Ministry of Education, Culture and Research

Technical University of Republic of Moldova

Faculty of Computer Science and Microelectronics

Department of Software Engineering and Automatics

Pathfinding- A* algorithm

Course work

At Algorithms' Design and Analysis

Performed by: Daniela Cojocari

Vlad Bantuș

St. gr. FAF-161

Checked by: Mariana Catruc

Senior lecturer

Chișinău, 2017

CONTENT

The aim: Analyzing A* algorithm and visualizing the shortest path

Objectives:

1. Comparing pathfinding algorithms
2. Highlighting advantages and usage or A* algorithm
3. Visualizing the result of A* algorithm using Unity 3D

1. Introduction

Nowadays we are living in the age of technologies, innovations and great inventions such as the location with GPS have become an integral part of everyone's life. Because of great result of finding the needed destination on the shortest path and moreover, because of saving money and time the number of the GPS users increased and is still increasing very rapidly. As a result, it is hard to imagine how the world could function without the GPS systems. For example, being in a new area (a city, a forest etc.) and having a meeting or being as a tourist, or more, imagine a road full of traffic jams or even being blocked in a labyrinth, without the GPS system the solution will not be easily found. With all said, the major advantage of the GPS over the traditional searching route to the target destination is the speed of action. GPS system consists of two basic components – digital maps and the shortest path search algorithms and the ordinary user does not even realize that whole system is based on algorithms.

2. Pathfinding concept and historical facts

2.1. Pathfinding algorithm

Pathfinding or **path search algorithms** are used to solve the problem of finding a traversable path through an environment with obstacles. This is a problem debated in many different fields of study such as robotics, video games, traffic control or even human decision making.

There have been numerous attempts at pathfinding in the past. For example, 'Travelling salesmen' problem which was mentioned by William Rowan Hamilton in the 1800s, one of the earliest approaches of pathfinding. The problem dealt with finding the shortest route through multiple cities by visiting them only once, which gained more attention only in the 1900s and solutions to this problem were encouraged, and one of the earliest solutions was known as the Depth First search, developed by Charles Pierre Tremaux. Following this, there appeared many new pathfinding algorithms such as – Breadth First Search, Bellman- Ford's, Dijkstra's and A*, which have a big usage nowadays.

Path problems were also studied at the beginning on the 1950's in the context of 'alternate routing', that is, finding a second shortest route if the shortest is blocked. On period 1946-1953, ***matrix methods*** were developed to study relations in networks, like finding the closure of a relation; that is identifying in a directed graph the pairs of point s, t such that t is reachable. Another approach, ***linear programming*** was during 1955-1957 and was presented by Dantzig and Minty like 'String model'. Later in 1959 appeared first $O(n^2)$ algorithm, well-known today as Dijkstra's algorithm, and in 1968, inspired by and improved on Dijkstra's Algorithm, was released A* algorithm.[9][10]

2.2. A* algorithm's history

A* algorithm was first described in 1968 by Peter Hart, Nils Nilsson and Bertram Raphael, as an extension of Edsger Dijkstra's 1959 algorithm.

In 1964 Nils Nilsson invented a heuristic based approach to increase the speed of Dijkstra's algorithm. This algorithm was called ***A1.*** In 1967 Bertram Raphael made dramatic improvements upon this algorithm, but failed to show optimality. He called this algorithm ***A2***. Then in 1968 Peter E. Hart introduced an argument that proved A2 was optimal when using a consistent heuristic with only minor changes. His proof of the algorithm also included a section that showed that the new A2 algorithm was the best algorithm possible given the conditions. He thus named the algorithm in Kleene star syntax to be the algorithm that starts with A and includes all possible version number or ***A*.***[10]

3. A* algorithm features

3.1. A* using heuristic function

Modern pathfinding algorithms used in games, are based on the Dijkstra's algorithm and the most popular is **A* algorithm**, because it is an improvement on Dijkstra as it uses heuristic to ensure better performance of the pathfinding. **Heuristic technique** is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy or precision for speed. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution. In a way, it can be considered a shortcut.[4][3]

A heuristic function, also called a heuristic is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow. It may approximate the exact solution. So, if the heuristic defined is the lower bound of the minimal cost

from the source note to the destination, the A* would find the minimum cost path to the goal. It is important to choose a good heuristic function.

3.2. A* algorithm's three 'scores'

Before analyzing advantages heuristic function is needed to mention node. **A node** has a positioning value (eg. x, y), a reference to its parent and **three 'scores'** associated with it. These scores are how A* determines which node to consider first.

The g score is the base score of the node and is simply incremental cost of moving from the start node to current node:

$g(n)=g(n.parent)+cost(n.parent,n)$

cost (n_1, n_2) = the movement cost from n_1 to n_2

The h score - the heuristic. Coming back to heuristic function is possible to highlight its advantages. The heuristic can be used to **control A*'s behavior**:[4]

1. If h(n) is 0, then only g(n) plays a role, an A* turns into Dijkstra's Algorithm, which is guaranteed to find a shortest path.
2. If h(n) is always lower than (or equal to) the cost of moving from n to the goal, then A* is guaranteed to find a shortest path. The lower h(n) is, the more node A* expands, making it slower.
3. If h(n) is exactly equal to the cost of moving from n to the goal, then A* will only follow the best path and never expand anything else, making it fast. A* will behave perfectly, if is known the given information.
4. If h(n) is sometimes greater than the cost of moving from n to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.
5. At the other extreme, if h(n) is very high relative to g(n) then only h(n) plays a role, and A* turns into Greedy Best- First-Search.

In a game, the properties of A* can be very useful. For example, in some situations, it would be needed to have a "good" path more than a "perfect" path. To shift the balance between g(n) and h(n), it possible to modify either one.

Another point to mention is the importance of scale in A* algorithm. Opening another parenthesis A* computes f(n) = g(n) + h(n), the third score, **the f score**, which is simply the addition of g and h scored and represents the total cost of the path via the current node. To add two values,

those two values need to be at the same scale. If g(n) is measured in hours and h(n) is measured in meters, then A* is going to consider g or h too much or too little, and the result won't get as good paths or A* will run slower than it could.

3.3. A* algorithm involving the distance heuristic

Analyzing heuristics for grid maps, is needed to check **the distance heuristic** that matches the allowed movement:

1. On a square grid that allows **4 directions** of movement, uses Manhattan distance (L_1).
2. On a square grid that allows **8 directions** of movement, uses Diagonal distance (L_∞).
3. On a square grid that allows **any direction** of movement, is needed or maybe not needed the Euclidean distance (L_2), depends on personal usage. If A* is finding paths on the grid but it is allowing movement not on the grid, possible to consider other representations of the map.
4. On a hexagon grid that allows **6 directions** of movement, uses Manhattan distance adapted to hexagonal grids.

In this work is considered 4 directions of movement that uses Manhattan distance for a square grid. Is needed to analyze cost function and to find the minimum cost D for moving from one space to an adjacent space. In a simple case, D can be set to be 1. The heuristic on a square grid where is possible to move in 4 directions should be D times the Manhattan distance:

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
return D * (dx + dy)
```

Choosing D, is needed to use a scale that matches the cost function. For the best paths, and an 'admissible' heuristic, set D to the lowest cost between adjacent squared. In the absence of obstacles, and on terrain that has the minimum movement cost D, moving one step closer to the goal should increase g by D and decrease h by D. After adding the two, f function will stay the same, proving that the heuristic and cost function match. Is also possible to give up optimal paths to make A* run faster by increasing D, or by decreasing the ratio between the lowest and highest edge costs.

In case the map allows diagonal movement is needed a different heuristic. The Manhattan distance for (4 east, 4 north) will be 8xD. However, is possible to move simply (4 northeast), so the heuristic should be $4xD_2$, where D_2 is the cost of moving diagonally.

```
function heuristic(node) =
        dx = abs(node.x - goal.x)
        dy = abs(node.y - goal.y)
                return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

In case is possible to move at any angle (instead of grid directions), then is needed to use a straight line distance, called also Euclidean distance:

```
function heuristic(node) =
dx = abs(node.x - goal.x)
dy = abs(node.y - goal.y)
return D * sqrt(dx * dx + dy * dy)
```

3.4. Uses for A*

First and most common usage is shortest path or most efficient path between two nodes, such as the shortest path between two tiles on map. A board game may need to know if a piece can reach some tile and how many movers it would require to get there.

Second one is flood fill. In situation when a path finding algorithm is asked to search a path to an unreachable destination, it won't be obtained a result, but it will still gain and present an useful information. The set of nodes the algorithms explored trying to find a path gives to the user all the nodes that are reachable from the starting location. So, it checks all possible reachable or existing nodes. If the graph represents for example a map, is possible to identify if two land masses are connected or find all the locations which are part of a lake.

The third usage is decision making. In a graph, each node supposes to represent some form of technology in a game's tech tree. A path finding algorithm can determine in such situation the cheapest series of upgrades required to reach a specific technology level. In other words, according to a condition it can reach the goal.

4. Comparing pathfinding algorithms

4.1. Dijkstra vs Bellman-Ford vs Floyd-Warshall

Nowadays, there are dozens of algorithms that solve the pathfinding problems. In dependence of the task or problem is chosen the adequate algorithm, but choosing only one from the numerous algorithms reported in the literature is a critical step in many applications. In a recent study, or in recent years, a set of two shortest path algorithms that run faster on real examples has been identified. These two are: A* algorithm and Dijkstra's algorithm.

First of all, A* algorithm has as base Dijkstra's algorithm. To ease the analyze is enough to highlight the Dijkstra's behavior in dependence of other algorithms. Taking in a parallel Dijkstra, Bellman-Ford and Floyd-Warshall is possible to identify following conclusions:

Table 1. Comparison between pathfinding algorithms

Algorithm	**Behavior**	**Special Features**	**Time complexity/Worst- case performance**
Dijkstra	Works with edges that have nonnegative lengths, and a source node and finds the shortest path from source node to every other node[6]	Finds out the shortest path between any two nodes in a graph.	$O(\|E\| + \|V\| \log \|V\|)$
Bellman-Ford	Finds the shortest paths from a single source vertex to all of the other vertices in a weighted digraph[8]	Networks with small number of nodes	$O(\|E\| \cdot \|V\|)$
Floyd - Warshall	Finds the shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles)[7]	Networks with small number of edges	$O(\|V\|^3)$

Comparing by table 1, is easily observed that Bellman –Ford and Floyd-Warshall are used in small graphs, because of their time complexity. So, overall and going back to the usage of GPS system, a better algorithm from all three is Dijkstra's algorithm. And taking in account that A* is based on Dijkstra's algorithm, the advantages of A* are close to Dijkstra ones in comparison with Floyd-Warshall and Bellman-Ford.

4.2. Dijkstra's algorithm vs A* algorithm

Before comparing Dijkstra and A*, it is important to know how the algorithms works themselves. **Dijkstra's algorithm** works by solving the sub problem k, which computes the shortest path from source to vertices among the k closest vertices to the source. For the Dijkstra's algorithm to work, it should be a directed-weighted graph and the edges should be non -negative. If the edges are negative then the actual shortest path cannot be obtained.

The algorithm works by keeping the shortest distance of vertex v from the source in an array, Dist. The shortest distance of the source to itself is zero. Distance for all other vertices is set to infinity to indicate that those vertices are not yet processed. After the algorithm finishes the processing of the vertices Dist will have the shortest distance of vertex from source to every other vertex. Two sets are maintained which helps in the processing of the algorithm, in first set all the vertices are maintained that have been processed i.e. for which is already computed the shortest path. And in second set all other vertices are maintained that have to be processed. [1]

A* algorithm is a graph search algorithm that finds a path from a given initial node to a given goal node. It employs a "heuristic estimate" h(x) that gives an estimate of the best route that goes through that node. It visits the nodes in order of this heuristic estimate. It follows the approach of best first search. The secret to its success is that it combines the pieces of information that Dijkstra's algorithm uses (favoring vertices that are close to the starting point) and information that Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, g(n)represents the exact cost of the path from the starting point to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal.[1][2]

Table 2. Difference between A* and Dijkstra's Algorithms

Parameters	**A* algorithm**	**Dijkstra's Algorithm**
Search Algorithm	Best First Search	Greedy Best First Search
Time complexity	Time complexity is O(n log n), n is the number of nodes	Time complexity is O (n^2), n is the number of nodes
Heuristics Function	f(n) = g(n) + h(n) g(n) represents the cost of the path from the starting point to the vertex. h(n) represents the heuristic estimated cost from vertex n to the g	f(n) = g(n) g(n) represents the cost path from the starting point to the vertex n.

Concluding using table 2, Dijkstra's Algorithm is the worst case of A* algorithm. A* is faster as compare to Dijkstra's algorithm because it uses Best First Search whereas Dijkstra's uses Greedy Best First Search. The major disadvantage of Dijkstra's algorithm is the fact that it does a blind search there by consuming a lot of time waste of necessary resources. Another disadvantage is that it cannot handle negative edges. This leads to acyclic graphs and most often cannot obtain the right shortest path. Since Dijkstra pick edges with the smallest cost at each step it usually convers a large area of the graph. This is especially useful when you have multiple target nodes but you do not know which one is the closest.

On other hand, A* expands on a node only if it seems promising. Its only focus is to reach the goal node as quickly as possible from the current node, not to try and reach every other node. A* is complete, which means it will always find a solution if that exists. A* can be morphed into another path-finding algorithm by simply playing with the heuristics it uses and how it evaluates each node. This can be done to simulate Dijkstra, Best First Search, Breadth First Search and Depth First Search. The only disadvantage is if there are many target nodes, because it needs to be run several times (once per target node) in order to get to all of them. However, this is a slight disadvantage which is almost covered by positive sides and results of this algorithm.

This project is based on A* algorithm that will find the oriented shortest path in an improvised labyrinth with obstacles. It will be represented in Unity 3D as a map with two destinations and the path between them.

5. A* implementation in Unity 3D

5.1. Unity - introduction

Before 2005, developing the 2D and 3D game apps for different platforms was quite a task as a mobile game development for various platforms involves a lot of time, efforts, and dollars. But, the launch of the Unity engine at Apple's Worldwide Developers Conference turned the table upside down.

Unity is a cross-platform game engine developed by Unity Technologies, which is primarily used to develop both three-dimensional and two-dimensional video games and simulations for computers, consoles and mobile devices.

Unity is a multipurpose game engine that supports 2D and 3D graphics, drag-and-drop functionality and scripting using C#. Two other programming languages were supported: Boo, which was deprecated with the release of Unity 5 and JavaScript which started its deprecation process in

August 2017 after the release of Unity 2017.1. Unity supports building to 27 different platforms, for example: iOS, Android, Windows, Mac, Linux, PlayStation 4 etc. [11]

According to a survey, "The Unity 3D engine has 45 % of Global Game Engine market share and preferred by 47% Game developers as the primary development tool."

5.2. Unity's features

Unity has some features, that make user work much more easier and flexible. For example, 2D and 3D mode settings. In dependence of project purpose is possible to switch between 3D and 2D mode by setting a Default Behavior Mode.

Another functionality of Unity is setting managers. The settings affect overall aspects of Unity's functionality, such as Graphics, Physics and the details of the published player. It covers parts as audio, editor, input, network, physics, quality, graphics, tags and layers etc.

Unity 5 adds a lot of new features for graphics. For lighting, there is now real-time global illumination - that is mobile, desktop and console ready. New improvements for shading include reflection probes and physically based shading. There are many new physics effectors like a point effector that attracts and repels and a surface effector that applies tangential force. There is an area effector that applies directional force and also a platform effector that allows for a one way collusion with zero side friction.

5.3 Unity 3D's advantages

The engine is highly preferred for its extended support to 27 platforms. The app developed and deployed can be easily shared between PC, web and mobile platforms.

The text editor is provided by IDE to write the code, but sometimes a distinct code editor is also used by the developers to alleviate confusion. Additionally, the integrated development editor support JavaScript and C# for scripting, and also offers notable features that are ideal for the game development.

The high quality audio and visual effects are supported by the engine that eases the game development. The visuals are adaptable on every screen and device without any distortion or compromise with the image quality.

The debugging and tweaking is amazingly easier with Unity game development because all the game variables are displayed during gameplay, which in turn allow the developers to debug the process at runtime.[12]

5.4. Unity 3D's disadvantages

In Unity 5 engine, the built-in support for the PhysX physics engine has some performance issues and lacks some important functionalities, which need to be added to craft the excellent game app. The developers need to have licenses for the best graphics, deployment and performance improvements.

The code is stable in Unity as opposed to other engines and packed with a great architecture that improves the game app performance. But, unavailability of the source code makes finding, addressing and fixing the performance issues difficult.

The game developed leveraging Unity engine consumes more memory, which in turn creates OOM errors and debugging issues in the apps.[12]

5.5. Unity 3D's methods and functions

The central component of any game, from a programming standpoint, is the ***game loop*.** It allows the game to run smoothly regardless of a user's input or lack thereof. Every game must and should have a game loop because a game must continue regardless of a user's input.

Awake and ***Start*** are two functions that are called automatically when a script is loaded. When a scene starts, ***Awake()*** is the function which is always called(once for each object in the scene) before any Start functions. Start is called after Awake, immediately before the first Update, but only if the script component is enabled. ***Start()*** is called before the first frame update only if the script instance is enabled.[13]

Update is the most commonly used function in unity. It's called once per frame on every script that uses it. ***FixedUpdate*** is a similar function to update but it has a few important differences. ***FixedUpdate()*** is often called more frequently than ***Update()***. It can be called multiple times per frame, if the frame rate is low and it may not be called between frames at all if the frame rate is high.[13]

5.6. A* algorithm implementation

Involving theoretical part, the code for A* algorithm for the shortest path is implemented using C# and visualized through Unity 3D. For the beginning, to visualize the process of finding the shortest path was needed to create a grid and two points: a start point (a seeker) and an end point(a target). In Unity world, the plane has its own coordinates, because the grid created on Unity's plane consists from units (squares), the size of squares can be changed by user. So, to be more clear for the user, the grid is divided in smaller squares measured in unit scale recognized by Unity, case each square has side

length of 10 meters and a diagonal of 14 meters (rounded value) and these empty spots or squares are colored in white.

This grid is created by function void ***CreateGrid()***, which has no parameter and creates a 2D Marix from Units (see annexes for code).

```
1. void CreateGrid()
```

Another particular class to mention is ***Unit class*** that stands for squares (see annexes for code).

Fields of Unit class:

public bool walkable – a variable which is true if seeker can go through the unit or false if it is an obstacle

public Vector3 realPosition – an object which contains unity world coordinates of unit.

public int x – a variable which stores the x index of unit in 2D Unit Matrix .

public int y – a variable which stores the y index of unit in 2D Unit Matrix .

public int gCost - a variable which stores the movement cost to move from the seeker's unit to a current unit .

public int hCost – a variable which stores the movement cost to move from current unit to target's unit.

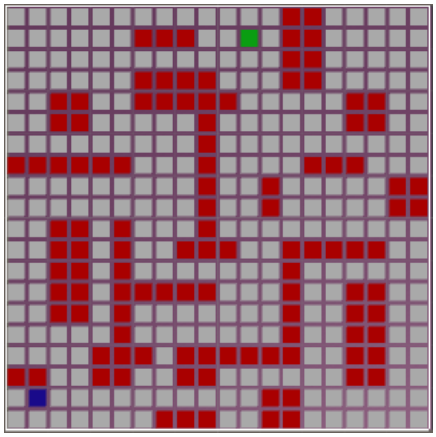
public Unit parent – an object which contains the parent unit of current unit .

public int fCost – variable which stores the sum of gCost and hCost .

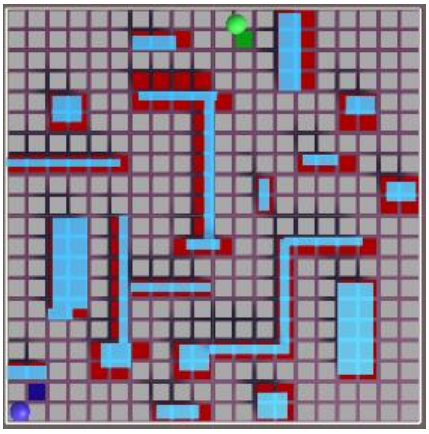
Constructors of Unit class **:**

public Unit(**bool** walkable, Vector3 realPosition, **int** x, **int** y) – it is a general constructor which requires parameters for object instantiation . The required parameters are explained above .

To complicate the situation and to test more A* algorithm, on grid are some obstacles that creates a labyrinth. Is possible to visualize them in 3D and in 2D, they will be substitute by red squares. Another point to mention is location of obstacles. If they are on the board between two squares, they will take both squares. So in 3D is seen only the wall, however in 2D it will be two rows of squares in red color.

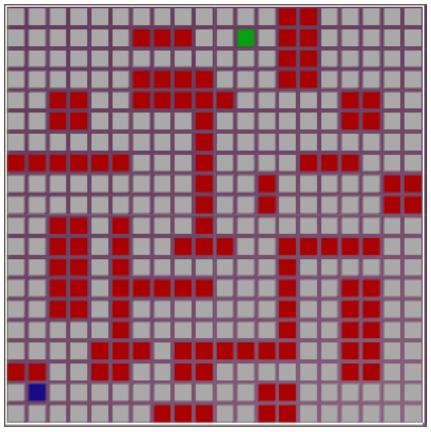


Picture No1. *2D visualization*



Picture No.2. *3D visualization*

To represent the destinations are used 2 spheres, a green one and a blue one. According theory, in A* algorithm is needed a target and a seeker, in other words a start and a stop. In this case, the seeker is blue and the target is green. Moreover, the square, on which they are set, will have the spheres' color (it will be green or blue).



Picture No3. *Representation of map with obstacles,*
seeker and target according mentioned colors above

Another important moment regarding grid, is real location of objects on squares. Is needed a function that will convert the unit dimensions into coordinates. In such a way, is possible to work with full squares, avoiding boundless checking of an object location in a square and optimizing the speed result of A* algorithm, cause it will check neighbor squares as nodes.

Calling back the theory, to find the shortest path is applied A* algorithm. The main function which contains all logic implementation of algorithm is: ***FindPath*** function, which has 2 parameter of type Vector3, which contains information about position of the seeker and the target (see annexes for code).

```
1.  void FindPath(Vector3 seekerPos, Vector3 targetPos)
```

For code clarity, some details of FindPath function will be revealed below:

```
3.                  Unit seekerUnit = grid.fromRealPosToUnit(seekerPos);
4.                  Unit targetUnit = grid.fromRealPosToUnit(targetPos);
```

On line 3 – 4, seeker and target coordinates are transformed in unit coordinates, in other words is find the units(squares) on which are placed seeker and target. After follows steps as in pseudo code (see annexes for pseudocode).

```
1.          List<Unit> openList = new List<Unit>();
2.          List<Unit> closedList = new List<Unit>();
3.          openList.Add(seekerUnit)
```

It is created an Open and a Closed List and added the unit where is placed seeker in the Open List. After follows the loop, which is executed while the Open List is not empty. In this loop is found the Unit with the minimum F – cost(Fcost = Hcost + Gcost). With:

```
1.              openList.Remove(currentUnit);
2.              closedList.Add(currentUnit);
```

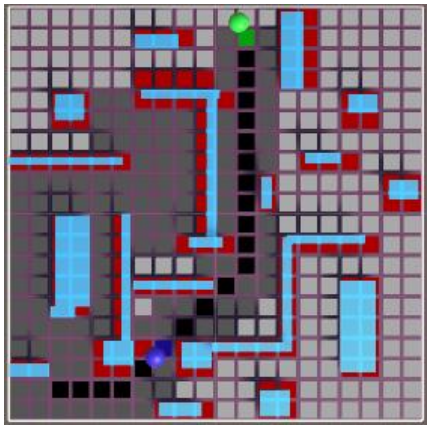
Is removed from Open List the Unit which was found in previous step (unit with the minimum F cost) and is added to the Closed set.

```
1.              if (currentUnit.realPosition == targetUnit.realPosition)
2.              {
3.                  ComputePath(seekerUnit, targetUnit);
4.                  return;
5.              }
```

This if statement checks if the unit which was added to Closed List is our Final unit, on which is placed target. If it is finished the search of path, is computed the path which was found and the next step in FindPath function is exit. If it is not the final unit, then it continues to search path.

Also, in this loop are checked all neighbors of current unit, which has the minimum F cost. If the neighbor unit is not walkable, in other words is an obstacle or it is already in the Closed List, then it is ignored, otherwise it moves on. It is also checked if the new computed path has the lower Gcost(distance from current node to it neighbor) and also if this neighbor is not in Open List then the required fields of the unit are updated.

Because of an additional implemented function, called ***Moving()***, seeker can move by found shortest path till it will reach the target.



Picture No.4. *On-mode 3D version seeker moving to target*

6. Conclusion

This report was directed at studying A* algorithm and its unique components, comparing A* algorithm with other pathfinding algorithms and highlighting A* algorithm's advantages and at last, visualizing the shortest path from 2 points on a map.

There is no winning pathfinding algorithm and the deciding factor for the best algorithm depends completely on the scenario of the implementation. In case of simulating the behavior of agents in a particular situation which requires a fast and the actual shortest path at a minimum cost, the best approach is A* algorithm. Taking a glance back to theory, Dijkstra's algorithm exceeds other pathfinding algorithms, and A* algorithm in the worst case is the same Dijkstra's algorithm. From these two statements, is possible to conclude that A* algorithm outstands over all pathfinding algorithms.

A* algorithm can provide more information than any other algorithm. It can analyze the open and the closed list and can find all solutions that are possible, because of neighbors check. Moreover, because of heuristic function, the process of finding the shortest path is faster and the inexpensive feedback is obtained. Also, with this algorithm is reduced the number of nodes visited or checked, in such a way saving time and resources.

Implementation of A* algorithm in Unity 3D environment give to the user a better understanding of usage and a more clear visualizing of the process of finding the shortest path. So, in 3D option is possible to visualize the target and seeker as 3D objects and the obstacles are presented as walls, which cannot be passed through. In 2D option, the map looks like a matrix or like a big square divided in small squares. In this case, obstacles, seeker and target can be identified by their own color. Another point to mark, is the shortest found path, which can be visualized as a path and because of aa moving function, on other map can be seen how seeker goes to target on the found path (see annexes for screenshots).

7. Bibliography

1. Comparing A* and Dijkstra's algorithm: https://www.slant.co/options/11585/~a-algorithm-review (06.01.2018)
2. A* pathfinding algorithm: http://www.growingwiththeweb.com/2012/06/a-pathfinding-algorithm.html (06.01.2018)
3. Heuristic theory: https://en.wikipedia.org/wiki/Heuristic_(computer_science)#Traveling_salesman_problem (07.01.2018)
4. A*'use of the Heuristic: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html (07.01.2018)
5. A* algorithm: https://en.wikipedia.org/wiki/A*_search_algorithm (05.01.2018)
6. Dijkstra's algorithm: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm (09.01.2018)
7. Floyd-Warshall algorithm: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm (09.01.2018)
8. Bellman- Ford algorithm: https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm (09.01.2018)
9. History of pathfinding algorithms: http://www3.cs.stonybrook.edu/~cse352/T5talk.pdf (09.01.2018)
10. On the history of the shortest path problem: https://www.math.uni-bielefeld.de/documenta/vol-ismp/32_schrijver-alexander-sp.pdf (09.01.2018)
11. Unity: https://en.wikipedia.org/wiki/Unity_(game_engine) (10.01.2018)
12. Unity advantages and disadvantages: http://www.potenzaglobalsolutions.com/blogs/5-rarely-known-advantages-and-disadvantages-of-unity-game-development (10.01.2018)
13. Unity and C#: https://onestopdotnet.wordpress.com/2014/04/19/unity-game-loop/ (10.01.2018)

8. Annexes

1. Pseudocode

```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
        current = node in OPEN with the lowest f_cost
        remove current from OPEN
        add current to CLOSED
```

```
if current is the target node //path has been found
        return

foreach neighbour of the current node
        if neighbour is not traversable or neighbour is in CLOSED
                skip to the next neighbour

        if new path to neighbour is shorter OR neighbour is not in OPEN
                set f_cost of neighbour
                set parent of neighbour to current
                if neighbour is not in OPEN
                        add neighbour to OPEN
```

2. Function **FindPath** implementation:

```
1.  void FindPath(Vector3 seekerPos, Vector3 targetPos)
2.      {
3.          Unit seekerUnit = grid.fromRealPosToUnit(seekerPos);
4.          Unit targetUnit = grid.fromRealPosToUnit(targetPos);
5.
6.          openListUnit = new List<Unit>(); // for computing unit of openList ( need to draw pat
    h )
7.          List<Unit> openList = new List<Unit>();
8.          List<Unit> closedList = new List<Unit>();
9.          openList.Add(seekerUnit);
10.         openListUnit.Add(seekerUnit); //...
11.
12.         while (openList.Count > 0)
13.         {
14.             Unit currentUnit = openList[0];
15.             for (int i = 1; i < openList.Count; i++)
16.             {
17.                 if ((openList[i].fCost < currentUnit.fCost) || (openList[i].fCost == currentU
    nit.fCost && openList[i].hCost < currentUnit.hCost))
18.                 {
19.                         currentUnit = openList[i];
20.                 }
21.             }
22.
23.             openList.Remove(currentUnit);
24.             closedList.Add(currentUnit);
25.
26.             if (currentUnit.realPosition == targetUnit.realPosition)
27.             {
28.                 ComputePath(seekerUnit, targetUnit);
29.                 return;
30.             }
31.
32.
33.             foreach (Unit neighbour in grid.GetNeighbours(currentUnit))
34.             {
35.                 if (!neighbour.walkable || closedList.Contains(neighbour))
36.                 {
37.                     continue;
38.                 }
39.
40.                 int pathToNeighbour = currentUnit.gCost + GetDistance(currentUnit, neighbour)
    ;
41.                 if (pathToNeighbour < neighbour.gCost || !openList.Contains(neighbour))
42.                 {
43.                     neighbour.gCost = pathToNeighbour;
44.                     neighbour.hCost = GetDistance(neighbour, targetUnit);
45.                     neighbour.parent = currentUnit;
46.                     openList.Add(neighbour);
47.                     openListUnit.Add(neighbour); // ...
48.                 }
```

```
            }
        }
```

3. Function ***void CreateGrid()*** implementation:

```
void CreateGrid()
{
    grid = new Unit[unitGridSizeX, unitGridSizeY];
    // right (1,0,0) , forward (0,0,1)
    Vector3 bottomLeftCorner = transform.position - Vector3.right * realGridSize.x / 2 - Vector3.forward * realGridSize.y / 2;

    for (int x = 0; x < unitGridSizeX; x++)
    {
        for (int y = 0; y < unitGridSizeY; y++)
        {
            //1.5 unit
            Vector3 unitPosition = bottomLeftCorner + Vector3.right * (x * nodeDiameter + nodeRadius) + Vector3.forward * (y * nodeDiameter + nodeRadius);
            // check if there is a collision
            bool walkable = !(Physics.CheckSphere(unitPosition, nodeRadius, obstacleMask));
            grid[x, y] = new Unit(walkable, unitPosition,x,y);
        }
    }
}
```

4. **Unit class** (squares)

```
public class Unit
{

    public bool walkable;
    public Vector3 realPosition;
    public int x;
    public int y;

    public int gCost;
    public int hCost;
    public Unit parent;

    public Unit(bool walkable, Vector3 realPosition, int x, int y)
    {
        this.walkable = walkable;
        this.realPosition = realPosition;
        this.x = x;
        this.y = y;
    }

    public int fCost
    {
        get
        {
            return gCost + hCost;
        }
    }
}
```