Astrogator: An Autopilot for Navigating 3D Space

Final Report

Submitted for the MEng in
Computer Science with Games Development

May 2016

by

Kuba Maruszczyk

Word Count: 11265

Table of Contents

1. Introduction

The game industry is the multi-billion-dollar business and still growing (Iabuk, 2014). In the past, it used to be the quality of graphics that determined the success of the game. However, at present, good graphics alone are insufficient to drive the sales (Cui and Shi, 2011a). Instead, players desire more realistic gaming experience. This leads to an increasingly important role of artificial intelligence (AI) in the success of games. One of the most visible types of AI in computer games is pathfinding, that is, determining a safe and efficient way of moving the entity to desired destination. As a classical AI problem in games development, it is a critical component for creating believable entities and providing the user with more challenging, real and enjoyable experience (Graham, McCabe & Sheridan, 2003). Nothing can spoil a first impression of a game faster than seeing a game character moving in a robotic manner, unable to navigate through a simple game world (Bourg and Seemann, 2004). Even though there are many existing algorithms available for finding optimal paths through a 2D space, the evolution of the 3D gaming universe in the last decade brought up a new quest for a quick path searching in 3D world. Although some possible solutions are starting to emerge, these tend to be restricted to the problem at hand and thus, not easily applicable to variety of game environments. For example, working perfectly in static game world but unable to navigate dynamic environments.

The purpose of this project is to develop an autopilot to successfully navigate a multilayered 3D environment, cluttered with dynamic obstacles and to make it flexible and easily applicable to a variety of game worlds. This report will begin with the presentation of the pathfinding problem in greater details, including a brief overview of the most popular search algorithms and their application to the current

project. Then, the aim and objectives of the current research will be outlined. This will be followed by a presentation of the design and implementation of the proposed solution, as well as an analysis of a series of experiments conducted to assess its efficiency. Finally, a critical evaluation of the project will be undertaken and ideas for future possible extensions will be presented.

2. Aim and Objectives

The aim of this project is to create a cluttered, multilayered 3D environment and develop an autopilot algorithm that can determine a safe and efficient path through it. Other factors such as avoiding hazards, dynamically altering the course to avoid moving objects, creating a believable path and accounting for the effects of gravitational pull will also be considered.

2.1. Primary Objectives

Objective 1 – Creating 3D environment

The aim of this objective is to create a simple visualization of a cluttered, multilayered 3D environment. The game world should be an open space in a form of asteroid field. The complexity of the environment should be gradually increasing, with number of obstacles easily adjusted. Both static and dynamic obstacles should be used in the scene.

Objective 2 – Developing an autopilot to navigate 3D space

The aim of this objective is to develop an autopilot to safely and efficiently navigate through multilayered 3D cluttered environment. There are several sub-objectives identified for this purpose:

- Assess current state of 3D pathfinding techniques
- Design and implement pathfinding algorithm that would account for 3D nature of the multilayered environment, rather than mapping 3D world onto 2D
- Introduce a dynamic obstacle avoidance mechanism for the entity to dynamically alter the course of the agent to avoid moving obstacles

- Run a series of experiments to assess performance of the autopilot under variety of circumstances e.g. different types of obstacles, amount of clutter etc. and use these tests to identify any potential problems and solutions
- Evaluate performance of the autopilot and discuss possible further improvements

2.2. Secondary Objectives

Objective 3 – Accounting for Gravitational Pull and Fuel Consumption

- Consider gravitational pull in the workings of the autopilot to make the simulation more realistic, possibly by restricting a number of consecutive vertical movements
- Aim to minimise fuel consumption by controlling for various factors, such as distance travelled, speed, pitch angle etc.

Objective 4 – Creating a naturally looking path

- Create a natural path for the entity to follow by smoothing out the path calculated by algorithm

Objective 5 – Creating Appealing and Informative User Interface:

- Extend user controls by allowing the user to navigate the world by selecting the camera, zooming in and out, changing the start and goal locations and even selecting preferred algorithm for a certain pathfinding task
- Include a number of flight control instruments such as heading indicator, artificial horizon, speed indicator and fuel level indicator to make the application more informative and interesting

- Implement stereoscopic graphics to create immersive user experience

3. Background

3.1 Pathfinding Terminology

Pathfinding refers to the process of finding an optimal route between two points (Bourg and Seemann, 2004). The first step of solving the pathfinding problem is to define the **search area**; a way to represent a game world for the search algorithm to calculate the best path. To this aid, the game world is mapped into a **data structure**, such as linked list or search tree, which consists of nodes. A **node** is an individual unit of the data structure. Nodes contain data and also may link to other nodes. The **links** to other nodes are often represented by references or pointers. In a search tree, a **root node** is the highest point of the tree structure (see: Figure 1). A **parent node** is a node, which extends to another node (child node). A **branch** refers to a section of the search tree, which is made up of one or several nodes.

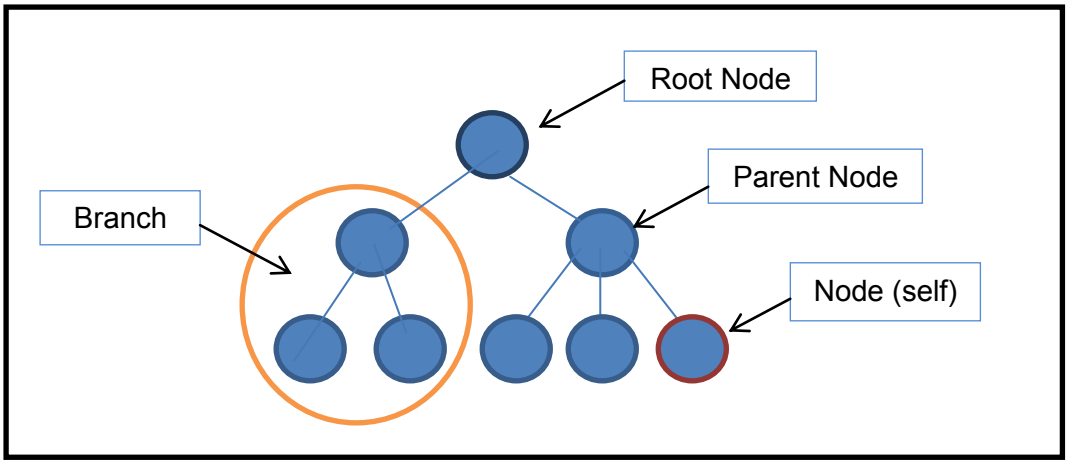


Figure 1 Basic Search Tree Structure

The time to get from one node to another is referred to as **cost**. This is typically derived from a distance between the two nodes. Each node is assigned a certain

cost. A **heuristic** directs the search towards the target; it is an estimate of the cost of getting from a given node to the final destination (Bourg and Seemann, 2004).

3.2 Search Area

Finding the most appropriate data structure to represent the game world is of a crucial importance, determining the efficiency of selected search algorithm (Cui & Shi, 2011b). A large data structures can occupy a lot of memory and thus, slow down the performance. A small data structure may be faster but less accurate if the game world representation is too simple. Therefore, there is a trade-off between optimality and simplicity.

There are number of ways the search space can be represented:

- **grid**
    - the map is divided into small regular shapes, usually tiles with each tile being a node in the search area (Bourg and Seemann, 2004)
    - evenly spaced grids are effective and simple ways of representing the data
    - due to their regular structure, they can be easily updated
    - however, depending on the grid size, they can suffer from large memory footprint
- **waypoint graph**
    - waypoints are points placed around the ‘walkable’ path
    - allow for customization and flexibility
    - each waypoint is a node
    - depending on the density of the waypoints the movement can become quite restricted

- **navigation mesh**
    - map is divided into a set of convex polygons, which describe the ‘walkable’ area of the game space
    - the path from the start polygon to the destination polygon is found first and then, a detailed path to the target is found inside the destination polygon (Cui & Shi, 2011a)
    - as each polygon can represent a large search area, the memory footprint is very low
    - however, the creation of the mesh is relatively complex and laborious

Current pathfinding solutions provide either a high speed search by using data structures that have a lower number of nodes to be explored or produce optimal paths by using detailed game word representations but at the expense of time and memory resources (Bultico et al., 2011). While the first solution is fast, the lower the number of nodes the bigger they are. Operating on such data structures restricts movement of the entity due to less accurate representation of the game world. This in turn, may lead to less believable, robotic paths to be produced. Creating detailed representation of the game world on the other hand, leads to more accurate searches likely to produce optimal solutions but using more time and resources.

One solution to this problem, would be to use more sophisticated subdivision techniques such as quadtrees (or octrees for 3D world). This method would allow for merging the nodes that are free of obstacles together. As nodes are merged, the search space decreases in size, resulting in more efficient performance of the search algorithm. However, this would also suffer from the problem of jagged paths as the merged tiles would introduce uneven node size, reducing quality of returned path.

The path would be severely affected as the cost of the large node would be much lower than going around it. Merging tiles would also become more problematic in dynamic game worlds. Effective avoidance of dynamic obstacles requires detailed, real-time information about changes in the environment. While improved partition of the data structure would have some advantages for static game worlds, it could interfere with successful avoidance of dynamic obstacles. Although it is possible to generate quadtrees (or octrees) dynamically, updating the data structure every time the environment changes would be time-consuming, especially is if large amount of objects needed updating often.

Due to all these constrains, it is quite common for game creators to use data structure designed particularly for their game worlds (Cui and Shi, 2011b). How to find the best data structure that would aid finding optimal path more efficiently is still an area of study.

3.3 Search Algorithms

There are many pathfinding algorithms that have been proposed to solve the problem of navigation through different kind of environments. These can be divided into two categories: undirected and directed (Graham et al., 2003).

Undirected algorithms refer to searches, in which all nodes are explored blindly, according to a certain search pattern but without any knowledge about the likely location of the target. These algorithms have been developed for finding the shortest way out of a maze and although may not be of use when creating an efficient and believable pathfinding, they can be utilized to get the agent moving while a more sophisticated algorithm is searching for an optimal path (Graham et al., 2003). It is important to note that none of these searches considers the cost of the explored path

and thus can only be used if no cost variables are involved; the distance between the nodes is the same for every link and the solution is returned in terms of path length. The two main undirected algorithms include:

- **Breadth-first search (BFS)** starts its search from the root node (or any initial node) of a search tree, which is expanded first, followed by all its successors and then their successors and so on (Hui, Prakash & Chaudhari, 2004). In basic terms, all nodes at each depth level of the search tree are expanded before any nodes at the next depth level. BFS basically expands the search by one depth level finding the smallest number of steps it takes to get to any given node from the initial node.
- **Depth-First Search** algorithm starts its search at the root node, exploring each branch of the tree as far as possible (exploring all the child nodes) before backtracking and moving to the next branch (Hui et al., 2004). This search is particularly useful for problems with multiple solutions, as it is capable to find the best path after exploring only a small portion of the search area (Graham et al., 2003).

Directed algorithms use more methodological approach to pathfinding by considering the cost of each explored path (Bourg and Seemann, 2004). There are two main cost variables used for directing the search:

- **g(n)** – the cost of movement from the start position of the search to a given node. The use of g cost minimizes the cost of the path, it is optimal but can be very inefficient.

- **h(n)** – the heuristic, an estimate of the cost of getting from a given node to the final destination. Although it can cut the cost of the search considerably, its use is not necessarily optimal nor guarantees the solution to be found.

The main directed algorithms include:

- **Dijkstra's algorithm** conceptually functions as breadth-first search but allows for assigning different cost to each link between the nodes (Hui et al., 2004). The nodes in all directions are expanded uniformly, with the g cost of each node being evaluated. The path with the lowest g cost is then selected as the optimal one. Although the search will return the optimal solution, it may take a substantial amount of time before all nodes are explored.
- **Best First Search** is an informed search, which uses a heuristic to expand the most promising node. Best first search ranks the nodes based on the h cost. The paths that are judged to be the closest to the target node are explored first. This significantly reduces the search time of the algorithm, however, in some instances may lead to less than optimal path to be found or in the worst case scenario, the algorithm being stuck in an infinite loop.
- **A* algorithm** is the best-established algorithm for the general searching of optimal paths (Cui & Shi, 2011). A* algorithm, first proposed by Peter Hart and colleagues (1968), is guaranteed to find the most optimal path (if one exists) between any starting position to a given destination. A* algorithm combines the advantages of Dijkstra's algorithm and best first search algorithm. In A* search each node's g value (Dijkstra's algorithm) and the heuristic estimate of the remaining cost from the node to the goal (best first search algorithm) are

considered. These two values are then combined to create an f cost, which is used by the algorithm to efficiently find the most optimal path:

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

The A* algorithm keeps track of all the nodes that need to be visited, in what is called an **open list**. The open list begins with only one node – the start node. The algorithm checks whether the adjacent nodes are walkable, that is, do not contain an obstacle, and adds them to the open list, while ignoring all the nodes that cannot be used. The f cost of each node is being evaluated. The A* also keeps a **closed list**, a list of nodes that have already been examined. To understand how the nodes on the open list are linked, the algorithm tracks the parent of each node. Once the target node is reached, A* uses the parent links to track back the cheapest path to the initial node (Bourg and Seemann, 2004).

Despite the success of directed algorithms such as A*, traditional pathfinding algorithms suffer from some limitations. Firstly, certain adjustments have to be made for these algorithms to be used in 3D worlds that are multilayered in nature. Secondly, a lot of pathfinding solutions are unable to handle dynamic worlds. For example, once a path is calculated, if a dynamic obstacle subsequently covers a node along the predetermined path, the entity would have no knowledge of this and would continue along that path, eventually colliding with the obstacle. Special dynamic obstacle mechanisms are needed to overcome this issue.

3.4 Multi-dimensional World

Even though A* search can efficiently solve the pathfinding problem in 2D game environment, 3D trend in games development requires a new, improved version of

the algorithm, which can be applied in 3D world. One possible solution would be to map 3D world into 2D expression and use original A* algorithm to search for the optimal path (e.g. Hui et al., 2004; Hu, gen Van & Yu, 2012). However, this method would only be applicable to simple 3D situations and would fail in more complex environments, where a number of overlapping layers may appear frequently, for example, underground, inner multi-floor building or in space (Niu and Zhuo, 2008). The 3D A* algorithm would be required to work the pathfinding problem in actual multilayered 3D world (see: Figure 2).

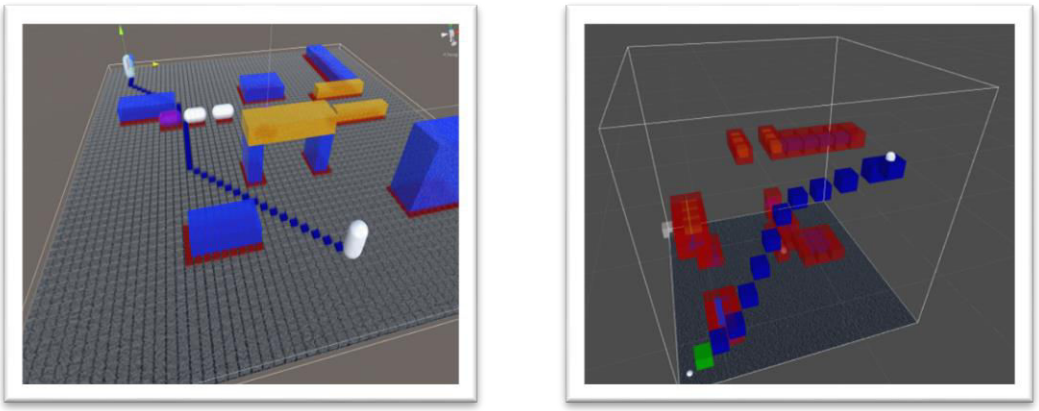


Figure 2 3D word with 2D algorithm capable to find path as the actual movement happens only in single plane (left hand side) and actual, open, multilayered 3D world, for which 3D algorithm is required (right hand side). It is the latter type that is the interest of this project.

3.5 Avoiding Obstacles

Safe pathfinding is as important as efficient pathfinding and thus, the agents need to be able to navigate through the cluttered world, where encountering an obstacle on their path is inevitable (Hui et al., 2004). There are two types of obstacles: static and dynamic. When the game world is fully observable and static, handling obstacles is relatively easy, with the nodes occupied by any obstacles marked as non-passable and simply excluded from the pathfinding process. While the majority of search

algorithms is able to work out the best route in static map, collision avoidance between entities or as a matter of fact, any other dynamic obstacle must also be considered. There are a number of algorithms created specifically to resolve the problem of avoiding the obstacles in dynamic environments or environments that are partially known. The most popular of these include:

- **Adaptive A***(Koeing and Likhachev, 2006) is a variation of A*, which calculates a new path from the agent current location if the current path is no longer viable. Adaptive A* updates the h cost of all the nodes in A* closed list. Once the increase in cost is detected, a completely new path is generated.
- **D*** algorithm (Stentz, 1995) and **D*Lite** (Koeing and Likhachev, 2005) recalculate a new path once the obstacle is met, however, it does it much faster than Adaptive A* by keeping an information about previously found paths, which can be then reused for future searches. D* algorithm is however very complex and thus, it was replaced by D* Lite.
- **Tree Adaptive A*** (Hernández et al., 2011) builds a tree of previously known paths, which then can be reused when the obstacle is met. Tree Adaptive A* has been reported to outperform D* Lite search, however, it requires specific data structure to be kept at runtime and thus, it is quite complex
- **Multipath Adaptive A*** (Hernández, Baier & Asin, 2014) is a simple extension of Adaptive A*, in that it reuses paths previously found by A* searches when the environment has changed and new path is needed. Once the new A* search encounters a node that belongs to previously known path and this path is still the optimal path to the goal, A* search is terminated early, which results in better performance. It is simpler than Tree Adaptive A*

and does not require a special data structure. However, so far this algorithm has only been used for partially known environments and not dynamic ones.

The idea of reusing previously found paths is a reoccurring theme among the workings of the algorithms that aim at effective pathfinding in either dynamic or partially known environments. This is not a new concept and was first described by Holte et al. (1996) in relation to hierarchical search. Adaptive A* works well in unknown environments with the small number of obstacles. However, recalculating the path each time the obstacle is met, can become frequent and time consuming if there is large number of obstacles present in the game world. As the Tree Adaptive A* and Multipath Adaptive A* remember previously found paths, they can incorporate that information into their future searches and thus, stop the A* search earlier once the newly found path meets the old path that can still be used (that is if it is still the optimal path to the goal). As the Tree Adaptive A* explicitly stores tree of path to the goal, such tree is actively maintained during run-time and requires specific structure (Hernández et al., 2011) . Multipath Adaptive A* benefits from its efficiency as it is faster than D* Lite search and simplicity as it is less complicated than Tree Adaptive A* (Hernández, Baier & Asin, 2014). However, as it has only been used for partially known environments, some modifications would be needed if it were to be used in dynamic game worlds.

3.6 Natural looking path

Once the optimal path is found it needs to be smoothed out to create naturally moving agent. There are two main ways to achieve this: splines or Bezier Curves (Hui et al., 2004). Both of them will be explored as a possible solution of creating a realistically looking path.

3.7 Technologies

In order to speed up and simplify development process, decision was made to use Unity3D in conjunction with Visual Studio 2015, as development tools. The language of choice was C#. This allowed for the main focus to be directed more towards the pathfinding problem, while spending less time on creation of basic environment, in which the navigation process takes place. In addition, Unity3D engine provided the developer with a set of useful libraries to aid various development tasks.

4 Technical Development

4.1 System Design

4.1.1 3D World Design

The graphical representation of the 3D world will be simple and flexible to allow for easy modification. The theme of the 3D world will be an asteroid field, with a space ship as a path following entity. Although the graphical representation of the world is not prioritised in this project, an effort will be made to create a scene that will appeal to the user. A number of 3D scenes will be created that will vary in the size, number of static and dynamic obstacles. These will be then used for experimental purposes.

4.1.2 Autopilot Design

The autopilot has been designed in such way that it is divided into separate modules, with each module encapsulating relevant data and functionality (see: Figure 3).

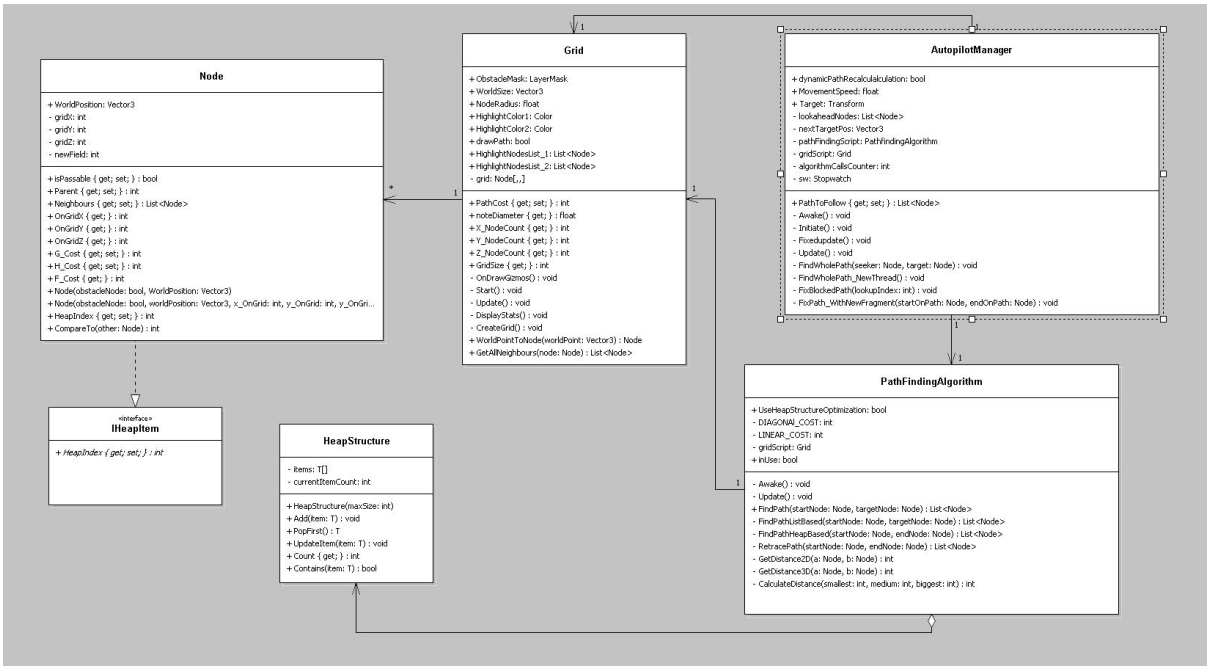


Figure 3 Initial UML Class Diagram of the Autopilot.

These modules can be classified as either low or high level components. Low level components include:

- Data Structure (Grid)

A world map will be represented by a regular grid. However, instead of dividing the map into tiles (like it is typically seen in 2D games), a cube will be used as building unit for the 3D world map. The shape of the cube will automatically define the number of its neighbouring units, which in three dimensions will include adjacent and diagonal neighbour nodes on the same height level and adjacent and diagonal neighbour nodes above and below (up 26 in total). At first, only one layer of cubes will be created to represent 2D plane. This will be further expanded by additional layers to create the map of multilayered 3D space.

- Node

Nodes are the prime elements of the data structure used by the pathfinding algorithm. Each node will contain information about its position and state, that is, whether or not the node is passable or obstructed by an obstacle, as well as information about any additional costs (for example: areas of bad weather conditions).

- Pathfinding Algorithm

A* algorithm will be used as a base for this project due to its reliability to always find the optimal path between the two points. However, a number of modifications will be made to optimize the algorithm and tailor it to account for the three dimensional, multilayered nature of the game world.

All these low level components will be effectively managed and combined by a high level Autopilot Manager. Above many, the Autopilot Manager will be responsible for calling the pathfinding algorithm when appropriate, remembering previously found

path and actually moving the entity along the path. The autopilot manager will include a highly sophisticated dynamic avoidance mechanism that will be employed in dynamically-changing environments.

4.1.3 Experimental Design

A series of experiments will be designed to assess the effectiveness and efficiency of the autopilot under variety of circumstances. The independent variables tested will include:

- Size of the World
- Amount of Clutter in the Environment (Static Obstacles)
- Amount of Dynamic Obstacles

The performance of the algorithm will be tested against the following criteria:

- Safety
- Time Taken to Find the Path
- Length of the Found Path
- Number of Times Algorithm is Called (dynamically-changing environments only)
- Fuel Consumption

4.1.4 User Interface Design

User interface will contain a set of controls allowing the user to monitor and set various parameters and interact with the simulation. Among many, a functionality will be implemented to allow the user to select different search algorithms, select the camera, zoom in/out etc. If the time permits, stereoscopic graphics will be implemented to create immersive user experience.

4.2 System Implementation

4.2.1 Graphics Implementation

As the Unity3D engine was used to develop the software, there was no need to focus on the lower level aspects of graphics programming. A number of different unity scenes was created in order to develop and test the application (see: Figure 4).

Figure 4 Example 3D world cluttered with dynamic asteroid obstacles.

- Models used included:
    - Asteroid, obtained from unity asset store (see: Figure 5)
    - Space Ship, free model found on the web (see: Figure 6)
    - Unity primitives provided by Unity

Figure 5 Asteroid Model

Figure 6 Space Ship Model

- Skybox used:
    - “Startday Skybox” obtained from unity asset store.

- Textures Used:
    - Various textures either made especially for this project by its author or obtained from unity asset store.
- Sounds:
    - Engine sound asset was recorded especially for this project needs by the author.
    - Sound controller script was created by the author to provide functionality which changes engines’ sound pitch in response to constantly changing space ship’s speed.
- Rain Particle Effect:
    - Rain particle animation obtained from unity asset store was tailored for specific needs of this application by the author.

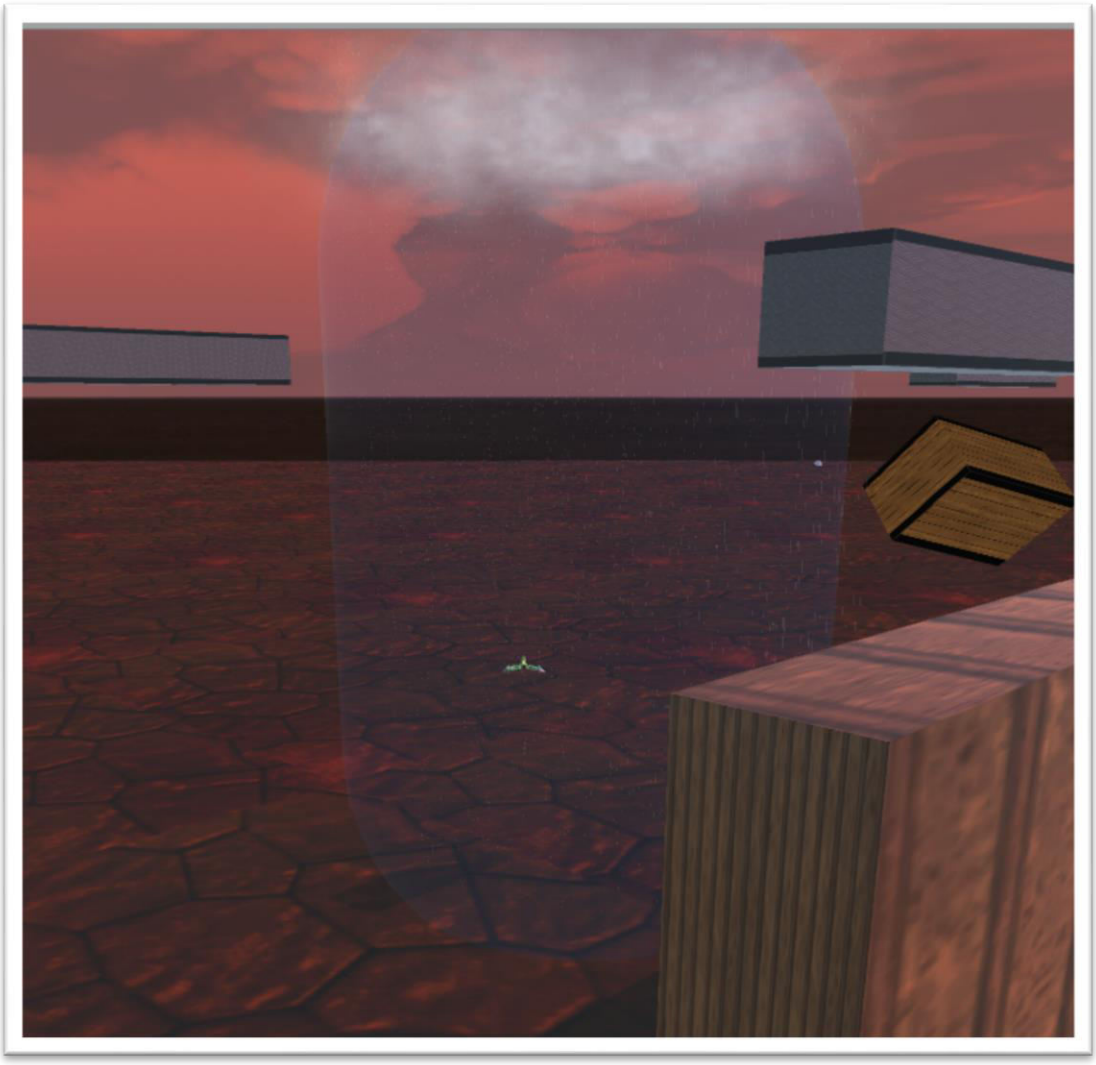


Figure 7 Example Rain Zone

4.2.2 Autopilot Implementation

The autopilot program structure distinguishes following elements:

NODE CLASS

Nodes are the basic elements of the data structure used by the pathfinding algorithm. All nodes have the same uniform size. Each node contains set of data fields used by the program:

- node’s position (in the world)
- node’s location in the Grid (or 3DGrid) data structure
- list of its neighbours (26 adjacent nodes)
- flag to inform whether the node represents empty area or area obstructed by an obstacle
- distance from the start point of the search (G_Cost)
- distance from the target node (H_Cost)
- overall cost (weight) of the node (F_Cost = G_Cost + H_Cost)
- any additional cost added to overall weight (in case of bad weather conditions)

GRID CLASS

The grid consists of a number of nodes placed inside three dimensional array and linked with each other in a way, that a single node has a list of references to all of its neighbours (maximum 26 nodes in 3D grid). Grid implementation allows certain parameters such as world size and size of the node to be specified. Grid structure is generated at the beginning of the program’s execution. During this process actual world inside the simulation is mapped onto a set of nodes that will make up the grid. Mapping algorithm searches the world for objects marked as obstacles and if such objects are encountered, corresponding nodes are marked as not passable.
Figure 8 shows the outcome of the mapping process in greater details.

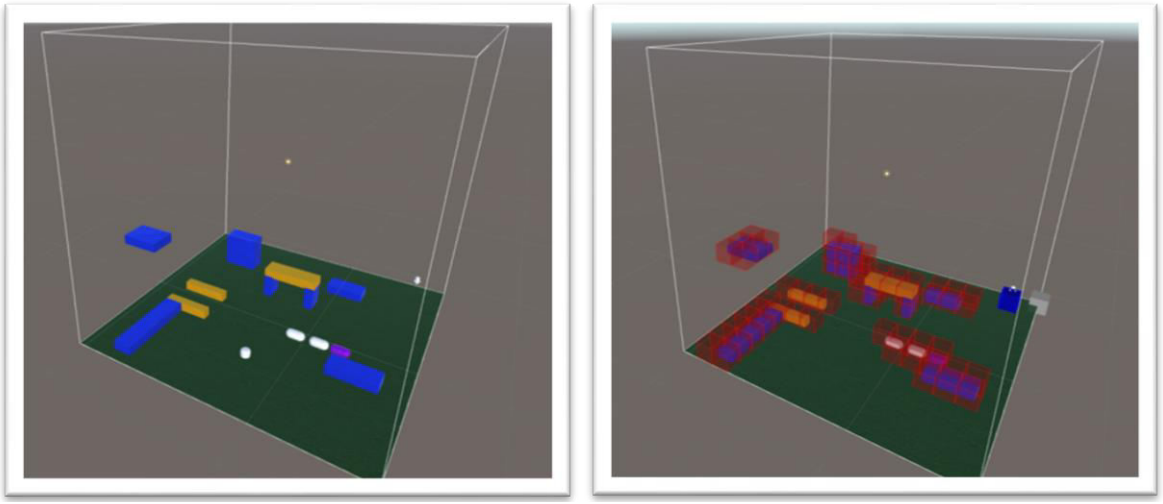


Figure 8 Example 3D world before the mapping process (left hand side) and the same world with not passable Nodes (obstacles) highlighted (right hand side).

Number of nodes within the structure is dictated by the size of the world as well as the size of the node. For example, the world size of 30x30x30 units length and node diameter of 2 unit length will require 3375 nodes to be created, whereas the same world size but with node diameter set to 1 unit length, will require 27000 nodes to be created (see: Figure 9).

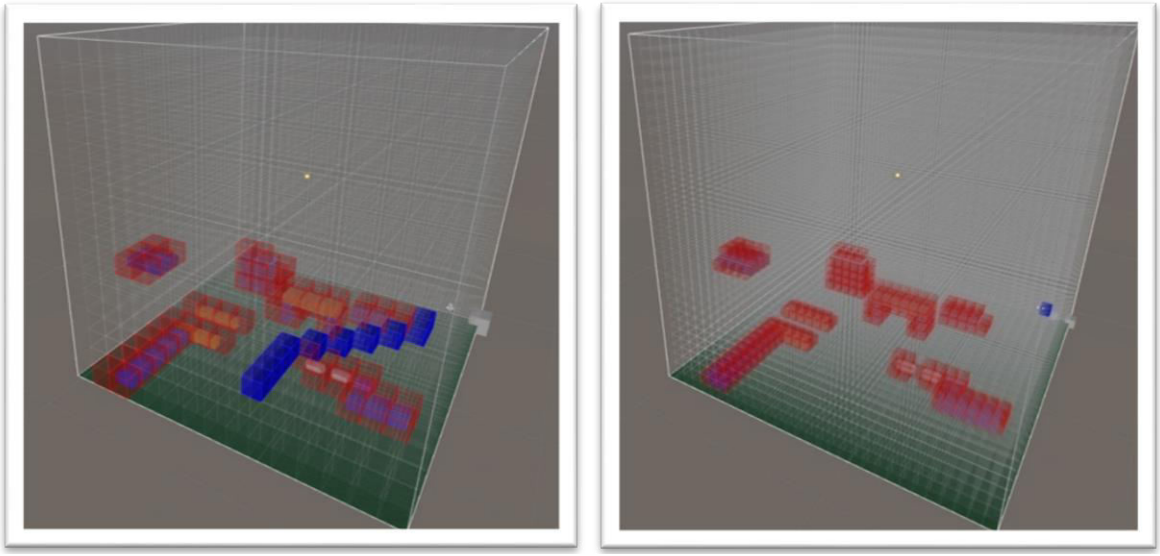


Figure 9 Example 3D world with size 30x30x30 with node of 2 units size (left hand side) and with node of 1 unit size (right hand side).

Grid class implementation also provides the method, which allows for direct mapping of any given point inside the world onto corresponding node from the data structure (`Node WorldPointToNode(Vector3)`). Methods, which provide functionality to draw the grid representation (or any of its elements) have also been added for diagnostic and debugging purposes.

PATHFINDING ALGORITHM CLASS

Implementation of A algorithm in 3D*

For the reasons discussed before, the algorithm chosen for the needs of current project was A* search. To start with, the algorithm was implemented in 2D and was then modified to account for the multilayered, open, 3D world.
The implementation is contained within `PathfindingAlgorithm` class.

Method `public List<Node> FindPath(Node, Node)` takes two nodes as its arguments and returns path in form of list of nodes (actually references to nodes). If the path is not found, the method will return `null`. `FindPath()` method internally calls an appropriate pathfinding sub-method, either: `FindPath_ListBased()`(see: Appendix C) or `FindPath_HeapBased()` (described later).

Class also contains two variants of method `public int GetDistance(Node, Node)`, which is responsible for calculating distance (expressed in nodes) between any given two nodes on the grid.
First variant calculates distance in 2D (using only X and Y coordinates).

The following pseudo code shows this operation in greater details:

```
GetDistance(Node1, Node2) – 2D VERSION

Step1: Calculate distance in nodes on the x axis using the following equation:

       Distance on X axis = Node1 X position - Node2 X position

Step2: Repeat the same for the y axis:

       Distance on Y axis = Node1 Y position - Node2 Y position

Step3: Evaluate (Distance on X axis > Distance on Y axis)

       IF YES:     --(Scenario1)--

              RETURN:

               Diagonal Cost Constant * Distance on Y axis +

               Linear Cost Constant * (Distance on X axis – distance on Y axis)

       IF NO:      --(Scenario2)--

              RETURN:

               Diagonal Cost Constant * Distance on X axis +

               Linear Cost Constant * (Distance on Y axis – distance on X axis)
```

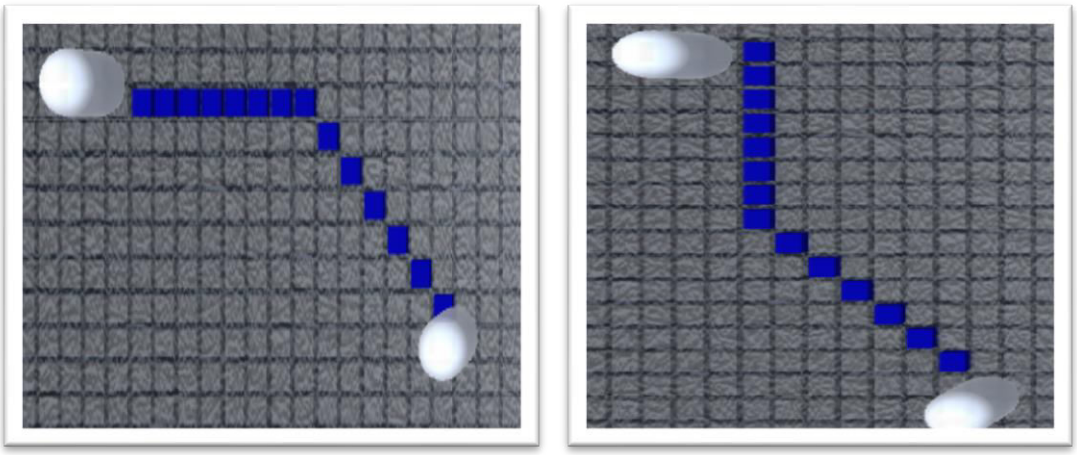


Figure10 Scenario 1 highlighted (left hand side) and scenario 2 highlighted (right hand side)

Second variant calculates distance between nodes on 3D grid (using X, Y and Z coordinates). The following pseudo code shows operation in greater details:

```
GetDistance (Node1, Node2) – 3D VERSION

Step1: Calculate distance in nodes on the x axis using the following equation:

        Distance on X axis = Node1 X position - Node2 X position

Step2: Repeat the same for the y axis:

        Distance on Y axis = Node1 Y position - Node2 Y position

Step3: Repeat the same for the z axis:

        Distance on Z axis = Node1 Z position - Node2 Z position


*** Results X,Y and Z distance calculation are placed in the three element Distance Array ***


Step5: Sort Distance Array from smallest to biggest value

Step6: Calculate distance using the following formula:

        Resultant distance =

                Diagonal Cost Constant * Smallest Distance +

                Diagonal Cost Constant * Medium Distance +

                Linear Cost Constant * (Biggest Distance – Medium Distance- Smallest Distance)


Step7: RETURN Resultant Distance.
```

Although the heart of the algorithm remains virtually unchanged for 2D and 3D versions (except for the distance measuring method mentioned before), the actual data structure, on which the algorithm operates differs. In 2D version, two dimensional array of type `Node` is the core of the Grid Structure, whereas in 3D version, three dimensional array serves as core. Further differences are associated with `Grid`'s member methods responsible for the grid creation, operation, and communication with the rest of the system:

1.Method: `private void CreateGrid()`

Method's first task is to create and populate structure's array of nodes, while also scanning the world area (defined by the given boundary) for static obstacles (such as mountains, buildings, etc). The process referred to as "scanning" is the process of checking the small, neighbouring, nodes size areas "one by one" for intersection with world objects marked as obstacles (objects with layer mask = "Obstacle"). When the new node is created, its state is set based on the current area scan result.

Method starts the scan at the world's bottom left corner and progresses from there using linear pattern. In 2D, world area is scanned column by column, until the right top corner is reached (see: Figure 11), whereas in 3D, world is scanned from left to right, slice by slice, with each slice being scanned by row (see: Figure 12). The process has the cyclomatic complexity of $O(n^2)$ for 2D grid and $O(n^3)$ for 3D grid.

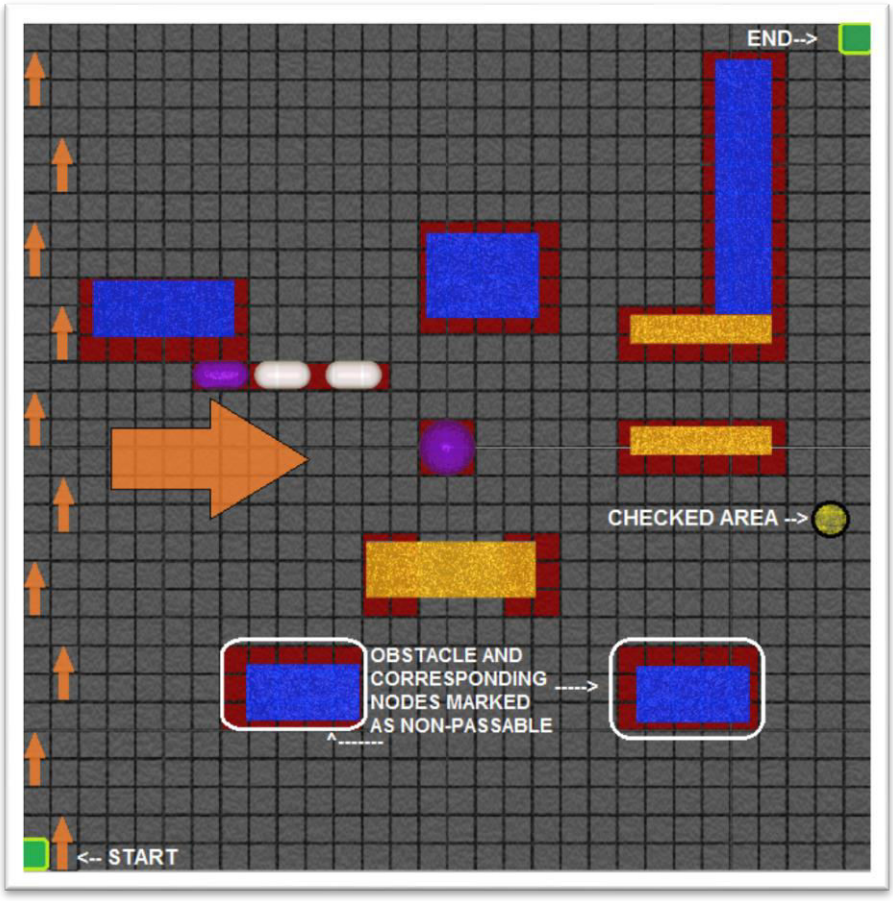


Figure 11 Create Grid Method in 2D

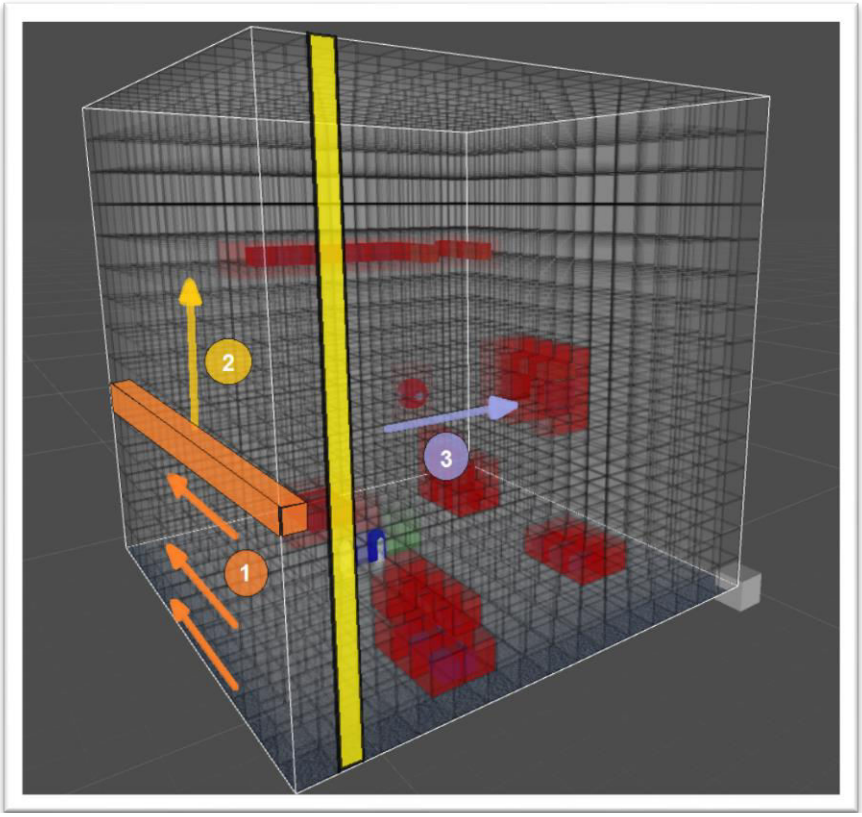


Figure 12 Create Grid Method in 3D

The second task of the method `CreateGrid()` is to assign neighbours to each node on the newly created grid and thus form the network of multi-connected nodes.

This process requires additional pass through the newly created node array. While the iteration takes place, neighbour assignment operation is performed for each node (see: Figure 13). Single node assignment uses set of nested for-loops (two for 2D, three for 3D) to determine the neighbours of a given node. The number of iteration for each for-loop is always 3. This is because the areas are checked on each side of given node in all dimensions. Single assignment has the cyclomatic complexity of $O(3^2)$ for 2D and $O(3^3)$ for 3D.

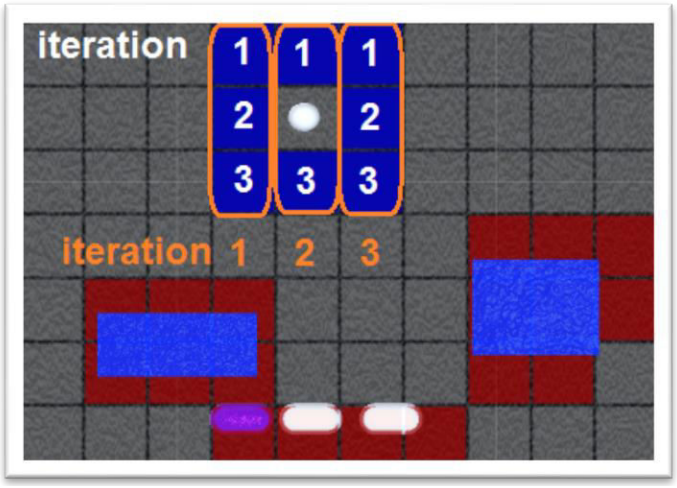


Figure 13 Assigning Neighbours in 2D

A optimisation using heap structure*

Usage of binary heap structure as a storage vehicle for open set can significantly increase algorithm speed by reducing the number of iterations needed to find the most promising node.

Binary heap pushes element with the highest priority to the top of the tree structure, allowing it to be accessed fast, without the need to search through other elements. Each new element is placed at the appropriate place in accordance to its priority. Although the underlying structure behind the heap is the one dimensional array with the elements ordered from left to right, it is the set of insert and access rules that defines its behaviour. The main rule is that each parent node's value must be less than any its two children. Each newly added node is inserted at the bottom of the tree (end of the array), then compared against the parent and if its value is bigger than parent's, swapped with it. This process is repeated until nodes value is no longer bigger than parent. This way the element with the smallest value will always be located at the top of the tree (first element in the array).

If the top node is "popped" from the heap, the balance is restored in two stages:

Stage 1:

The node assigned to the last element of the array is reassigned to the first element.

Stage 2:

The new top element is compared against its children and swapped with the one that has lower value. This process is repeated until the node is placed at the correct place.

The number of operations needed to complete this above actions is relatively small if compared with number of iterations needed to navigate through list of the same size. Heap structure complexity is presented in Table 1.

	Average	Worst
Push (add)	O(1)	O(log n)
Pop	O(1)	O(1)

Table 1 Complexity of Heap Structure expressed using Big O Notation.

To optimise the A* search a new method was created (`FindPathHeapBased(Node,Node)`, that uses heap structure rather than the list (see: Appendix D). A significant increase in algorithm speed was found (see: Table 2 in Testing section).

A flexibility (switching between the algorithms)*

As the A* algorithm is a combination of Dijkstra's algorithm and Best First Search algorithm, it can be easily modified to work as any of these, provided some alterations of the F_Cost in the Node Class.

To work out the F_Cost of each node A* search uses the G_Cost (distance from the start node to the current node) and H_Cost (distance from the current node to the goal):

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

Dijkstra's algorithm uses only the G_Cost to inform its search, and thus if Dijkstra's algorithm behaviour is desired, F_Cost of each of node must be set equal to its G_Cost, as it does not have the H_Cost:

$$F(n) = g(n) + 0$$

Best First Search on the other hand uses only H_Cost to navigate its search. For Best First Search behaviour, each node's F_Cost must be set equal to its H_Cost, as it does not have the G_Cost:

$$F(n) = 0 + h(n)$$

All three options were included in the implementation of the autopilot. The flexibility of the system design (the cost of the nodes stored in the Node Class rather than in the algorithm itself) allowed for switching between the algorithms without any changes being made to the actual algorithm implementation. This also allowed for further cost alterations to account for various environment factors such as bad weather, terrain type etc.

AUTOPILOT MANAGER CLASS

This module is responsible for the appropriate usage and application of the algorithm in order to make the pathfinding process as efficient and as robust as possible.

Controlling Movement

Pathfinding Manager is implemented as unity's `MonoBehaviour` script and therefore, designed to be attached to unity's game object, which will be controlled by it (directed towards the target). It is responsible for actual movement of the entity, including its heading, rotation and speed.

Calling Algorithm

Pathfinding Manager is responsible for calling the pathfinding algorithm, anytime the path calculation is needed.

Monitoring Pathfinding Efficiency

Pathfinding Manager also provides functionality to monitor various parameters, which define the efficiency of the pathfinding process, such as number of algorithm method calls, total pathfinding time, average pathfinding time, number of obstacles encountered. It also uses a method, which saves these results into .csv file.

Dynamic Avoidance Mechanism

The most important role of this component is to employ A* algorithm in such way that it becomes useful when avoiding dynamic (moving) obstacles. As the development of the dynamic obstacle avoidance was the most challenging part of the project, it will be now discussed in detail.

Version 1 Recalculating path with each move.

Dynamic obstacles can appear on the entities way at any given time. One way of dealing with this problem was to keep recalculating the path as often as possible to avoid surprise collision. However, as the biggest concern here is the time needed for path to be recalculated (which is especially true for 3D environment, where the number of nodes can easily reach couple of millions), there was a need for a efficient way of doing so. It became apparent that recalculating a new path with each program cycle was not a viable option. Instead, the algorithm was called only once, each time the entity moved from one node the next one (see: Figure 14).

Problem: This solution still suffered for constant recalculation of the path, which was not acceptable solution.

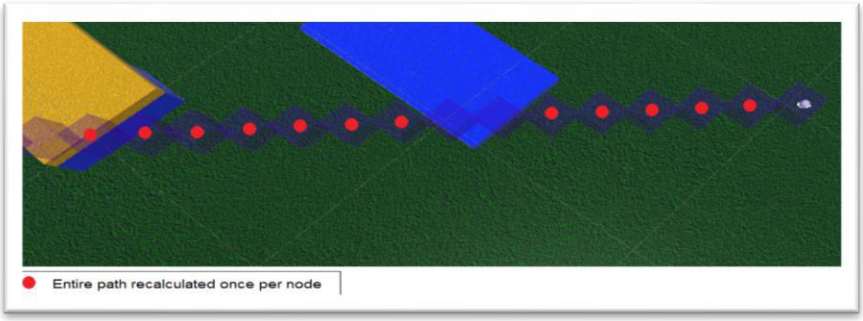


Figure 14 Points at which new path was recalculated.

Version 2 Path Lookup Mechanism

In order to cut down algorithm calls, Path Lookup Mechanism was introduced. Algorithm was run once at the beginning of the program's execution and the path that was found was stored by the Autopilot Manager inside the dedicated list. The entity would follow the path node by node; at each arrival removing passed node from the list (path), and check if any of the remaining nodes in the path changed their state to not passable. If such situation occurred, the path was recalculated (see: Figure 15).

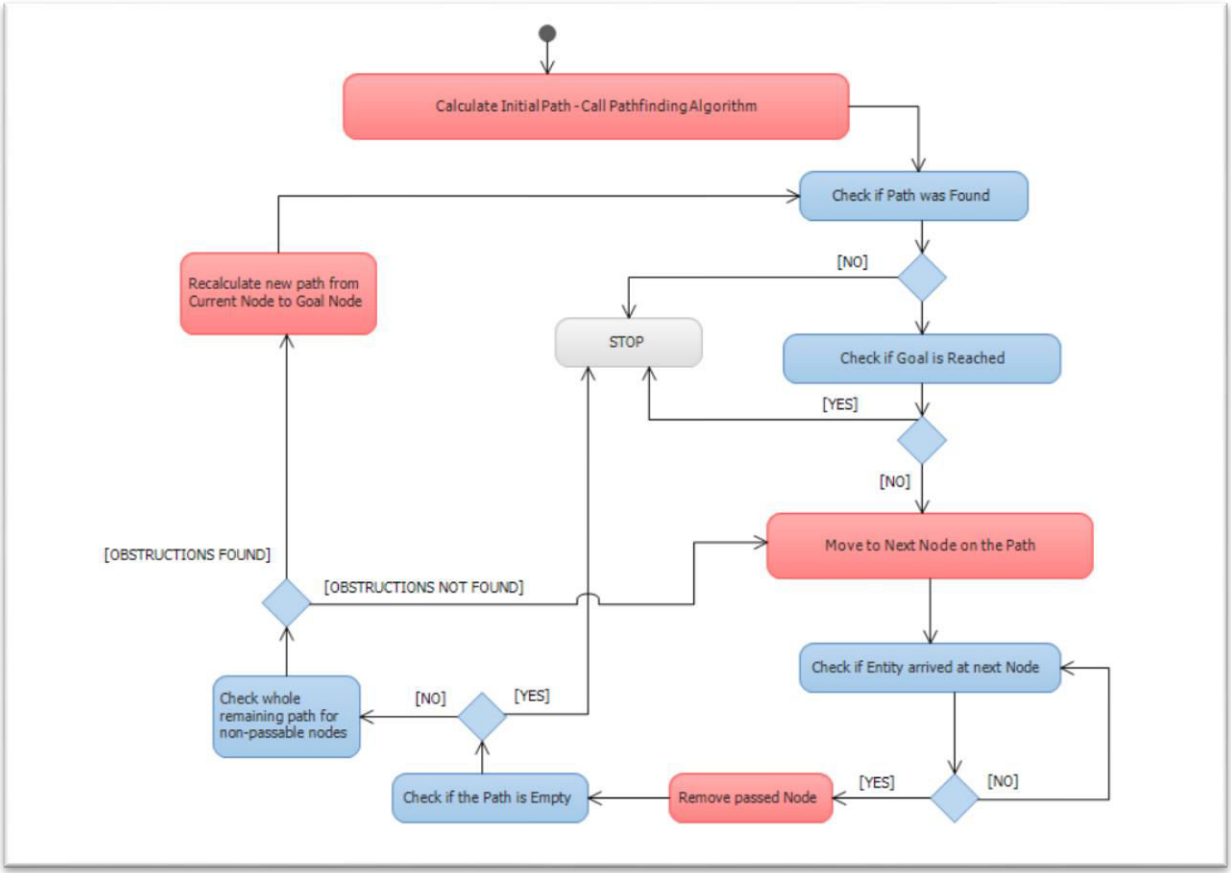


Figure 15 Path Lookup Mechanism Flow Chart

Problem:

This approach was working relatively well if there was a small number of dynamic obstacles present and not frequently crossing entity's path. However, when the

obstacle count and spawn frequency was increased, number of algorithm method calls quickly overburdened the processor.

Version 3 Recalculating Only Path Fragment

Improvement to the Path Lookup Mechanis was introduced to accommodate for a large number of obstacles. After arrival from one path node to the next one, only first few (5) nodes (of the remaining path) were checked for state changes. This helped to reduce number of algorithm method calls significantly, as it was only called when the moving obstacle crossed the entity's path in close proximity. At this point another optimization was introduced. It was discovered that there was no need to recalculate the whole path each time when close proximity crossing event occurred. Functionality was implemented to recalculate only the part of the path that became obstructed. As a result, the old obstructed fragment of the path is removed and newly recalculated one is added to the path followed by the entity (see: Figure 16).

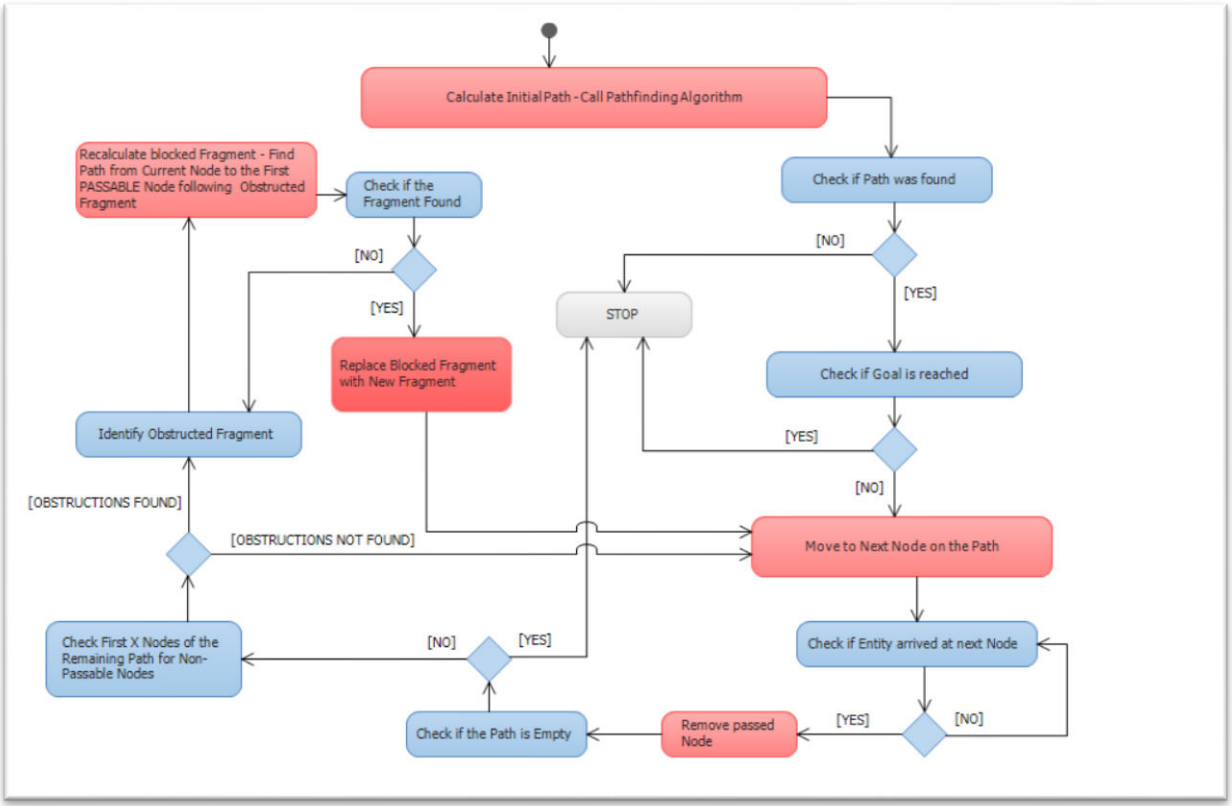


Figure 16 Updated Flow Chart for Improved Path Lookup Mechanism

True Dynamic obstacles

Initially dynamic obstacles were simulated by simple algorithm designed to mark given number of nodes along the path, at given time rate, as not passable. Although this testing approach proved helpful during initial stages of the development, more realistic and sophisticated obstacle emulation method was implemented. New method allowed for dynamic obstacle creation and simulation. The method uses object “spawners” that can instantiate any number of objects from given (unity) prefab. Virtually unrestricted number of obstacle “spawners” can be created (if factors such as hardware limitations are not considered) and placed at any given position in the simulation world. Each “spawner” can be set to “produce” chosen number of obstacles per given volume, per given amount of time . Obstacles are destroyed once they move outside the arbitrary boundary, or after certain amount of time has passed. This is done for performance reasons. Asteroid model of diameter

approx. 1 unit was chosen to represent an obstacle and was set to move at speed around 0.8 Unit per second (see: Figure 17).

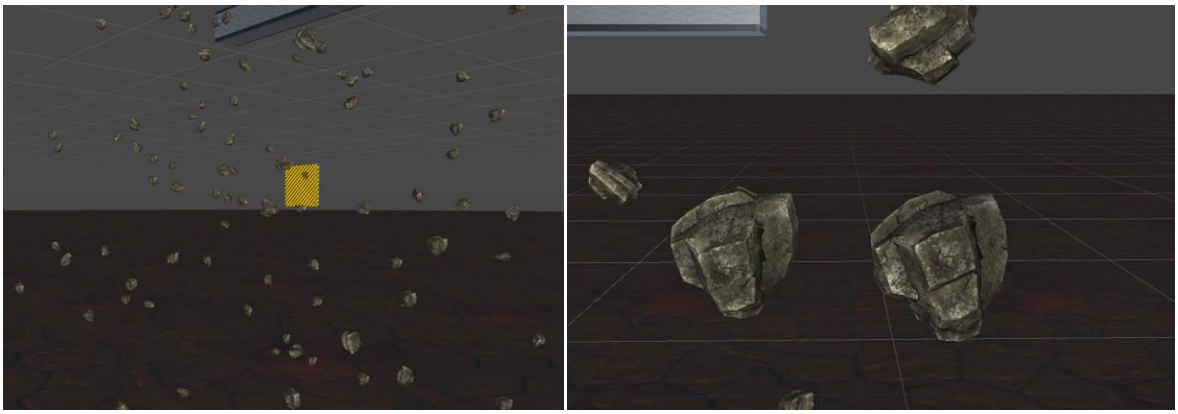


Figure 17 Dynamic Obstacle Spawner (left hand side) and dynamic obstacle close up (right hand side)

Once more realistic obstacles were introduced it became necessary to equip avoidance system with the tool, which will allow it to discover and recognize those obstacles when they in close proximity. First attempt was to look few nodes ahead in the current path for possible collision with the node sized sphere collider. If the collision occurred node was marked as not passable and path was fixed with new recalculated fragment (see Figures: 18, 19, 20).

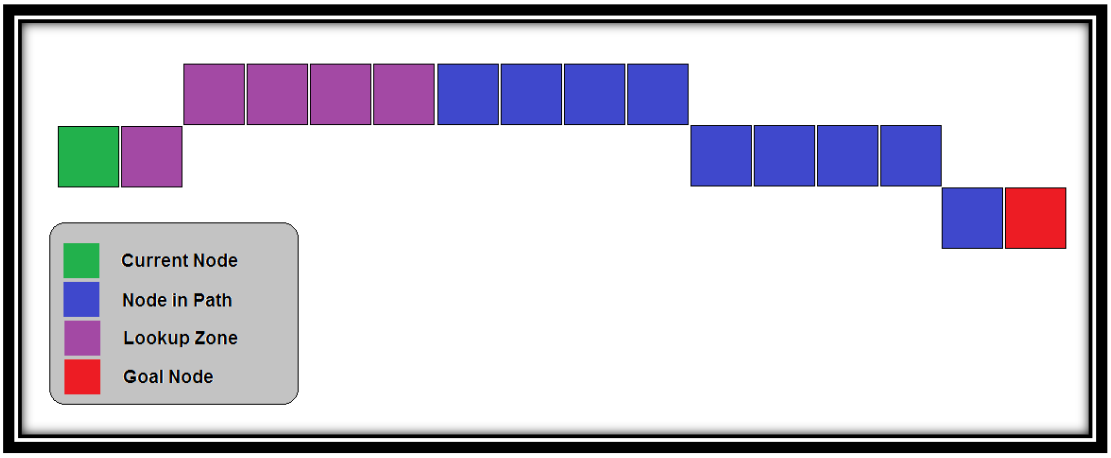


Figure 18 – Lookup zone identified.

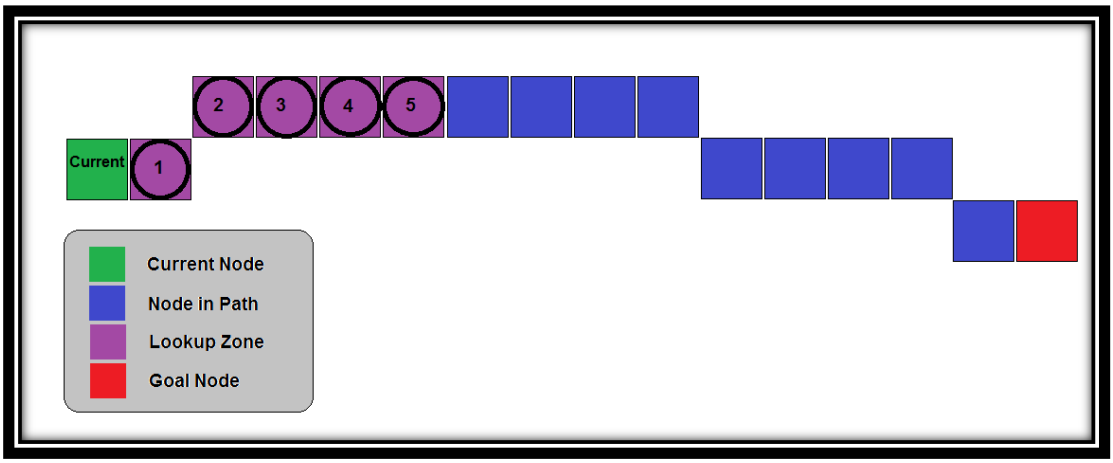


Figure 19 – Lookup zone being checked for obstacles.

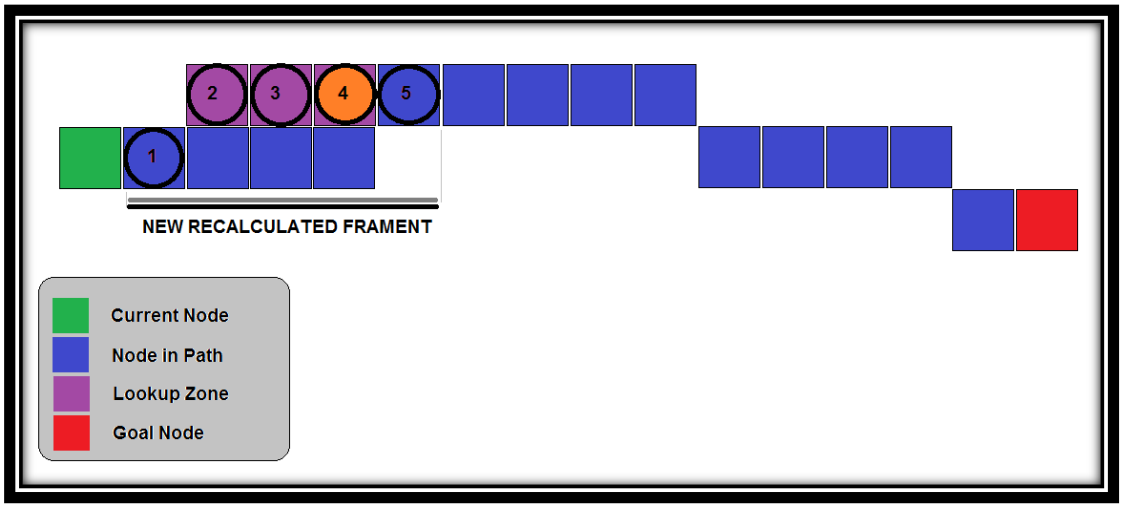


Figure 20 – Obstruction found at node four, path fixed with new fragment.

Problem:

Although promising, this approach proved to be rather ineffective in avoiding, even relatively small amount of asteroids crossing entity's path, while the entity was travelling through low density asteroid field (50 asteroids per 27000 Distance Units Cubed(30x30x30)). The lack of effectiveness could be explained by system's inability to "see" obstacles that are not yet crossing entity's path but are in close proximity and very likely to do so. By the time they are detected there is no room for taking evading action and the collision with obstacle occurs (see: Figure 21).

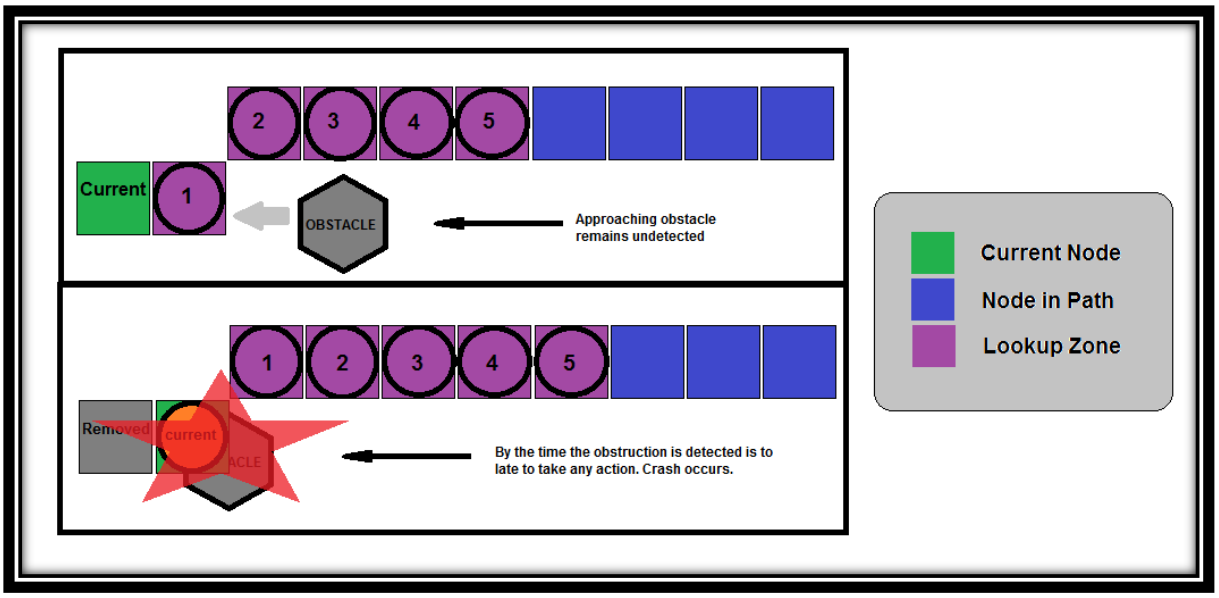


Figure 21 – Shortcomings of Path Lookup Mechanism.

Version 4 Extended Path Lookup

The alternate mechanism was introduced in which not only nodes in the path are checked for collision but also their neighbours. Nodes that are found to contain obstacles are temporarily marked as not passable and added to dedicated short-hand reset list. Then pathfinding algorithm is called to update (fix) the affected path fragment.

Once per given amount of time (in this case 0.5 second) nodes in reset list have their passable state reset and the list is cleared. Separate dedicated thread is assigned to perform this task in the background. This is done to account for dynamic nature of the environment. As asteroids (obstacles) move constantly from place to place (from one node to another), marking each node that happened to contain an obstacle at certain moment, permanently as non-passable would result in a inaccurate grid state

and suboptimal pathfinding results. Clearing dynamically altered node states periodically, appeared to address this issue.

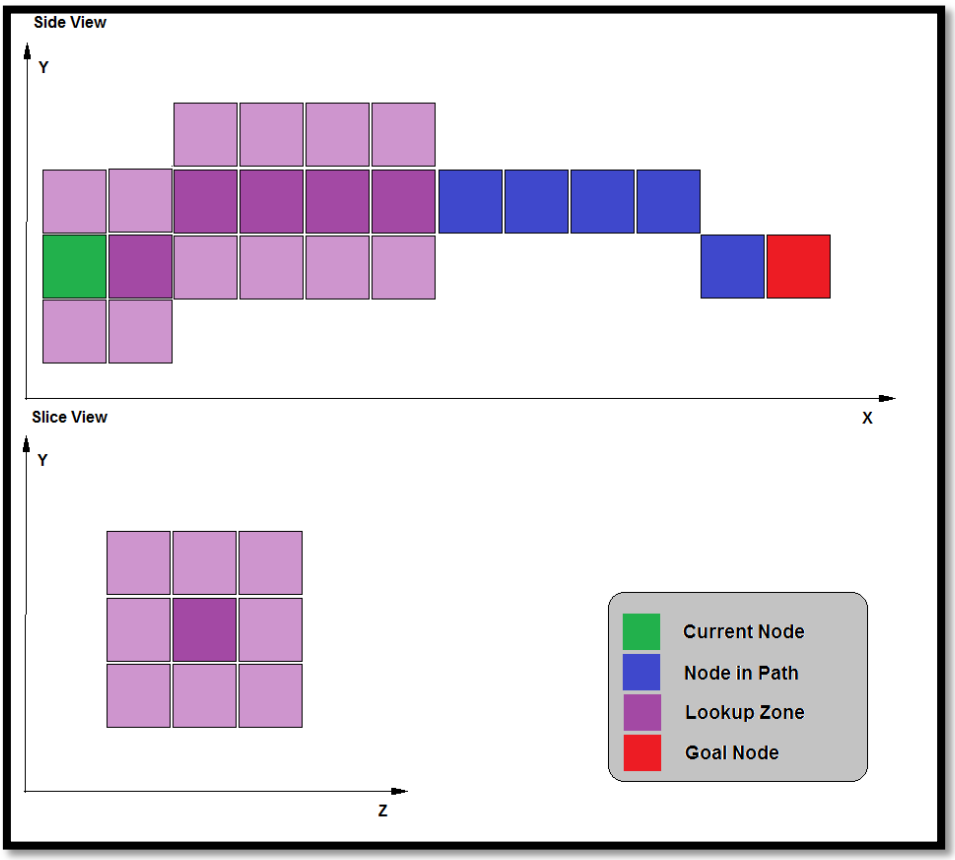


Figure 22 Extended Lookup Mechanism
- Neighbours of Lookup Zone Nodes also checked for obstacles.

Problem:

Although this version introduced substantial improvement in avoiding moving obstacles, some collisions were still occurring especially when the entity was flying trough denser asteroid fields (300 asteroids per 80000 Distance Units Cubed(40x40x40)).

Version 5 Speed Control System

Next version introduced autopilot speed control system and collision avoidance system. The latter simulates short distance radar and is realized in the program with a simple sphere collider (see: Figure 23). If an obstacle appears in the near proximity of the ship (four units for this setting) the autopilot will get notified and accelerate the in order to move away from it (see: Figure 24). Autopilot uses provided space ship specification data such as: Normal Cruising Speed [Unit/second], Maximum Speed[Unit/second] and Acceleration Rate[Unit/Second^2]. As the path fragment is recalculated each time the movement from node to next node is completed, increasing speed implicitly enforces higher rate of path recalculation, which serves as additional protection mechanism.

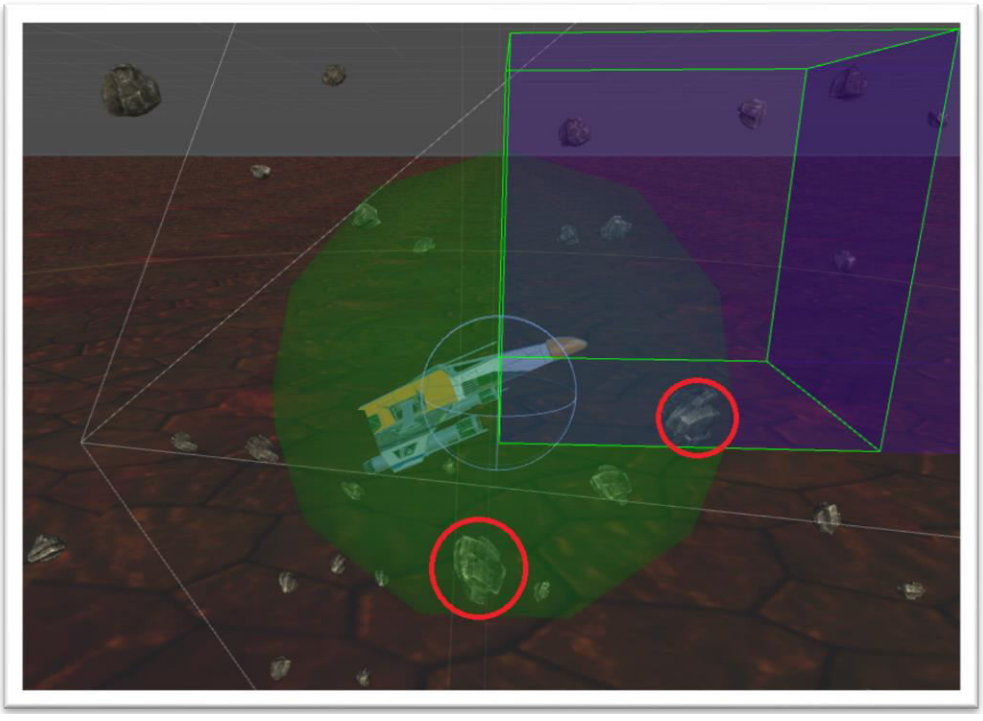


Figure 23 Collision Avoidance System (Radar)

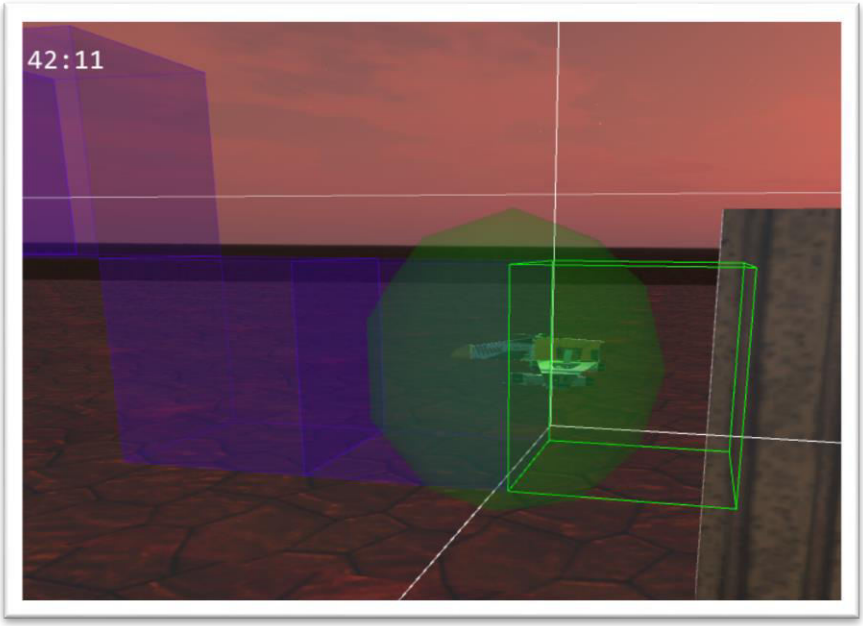


Figure 24 Speed Control System

WEATHER AVIODANCE

Weather Avoidance System was developed in order to maximise autopilot's safety whilst minimising the fuel consumption. Nodes encompassing areas of bad weather such as rain or fog have additional cost added. This forces the pathfinding algorithm to disfavour these nodes and consider them only as the last resort.

As it relies on dynamic avoidance mechanism it can only be employed when it is active.

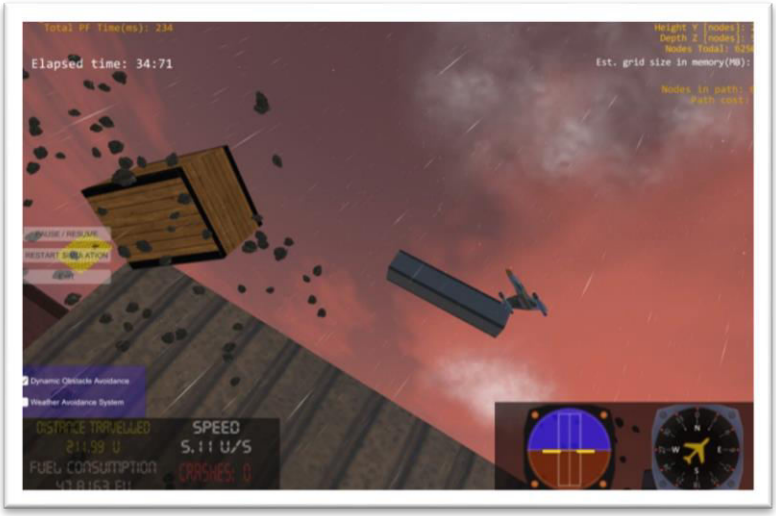


Figure 25 Example Rain Scene

FUEL CONSUMPTION MONITOR

During the later stages of development factors such as fuel consumption and fuel efficiency became subject of focus. Fuel Management class was created in order to calculate the amount of fuel burned and keep track of it.

Factors accounted for included:

- Distance travelled
- Fuel burning factor (amount of fuel burned per unit distance travelled)
- Gravity pull (types of movement: ascending, cruising, descending)
- Speed
- Weather conditions

The fuel burning factor per unit distance was set to 0.2, which allowed to calculate fuel burned for any distance travelled in straight flight (pitch angle = 0). Amount of fuel burnt is calculate per frame and thus, allows fuel indicators to updated in real time. In order to obtain change in fuel burnt per frame change in distance per frame had to be calculated first:

$$\Delta S = \Delta V \times \Delta t,$$

where S is the distance travelled, V is the speed and t is the time.

Fuel burned change per frame was then calculated by the equation:

$$\Delta \text{ Fuel Burned} = \text{Burning Factor per Unit Distance} \times \Delta S.$$

To account for gravity pull, pitch angle of the ship had to be considered (see: Figure 26). Maximum pitch of angle of this particular ship was set to 45°. At the maximum pitch angle (45°) fuel consumption was assumed to be 150% of the normal cruising fuel consumption rate (burn coefficient 1.5). At the minimum pitch angle (-45°) fuel consumption was assumed to be 75% of the normal cruising fuel consumption rate (burn coefficient 0.75).

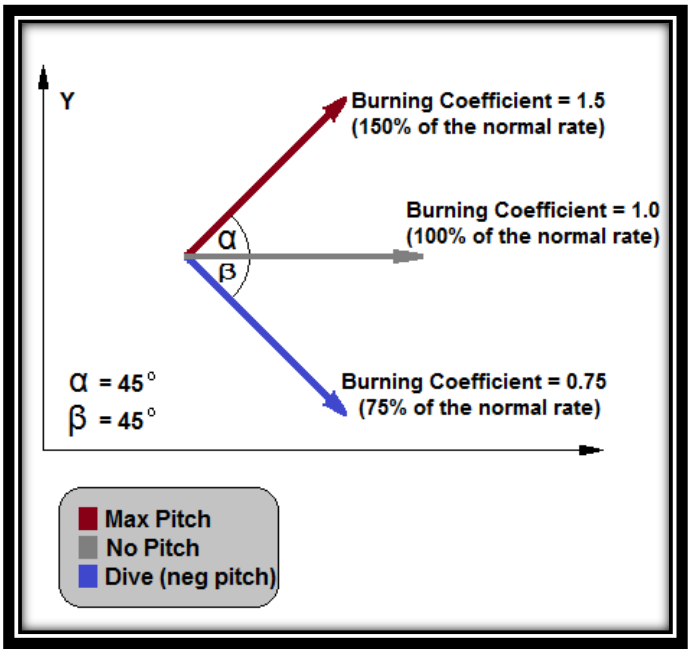


Figure 26 Fuel Burning at Corresponding Pitch Angles

The following formulas were used to calculate fuel burning coefficient (which is) based on the ship's pitch angle:

$$Max\ \Delta\ for\ Pitch = \ Burning\ Coefficient\ for\ Max\ Pitch - 1$$

Then:

$$Pitch\ Burning\ Coefficient = 1 + \frac{Max\ \Delta\ for\ Pitch\ * PitchAngle}{Max\ Pitch\ Angle}$$

In order to achieve linear range from 75% to 150% pitch angle significance had to be decreased by the factor of two when the angle was below zero.

If(pitch angle is negative) pitch angle = pitch angle/2.

If the ship is travelling through bad weather conditions, the amount of burned fuel calculated is multiplied by the factor of two.

AUTOPILOT FINAL VERSION – Combining it all together

Final version of the autopilot included following features described earlier:

- Extended Path Lookup Mechanism
- Speed Control System
- Collision Avoidance System
- Weather Avoidance System
- Fuel Consumption Monitor

Combining all these features together provided maximum safety while maintaining the efficiency of the autopilot. For the static game worlds, the dynamic obstacle avoidance mechanisms (such as Path Lookup) can be switched off, improving the speed of the pathfinding process.
For the dynamic game worlds, weather avoidance system can be switched off is there is no need for it. All these, result in the autopilot being flexible and easily adjustable for variety of game worlds.

Figure 27 Final Autopilot Mechanism at Work with the Path Visible

4.2.3 User Interface Implementation

User interface implementation utilises the functionality provided by Unity's canvas system. It contains a set of indicators and interactors allowing the user to monitor and control various aspects of the pathfinding and simulation processes (see: Figures 28 and 29).

Figure 28 As weather avoidance system is not selected, the ship is flying through the rain zone.

Figure 29 The change in artificial horizon can be seen as the ship turns.

Indicators can be divided into two groups:

- First group located at the top of the screen shows algorithm-specific information such as pathfinding time, data structure size and memory footprint as well as number total of A* calls.
- Second group of indicators located at the bottom of the screen shows the autopilot and flight related information such as speed, heading, artificial horizon, distance travelled and the fuel burned. During design and the creation stages of this part of the UI, the best effort was made to produce indicators that would reassemble real-life in-flight instruments and be truthful and informative at the same time. For both artificial horizon and heading indicators a set of bitmaps representing various parts of the instrument was created. These were superimposed on each other in order to create desired appearance. Used images can be subdivided into two categories: body images, and moving part images. Parameters such as ship rotation vector, pitch angle and heading vector are used to transform position and rotation of

the of the moving instrument parts and thus producing the desired visual effect (see: Figure 30).

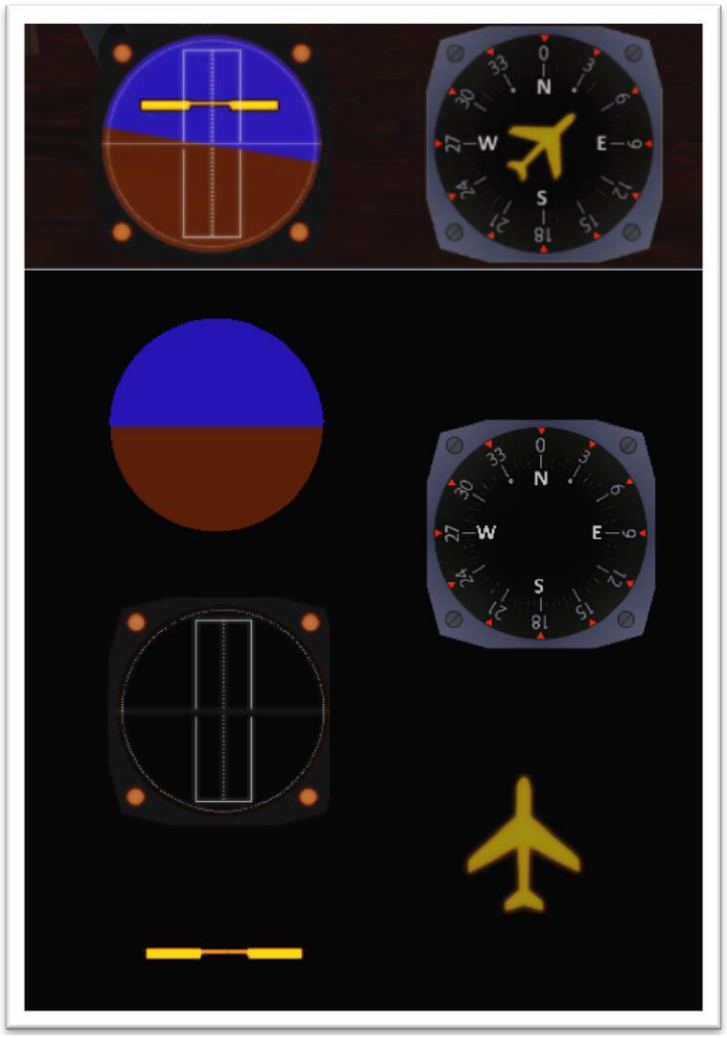


Figure 30 Bitmaps Used to Create Indicators

Interactors can also be divided into two categories:

- First category encompasses application and simulation controls. It is basically a set of buttons allowing the user to pause, resume or restart the simulation and exit the application when required.
- Second category represents flight-specific controls. In this case there are two check boxes responsible for activation/deactivation of Dynamic Obstacle Avoidance System and Weather Avoidance System. As Weather Avoidance

System will only work when the Dynamic Obstacle Avoidance System is activated, the check box will be inactivated if the latter is switched off.

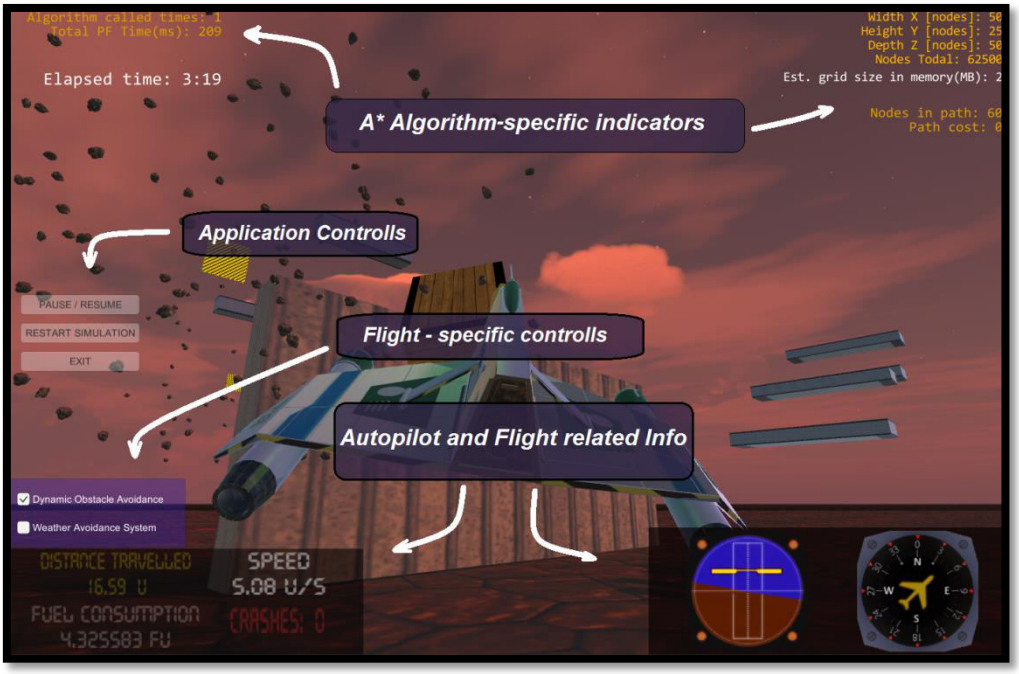


Figure 31 Specific Parts of User Interface Described.

4.3 Experiments

A number of experiments was run to assess the efficiency of the proposed algorithm. Each of this will be now discussed in turn.

4.3.1 Experiment 1 List vs. Heap

This experiment was to compare the efficiency of the A* algorithm implemented with a list versus a heap structure. The results of this test can be seen in a Table 2. The implementation of the A* algorithm that uses heap structure proved to be much faster than implementation of the A* that uses the list. The improvement is the bigger the larger the world size.

	List	Heap
World Size 30x8x30		
No Obstacles	6 ms	1 ms
Static Obstacles	265 ms	15 ms
World Size 60x15x60		
No Obstacles	17 ms	1 ms
Static Obstacles	4190 ms	77 ms
World Size 120x30x120		
No Obstacles	63 ms	4 ms
Static Obstacles	86078 ms	194 ms

Table 2 Average Pathfinding Time (in milliseconds) for A* algorithm as implemented with either list or heap structure (as averaged over a 100 runs).

4.3.2 Experiment 2 Dijkstra's, Best First Search and A* Comparison

This experiment was conducted to compare pathfinding time of three main direct search algorithms: Dijkstra's Search, Best First Search and A* search. The results of this test can be seen in Table 3.

	Dijkstra's Search	Best First Search	A* Search
World Size 30x8x30			
No Obstacles	50 ms	1 ms	1 ms
Static Obstacles	45 ms	1ms	15ms
World Size 60x15x60			

No Obstacles	475 ms	2 ms	1 ms
Static Obstacles	397 ms	2 ms	77 ms
World Size 120x30x120			
No Obstacles	3980 ms	8 ms	4 ms
Static Obstacles	N/A	18 ms	194 ms

Table 3 Average Pathfinding Time (in milliseconds) for each algorithm (as averaged over a 100 runs).

The Dijkstra's algorithm is the least efficient one, with its performance significantly lower than both best first search or A* algorithm. For the world free from obstacles, the A* search performs the best, however, it is outperformed by best first search when obstacles are introduced. This can be due to number of reasons; specific configuration of obstacles on the map, low density of obstacles etc. However, as it was mentioned before even though the particular workings of best first search may speed up its search performance, it may not necessarily find the most optimal path. Indeed, after a closer examination, it was found that the paths found by best first search were longer than these found by A*. The difference was the bigger, the larger was the search space (see Table 4).

World Size	Best First Search	A* Search

30x8x30	33	28
60x15x60	67	53
120x30x120	156	114

Table 4 The comparison of path length returned by Best First Search and A* search in differently-sized worlds cluttered with static obstacles.

4.3.3 Experiment 3 Dynamic Obstacle Avoidance Mechanism

This experiment aimed at comparing different versions of dynamic obstacle avoidance mechanism in terms of its efficiency and safety. Every version of the mechanism was tested on the same multilayered, 3D world with large number of both static and dynamic obstacles present. This was done in order to assess the limitations of each version of the mechanism and aid continuous improvement of the autopilot.

Parameters of Test World:

- Size 200x100x200 (2MB in memory)
- Total Nodes 62500
- Static Obstacle Nodes 2079

A number of dynamic nodes (obstacles) crossing the path of the autopilot was also calculated. A set number of dynamic obstacles was being produced per given

volume per given amount of time. The dynamic obstacles spawners have been positioned in such way to throw the obstacles towards the entity's way. However, the was no actual control over how many of these obstacles would actually cross entity's path. For this reason, each version of the dynamic mechanism was run 20 times. The results presented below are the average results across these 20 runs.

Experiment 3a

This experiment was a control run and involved autopilot navigating through the dynamically changing environment without any avoidance mechanism.

	Control Group
Pathfinding Time (ms)	204
Algorithm Calls	1
Path Length	60
Collisions	4.7

Table 5 Average Pathfinding Results for Control Group for the Dynamic Avoidance Mechanism (as averaged over 20 runs).

As there is no dynamic avoidance mechanism employed in control runs, the algorithm is called only once at the beginning to calculate the path around static obstacles. As expected, there is a high number of collisions with dynamically moving obstacles (see Table 5).

Experiment 3b

This experiment was a comparison of recalculating path fragment with every move and recalculating path fragment only when obstacle on that fragment has been detected.

	Control Group	Recalculating Path Fragment with Each Move	Recalculating Path Fragment when Obstacle Detected
Total Pathfinding Time (ms)	204.2	226.5	222.6
Number of Algorithm Calls	1	61	15.83
Path Length	60	60	60
Nodes Processed as Dynamic	NA	27.3	26.7
Collisions	4.7	3.6	1.5

Table 6 Average Pathfinding Results for Path Lookup Mechanism with different method of path recalculation (as averaged over 20 runs).

In comparison to control group, total pathfinding time has increased. This is to be expected, as the dynamic avoidance mechanism requires additional algorithm calls to safely navigate the world. Although total pathfinding time did not differ too much with the autopilot recalculating a new path fragment only when obstacle is detected rather than with every move, the number of algorithm calls decreased greatly and most importantly, safety of the dynamic mechanism has increased (see: Table 6). This can be due to the fact that recalculating new path fragment only when obstacle is detected makes the autopilot movement more balanced and controlled, thus less likely to overreact to changes of environment that might have not had impact on its path.

Experiment 3c

This experiment assesses the effectiveness of Extended Path Lookup mechanism, in which not only the path fragment is considered but also neighbouring nodes of that path fragment.

	Control Group	Recalculating Path Fragment when Obstacle Detected	Extended Path Lookup Mechanism
Total Pathfinding Time (ms)	204.2	222.6	221.5
Number of Algorithm Calls	1	15.83	13.5
Path Length	60	60	60
Nodes Processed as Dynamic	NA	26.7	589.83
Collisions	4.7	1.5	0.3

Table 7 Average Pathfinding Results for Extended Path Lookup mechanism in comparison to original Path Lookup mechanism (as averaged over 20 runs).

The Extended Path Lookup mechanism has a great effect of the safety of the autopilot reducing the average collision rate by 80% while maintaining the total pathfinding time at the same level (see: Table 7).

Experiment 3d

This experiment was aimed at assessing the Autopilot Speed Control System and Collision Avoidance System improvements that allow the autopilot to accelerate

when in immediate danger of collision. As it can be seen in Table 8, again the safety of the autopilot substantially increased, with over 80% less collisions.

	Control Group	Extended Path Lookup mechanism	Autopilot Speed Control and Collision Avoidance
Total Pathfinding Time (ms)	204.2	221.5	213
Number of Algorithm Calls	1	13.5	11.3
Path Length	60	60	60
Nodes Processed as Dynamic	NA	548	535
Collisions	4.7	0.3	0.05

Table 8 Average Pathfinding Results for Autopilot Speed Control and Collision Avoidance mechanism in comparison to Extended Path Lookup mechanism on its own (as averaged over 20 runs).

4.3.4 Experiment 4 Final Autopilot Mechanism

Since the dynamic obstacle avoidance system with Extended Lookup Mechanism in combination with Autopilot Speed Control have been found to be the most effective, it will be used for the final tests to assess its flexibility under different circumstances.

Experiment 4a

The aim of this experiment is to compare the effectiveness of the autopilot system in words varying in size but we the same percentage of static obstacles present. As it can be seen in Table 9, the safety of the algorithm has not changed. The A* pathfinding algorithms was called less times in the larger world, however, the total pathfinding time increased greatly.

World Size % of Static Obstacles	Size 200x100x200 (4%)	Size 300x100x700 (4%)
Total Pathfinding Time (ms)	213	1587
Number of Algorithm Calls	11.3	7
Path Length	60	158
Nodes Processed as Dynamic	535	784
Collisions	0.05	0.05

Table 9 Average Pathfinding Results for Final Autopilot Mechanism in differently-sized worlds but with equal number of both static and dynamic obstacles (as averaged over 20 runs).

Experiment 4b

In this experiment, the size of the world remained the same, however the number of static obstacles was different.

World Size % of Static Obstacles	Size 200x100x200 (4%)	Size 200x100x200 (16%)
Total Pathfinding Time (ms)	213	353
Number of Algorithm Calls	11.3	8.75
Path Length	60	112
Nodes Processed as Dynamic	535	1458
Collisions	0.05	0.1

Table 10 Average Pathfinding Results for Final Autopilot Mechanism equally-sized world, same number of dynamic obstacles but with varying number of static obstacles – maze like (as averaged over 20 runs).

In the first world, the amount of obstacles was 4%, while it the second world the amount of obstacles quadrupled totalling to around 16%, set up in a maze like manner. As it can be seen, in the very cluttered world the safety of algorithm

decreased. However, this was due to occasional collisions with dynamic obstacles. This could have been explained by the fact that the maze like world did not have much space to manoeuvre around. As a result, a large number of dynamic obstacles had an overwhelming effect on the autopilot.

World Size % of Static Obstacles	Size 200x100x200 (4%)	Size 200x100x200 (16%)
Total Pathfinding Time (ms)	213	277
Number of Algorithm Calls	11.3	6.6
Path Length	60	112
Nodes Processed as Dynamic	535	1344
Collisions	0.05	0.05

Table 11 Average Pathfinding Results for Final Autopilot Mechanism equally-sized world, same number of dynamic obstacles but with varying number of static obstacles (as averaged over 20 runs).

Indeed, when the same number of obstacles is spread more evenly around the world, the safety performance of the autopilot remains equally good (see: Table 11). Having enough space to evade the obstacles, means that all the dynamic mechanism features can be used in full.

Overall, the autopilot effectively navigates through variety of game worlds, with varying number of static and dynamic obstacles. While there are specific types of game maps that can be more challenging for the autopilot to navigate through, e.g. maze-like maps, even under these circumstances its performance is only slightly affected. Since the algorithms used by the autopilot is the A* search, optimal path to the goal is found each time the pathfinding process is called. A number of protective

mechanisms combined together in the workings of the autopilot produced a reliable and well-built solution to the pathfinding problem in open, multilayered 3D world cluttered with dynamic obstacles.

5 Evaluation

5.1 Project Achievements

The aim of this project was to create a cluttered, multilayered 3D environment and develop an autopilot algorithm that can determine a safe and efficient path through it. Since all primary and most of secondary objectives have been realised, the project can be considered a success. The sub sections below describe in detail how each objective has been fulfilled and address any particular technical challenges, which had to be overcome during the development process. Areas of improvement that could be considered for further work are also identified and discussed.

5.1.1 Review of Primary Objectives

Objective 1 – Creating 3D environment

The aim of this objective was to create a simple visualization of a cluttered, multilayered 3D environment. To this aid the game world was created as an open space in form of asteroid field. Different type of obstacles have been introduced to the scene, including both static and dynamic obstacles, which can be easily manipulated in number and location. Dynamic obstacle spawners are especially interesting feature of the 3D word, with the number of dynamic asteroids per unit in time regulated, which is very useful for experimental purposes. The areas varying in weather conditions such as rain and fog have also been created. The game world has a very appealing feel with sounds complementing the scene.

Objective 2 – Developing an autopilot to navigate 3D space

The aim of this objective was to develop an autopilot to safely and efficiently navigate through a 3D cluttered environment. This objective has been fully met as

follows. Firstly, the state of 3D pathfinding techniques was assessed. Based on the knowledge gained from the background research the autopilot system was designed and successfully implemented to account for 3D nature of the world. The autopilot was developed to work in actual 3D open world rather than 3D world mapped onto 2D world, which is very important to note. A dynamic obstacle avoidance mechanism was effectively introduced, with the entity successfully manoeuvring around encountered dynamic obstacles. A series of experiments was run to assess performance of the autopilot under variety of circumstances e.g. different types of obstacles, amount of clutter etc. and all experimental results confirm the safety and efficiency of the proposed solution. The opportunities for further refinement are described in section 5.2.

5.1.3 Review of Secondary Objectives

Objective 3 – Accounting for Gravitational Pull and Fuel Consumption

The aim of this objective was to consider gravitational pull in the workings of the autopilot to make the simulation more realistic and to minimise fuel consumption by controlling for various factors, such as distance travelled, speed, pitch angle etc. To account for gravitational pull the cost of vertical movement was made more expensive. Fuel Consumption Monitor was created to calculate the amount of fuel burned during the flight. As the travelling through the bad weather conditions would increase the amount of fuel burned, the weather avoidance mechanism has been put in place to account for this and aim at minimising the amount of fuel burned.

Objective 4 – Creating a naturally looking path

The aim of this objective was to create a natural path for the entity to follow by smoothing out the path calculated by algorithm. This objective was only partially met due to the time constrains. The entity smoothly moves from node to node adjusting its rotation accordingly to its velocity vector. However, the rotation can be sometimes quite abrupt if the path changes rapidly due to dynamic obstacles.

Objective 5 – Creating Appealing and Informative User Interface

The aim of this objective was to create appealing and informative user interface. This objective was fully met with a number of additional features added during the implementation process. A number of application and simulation controls was created to aid user interaction with the simulation. The user is able to pause, resume or restart the simulation and exit the application when required, zoom in/out, select the camera as well as to choose preferred autopilot options. In this case there are two check boxes responsible for activation/deactivation of Dynamic Obstacle Avoidance System and Weather Avoidance System. A wide range of indicators is also available to the user. First group located at the top of the screen, shows algorithm-specific information such as pathfinding time, data structure size and memory footprint as well as number total of A* calls. Second group of indicators located at the bottom of the screen shows the autopilot and flight related information such as speed, heading, artificial horizon, distance travelled and the fuel burned. The best effort was made to produce indicators that would reassemble real-life in-flight instruments and be truthful and informative at the same time. Immersive user experience was created by the use of stereoscopic graphics.

5.2 Further Work

There are still many ways, in which developed autopilot could be improved and extended.

Some optimizations could involve restricting search area to minimize the cost of CPU and memory used:

- The grid data structure could be divided into regions. Each region could be considered an independent search area, with multi-threading used to calculate each chunk of the optimal path.

- The concept of layer could be introduced (Niu & Zhuo, 2008). The layer would consist of cubes, which have the same vertical attribute and are in one horizontal plane. Moving along the layer would be the same as moving in 2D plane, with only x and y directions considered. The real pathfinding process could be employed only if finding the path along that layer would fail.

However, all these solutions, as much as worth exploring have their drawbacks, as restricting entity's movement would always pose the risk of finding suboptimal paths. The challenge here is to find the balance between the efficiency and optimality.

In the game scene many entities may need to find path at the same time. As the autopilot in this project was used to calculate a single path at any given time, future extension could address multiple entity pathfinding. Processing multiple pathfinding request at the same time could be costly to CPU and thus, a queue mechanism could be created, which would handle requests from entities in turns, based on their

priority (Chen, Shi & Liu, 2009). Another interesting option would be to allow communication between entities. If the game world contains frequently used paths, sharing results from one entity's search could be accessible for other entities to avoid repetitive tasks.

Graham and colleagues (2003) argued that rather than exploring new solutions to overcome pathfinding problem, researches tend to over rely on A* algorithm, which often results in restricting the game world, for example, by reducing the number of dynamic obstacles. One solution to this problem could be the use of neural networks or other machine learning techniques to assimilate pathfinding behaviours. However, there is no guarantee that machine learning algorithms will always generalise for situations that did not crop up in the training process. Machine learning will fail, if the right machine learning algorithm is not applied correctly. Additionally, a lot of training data is often needed for learning to be successful.

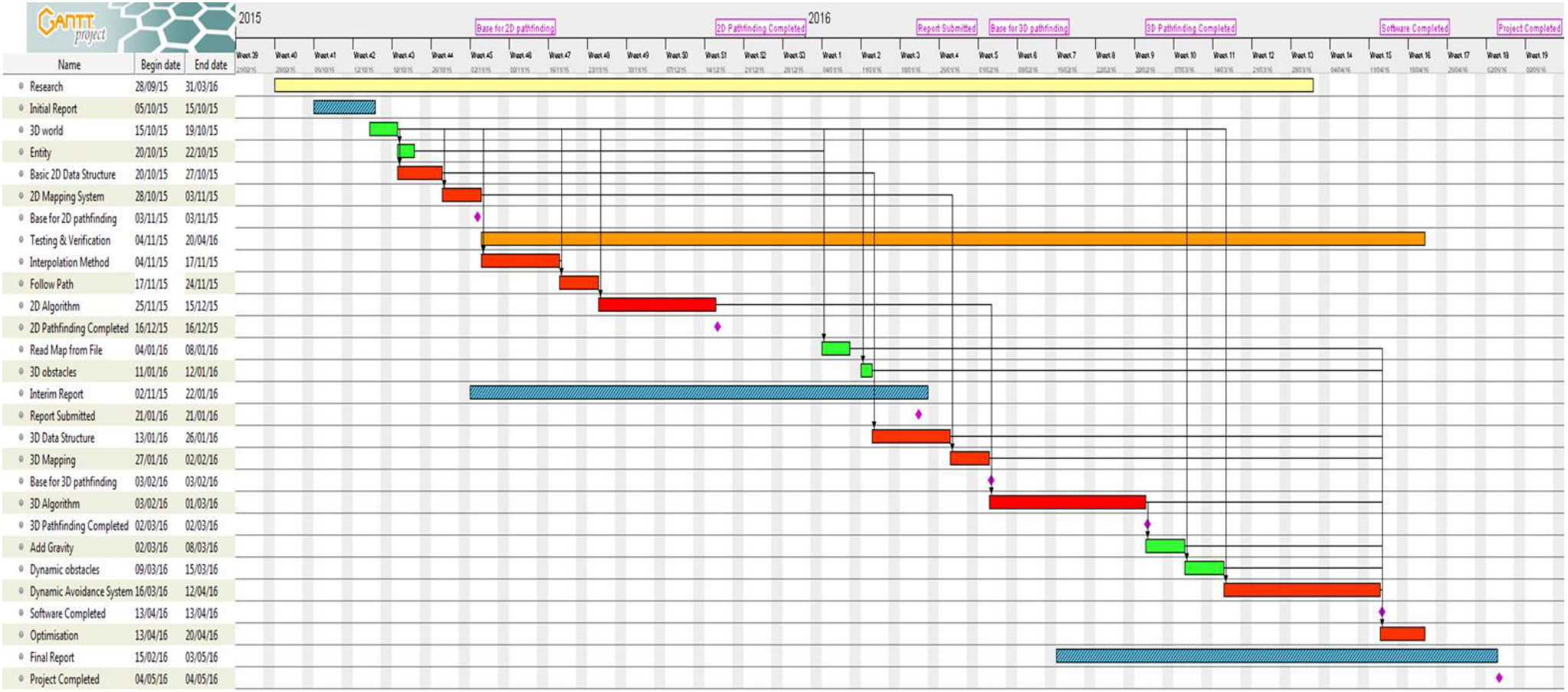
6. Conclusion

This project presented autopilot algorithm able to safely and efficiently navigate through the multilayered, open, 3D world cluttered with static and dynamic obstacles. The autopilot works by selectively assessing a part of the environment that is relevant to its path. When changes in the game world are encountered, it is able to effectively repair its previous solution or in case of surprise encounter employ a number of safety mechanisms to escape the collision. Although opportunities for further refinement have been discussed, the experimental results have shown its flexibility under a variety of circumstances, making it a valuable addition to the pathfinding algorithms.

Appendix A: Initial Task List

#	Task Name	Description	Duration (weeks)
1	Research	Conduct the background research on previous work in the area	26
2	Initial report	Write initial report deliverable	1
3	3D World	Create simple 3D world with static obstacles located on the ground (2D plane)	0.5
4	Entity	Create simple entity which will follow the future path	0.5
5	Basic 2D Data Structure	Implement basic data structure ("Node" class, "Grid" Class and "Path" Class) for pathfinding in 2D plane	1
6	2D Mapping System	Create mapping system to translate between world coordinates and position on the grid	1
7	Testing & Verification	Continuous testing of the created software	25
8	Interpolation Method	Implement 2D prototype of interpolation method to interpolate between nodes	2
9	Follow Path	Implement functionality that will allow the entity to follow the 2D path at given speed	1
10	2D Algorithm	Implement optimal pathfinding algorithm operating in 2D	3
11	Read Map from File	Implement "Read map from file" functionality	1
12	Obstacles in 3D	Add more obstacles at different heights (different planes)	0.5
13	Interim Report	Write interim report deliverable	12
14	3D Data Structures	Modify data structures to account for pathfinding in 3D environment	2
15	3D Mapping	Modify world-grid coordinates mapping system to account for pathfinding in 3D environment (world-cube coordinates)	1
16	3D Algorithm	Modify the pathfinding algorithm to make it operate in three dimensions	4
17	Add Gravity	Modify the pathfinding algorithm to consider gravity - make vertical upwards movement more costly	1
18	Dynamic Obstacles	Add moving obstacles to the environment	1
19	Dynamic Avoidance System	Add Moving Object Avoidance System	4
20	Optimization	Optimization/Extra Features	1
21	Final Report	Write final report deliverable	12

Appendix B: Initial Time Plan



Name	Begin date	End date
Research	28/09/15	31/03/16
Initial Report	05/10/15	15/10/15
3D world	15/10/15	19/10/15
Entity	20/10/15	22/10/15
Basic 2D Data Structure	20/10/15	27/10/15
2D Mapping System	28/10/15	03/11/15
Base for 2D pathfinding	03/11/15	03/11/15
Testing & Verification	04/11/15	20/04/16
Interpolation Method	04/11/15	17/11/15
Follow Path	17/11/15	24/11/15
2D Algorithm	25/11/15	15/12/15
2D Pathfinding Completed	16/12/15	16/12/15
Read Map from File	04/01/16	08/01/16
3D obstacles	11/01/16	12/01/16
Interim Report	02/11/15	22/01/16
Report Submitted	21/01/16	21/01/16
3D Data Structure	13/01/16	26/01/16
3D Mapping	27/01/16	02/02/16
Base for 3D pathfinding	03/02/16	03/02/16
3D Algorithm	03/02/16	01/03/16
3D Pathfinding Completed	02/03/16	02/03/16
Add Gravity	02/03/16	08/03/16
Dynamic obstacles	09/03/16	15/03/16
Dynamic Avoidance System	16/03/16	12/04/16
Software Completed	13/04/16	13/04/16
Optimisation	13/04/16	20/04/16
Final Report	15/02/16	03/05/16
Project Completed	04/05/16	04/05/16

Appendix C: Pseudo code for A* using C# list for its open set.

```
List<Node> FindPathListBased(Node,Node).

--- BEGIN ---

Step1: Check if both start node and target node are passable.
       If no return path not found.
Step2: Create containers for open set and closed set.
Step3: Add start node to open set.
Step4: Loop while open set is not empty..

       Step5: Set currentNode to first element in open list .
       Step6: Loop for each element E in the open set:

              -check if (E's F_Cost < currentNode's F_Cost).

                  IF YES: set currentNode to E

              -check if(E's F_Cost == currentNode's F_Cost)
                         AND (E's H_Cost < currentNode's H_Cost).

                  IF YES: set currentNode to E

       Step7: Remove current node from open set.
       Step8: Add current node to closed set.
       Step9: Check if current node == target node.

                IF YES: return - PATH FOUND

       Step10: Loop for each of the neighbours "N" of current node.

               -check if (N is passable)OR(N is in closed set)

                    IF YES: skip current iteration

               -calculate:
                   newMovementCostToNeighbour =
                     current node G_Cost + distance between(current node,N).

               -check if (newMovementCostToNeighbour < N's G_Cost)
                         OR (neighbour IS NOT in open set)

                    IF YES:
                          Set: N's G_Cost = newMovementCostToNeighbour
                          Set: N's H_Cost = distance between(N, target node)
                          Set: N's ParrentInPath = current node
                          -check if(open set contains N)
                               IF NO: add N to open set
 --END LOOP FROM STEP 4

Step11: Return null - path not found.

--- END ---
```

Appendix D: Pseudo code for A* using binary heap for its open set.

```
List<Node> FindPathHeapBased(Node,Node).
```

--- BEGIN ---

```
Step1: Check if both start node and target node are passable.
        IF NOT return path not found.
Step2: Create heap structure for open set and hash set for closed set.
Step3: Add start node to open set.
Step4: Loop while open set is not empty..

       Step5: Pop current node from open set.
       Step6: Add current node to closed set.
       Step7: Check if current node == target node.

                IF YES: return - PATH FOUND

       Step8: Loop for each of the neighbours "N" of current node.

               -check if (N is passable)OR(N is in closed set)

                    IF YES: skip current iteration

               -calculate:
                   newMovementCostToNeighbour =
                     current node G_Cost + distance between(current node,N).

               -check if (newMovementCostToNeighbour < N's G_Cost)
                         OR (neighbour IS NOT in open set)

                    IF YES:
                          Set: N's G_Cost = newMovementCostToNeighbour
                          Set: N's H_Cost = distance between(N, target node)
                          Set: N's ParrentInPath = current node
                          -check if(open set contains N)
                               IF NOT: add N to open set
                               IF YES: update N's position in open set

 --END LOOP FROM STEP 4

Step9: Return null - path not found.
```

--- END ---

References

Bulitko, V., Björnsson, Y., Sturtevant, N.R., & Lawrence, R. (2011). Real-time heuristic search for pathfinding in video games. *Artificial Intelligence for Computer Games*, pp. 1-30. New York: Springer.

Bourg, D. M., & Seemann, G. (2004). *AI for Game Developers.* Sebastopol: O'Reilly Media.

Chen, S., Shi, G., & Liu, Y. (2009) Fast path searching in real time 3D game. *Intelligent Systems*, 3, pp. 189-194.

Cui, X., & Shi, H. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1), pp. 125-130.

Cui, X., & Shi, H. (2011). Direction oriented pathfinding in video games. *International Journal of Artificial Intelligence & Applications,* 2(4), pp. 1-11.

Graham, R., McCabe, H., & Sheridan, S. (2003). Pathfinding in computer games. *ITB Journal*, 8, pp. 57-81.

Hart, P. E., Nilsson, N. J. & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics,* 4(2), pp. 100-107.

Hernández, C., Baier, J.A. and Asın, R. (2014). Making A* Run Faster than D*-Lite for Path-Planning in Partially Known Terrain. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS)*.

Hernández, C., Sun, X., Koenig, S. and Meseguer, P. (2011). Tree adaptive A*. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pp. 123-130. International Foundation for Autonomous Agents and Multiagent Systems.

Holte, R.C., Perez, M.B., Zimmer, R.M. and MacDonald, A.J. (1996) Hierarchical A*: Searching abstraction hierarchies efficiently. In *The Thirteenth National Conference on Artificial Intelligence,* 1, pp. 530-535.

Hu, J., gen Wan, W. & Yu, X. (2012). A pathfinding algorithm in real-time strategy game based on Unity3D. *Audio, Language and Image Processing (ICALIP), 2012 International Conference*, pp. 1159-1162. IEEE.

Hui, Y. C., Prakash, E. C. & Chaudhari, N. S. (2004). Game AI: artificial intelligence for 3D path finding. *TENCON 2004. 2004 IEEE Region 10 Conference*, pp. 306-309. IEEE.

Iabuk. (2014) *Games Industry in Numbers.*[Online] Available form: http://www.iabuk.net/research/library/gaming-revolution/. [Accessed on 27th of March 2016].

Koenig, S. and Likhachev, M. (2005). Fast replanning for navigation in unknown terrain. *Robotics, IEEE Transactions on*, *21*(3), pp.354-363.

Koenig, S. and Likhachev, M. (2006). Real-time adaptive A*. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pp. 281-288. ACM.

Niu, L. & Zhuo, G. (2008). An improved real 3D A* algorithm for difficult path finding situation. *The Inter. Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 37.

Stentz, A., 1995, August. The Focussed D* Algorithm for Real-Time Replanning. *IJCAI*, 95, pp. 1652-1659.