Sampsa Oksa

Pathfinding in a 3D-environment Using Unity3D

Thesis
Kajaani University of Applied Sciences
School of Business
Business Information Technology
Autumn 2014

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteiden ala	Koulutusohjelma Tietojenkäsittely
Tekijä(t) Sampsa Oksa	
Työn nimi Reitinhaku 3D-ympäristössä Unity3D pelimoottorilla	
Vaihtoehtoiset ammattiopinnot Peliohjelmointi	Ohjaaja(t) Leena Heikkinen
	Toimeksiantaja -
Aika Syksy 2014	Sivumäärä ja liitteet 37

Opinnäytetyön tavoitteena oli luoda toimiva reitinhakualgoritmi 3D-ympäristöön käyttäen Unity3D pelimoottoria. Täysin toimiva A* reitinhakualgoritmi toteutettiin niin, että se pystyi suorittamaan kaksi eri testiä useammalla eri graafilla. Opinnäytetyö keskittyy reitinhaun toteutukseen ja siitä saatujen tulosten arviointiin. Tämän vuoksi opinnäytetyö ei keskity liiaksi eri variaatioihin ja optimointimahdollisuuksiin joita A* tarjoaa.

Opinnäytetyö alkaa reitinhakuun tutustumisella ja tarjoaa perustietoa siitä, mitä se on ja miten sitä voidaan käyttää. Kun perustiedot on käyty läpi, opinnäytetyössä esitellään vanhempia hakutekniikoita, jotka ovat muokanneet tietä nykyaikaiselle reitinhaulle. Tämän jälkeen A*-algoritmia tarkastellaan tarkemmin, selittäen sen toiminnallisuutta ja avainominaisuuksia. Opinnäytetyössä tutkitaan tämän jälkeen eri tapoja, joilla A*-algoritmia voidaan muokata, jotta se toimisi parhaimmalla mahdollisella tavalla siinä sovelluksessa, jossa sitä käytetään. Osaa tiedoista käytetään myöhemmin opinnäytetyön toiminnallisessa osassa.

Kun kaikki tekniikat, joita A* voi käyttää, on esitelty, opinnäytetyön toiminallinen osa esitellään. Se määrittää toiminnallisen osan tavoitteen ja miten se toteutetaan. Esittelyssä kuvataan, miten reitinhaku lisätään ympäristöön ja kuinka siitä saatuja tietoja myöhemmin analysoidaan.

Opinnäytetyön loppu keskittyy projektin dokumentoimiseen ja testiympäristöjen järjestelyyn sekä itse testeihin. Dokumentaatio selittää myös minkä vuoksi tiettyjä asioita on käytetty ja millä tavalla ne vaikuttavat testien lopputulokseen. Tämän jälkeen tuloksia analysoidaan ja koko opinnäytetyön onnistumista arvioidaan.

Kieli	Englanti
Asiasanat	Reitinhaku, A* algoritmi
Säilytyspaikka	☒ Verkkokirjasto Theseus ☐ Kajaanin ammattikorkeakoulun kirjasto

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

THESIS
ABSTRACT

School Business	Degree Programme Business Information Technology
Author(s) Sampsa Oksa	
Title Pathfinding in a 3D-environment Using Unity3D	
Optional Professional Studies Game programming	Instructor(s) Leena Heikkinen
	Commissioned by -
Date Autumn 2014	Total Number of Pages and Appendices 37

The goal of this thesis was to create a working pathfinding algorithm for a 3D environment running in Unity3D game engine. A fully working A* pathfinding algorithm was implemented so that it was able to run two different types of tests using two different graph representations. The thesis strictly focused on the implementation and analysis of the results produced by the algorithm. Because of this, the thesis did not delve too much in to the different variations and optimizations available for the A*.

The thesis starts with an introduction to pathfinding and provides some basic information what it is and how it can and should be used. After the basics the thesis represents some older search techniques which have been in a major role for the modern pathfinding algorithms. After that the A* algorithm is reviewed in a more detailed manner, explaining its functionality and some of its key factors. The thesis then explores different ways the A* algorithm can be modified to satisfy the needs of the game it is applied to. Some of the findings are used later in the practical part of the thesis.

After introducing all the techniques the A* can use, the practical part of the thesis is presented. It defines the goal of the project and how the execution will happen. The introduction of the project describes how the pathfinding will be added to the environment and how the received results are later analyzed.

The end of the thesis focuses on documenting the process of setting up the test environments and running the tests. The documentation also explains why certain things are made and how they affect the outcome of the tests. The conclusion of the thesis analyzes the results gained from the project and reviews the success of the thesis as a whole.

Language of Thesis	English
Keywords	Pathfinding, A* algorithm
Deposited at	☒ Electronic library Theseus ☐ Library of Kajaani University of Applied Sciences

CONTENTS

LIST OF SYMBOLS

FPS	Frames Per Second, the frame rate of a game.
Node	Node is a term in pathfinding for a point which is used in the pathfinding. A node holds information about its location and different values which are used to calculate the best route. Additional information related to the node can also be stored in it.
Graph	A graph is some kind of a form in which the nodes used in the pathfinding are presented.
Heuristic	A heuristic means the approximation of distance in pathfinding. It is an estimation how much it costs to get to the end node from a particular node.
DFS	Depth-first search, a search algorithm that visits nodes as deep as it can before switching to another branch.
BFS	Breadth-first search, a search algorithm that goes through nodes one level at a time before delving deeper in a tree graph.
AI	Artificial Intelligence, behavior of the computer controlled units in a game which tries to mimic intelligent behavior.

1 INTRODUCTION

Pathfinding is an important part of the AI (Artificial Intelligence) systems present in video games. It adds value and works as a pretty strong backbone for any AI unit which attempts to replicate intelligent behavior in some manner. While pathfinding as a concept seems pretty straightforward, it does have many things that need to be taken in to consideration when it is designed and implemented. In this thesis a pathfinding algorithm is created and used in a 3D environment. The practical part of the thesis is restricted to a simple pathfinding implementation in two different ways to demonstrate the flexibility of the algorithm.

This thesis explores the core functionality of pathfinding and ways to modify and improve it to fill the needs of a particular game it is implemented to. Because the main focus of this thesis is the practical part, none of the information provided in the theory part is explored too deeply because it is beyond the scope of this work. Later in the thesis this information is, however, used to create a working A* algorithm that can be used in 2D and 3D video games. The goal is to create a pathfinding algorithm which finds optimal routes in reasonable time so that it is theoretically possible to implement to an actual game.

The created pathfinding algorithm was developed solely for this thesis using the Unity3D game engine. Unity3D was chosen for its easy visual editor which allows an easy way to document and visualize the algorithm in runtime. The engine's visual 3D tools also enabled fast prototyping and modification of the test environments so that the algorithm could be tested in a myriad of ways. All the findings of the thesis are applicable to future projects involving pathfinding and AI programming.

2 BASICS OF PATHFINDING

What is pathfinding? At the simplest level, pathfinding is a way to find a route between two points. In modern video games this is used as a part of the artificial intelligence (AI) to control the movements of non-player characters (NPC). Pathfinding is required to avoid unintelligent behavior which might lead to breaking the immersion of the player. (Anguelov 2011.)

Since video games are largely visual experiences, a lot of improvements have happened in the recent years to provide players better experiences. Games are usually required to run at least at 30 FPS (Frames Per Second), to appear smooth, and it sets a lot of performance constraints for the AI in general to be able to complete all the necessary tasks every frame. This means that the time given for the pathfinding process is quite a small amount of milliseconds. Since computers also have a finite amount of memory, optimization is in place when making large features like pathfinding. This is why different algorithms have been developed to find the most efficient paths. (Anguelov 2011.)

If pathfinding is not present in a little more complicated environment, it might lead to unintelligent behavior. If a NPC moves around only by trying to reach its goal using a direct line between points, it might get stuck behind a wall or a tree. This kind of behavior breaks the illusion of thinking creatures which also means that the gaming experience is interrupted. (Anguelov 2011.)

What pathfinding does is that it calculates possible routes between two points and finds the most optimal solution. It takes into account different obstacles that might block the way or slow down the movement of the unit. Depending on the algorithm used, the found path may vary a lot from a path found with another algorithm. This requires that the programmer gives the program some kind of understanding of directions and distances and how to use them. (Patel 2014.)

Figure 1 shows a basic difference between the moving with and without pathfinding. The blue line represents the movement with pathfinding on and the red line without pathfinding.

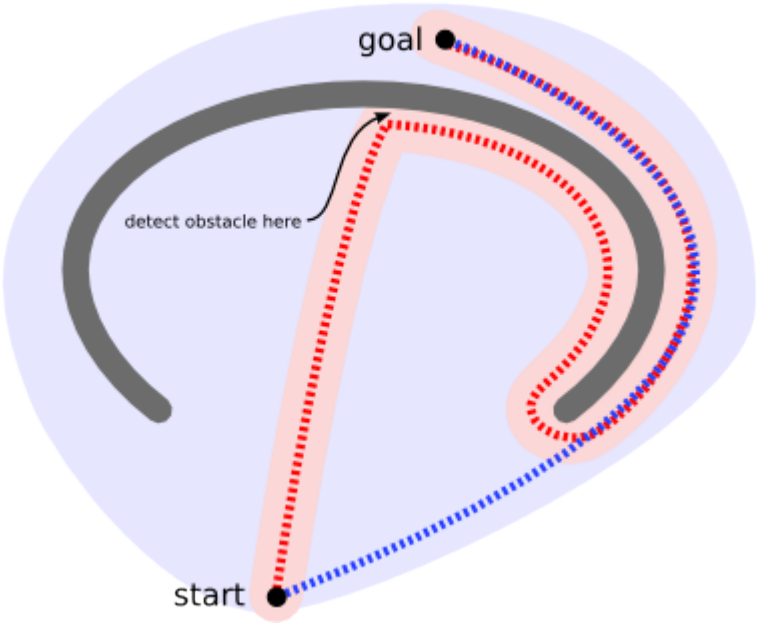

Figure 1. Movement with and without pathfinding (Patel 2014)

Other applications also benefit from pathfinding, most notably real life navigation systems that use GPS to track their user's position. These applications give the user different route suggestions which they can then use to get to certain locations. Usually the algorithms have different variables in them, which enable them to filter out different results. These results might include the most ecological route, shortest route and the fastest route. (Puthuparampil 2007.)

Because of the performance constraints set by the computers it is nearly impossible to make a pathfinding algorithm that is suitable for every purpose. That is why the algorithms are usually tailored for their specific purposes. This allows the algorithm to work as optimally as possible and do just the necessary job that is required of it. (Patel 2014.)

3 EARLY PATHFINDING METHODS

Mathematicians have explored different types of pathfinding algorithms for decades and found a myriad of solutions. The basic algorithms are quite simple to understand while the more complex ones are very challenging to comprehend. The early algorithms were uninformed graph searches that did not take into consideration any costs when traversing nodes. The more modern algorithms are cost based, meaning they can more precisely analyze the cheapest paths to reach the target destination. (Buckland 2005, 209-210.)

3.1 Depth-first search (DFS)

The depth-first search is an algorithm that searches through a graph by going as deep as it possibly can. The algorithm follows a branch until it reaches a dead end or the destination. If the branch is a dead end, the algorithm then backtracks to a more shallow depth where the next branch begins and searches that branch in full depth. (Buckland 2005, 210 - 213.)

Since DFS (Depth-first search) is an uninformed algorithm, it does not take into consideration where the end destination is. Even though DFS usually finds a path, it is not guaranteed to be optimal in any way. Using DFS instead of an intelligent cost based algorithm may lead to unwanted behavior. (Eranki 2002.)

The problem DFS has is that it may or may not find the path very quickly. If the algorithm has luck and it picks the correct branch in the very beginning, it might find a route to the destination very fast. On the other hand it might also search through the whole graph before finding the way. If the graph size is very large this usually leads to a situation where the algorithm takes a very long time. In the worst case scenario, the algorithm might also end up in an infinite loop without ever finding a path. (Buckland 2005, 213 - 224.)

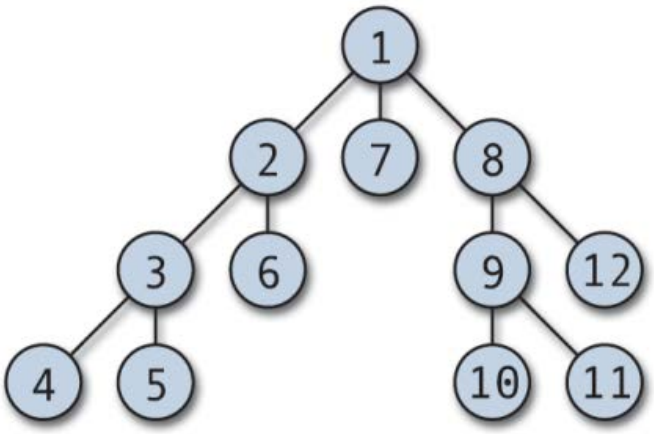

Figure 2. DFS and the search order of the nodes (Esser 2005)

3.2 Breadth-first search (BFS)

Breadth-first search is much like DFS but instead of trying to reach the end going through the graph one branch at a time, BFS (Breadth-first search) fans out from the starting point. BFS searches through all the nodes in one depth before moving on to the next depth. By searching through the graph this way the algorithm is guaranteed to find a shortest path to the destination if it exists. (Buckland 2005, 224.)

The real problem with BFS is that this kind of systematic search takes a very long time when the search spaces are larger. Because BFS searches through the graph one depth at a time, large searches where there is a high branching factor start to slow down. So even though the algorithm guarantees a path, it might take a very long time to actually find it. (Buckland 2005, 230 - 231.)

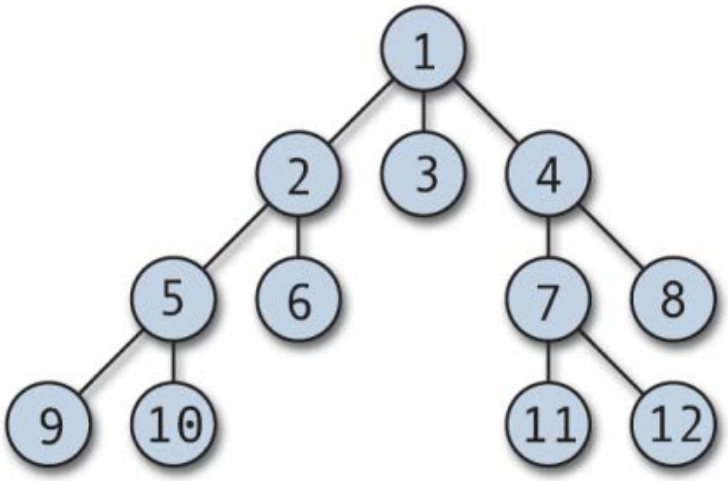

Figure 3. BFS and the search order of the nodes (Drichel 2005)

3.3 Dijkstra's algorithm

Edsger Dijkstra published his algorithm in 1959 to answer the shortest path problem in a weighted graph that does not use negative cost values (Anguelov 2011). In a graph this means that the edges between nodes represent the cost to travel to that specific node (Dijkstra 1959). This makes the algorithm a cost-based graph search and the way the algorithm works guarantees a shortest path if one exists. (Buckland 2005, 231 - 234.)

Dijkstra's algorithm works by handling one node at a time. The node gets a value that describes how much it costs to move to that specific node from the source node using a specific path. The values are cumulative, meaning that the further the algorithm searches, the larger the values get. By storing these values the algorithm can evaluate whether another path is better just by comparing if the cumulative value would be smaller using the new route (Puthuparampil 2007). The process will be terminated as soon as the algorithm reaches the destination node. (Buckland 2005, 234.)

Despite the fact that Dijkstra's algorithm gives the shortest path, it is not very efficient and is therefore not usually used in video games. Because the algorithm always takes the closest node under examination, the searched area expands to unwanted directions. This is simply because the algorithm does not have a sense of direction. The algorithm searches a lot of unnecessary nodes similar to BFS. (Buckland 2005, 240 - 241.) Figure 4 shows an example

of a search using Dijkstra's algorithm. The red square is the starting node, the blue square is the end node and the teal colored area marks all the visited nodes required to find the shortest path.

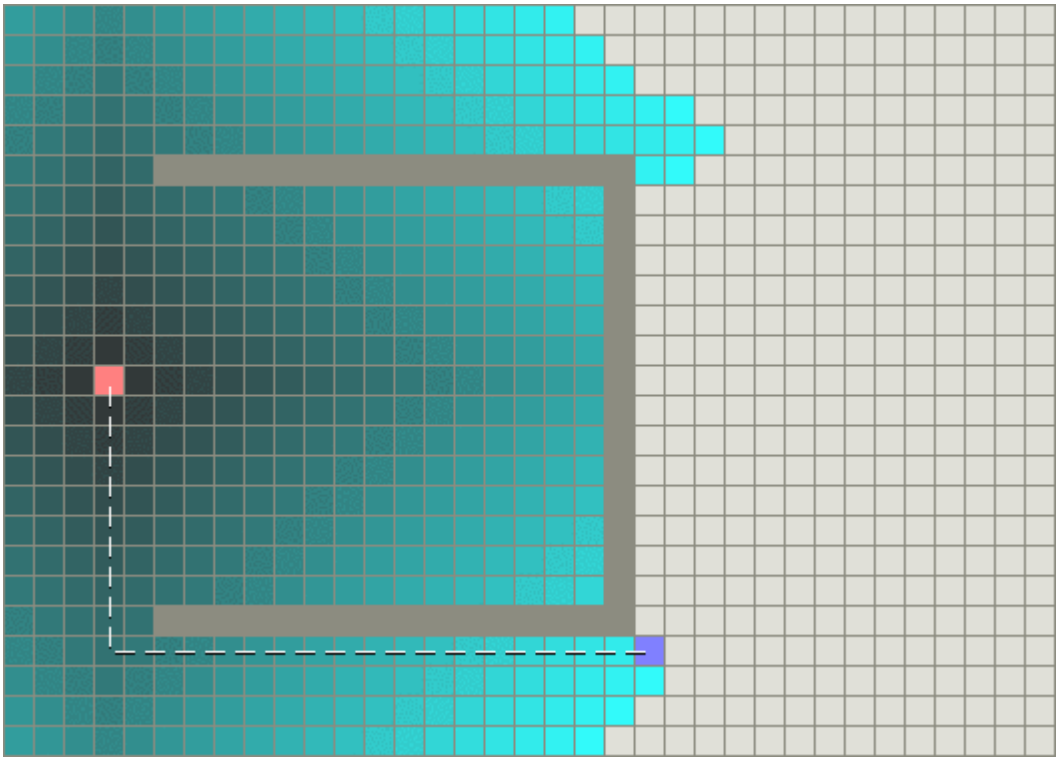

Figure 4. Dijkstra's algorithm (Patel 2014)

3.4 Greedy best-first search

Greedy best-first search differs from Dijkstra's algorithm so that it does not keep track how much it costs to traverse from the start node. Instead it uses an estimate how much it costs to traverse from a particular node to the end node. This estimation is called a heuristic. An algorithm is considered greedy when there is no long term planning involved. Greedy algorithms choose the option which seems to be the best at the current time. (Madhav 2013, 185.)

Because of the heuristic evaluation, greedy best-first search is much faster than Dijkstra's algorithm. The heuristic guides the algorithm towards the end node which reduces the amount of nodes the algorithm has to visit. However, the path found by the algorithm is not guaranteed to be the shortest path. (Patel 2014.)

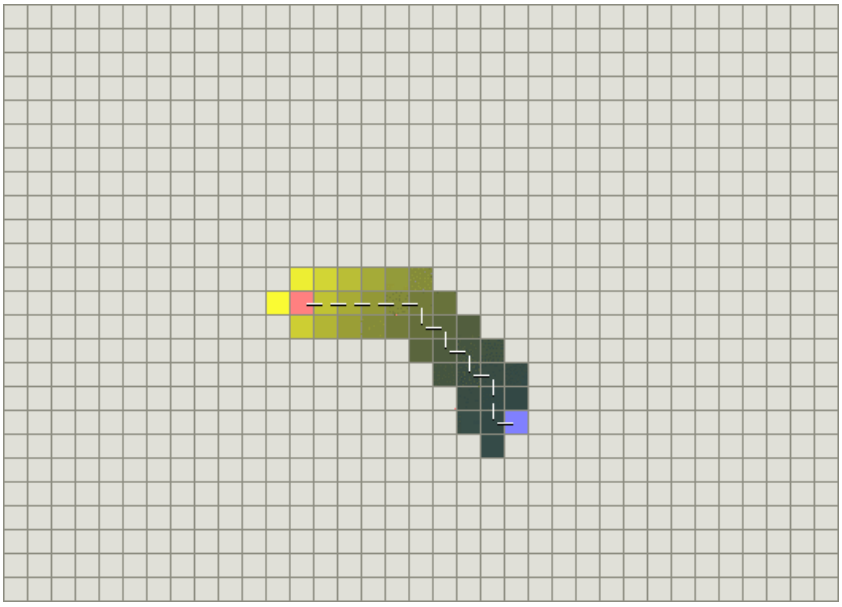

Figure 5. Greedy best-first search without obstacles (Patel 2014)

The greedy best-first search is not trouble free. Due to the fact that it does not take the already traversed path into consideration like Dijkstra's algorithm, the algorithm might keep on going straight towards the goal even if the path is not even remotely optimal. This might lead to weird path choices and especially different types of obstacles cause problems for the algorithm. (Patel 2014.)

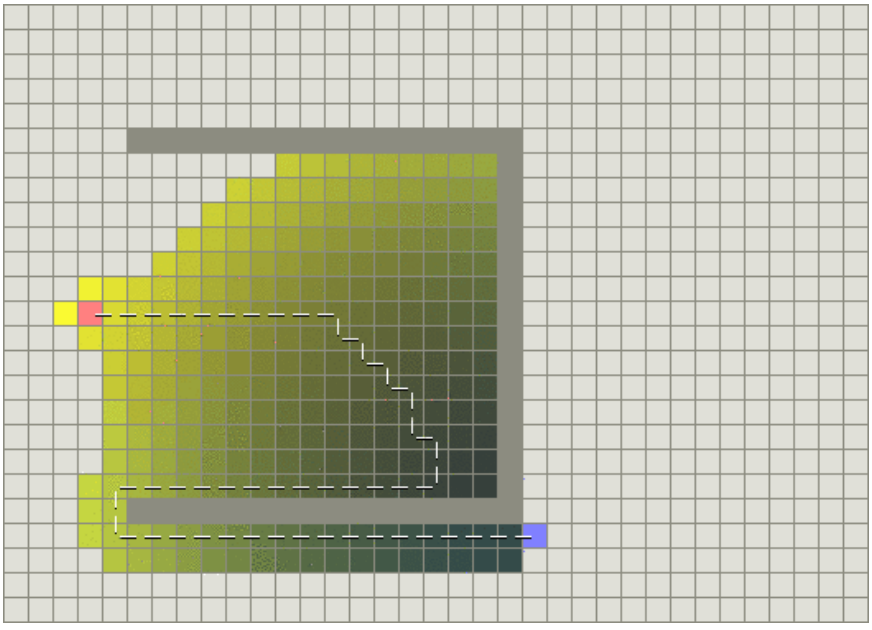

Figure 6. Greedy best-first search with an obstacle (Patel 2014)

4 A* ALGORITHM

A* (pronounced A-star) is the most popular pathfinding algorithm used in video games. While it is in its core very much alike Dijkstra's algorithm, it also utilizes the heuristic approximation used in greedy best-first search. A* is fairly flexible and easy to modify, which makes it ideal for video games. It can find the shortest paths like Dijkstra's algorithm and perform fast like the greedy best-first search. A* makes use of three values: G-value which is the actual cost to travel to a node, H-value which is the heuristic estimate and F-value which is the combination of the previous two. (Patel 2014.)

4.1 A* basics

A* is guaranteed to give optimal paths if the heuristic estimate is lower or equal to the actual cost left to travel to the end node. Using the information of the actual cost to traverse to a node from the start node and the estimated cost left to traverse to the end node gives the algorithm accuracy and performance, because the actual cost keeps the algorithm on the right path and the heuristic directs the algorithm to the right way. (Buckland 2005, 241 - 242.)

The G-value marks the true cost to traverse to a node from the starting node. Each node stores the value how much it costs to traverse to it using a specific path. When the algorithm examines a node, it searches through the surrounding nodes connected to it. If the surrounding nodes would get a better value when traversed through the node under examination, then the path to the surrounding nodes is redirected through the examined node. Since the G-value is responsible for the accuracy of the path, it is the most important value of the algorithm in that sense. (Anguelov 2011.)

H-value, which is the heuristic value, is an approximation of the distance still needed to travel to get to the end node. Heuristic value directs the A* to the most promising direction and allows the algorithm to ignore nodes further from the end node. This of course requires that the solution, optimal path, is found from the direction the heuristic value points to. If the path cannot be found from the initial direction, the algorithm will search the less potential nodes later on. (Anguelov 2011.)

The total value F consists of the actual cost G to travel to that specific node from the starting point and the heuristic estimation H how much it costs to get to the end node. Therefore F(n) = G(n) + H(n). When the different values are utilized in this manner, the algorithm is directed towards the end node in a fashion which gives the algorithm speed and precision. (Buckland. 2005 241-242; Anguelov 2011.)

Fewer nodes need to be examined since the algorithm does not expand to all directions like Dijkstra's algorithm and this is the main difference between the two. When using the F value to get the best node out of the list of open nodes, A* first tries the most obvious solution without discarding potential nodes too soon. This feature allows the algorithm to find an optimal route even if the first searched direction ends up being a dead end. (Buckland 2005 242; Anguelov 2011.)

4.2 Heuristics

There are various types heuristic functions that can be used in the A* algorithm and they can be used to control A*'s behavior. By changing the heuristic function and how it operates, the algorithm's speed and precision can be drastically altered. The performance and precision go hand in hand and usually improving the other leads to the decreasing of the other. (Patel 2014; Anguelov 2011.)

Heuristics affect the behavior of the algorithm in the following ways:

- The extreme situation where the value of H is 0, only the G value of the A* affects the algorithm's behavior. This turns the algorithm to Dijkstra's algorithm and therefore is guaranteed to find a shortest path.

- If the heuristic function always estimates the cost to be lower or equal to the actual cost left to travel to the end node, the heuristic function is admissible and will find a shortest path. The lower the estimated cost the larger the search area of the algorithm is. This will also make the search slower.

- If the approximated cost is exactly the actual cost left to travel to the end node then A* will only follow the most optimal path never expanding its search to unnecessary nodes making it very fast and optimal. The problem is that this is achievable only in

certain special circumstances, but it is good to know that A* will behave perfectly, given the right conditions.

- If the estimated cost is sometimes larger than the actual cost, the search is a little more biased towards the H value making it behave more like a greedy search. This will not guarantee a shortest path but it will make the search perform faster.

- At the other extreme where the H value is way greater than the G value, only the H values have any weight in the algorithm turning it completely into a greedy search. (Patel 2014.)

A* algorithm is quite flexible and easily modified by changing the heuristic function. This is especially useful in video games, where it is often unnecessary to find a perfect route but rather a solution that is close enough to the best thing. By utilizing the heuristic function in different ways, speed and precision can be achieved to get the result that is needed for a specific game or situation. (Patel 2014.)

Since the heuristic function can be modified quite freely, it can be used to filter out certain types of paths easily. Nodes that are located in these kinds of paths are simply given larger estimated values making the algorithm to avoid them. Another example of modifying the function is giving the function some kind of weight. By weighting the values over time or by other conditions, the algorithm can be more precise in wanted situations and in other situations focus more on the performance. Figures 7, 8 and 9 show the behavior of A* when using different heuristic functions on a grid graph. (Patel 2014.)

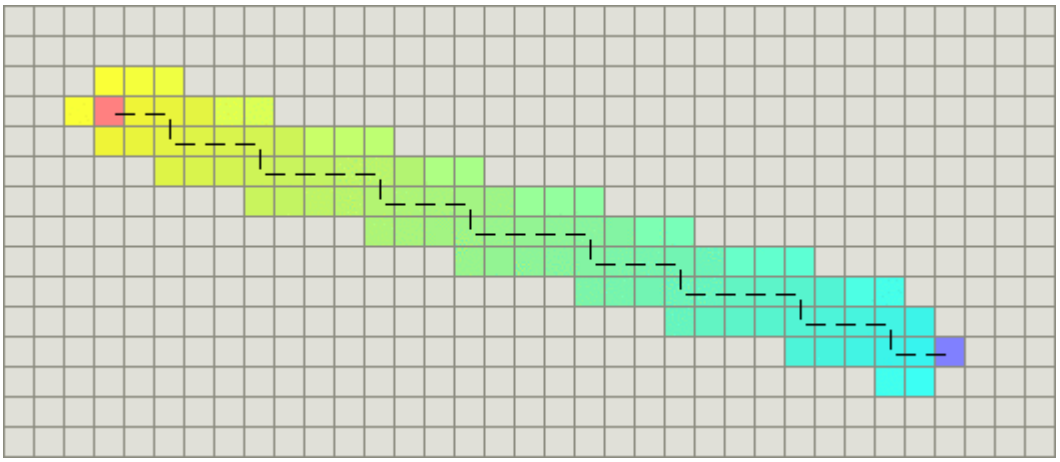

Figure 7. A* using Manhattan distance (Patel 2014)

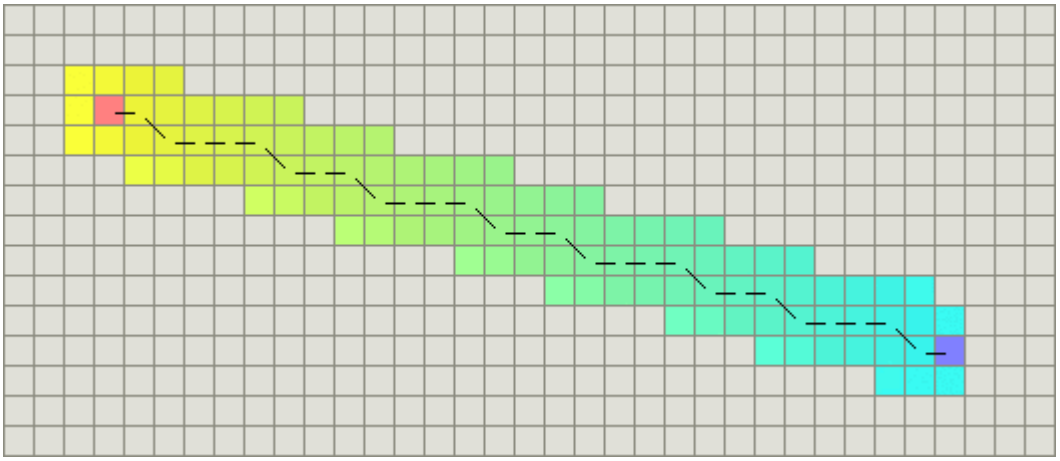

Figure 8. A* using Diagonal distance (Patel 2014)

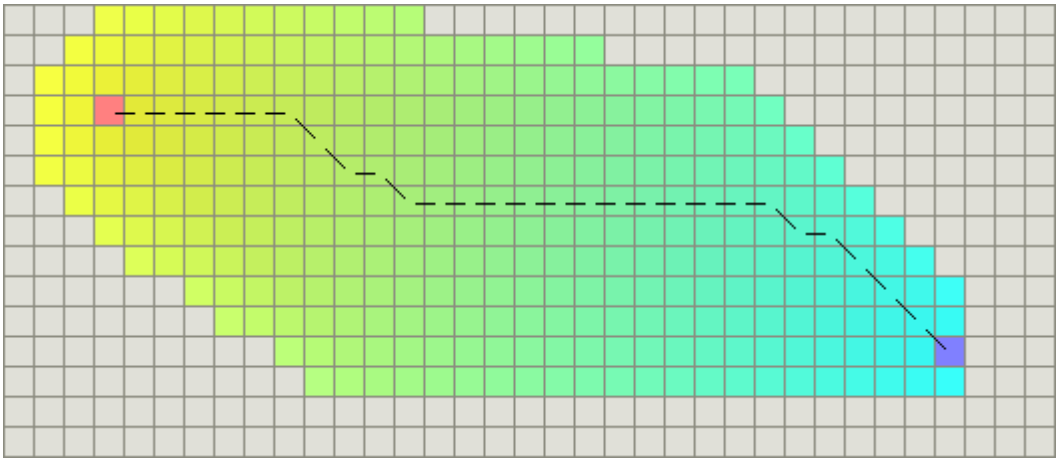

Figure 9. A* using Euclidean distance (Patel 2014)

4.3 A* core loop

Even though the A* algorithm may seem like it is a lot of work, the basic implementation and functionality is quite simple. The algorithm uses two sets, open set and closed set. The open set contains the potential nodes for examination and the closed set contains all the nodes that have already been examined. The best node is pulled from the open set to be examined and put in to the closed set. The algorithm then searches through the adjacent nodes and adds them to the open set if needed. All the searched nodes are given G, H and F values, if they do not already have any or if the existing values are worse, based on how much the costs would be if travelled through the examined node. When all the adjacent nodes have been searched, the algorithm picks the next best node from the open set and

searches the adjacent nodes of that node. This will go on until the end node is reached. The path construction is based on the information that every node has a link to its parent node, the node that was used to get to that specific node with the specific values. (Patel 2014.)

The A* loop is easy to comprehend when it is broken down into a few points:

- Initially only the starting node is added to the open set.
- The following is repeated:
    - The best node (the one with the lowest F value) is picked from the open set to be the current node under examination.
    - The node is moved to the closed set.
    - The adjacent nodes connected to the current node are searched through. If the adjacent node is blocked or if it is already in the closed set, the node is skipped.

      If the adjacent node is traversable and it is not yet in the open set, it is calculated the correct values, the current node under examination is set as its parent node and it is then added to the open set.

      On the other hand if the node is already in the open set, its G value is compared to the G value it would get if travelled to through the current node. If the G value it would get when traversed through the current node is better than its existing value, the adjacent node is calculated with new values and its parent is set to be the current node. Otherwise no action needs to be taken.
    - The search is terminated once the node picked up from the open set is the end node and it is moved in to the closed set. Alternatively the search ends if there are not any more nodes available in the open set meaning that a path does not exist.
- If a path is found then the path is constructed by tracking down each nodes parent node starting from the end node. The path is constructed when the starting node is

reached. The starting node can be identified easily since it does not have a parent node. (Lester 2005.)

While the basic implementation is easy to understand, the simple example does not take into account other factors that affect the algorithm. These include other moving units, varying terrain costs and smoothing of the found path. When the computers' limited processing power and memory is taken into account, optimization of the algorithm in every possible field should be taken seriously. (Anguelov 2011; Lester 2005.)

5 MAP REPRESENTATIONS

Grid graph is not the only option when using A*. A* can be used quite easily in many kinds of graphs which all have their own advantages. Therefore it is important to choose a graph that best suits the needs of the game A* is applied to. (Anguelov 2011.)

5.1 Grid graph

Grid graph is a very common representation used in games. This is simply because game worlds consist of different types of tiles which can then be used like a grid graph. Grid graphs are very useful when different tiles have different sets of rules that affect the path the algorithm calculates. Grid graph also allows multiple ways for the units to move in the game, since they can use the tiles' center points, line centers or even the joining points of the tiles. (Patel 2014.)

Grid graph does have its bad sides also. It is not very optimal when the map sizes get very large. The amount of nodes the algorithm has to go through gets larger so the processing speed may not be the best. If the movement of the units is not constrained to tile movement, then searching through large open areas node by node is quite useless using grid graph, when a large amount of those nodes could be skipped in the calculations. When A* searches through that kind of an area using grid graph, it is not very optimal and it also produces jagged movement. (Patel 2014.)

5.2 Polygonal graph

Polygonal graph is a great easy-to-use alternative to grid graphs. Polygonal graphs is especially useful when the units in the game can move to any point in the game world from another point. In polygonal graphs, the nodes are placed either manually or by using the information different game objects provide. (Patel 2014.)

Nodes placed in polygonal graph do not necessarily have any information whether they are blocked somehow, if they are leading to a dead end and what they are connected to.

Therefore it is the maker's responsibility to make sure that the nodes have all the information they need and that they are somehow available to other nodes. The actual connections between nodes are usually determined by a visibility check. Visibility check calculates which other nodes a specific node sees from its location. Graphs that are created this way can be very fast and simple, since they have only the needed amount of nodes in them. This means that large open areas can be crossed very fast in the algorithm, since there does not need to be additional nodes in the middle of that area. (Patel 2014.)

Downsides of the polygonal graph are the implementation and the complexity management. If the polygonal graph is crafted by hand, meaning the nodes are placed manually, it can be quite a heavy task to do if the game has multiple levels. It would require the maker to place the nodes for each of the levels, which could be very time consuming if the levels are very large or if there is just a large amount of them. Also if the level has a lot of nodes in a small area, the visibility calculations can make a very tangled web of the connections between the nodes. This mainly uses more memory and is therefore quite costly. (Patel 2014.)

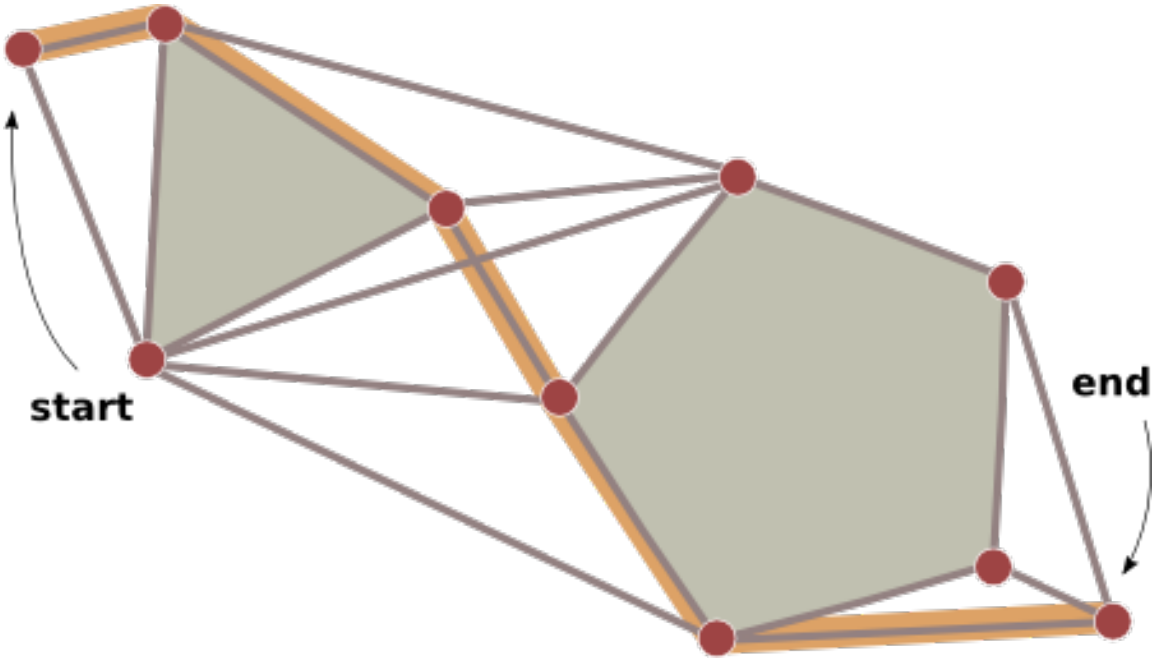

Figure 10. Nodes placed in a polygonal graph (Patel 2014)

5.3 Navigation mesh graph

Navigation meshes are maps created from the game world data. It uses the mesh data from the static objects in the level and calculates a mesh for itself. Since each mesh consists of different amounts of triangles, the navigation mesh uses those as tiles that are then used in the A*. Navigation mesh behaves much like a grid graph with the exception that the tile sizes are not preset and therefore they can be modified when creating the map so that large areas one big tile, making it sort of a combination of the grid graph and the polygonal graph. (Patel 2014.)

Navigation meshes are great when there are a lot of levels or very large maps. This is because navigation meshes are usually calculated beforehand programmatically and they are then stored as another file. This file is then read when the pathfinding starts and A* uses the data provided by the navigation mesh to find routes. Because the mesh uses data provided by the game world, it can store additional data easily, such as varying movement costs or other info the unit might need to know about that specific tile. (Anguelov 2011.)

The creation of navigation data based on the world data is the most efficient way to represent complex environments, since it does not matter how the world is structured for it to make connections based on the principles that have been set for it. Navigation meshes are able to minimize the amount of nodes effectively and there will be no unnecessary connections since visibility checks do not have to be made. (Anguelov 2011.)

The major disadvantage navigation mesh has is that it is not very well suited for dynamic environments where the accessibility of a node might change due to other gameplay factors. This is simply because the maps are pre-calculated and very heavy, so there is usually no way to redo them at runtime. (Anguelov 2011.)

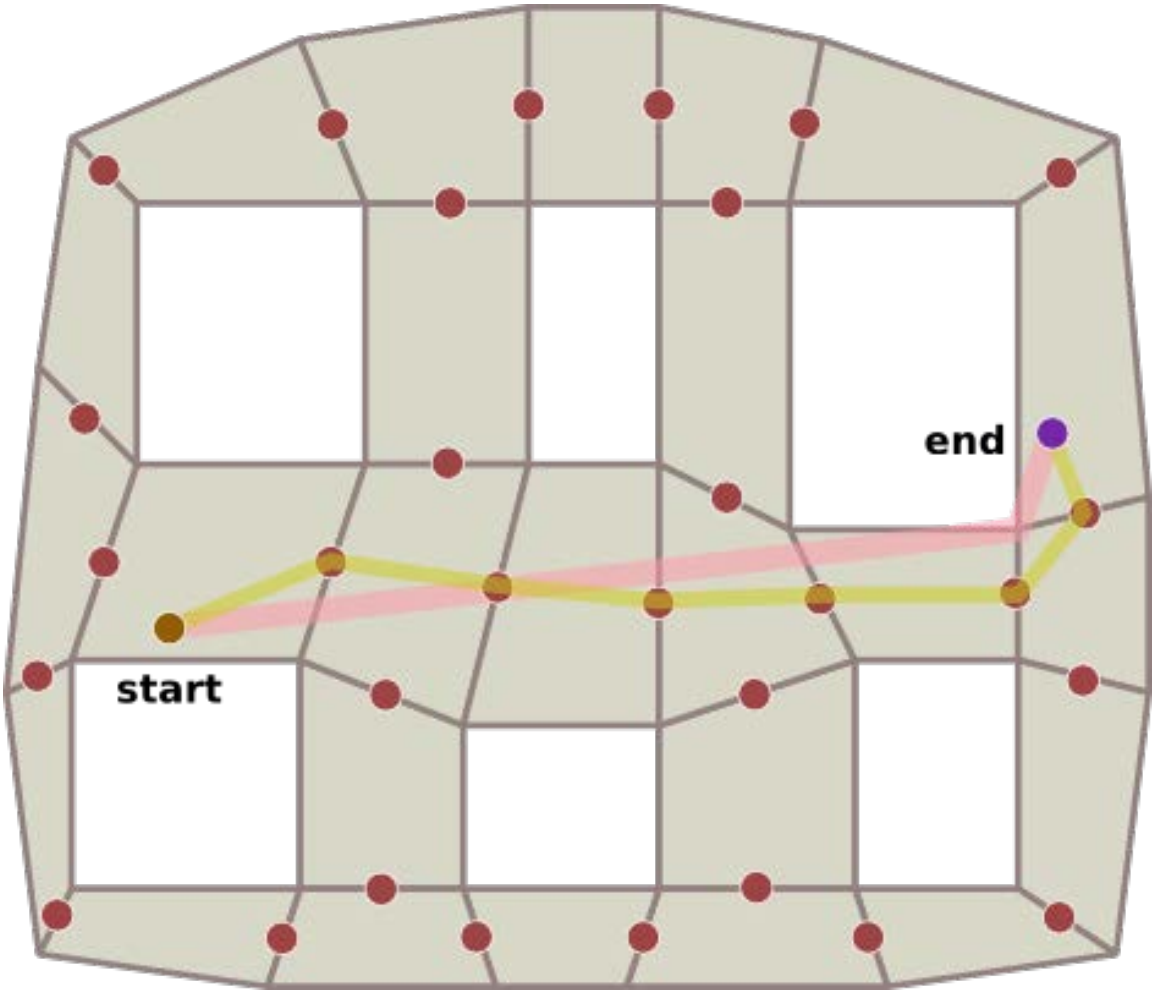

Figure 11. A navigation mesh based on the level geometry (Patel 2014)

6 OPTIMIZATION TECHNIQUES

As stated before, pathfinding can be quite costly for a game's performance considering the memory usage and execution speed. There is, however, ways to speed things up and make the algorithm run more smoothly. Some of these include better container options, optimization of the heuristic function and modification of the search area used in the algorithm. (Anguelov 2011.)

6.1 Open list operations

The open list operations in A* are expensive. This is because the search uses certain operations inside it, which are time consuming. A* constantly finds the best node from the open set and removes it. When checking the adjacent nodes, A* goes through the open set to see if the node already exists in the set and inserts it if it does not. The last operation is when the adjacent node exists in the open list but it gets a better value using the current route, then the values of that node need to be changed. If the container sorts the nodes based on their F value, in the worst case scenario the node has to be removed and inserted again with the new values. (Patel 2014.)

The usual containers, such as arrays, are not very well fit for these requirements. The dynamic changes in the sets and the required iteration operations make simple implementation slow and inefficient. A generic way to deal with all of the issues is to implement a binary heap, which uses indexing to sort and access nodes quickly. A binary heap works pretty well in most cases and is therefore recommended for a generic use of the A*. (Anguelov 2011.)

It is important to note that even though A* loop adds requirements to the handling of the open set, over optimizing the open set handling may not always work in the most favorable way. If the set implementation uses sorting and is ordered, the time spent in the optimization functions may be more expensive than simply iterating through the set, turning it into a de-optimization. (Anguelov 2011; Patel 2014.)

6.2 Heuristic function

In addition to the costly open set operations the one calculation that marks the optimality of the algorithm is its heuristic function. In some cases the cost of calculating the heuristic value for all the searched nodes may outweigh the cost that the open list operations take. (Anguelov 2011.)

By increasing the accuracy of the heuristic function fewer nodes are added to the open set making the algorithm perform faster. This means that the result of the heuristic function should be as close to the actual path cost as possible, not the distance. The other restriction for the heuristic function is the optimality of the function itself. Complex calculations make the heuristic run slower, so the function has to be optimal. Other way to optimize it is to cache the result of the function after the first calculation since it cannot change during a search. (Patel 2014.)

6.3 Search optimizations

A* can be optimized in other ways too. Some of these include modifying the search area in some way and modifying the search itself a little bit. Using them may affect the accuracy of the algorithm but as stated before, it is usually enough to have movement that looks ok instead of being perfect to give the A* a better performance. (Anguelov 2011; Patel 2014.)

The search area has a few possible ways to be optimized. One way is to limit the search area to a smaller set of nodes based on the start and ending positions ignoring all the nodes that are not included in that specific area. Other way is to use hierarchical maps, where the game world is divided into different layers to make the search areas stay inside a certain limit. In this way the game may use a world map to move greater distances and smaller maps to present the normal movement of players, for instance in a city. The A* is more effective with smaller sets and therefore it is usually unnecessary to know smaller navigation points when moving in a larger scale. (Patel 2014.)

The algorithm itself can be modified in many ways to make it more optimal. One example of this is the fast expansion of the successor nodes. The fast expansion means that when going through the adjacent nodes of a certain node, the adjacent node's F value is compared to the

current node's F value. If the F value is the same as the current node's, that node is immediately expanded without it being ever added to the open list. This is done because it is likely that the adjacent node would anyway be the next one to be searched. Since this saves some computational tasks from the open list operations, it is a fairly good way to optimize the A* in some cases. (Anguelov 2011.)

7 PROJECT DOCUMENTATION

The practical part of the thesis focuses on pathfinding in a 3D-environment using Unity3D game engine. 3D pathfinding differs from 2D pathfinding in one major way, which is the third dimension. The programming needed to implement it is not that far from the 2D one, but since the third dimension makes the game world so much more dynamic it does add some minor calculations to the algorithm.

The aim is to analyze the results of the pathfinding by comparing the optimality of the found routes and the execution speed of the algorithm. To get more details to determine these factors, two different tests are made. The first is a stress test to see how the execution speed and the precision of the algorithm vary when using different values in the algorithm. The second one shows the algorithm in action in a 3D environment with two different implementations.

7.1 Preparations

To do pathfinding, three things are required: a representation of a navigation node in the world, the pathfinder that uses a pathfinding algorithm for the given nodes and some kind of a test environment where the pathfinding can be executed. Since the tests will require different setups, the implementation of the node needs to be so that it can be placed on a grid graph or a polygonal graph. This also puts some requirements for the map implementation so that it can also display information using both types of graphs.

7.1.1 Node class

The node holds information about itself that is needed for the pathfinding algorithm. The basic required variables are the G, F and H values the node has during the search. A node also needs a reference to the node it is connected to which is called its parent node. The parent node is needed when the search is over and the path is constructed.

To make a more generic implementation that supports different graphs, the neighboring nodes are stored inside the node. These nodes mark all the nodes that can be traversed to from this particular node. The node class also stores information if the node is unreachable or blocked somehow. A ground multiplier is also set for the node, which is used in the second test to determine different ground types that have different costs. That way it is possible to make preferred roads the pathfinder uses.

7.1.2 Pathfinder class

The pathfinder class is the implementation of the pathfinding algorithm. A basic A* algorithm is used for these tests without any major modifications to the algorithm itself to make it run smoother. The simple implementation was chosen to demonstrate how the algorithm works in its most crude form.

Pathfinder class consists of two static functions: the search itself and path building. The search function follows the A* core loop. It creates the open and closed node sets and adds the start node to the open set. It then follows the loop where it gets the best node out of the open set, adds it to the closed set and searches through its neighboring nodes. The neighboring nodes get their updated cost values and are added to the open set if they are not already added. The search ends immediately when the node removed from the open set is the end node. Once the end node is picked, the path is constructed with the path building function following the parent nodes starting from the end node.

For visual feedback the nodes are colored to demonstrate how the search has operated:

- Grey nodes are unsearched nodes.
- Black nodes are blocked nodes.
- Blue nodes are searched nodes that have been added to the open set.
- Magenta nodes are visited nodes.
- Yellow nodes are part of the found path.
- Green is the starting node.

- Red node is the end node.

Since pathfinding needs to be optimal, the implementation also has a modified version of the search which uses a priority queue instead of a simple dynamic array. The priority queue uses a heap to give performance for inserts and removals. It also sorts the sets fast keeping them in order. Using a priority queue as a comparing point gives a great example how easily the performance of the algorithm can be increased.

7.1.3 Map setup

With the two different tests, the map has to support two different kinds of graphs. The node class takes care of much of the logic of providing such a possibility, but the area on which the nodes operate has some handling for itself. For these testing purposes Unity's terrain object was used to enable an environment which is easy to modify.

The stress test requires an area where a large amount of nodes can be placed in a grid graph and the second test needed an environment with some height differences and varying ground types to simulate a more dynamic level. Unity's terrain has some crafty tools to do the latter while it can also be used as a simple plane. As a plane the terrain works perfectly for the stress test environment. The terrain tools allow easy terrain modifications to create hills and ponds and also a way to paint textures directly on the terrain.

When the nodes are placed on the terrain, texture data is read from the terrain data. This texture data is used to determine the ground multiplier of the nodes, so that the traversing costs to certain nodes are larger than for others. This way roads drawn on the terrain are preferred when the pathfinder searches for the most optimal path. This does not guarantee the usage of roads however, which is intended behavior since sometimes a detour may be a better option.

7.2 Pathfinding tests

The tests that simulate the algorithm work a little differently from each other and they are supposed to analyze different things. The stress test demonstrates how the A* performs

with a large amount of nodes whereas the second test rather observes the behavior using the best applicable methods.

7.2.1 Stress test

The stress test environment consists of 10000 nodes on a grid graph. The nodes are placed in a 100x100 grid where each one of them is connected to the nodes surrounding them. This allows movement in eight directions. Some of the nodes in the grid are then marked blocked to give the algorithm some obstacles it has to get around to get to the end node. In the scenario the pathfinder searches for a path from the upper left corner to the lower right corner of the map. The setup can be seen in figure 12.

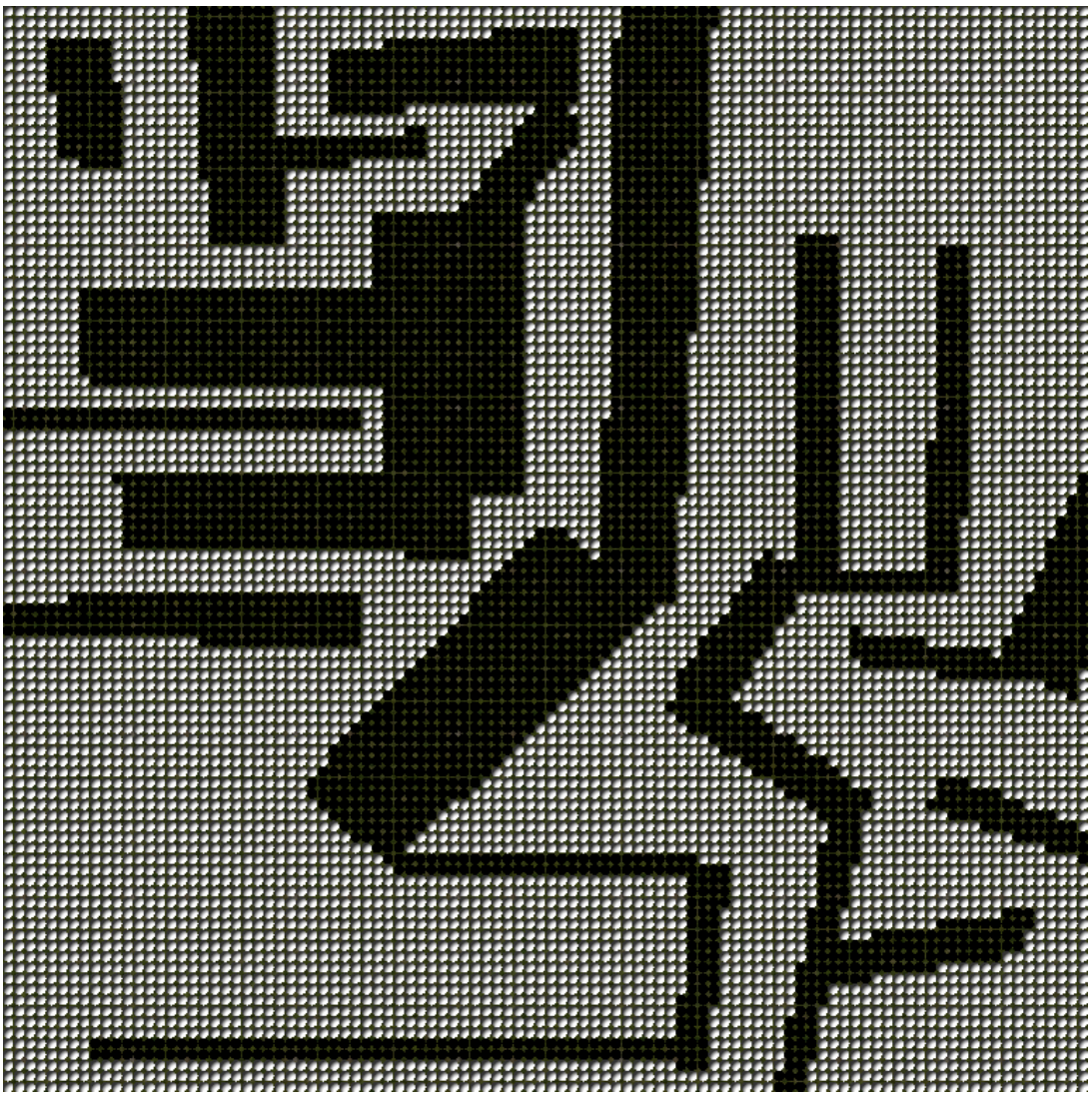

Figure 12. Stress test setup

The stress test determines what kind of paths we can produce using three different heuristics: Manhattan distance, Diagonal distance and Euclidean distance. Using different heuristics should produce different kinds of paths. The pathfinder also searches a different amount of nodes depending on how accurate the heuristic estimate is. The test is run a total of ten times for one heuristic, where five tests use the dynamic array and five use the priority queue for the open sets.

Manhattan distance uses the delta difference on the X axis and the Y axis between the searched node and the end node to produce the heuristic cost. The two values are added together to form the H value of the node. Since the heuristic does not guarantee that the estimate is always shorter or equal to the actual cost the Manhattan heuristic is not an admissible heuristic. Using this kind of a heuristic means that it might not produce the shortest path but it should perform slightly faster under favorable conditions. In this particular scenario the heuristic works pretty well and produces a clean path without any odd looking behavior. Using the dynamic array the pathfinding took an average of 1.8900146 seconds and using the priority queue only 0.02337647 seconds. You can see the found path in figure 13.

The Diagonal distance works similarly to Manhattan distance, but it takes the possibility of diagonal movement into account when calculating the heuristic estimate. This makes the H value to be closer to the actual cost to get to the end point but it also cannot guarantee a shortest path. This heuristic also produces a quite nice path in the test while the execution time average is 1.9288332 seconds for the dynamic array and 0.02272949 seconds for the priority queue. Path found using the diagonal distance can be seen in the figure 14.

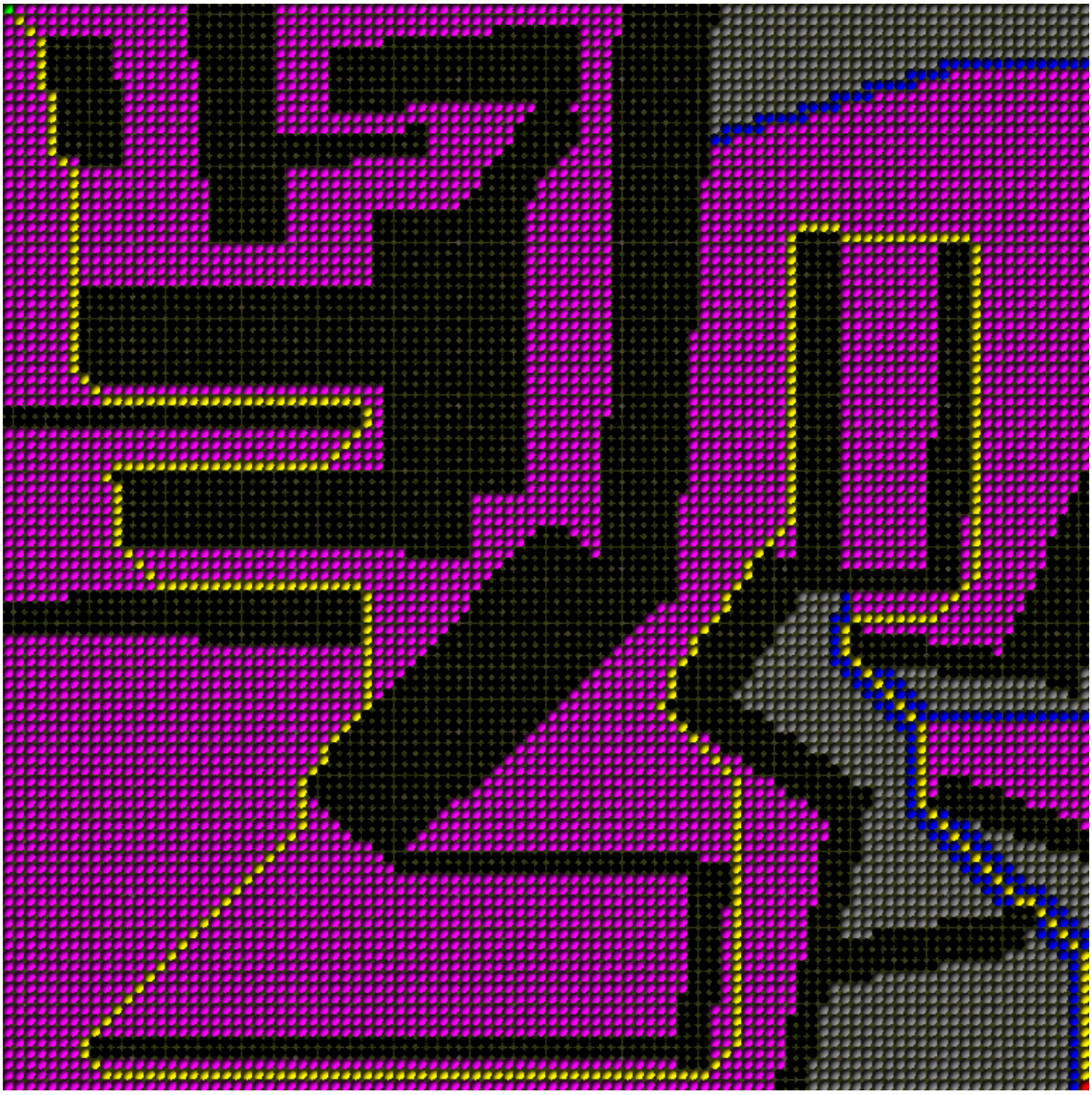

Figure 13. Path found using Manhattan distance

The last heuristic function is the Euclidean distance. Euclidean distance gets the direct line length from the searched node to the end node and uses it as the estimate. This makes the heuristic an admissible heuristic which is guaranteed to produce a shortest path. Euclidean heuristic is important because it is the one usually used in 3D environments when searching for a path. The problem with the Euclidean distance is that it makes the A* search nodes in a larger area than the Manhattan or Diagonal distance. The computational cost of the heuristic is also a bit heavier since it has to make an expensive square root operation. This heuristic does find a guaranteed optimal path that looks nice which took 2.06105596 seconds on average using the dynamic array and 0.02224121 seconds using the priority queue. Figure 15 shows the path.

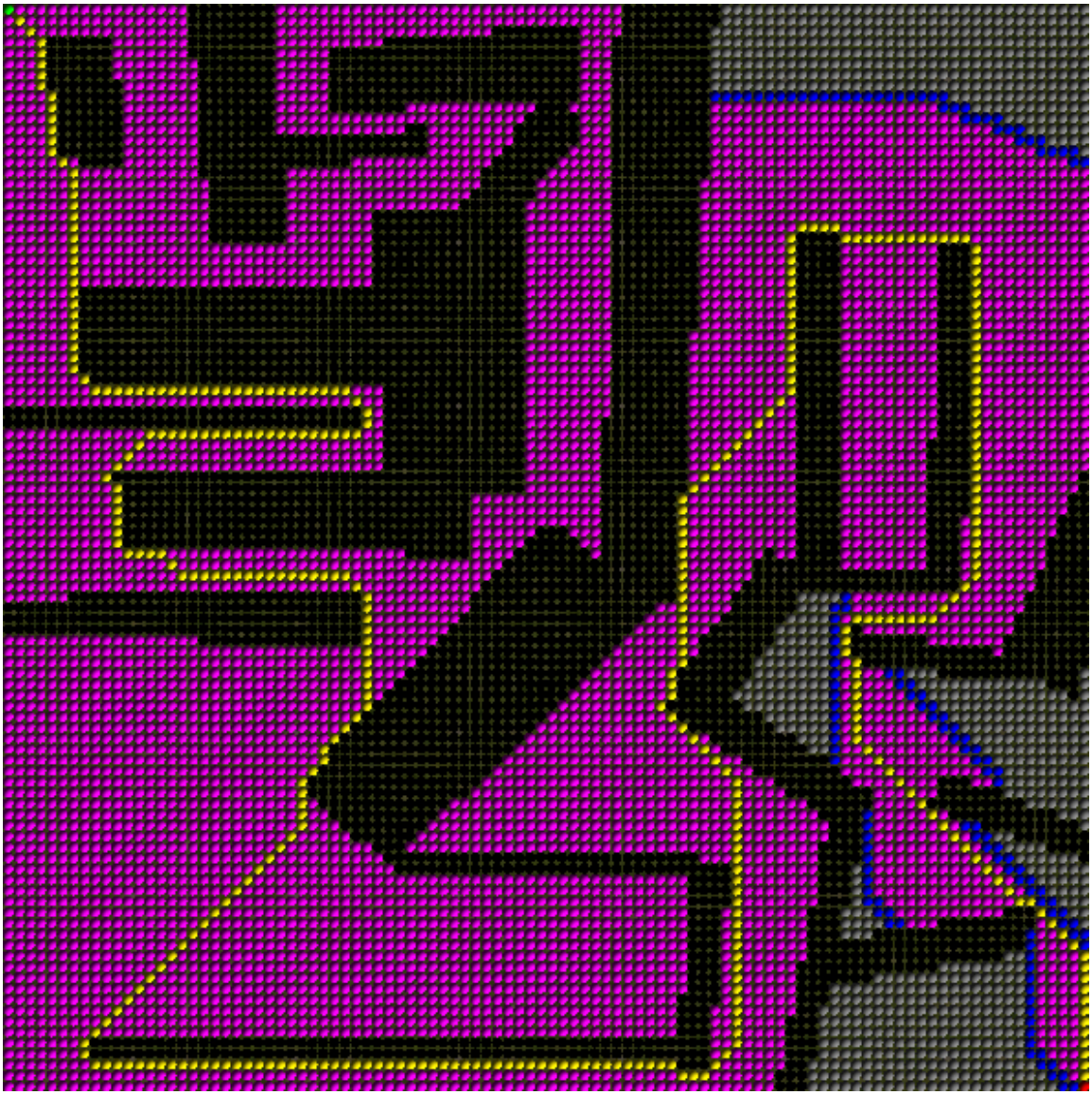

Figure 14. Path found using Diagonal distance

The difference between using a dynamic array and a priority queue is enormous. Priority queue performed 80 to 92 times faster than the dynamic array in all of the tests executed. All of the heuristics produced paths that seemed like acceptable behavior for an AI unit in a game. Figure 16 shows more detailed breakdown of the execution times. The green cells show the average time it took to run the test using the given parameters.

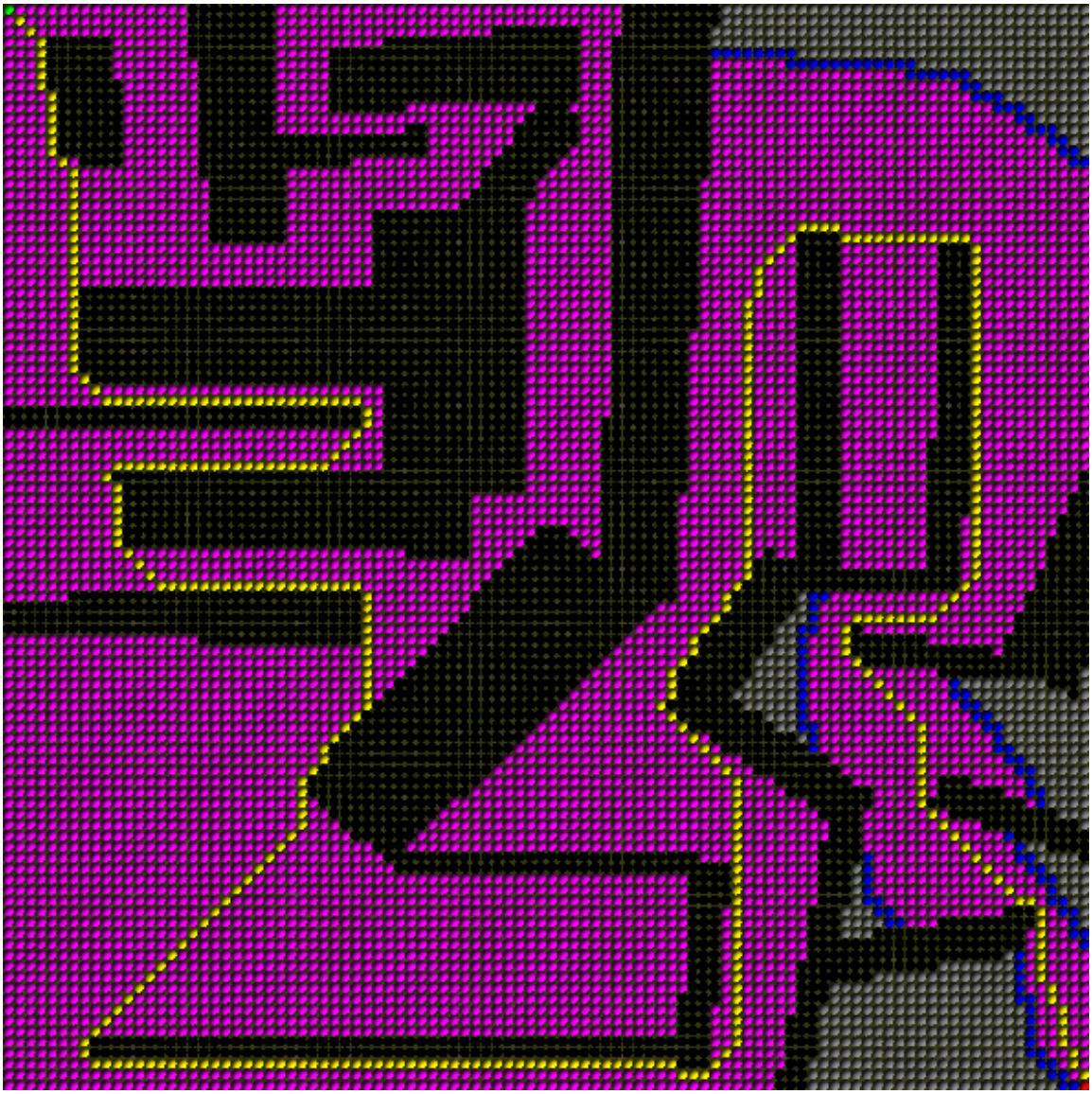

Figure 15. Path found using Euclidean distance

Manhattan		Diagonal		Euclidean	
Dynamic array	Priority queue	Dynamic array	Priority queue	Dynamic array	Priority queue
1,88269000	0,02435303	1,93579100	0,02258301	2,05871600	0,02185059
1,89758300	0,02252197	1,92504900	0,02246094	2,06958000	0,02197266
1,87396200	0,02264404	1,94201700	0,02282715	2,06030300	0,02233887
1,89874300	0,02374268	1,92004400	0,02282715	2,05944800	0,02258301
1,89709500	0,02362061	1,92126500	0,02294922	2,05725100	0,02246094
1,89001460	0,02337647	1,92883320	0,02272949	2,06105960	0,02224121

Figure 16. Table of the execution times of the algorithm

7.2.2 Application in a 3D-environment

The stress test results showed that the priority queue is an obvious choice for the open set operations using the chosen setup. For the 3D environment application the terrain had to be modified to represent a 3D game more properly. Using Unity's terrain tools, the terrain was changed to include some height differences and different types of ground. The result is shown in figure 17.

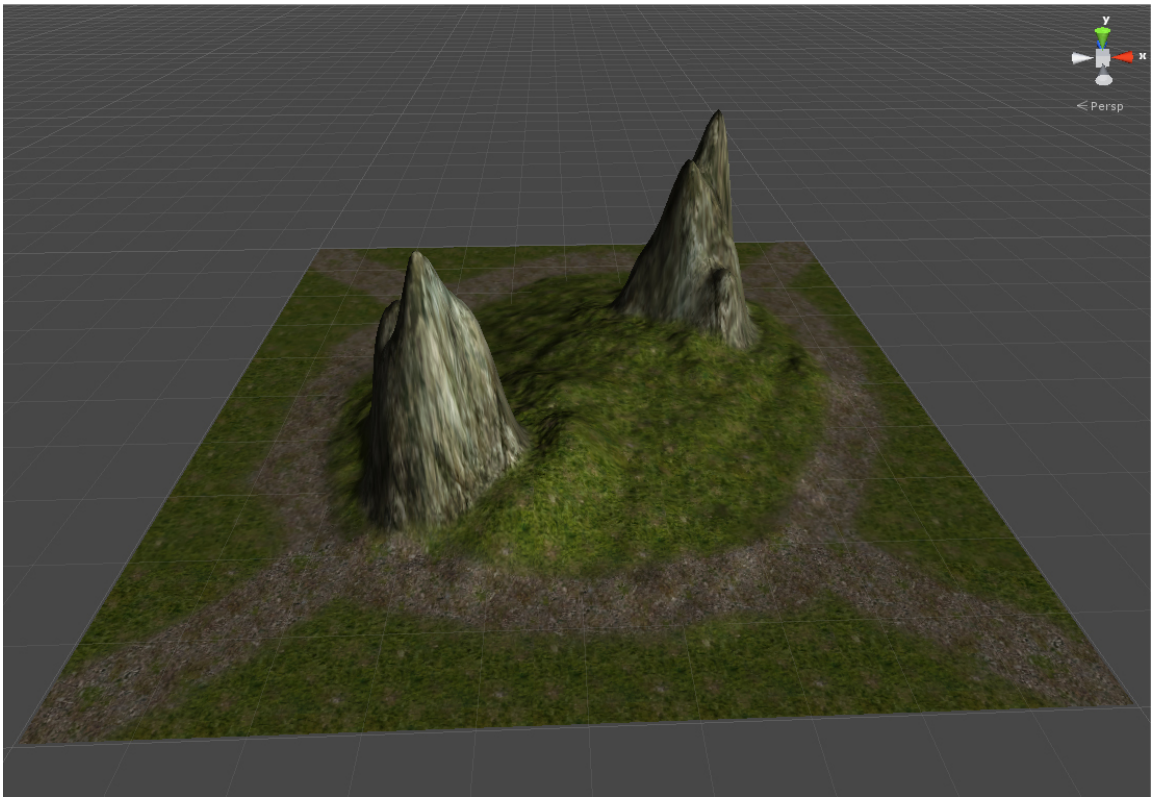

Figure 17. Test environment for the 3D environment test

Pathfinding can be used in different kinds of graphs. Usually 3D games with free movement use either polygonal graphs or navigation meshes for their pathfinding since they perform faster than a grid graph and fit better to the game world. The calculation of navigation meshes is quite tedious and time consuming so this test focuses on comparing the polygonal graph and grid graph in a 3D environment.

Using the same node generation as the stress test, 10000 nodes are placed on the terrain taking the shape of the terrain into account. The nodes are connected to the neighboring nodes regardless how high or low they are on the map. The nodes then use the data provided by the terrain to determine whether they are on the grass, the road or the mountain

rock. Nodes are then colored accordingly so that it is easier to see how they are located as shown in figure 18.

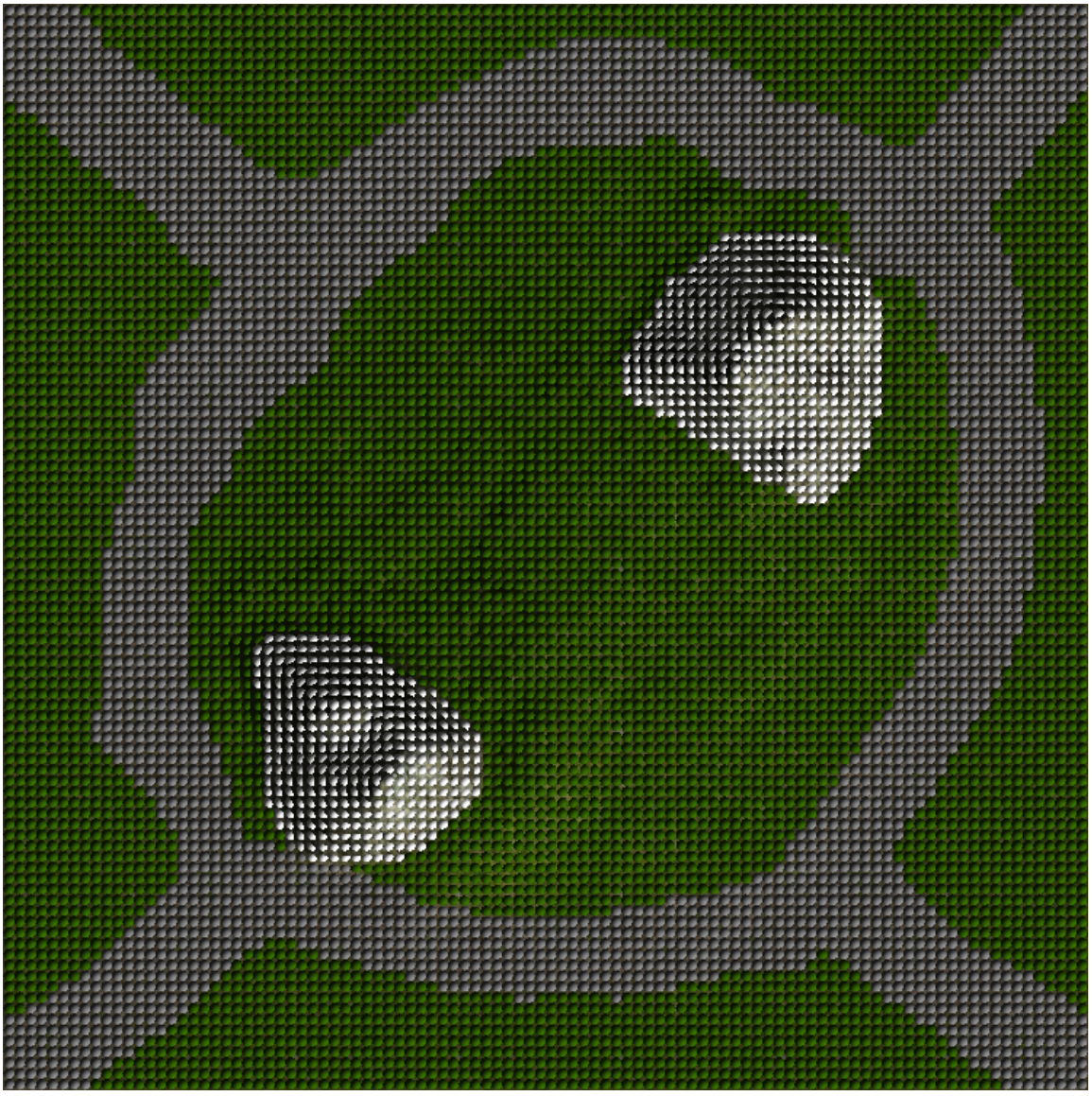

Figure 18. Grid graph representation viewed from above

In this setup nodes located on a road have normal costs. Grass nodes have a multiplier of .25 added to them to simulate slower movement and the mountain rock nodes have a multiplier of .75 added to them to mark them as places to avoid. Even though the map does not have any obstacles the A* should still avoid a straight path from one corner to another if a faster path exists by going around something.

Because the third dimension would make the usage of Diagonal and Manhattan distances produce costs that are quite far from the truth when talking about node to node travel, the

chosen heuristic is the Euclidean distance. Euclidean distance works exactly the same as it does in a 2D environment so no extra work is needed to make it work. The A* does take a small modification to make uphill and downhill movement more realistic so that the pathfinding will take them into account when searching for a path. The G cost is modified by the delta y value between the adjacent node positions. This way uphill movement is more expensive and downhill movement cheaper.

Using the grid graph the algorithm searches for a best path between nodes in the opposite corners of the map. A* performs similarly as it did in the stress test using the Euclidean heuristic and priority queue. Instead of going over the mountains to get a straight path the algorithm rather finds a faster way by following the road. When the start and end nodes are switched to other corners the algorithm faces a dilemma where it has to check whether a path straight between the large mountains with height differences produces a better route than by using the roads. These two situations are shown in figures 19 and 20.

The ten thousand nodes may be a slight overkill in this situation. There are not that many places in the map that needs difficult maneuvering skills and it is basically one large area without any obstacles. The amount of nodes can be easily reduced by introducing a polygonal graph. In the polygonal graph only a needed amount of nodes are placed on the map in strategic positions that require a node to guide the movement of a unit. In the map used in the test this means that nodes placed so that movement goes around the mountains.

Placement of the nodes is easy and they can be placed anywhere. In this map nodes were first placed in certain intervals on the roads and afterwards with sparse spacing on the grass area. A total of 75 nodes were placed to demonstrate the routes found by using a polygonal graph. The nodes made an easy visibility check to see which nodes they are connected to. Because of the way the pathfinder and node class were implemented, they required no additional changes to work in a different graph. The polygonal graph is displayed in the figure 21.

Figure 19. 3D environment search using a grid graph

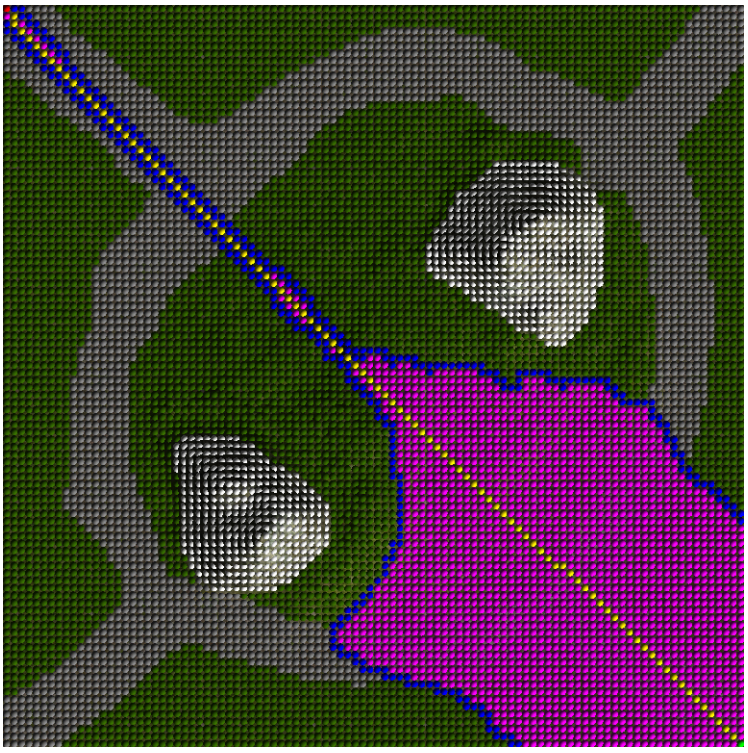

Figure 20. 3D environment search using a grid graph

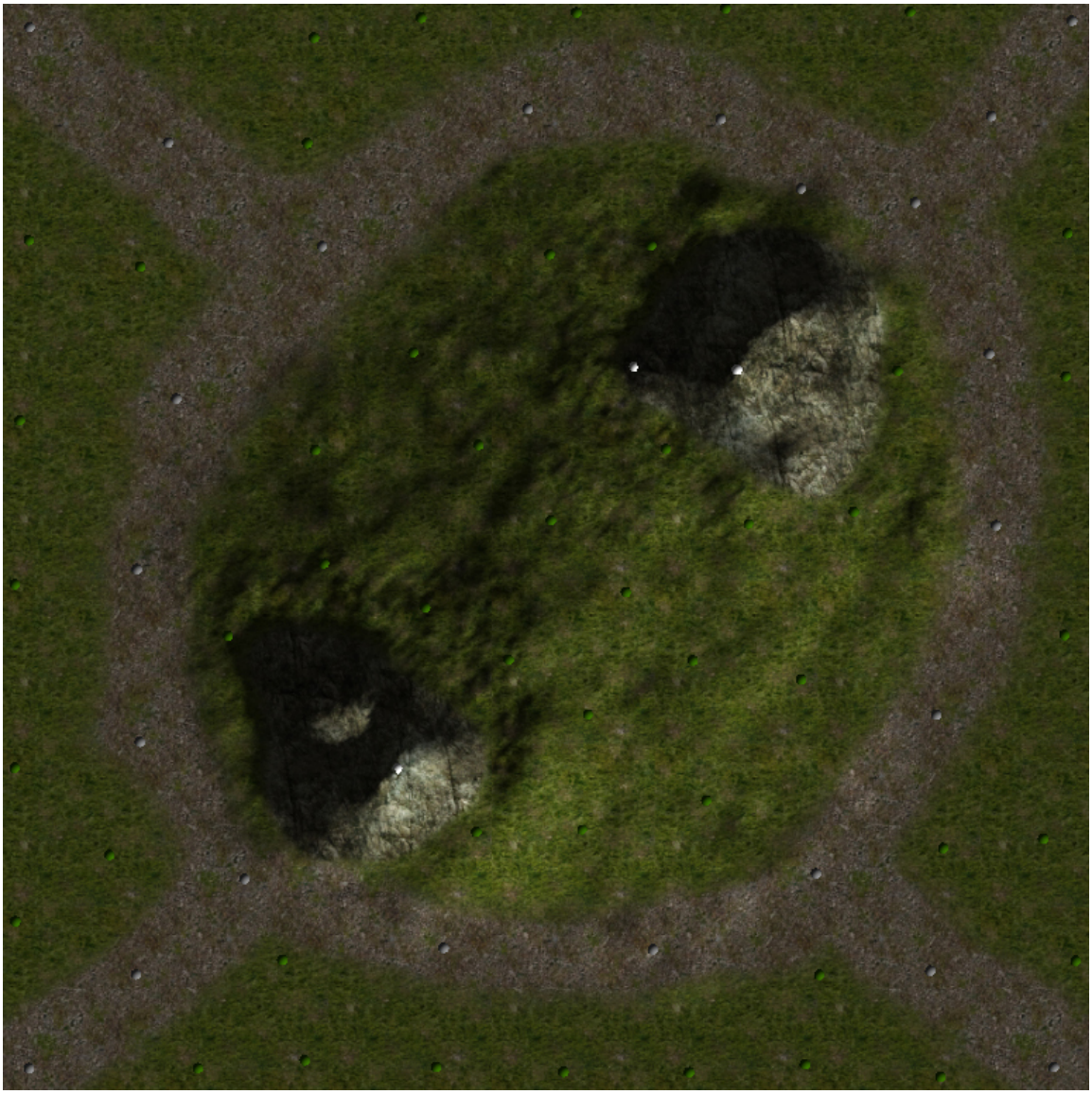

Figure 21. Polygonal graph representation of the nodes

Even though the amount of nodes was reduced to a less than one percent from the grid graph implementation the A* worked well and produced similar paths. The first scenario produced a path which followed the roads and in the second scenario the best path went between the mountains. When the amount of nodes was reduced the algorithm performed four to ten times faster. Figure 22 shows the path that was found for the second scenario using the polygonal graph.

Figure 22. Path found using a polygonal graph

Pathfinding in a 3D environment does not differ that much from 2D pathfinding. It does however enable more possibilities to introduce intelligent behavior to games. By using a setup that best suits the game in question the A* will also work very efficiently.

8 CONCLUSIONS

The objective of this thesis was to show and analyze the behavior of the A* pathfinding algorithm in a 3D environment. Since pathfinding as a subject is quite large, the work was limited to a very simple implementation of A*. In this sense the thesis was a success. The practical part of the thesis did show and analyze the behavior of the algorithm in different environments where a lot of data was gathered to support the theoretic part of the thesis.

The initial design for the practical part was to demonstrate more ways to improve the algorithm's performance speed, but after learning there were a myriad of different ways to improve it, it became obvious it was beyond the scope of this thesis. The creation of the test project also took long to make. The major reason for this was the requirements for it to run two different grid types to properly show the functionality of the two. Also the amount of ideas for the analyzable parts of the algorithm was bloated with ideas that were in no way achievable in the given time.

In the beginning writing the theory and figuring out the full context of the practical part took very long. Before being able to write theory it had to be learned so that it was not just writing of other people's thoughts. The algorithm itself also includes a lot of logic to be understood.

All in all, the writing of this thesis was a great learning experience. It taught the writer a lot of new information about pathfinding that was not known before. Therefore, it is important to note that the information provided by the thesis is not to be taken as the only valid option to do pathfinding. Even though the test showed that using a polygonal graph instead of a grid graph produced similar routes with faster processing time, it does not mean that the grid graph does not have its places of use.

In summary, the thesis did serve its purpose by providing information of the behavior of A* and by testing and analyzing its use in a 3D environment. The thesis did not, however, give examples of all the possible ways to make a pathfinding algorithm work more efficiently but it does give a good base to append to.

REFERENCES

Anguelov, B. 2011. Video game pathfinding and improvements to discrete search on grid-based maps. (Master of Science, University of Pretoria). , 62. Url: http://takinginitiative.files.wordpress.com/2011/05/thesis.pdf (read 24.6.2014)

Buckland, M. 2005. Programming Game AI by Example. United States: Wordware Publishing, Inc.

Coles, A., & Smith, A. 2007. Greedy best-first search when EHC fails. Url: http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume28/coles07a-html/node11.html#modifiedbestfs (read 24.6.2014)

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs.1, 269. Url: http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf (read 24.6.2014)

Drichel, A. 2005. Breitensuche.png. Url: http://commons.wikimedia.org/wiki/File:Breadth-first-tree.png

Eranki, R. 2002. Pathfinding using A* (A-star). Url: http://web.mit.edu/eranki/www/tutorials/search/ (read 24.6.2014)

Esser, W. 2005. Tiefensuche.png. Url: http://commons.wikimedia.org/wiki/File:Depth-first-tree.png

Lester, P. 2005. A* pathfinding for beginners. Retrieved from http://www.policyalmanac.org/games/aStarTutorial.htm (read 24.6.2014)

Madhav, S. 2013. Game Programming Algorithms and Techniques. United States: Addison-Wesley Professional

Patel, A. 2014. Amit's A* PAges. Retrieved from http://theory.stanford.edu/~amitp/GameProgramming/ (read 24.6.2014)

Puthuparampil, M. 2007. Report dijkstra's algorithm. (New York University). Url: http://www.cs.nyu.edu/courses/summer07/G22.2340-001/Presentations/Puthuparampil.pdf (read 24.6.2014)