This extra turn cost is also added when the estimated distance to the goal is calculated. There are a number of different ways to do this, but the computations for these tend to be rather demanding. The tricky part is thus finding a proper representation of this cost that can be calculated rather fast.

![Diagram](image.png)

**Figure 16: Calculating the turn cost.**

I have solved this by calculating the angle between the direction vector, \( d \), and the distance vector, \( v \) and let the cost be obtained from some simple function of this angle, \( f(\omega) \). See figure 16 for a further explanation. This can be easily done with the help of the scalar product (a lookup table speeds up the calculations).

\[
\omega = \frac{\arccos(d \cdot v)}{||d|| ||v||}
\]  

(5)

The cost function should preferably be very close to zero in a vicinity of zero such that small angles will not add any major cost. However, for great angles the extra cost should be large. An example of this function is:

\[
f(\omega) = k \cdot \omega^n \quad \text{where} \quad n \in \mathbb{N} \setminus \{0\}
\]  

(6)

In my applications I have used \( n = 2 \).

5.3.4 Other Search Restrictions

There is a possibility that there are other restrictions for the pathfinding that may be character specific. This can for example be that some of the characters currently moving around in the scene is forbidden to move in strict vertical directions. If this is the case the characters can have different search methods where a specific criteria needs to be fulfilled for each node that is pushed on the open list in the A* algorithm.

5.3.5 Searching in the Detailed Grid

When searching in the detailed grid every search node represents a grid cell. A search node can be expanded into either six new nodes (not allowing diagonal movement) or into 26 nodes (allowing diagonal movement). The latter has more child nodes to expand but may find its way to the end node faster since it usually visits less nodes during the search (less popped nodes from the open list). It also
results in a smoother path. On the other hand, if diagonal movement is allowed, some extra cells have to be evaluated since diagonal paths will interfere with cells outside the path nodes. None of these are suitable for high demand real time purposes.

Given a starting point and a goal point, the first thing to do is to find the corresponding cells for these. When searching for a path, collision detection has to be done for every step to check if the node can be reached without bumping into obstacles. There is a simple but somewhat non-generalizing way to deal with this. The free space is calculated with regard to the agents bounding volume (chapter 5). If the scene contains agents with different bounding volumes, the use of the largest one always results in a collision free $F$ for all the agents. Alternatively, different representations can be done for different agents but if you are dealing with limitations of memory, this is probably not the smartest thing to do. Figure 17 shows an example of a path in the detailed grid.

Figure 17: A path has been found in the grid.
5.3.6 Searching in the Octtree

When searching in the octtree, each search node stores an octtree cell. The search tree is built the same way as in the case of uniform cells with the restriction that diagonal movement is not allowed. This is done to prevent paths where the character may intersect with closed octtree cells (see figure 18).

Figure 18: Diagonal movement between cells may cause intersection with closed octtree cells.

In the undirected search, two search nodes are considered the same if they refer to the same octtree cell. In the directed case, however, two search nodes are the same if they refer to the same octtree cell and if this cell has been reached from the same octtree cell.

Figure 19: A path has been found in the octtree.
The path point that is stored for the resulting path could, for example, be the first point in the octree cell that is entered. These help positions are always situated just inside a midpoint to one of the octree cells walls. Storing these help positions as path points instead of using the octree cells midpoint also gives the path a smoother appearance. Two directed search nodes are the same if they have the same help position which means that they have the same parent octree cell. For the undirected search it is a good idea to use these help nodes instead of the octree cells center positions. This gives a shorter path and a smoother appearance (figure 19 shows a path with help nodes). In the directional search we also introduce the turning cost.

5.3.7 Searching in the Visibility Graph

In the search through the visibility graph, each search node stores a visibility graph node. In the undirected search, two search nodes are the same if they share the same visibility graph node. In the directed case, two nodes are the same if they share the same visibility graph node and have the same parent visibility graph node. As in the case of the directed octree search, there is also the addition of the turning cost. Figure 20 shows a path returned by the a search through a visibility graph. Note the chain of help nodes.

![Visibility Graph Diagram]

Figure 20: A path has been found in the visibility graph.
6 Results

Memory requirements and computational time are crucial factors when developing computer games. This becomes even more important if the product is developed for some kind of console (e.g., XBOX, PS2 or Gamecube). My main approach was therefore to base the prototype around the visibility graph representation, since this would be the ideal solution in terms of memory demands. The fact that "the less the search space, the faster the search" was also a big factor for this decision. I also knew that the search in the detailed grid would be very expensive in terms of CPU power since there would be a lot of node expansions, even for short search distances. All the results presented in this chapter are based on measurements done on the test scene and are therefore only valid for this scene. Although I believe that the general behaviour of the measurements show tendencies that are valid for other scenes as well. In some figures I have plotted a function as a reference. These functions should not be taken too seriously but rather as hints of the behaviour of the measured data. Some of the results are intended to show the complex structure of the pathfinding problem. The measurements are done with a non optimized program and therefore, there is no point in paying attention to the actual values of the data but rather consider the trends. Throughout the measurements I have used a cubic bounding box for the character and all sizes are measured in grid cell units.

6.1 Uniform Cells

Since the uniform cell representation (the detailed grid) already is a part of the SBZ engine, there has been no reason to investigate it in terms of memory requirements. At an early stage, I realized that this representation would not be interesting to further investigate due to extremely long search times. It is, however, partly used in the process of inserting the start and end point.

6.2 The Octree

The octree is a better representation than the uniform grid in terms of search times since it is actually depending on the structure of the scene (e.g., a search through a large open area might pass through only a couple of octree cells, while a search in the uniform grid has to pass through a lot of grid cells). The smaller the minimum cell size is, the more detailed information is captured about the scene. On the other hand, a smaller minimum cell size results in more octree cells and a larger octree. Since the whole octree has to be stored it actually may require more memory than the uniform grid representation. The number of cells is also, of course, very depending on the scene and the size of the desired character size. Figure 21 shows the number of open and closed cells generated with the minimum cell size equal to the character size. The number of open cells is relevant for computational times in the octree search, the number of closed cells can be related to the number of nodes when creating the visibility graph and the total number of cells is important in terms of memory demands. As a reference, a function of type \( y = c_1 + c_2 x^{-3} \) is plotted in the figure. Since the total number of cells, \( N \), of minimum cell size, \( s \), that can be fitted into a scene of size \( S \) is \( N = \left( \frac{S}{s} \right)^3 \), it is not very surprisingly that the measured data matches this type of function rather well.
Figure 21: Number of cells depending on the minimum cell size and character size. Character size and minimum cell size are equal.

Figure 22 shows the number of octtree cells for different minimum cell sizes. The data has been generated from the small test scene for a fixed character size of $3 \times 3 \times 3$ grid cell units. As before a function of type $y = c_1 + c_2 x^{-3}$ has been plotted as reference and matches the measured data rather well.

Figure 22: Number of cells depending on the minimum cell size. Character size is fixed.

The size of the octtree is also heavily dependent on the character size for which it is generated. In figure 23 the minimum cell size has been kept fixed ($11 \times 11 \times 11$ grid cell units) and the character size has been varied. With only five measurements it is hard to come to any real conclusions but the tendency seems
to be a more linear relationship.

![Image](image.jpg)

Figure 23: Number of cells depending on the character size. Minimum cell size is fixed.

### 6.3 The Visibility Graph

The ideal thing about the visibility graph is that it is very compact and in its purest form can be stored in almost no space at all. The drawback is that the start and end node have to be inserted in a fast way. My way of dealing with this situation is to use the octree for faster insertion. By doing this, the memory advantage of using the visibility graph is lost and the only way this representation could be considered better than the pure octree representation is if the search times are a lot faster.

Figure 24 shows the number of nodes (vertices) for the visibility graph. Here, the character size and the minimum cell size are equal. The data is generated from the small test scene. The relationship is not quite linear and the reference function plotted in the figure is of type $y = c_1 + c_2 \log(x)$. 

32
Figure 24: Number of nodes depending on the minimum cell size and the character size. The minimum cell size and the character size are equal.

Although there is a strong wish to keep down the number of nodes to as few as possible, it is actually not the number of nodes we worry about, but rather the number of edges. In the worst case, there is an edge between every pair of nodes resulting in \( n(n - 1) \) edges. Figure 25 shows the number of edges prior to and after the zone culling. The reference function this time is \( y = c_1 + c_2 r^{-1} + c_3 r^{-2} \).

Figure 25: Number of edges depending on the minimum cell size and the character size. The minimum cell size and the character size are equal.

Since the number of octtree cells is depending on the minimum cell size and character size and the number of corners for the visibility graph are dependent
on the number of closed octree cells, the number of edges is depending on the minimum cell size and the character size. Figure 26 shows the relation between the minimum cell size and the number of edges with the reference function \( y = c_1 + c_2x^{-1} + c_3x^{-2} \).

![Graph showing the relation between MinCellSize and NumOfEdges.]

Figure 26: Number of edges depending on the minimum cell size. The character size is kept fixed.

Figure 27 shows how the number of edges depends on the character size. As in the case of the relationship between the number of octree cells versus the character size, the relationship seems to be linear.

![Graph showing the relation between CharSize and NumOfEdges.]

Figure 27: Number of edges depending on the character size. The minimum cell size is kept fixed.
As seen in the figures above, the zone culling removes about 30 percent of the edges and is even more effective the more edges we have from the beginning. With the help of alternate path culling there may be even more edges to remove.

![Figure 28](image1.png)

**Figure 28:** Culling with alternate paths. The minimum cell size is kept fixed. Here with two different length factors.

In figure 28 the number of edges after the alternate path culling have been plotted against a polynomial of type \( y = c_1 + c_2 \alpha^{-1} \).

Since a search through a graph rarely explores more than a small part of the nodes it is not the total number of edges that is crucial but rather the number of edges per node — the edge density. Even in this case there is a lot to gain by culling with alternate paths.

![Figure 29](image2.png)

**Figure 29:** Density of edges prior to culling with alternate paths. Minimum cell size and character size = 5.

Figure 29 shows the edge density prior to culling with alternate paths. A majority of the nodes have between 20 and 50 edges, about one fourth has between 50
and 100 edges and two of the nodes even have more than 100 edges! Expanding 100 nodes in a A* search cycle is certainly not wanted and, luckily, as seen in figures 30 and 31 this can be handled by the alternate path culling. Another nice property of the alternate path culling is that a harder culling (larger length factor and larger cull angle) lowers the overall edge density.

Figure 30: Density of edges for different cull angles. Length factor = 1.1. Minimum cell size and character size = 5.

Figure 31: Density of edges for different cull angles. Length factor = 1.2. Minimum cell size and character size = 5.
6.4 Searching

For the A* search in the octree I have tried two different strategies, with or without storing the neighbours to each octtree. The storage of neighbours speeds up the search but also requires more memory. As a matter of fact, this extra amount of memory is unacceptably high. Consider, for example, a very large octtree cell surrounded by many octtree cells of minimum size. To store each of its neighbours the octtree cell has at least to store a 2-byte integer (if the octtree cells are stored in an indexed array) or a 4-byte pointer. For complex scenes with a lot of open air space, this is devastating. There has been at least two measurements for each search series used in the analysis. This has been done to minimize temporary slow searches due to uneven load on the processor.

Figure 32 shows some searches done with either pre-stored or online determination of the neighbours. Strangely enough, in some cases the online determination is actually faster, but overall the search times are about the same.

![Search times in the octree for stored neighbours and online calculated neighbours. Minimum cell size = 10 and character size = 5.](image)

Figure 32: Search times in the octree for stored neighbours and online calculated neighbours. Minimum cell size = 10 and character size = 5.

Figure 33 compares the search times for a series of searches in the octtree and the visibility graph. In general the search in the visibility graph is about 40% faster than the search through the octtree although in some cases the octtree search is more than four times faster.
Figure 33: Search times for the octree search and the visibility graph search. Cost weight $= 1.0$ and heuristic weight $= 2.0$. Minimum cell size and character size $= 5$.

With the addition of direction we add another dimension to the search nodes. Now, a search node is no longer unique by its position in space. Because of this an octree node can be represented by as many search nodes as it has neighbour cells and a visibility graph node may be represented by as many search nodes as it has edges. As seen in figure 34, this is rather devastating for the search times. In some cases the search times become almost 100 times slower. In particular, the octree search gets affected by this with search times now surpassing 30 seconds.

Figure 34: Search times for the directed octree search and the visibility graph search. Cost weight $= 1.0$ and heuristic weight $= 2.0$. Minimum cell size and character size $= 5$. 
Figure 35: Search results for the different heuristic weights. Cost weight = 0.0, Minimum.
Figure 36: Search times for different turn cost factors. The turn cost factors are measured as a multiplier of the character size. Cost weight = 1.0 and heuristic weight = 2.0. Minimum cell size and character size = 5.

Figure 37: Allowing reinsertion in the open list or not. The upper figure shows the octree search, the lower the visibility graph search. The turn cost factor = 2.5. Cost weight = 1.0 and heuristic weight = 1.0. Minimum cell size and character size = 5.

The A* algorithm always stores the shortest path to every search node. If it finds a shorter path to a previously visited node it deletes the old path and stores the new one instead. By not allowing this it is possible to speed up the search times.
This is accomplished by not allowing reinsertion into the open list. Figure 37 shows how this affects the search times for the octtree search and the visibility graph search. As seen, it lowers some of the slowest searches dramatically. This is especially evident in the octtree search.
7 Conclusions and Recommendations

As stated earlier, the main purpose of this work has been to investigate different solutions to the 3D pathfinding problem. With the addition of restrictions further realism has been added to the different prototypes. Adding realism to computer games (or any other real-time application) is almost always devastating. Even though the processors of today are quite powerful and a lot of rendering computations have been moved to the graphics hardware, they still have limitations. The general strategy to deal with this kind of problems is to do what you have always learned that you should not do - simplify and “cheat”. “Cheating” is sometimes the most powerful way to get past a difficult problem and computer games are one of the best examples of this. Computer game developers exclusively use the rule of thumb “as long as it works and looks good - use it”. Certainly, this kind of “cheating” is a form of art by itself.

The main issues when evaluating my different solutions have been memory and speed. Since my visibility graph solution stores an underlying octree for faster insertion of the start and end node, the pure octree solution is actually a better alternative in terms of memory demands. The memory requirements for the octree depends on both the character size and the desired minimum cell size. Especially the choice of the minimum cell size will affect the size of the octree. With a larger minimum cell size a lot of space may be spared to the cost of losing detailed information about the scene. When searching the octree, an octree cell's neighbours should be determined online. Letting each cell store its neighbours hardly gains any speed performance but certainly uses a lot more memory.

In terms of search speed, the visibility graph representation is definitely the best solution. By using the different culling techniques presented in section 5.2.3 the edge density can be brought down to really nice levels. This speeds up the search considerably since the A* search needs to do less node expansions each turn. The strength of the visibility graph search becomes even more evident with the addition of direction and other search restrictions. Although the addition of direction may add realism to the search, the penalty in performance is way too high to be accepted. However, the addition of a penalty term may at least be a less expensive way to force the search to choose smoother paths.

To improve the speed performance of the search, the A* algorithm can be modified by adding weights to the cost function. Shifting the weight towards the heuristic term may speed up the search considerably. The bad side of this is that the returned path is not always optimal. However, finding the optimal path is not always necessary. The general player will not study the pathfinding in detail. He will accept any type of pathfinding as long as the movement of characters is reasonable. A heuristic weight of 2.0 - 2.5 can make drastic improvements and also return quite nice paths. Another way to speed up the search is by not allowing reinsertion of previously visited search nodes into the open list in the A* search. As in the case of manipulating the A* with weights, the returned path may not always be the optimal, but the difference is almost always acceptable.

To wrap things up, we can conclude that the octree solution is the best solution.
in terms of memory and the visibility graph solution is the best one in terms of search times. Also, the addition of direction for the search nodes is devastating for the search times and should be avoided. If a better way to handle the insertion of the start and end node is found, the visibility graph solution definitely should be the choice of solution. Otherwise, I strongly recommend the octree. Even though the search times is about 65% slower than the visibility graph, the memory issue certainly is the most critical factor. To get faster searches a heuristic weight of 2.0 - 2.5 should be used. Also, reinsertion of previously visited search nodes should not be allowed.

7.1 Further Improvements

Since the search times are a lot better for the visibility graph search, this solution definitely may be worth to further investigate. Most game engines use some kind of BSP-tree (Binary Space Partitioning Tree) and PVS (Potential Visibility Set) for faster collision detection, render optimisation etc. Perhaps the insertion of the start and end node could be solved in a better way by using any of these. In that case, the octree could be deleted after the generation of the static visibility graph and solve the memory issue.

In any of the solutions presented, the representation should be checked for connectivity as an extra precaution. If the representation is not connected (e.g. it consists of two disconnected parts) a search from a position in one of the parts to a position in another part always will result in a search failure. If the representation is not connected, a smaller minimum cell size may solve the problem. Otherwise, the problem has to be dealt with in some other way.

In my work I have not considered dynamic changes in the environment. Dynamic changes make everything even more complicated. In the case of dynamic environments (e.g. buildings may be destroyed) changes have to be made to the representation. In the case of the octree, there might be a good idea to introduce another type of cell - the DYNAMIC cell. These cells may be either open or blocked depending on the circumstances. For the visibility graph the same can be accomplished by the addition of dynamic edges.

The path returned from the pathfinding is edgy and crooked. Prior to moving a character along the path it might be a good idea to do further processing of the path. For example a waypoint may be removed if it is possible to travel from its previous waypoint to the following one without passing through the current one. This becomes evident in particular in the beginning and the end of the path where the start and end node has been inserted and if there are long chains of help nodes. For the smoothing of the path each turn may be rounded by either using the path waypoints as control points for a spline or by letting each path waypoint be situated on a circle as described in the Appendix section (a somewhat similar approach can be found in [14]).

By using the circle smoothing described above it is also possible to introduce speed restrictions. In each turn the character is exposed to a centripetal force. This force is depending on the speed of the character and the radius of the circle. By letting the radius of the circle r depending on the turn angle and defining a
maximum centripetal force (maximum acceleration) $a_{\text{max}}$, the maximum speed $v_{\text{max}}$ for taking the turn may be calculated as $v_{\text{max}} = \sqrt{r a_{\text{max}}}$. The character may then follow the path by accelerating between the turns to fulfill the speed restrictions associated with each turn.
References


A Appendix

Consider two points, \( p_1 \) and \( p_2 \). To the starting position, \( p_1 \), there is also a direction, \( d \), representing the direction in which the character is facing. If we let this direction be a tangent to a circle, lying in the plane that is spanned by the \( d \) and the vector \( v = p_2 - p_1 \). The vector \( v \) should intersect with this circle. The radius of the circle should represent the characters minimum turning radius. The length of the line from the center of the circle, \( c \), to the point \( p_2 \) is \( B \). A tangent to the circle leaves the circle at a point \( p_3 \) and passes through \( p_2 \). Instead of calculating the cost as the length of the vector \( v \), the cost should now be calculated as the length of the arc from \( p_1 \) to \( p_3 \) plus the length of the line from \( p_2 \) to \( p_3 \), which we call \( A \). See figure 38 for a further explanation.

![Figure 38: Turn radius.](image)

The path length, \( l \), can then be calculated in the following way:

\[
B^2 = r^2 + ||v||^2 - 2r||v|| \cos \beta \quad \text{(Law of Cosines)}
\]

\[
\cos \beta = \sin \left( \frac{\pi}{2} - \beta \right) = \sin \left( \pi - \left( \frac{\pi}{2} - \beta \right) \right) = \cos \omega
\]

\[
\Rightarrow B^2 = r^2 + ||v||^2 - 2r||v|| \cos \omega
\]

\[
\cos \omega = \frac{\partial v}{||v|| ||v||}
\]

\[
\Rightarrow B^2 = r^2 + ||v||^2 - 2r \frac{\partial v}{||v|| ||v||}
\]

\[
\frac{\sin \gamma}{||v||} = \frac{\sin \beta}{B} \quad \text{(Law of Sines)}
\]

\[
\sin \beta = \cos \left( \frac{\pi}{2} - \beta \right) = -\cos \left( \pi - \left( \frac{\pi}{2} - \beta \right) \right) = \cos \omega
\]

\[
\Rightarrow \sin \gamma = \frac{||v|| \cos \omega}{B} = \frac{\partial v}{||v||}
\]

\[
\Rightarrow \gamma = -\arcsin \left( \frac{\partial v}{||v||} \right)
\]

\[
A^2 + r^2 = B^2 \quad \text{(Pythagorean Theorem)}
\]

\[
\frac{\sin \alpha}{A} = \frac{\sin \frac{\pi}{2}}{B} \quad \text{(Law of Sines)}
\]
\[\Rightarrow \delta = \arcsin \left( \frac{\sqrt{u^2 - r^2}}{B} \right) = \arcsin \left( \sqrt{1 - \frac{r^2}{B^2}} \right)\]

\[\phi = 2\pi - (\gamma + \delta)\]

\[l = r\phi + A = r \left( 2\pi - \left( \arcsin \left( \sqrt{1 - \frac{\phi^2}{B^2}} \right) \right) \right) - \arcsin \left( \frac{\phi}{2B} \right) + \sqrt{B^2 - r^2}\]