Mathias Braathen
Eyvind Nikolai Holt

A Study on AI Behaviour & Complex Navigation

Bachelor's project in Computer Engeneering
Supervisor: Ole Christian Eidheim
May 2020

Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Mathias Braathen
Eyvind Nikolai Holt

A Study on AI Behaviour & Complex Navigation

Bachelor’s project in Computer Engeneering
Supervisor: Ole Christian Eidheim
May 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Preface

Riddlebit, a fairly new game development company, is in the making of a game called Setback. It's a first person shooter game, with a goal of reaching a set amount of checkpoints in a map and eliminate other potential enemies on the way. As they want players to be able to practice, the option of being able to play against a computer controlled enemy is desired. We were therefore tasked with finding a solution for an Artificial Intelligence, that can take on the role of an enemy in the game.

Our initial goal was to implement a fully functioning AI for the game, but the game world isn't just flat with a few slopes. There are several obstacles and unique ways to move around, and so figuring out a practical way to represent the world, quickly became the largest task at hand. The focus of the paper was in the early stages originally to present a working AI for Riddlebit, but after discovering the insufficiency of Unreal Engine's navigation system, we decided to change focus to showing what is possible with Unreal Engine and started research into a suitable improvements or alternatives.

Acknowledgments

We would like to thank Riddlebit for continuous cooperation and help throughout the writing of this thesis, especially from Jonathan Jørgensen.

We would also like to thank our supervisor Ole Christian Eidheim for feedback and guidance.

Mathias Braathen E. Nikolai Holt

Mathias Braathen Eyvind Nikolai Holt

Problem Description

Our task is to create a solution for bots for Setback, to provide players with an option to play against a computer controlled enemy for practice. The contribution will be in the form of a plugin for Riddlebit to use in their development of Setback, using the game development engine Unreal Engine 4.

We will in addition to our solution for the bot using the systems currently available in Unreal Engine, explore other possible approaches for AI behavior and navigation. The field has substantial research, but there has yet to be implemented a complete solution. Therefore the the focused changed from the development of a plugin into research on how to represent the environment in a graph.

Abstract

AI behavior and navigation is a relatively new subject in the scope of video games, as their popularity has risen since the late 20th century. There are many different methods used for controlling an AI, and representing the game world for the pathfinding method to give the AI a route to follow. In the thesis, we explore different system for controlling AI behavior and compare them, and test out if the behavior tree that Unreal Engine provides is sufficient. We implement navigation for the AI using what system are already available in Unreal Engine, and compare them to other implementations and potential alternatives. We observe how jumping and other actions related to navigation are difficult to implement and limited with the current Navigation Mesh system. Then discussing some of the improvements that can be done to the connections in the Navigation Mesh or if voxels is a better solution.

Sammendrag

AI oppførsel og navigasjon er et relativt nytt tema innenfor dataspill, siden populariteten deres har steget betydelig siden slutten av 1900-tallet. Det finnes mange forskjellige måter å kontrollere AI på, samt å finne en måte å representere verden i spillet på slik at det er mulig å finne en vei AI'en kan følge. I denne oppgaven skal vi utforske systemer for å kontrollere AI oppførsel og sammenlikne dem, for å teste om Behavior treet Unreal Engine tilbyr er tilstrekkelig. Vi ser på hvordan hopping og andre handlinger knytttet til navigasjon er vanskelig å implementere og begrenset med det eksisterende Navigation Mesh systemet. Vi diskuterer deretter noen av de forbedringene som kan gjøres med kobling av Navigation Mesh eller om voxler er en bedre løsning.

Contents

List of Figures

List of Tables

Listings

Definitions	
Complex Navigation	Navigation that requires more than just walking on the navmesh. An example is jumping.
Unreal Engine 4 (UE4)	A Game development platform and a 3D graphics engine.
Character	A representation of a person in the game world, often also referenced as a pawn.
Behavior tree	A visual representation of a tree that determines a decision or behavior.
Environment	The world that our character moves around in.
Navigation Mesh	A representation of the environment that we can traverse with a pathfinding algorithm. Most often referred to as a navmesh.
Discretization	Making a continuous environment into a graph that can be traversed.
Link	A way to connect two unconnected edges of a navmesh.
Portal	The connection that is between the edges of the polygons that make a navmesh.
Waypoint	A point in the environment that acts as a goal to move towards for an agent. Is also a type of discretization
Checkpoint	The game representation of a waypoint, in Setback the nodes a player need to reach.
Jump pad	A pad that launches a character higher than it can normally jump.
Runtime	The time it takes for a action to complete in milliseconds.
Frame	A picture that is shown on a screen
Heuristic	A way to solve a problem that may not be optimal, here used for a method used to determine the length between two points.
Euclidean distance	The straight line between two points
A.I.	Artificial Intelligence, in this thesis this is a character controlled by a behavior tree and is navigating on a navigation mesh.
Controller	The term in UE4 that says who is controlling a character.
Agent	An agent is what we call a character where the controller is a behavior tree and not a person.
Blueprint	A node-based scripting interface system for creating gameplay elements in the UE4 editor.
AIPerception	A component in UE4 which allows an agent to perceive certain objects or characters around it in certain ways
AISense_Sight	An element that can be added to the AIPerception component, giving it the ability to making an event trigger when something enters within its sight.
AIPerceptionStimuliSource	A component i UE4 which allows an agent to be perceived by other agents.

Action space	A list of the actions an agent may take, we also use it as the umbrella term for the technique that decides which action should be taken.
Line Trace	A way to determine if there is an object between two points. A line is drawn between the two points and determines if it hits anything.
Sphere Trace	The same as a line trace, but with a sphere instead of a single line. The sphere has a radius.

1 Introduction

From the old classic games like Pong and Super Mario Bros, to the more modern approaches to video games like Skyrim and the Witcher, the AI in the game is a central part of all of them. Whether its to get a ping pong ball past the opponents side, or lure the player into a dangerous area where they can be killed, AI can be what sets these games apart. Nobody wants to get their asses kicked every time they press "play", but its not like they want to easily take the win every time either. Providing a challenging, but fair environment to the player is not always an easy task. Therefore, the demand for a system which provides a game with a suitable AI is high.

In this paper we will be taking a look at the two essential parts of what makes an AI what it is. The first one is behavior; what makes the AI do what, and when? Since the era of video games started, there have been invented different ways of designing AI behavior, and we will explore some of them. The next is navigation; how does the AI maneuver through the world? Moving from point A to point B in itself is no big hurdle, but we want the enemies in the game to seem intelligent and act like they know where they are going.

Development and testing will be done through the Unreal Engine 4 (UE4) game development platform. The environment we will be using is the game Setback[1]. Setback gives us some of the challenges through their complex level design, movement system and multiple weapons. By complex level design and movement, we refer to the fact that there is more to it than just walking on a flat floor. Obstacles and alternate ways of traversing the map, like walking on jump pads or swinging across gaps with your grappling hook, are very much a big part of moving around in the game.

1.1 Research questions

Winning in a game does directly correspond to fun for the player, and neither does losing - overcoming a challenge does. How the AI in a game behaves and reacts to the player in a fitting manner is a central piece in the puzzle to give that player an enjoyable experience. While defining our research question for the behavior part of this paper, we need to make sure we have the right focus.

- What systems are available for implementing behavior for an AI, and which one is most fitting for a game like Setback?

There are several methods aimed at AI behavior implementation, and so we need to seek out the most relevant ones and compare them. In addition, we also need to figure out which one will be the best option for Setback, which will be the environment used for this thesis.

An agent needs to be able to find the best path to its current goal. Given the complex environment that Setback has there are some limitations to the representation that the current solution provides. Our second research question then becomes:

- How well can we navigate in the current representation of the environment that Unreal Engine gives us?

By this we mean that we are mainly focused on how well the graph represents the environment for the pathfinding. Are there parts of the environment that are not represented, and how do we then represented these? And with these representations, we look at how long the agent uses around a route.

[1]Link to Setback page: https://setback.riddlebit.net

1.2 Thesis Outline

This paper is constructed of 7 chapters, and some appendices at the end. The first chapters, 1 and 2, are theory and introductions, and will acquaint the reader with necessary knowledge to properly understand the following chapters. Chapter 3 introduces to the related work on the subject. Many others have explored this subject, and the ones most related to our focus are mentioned here, with their methods and results. Following is chapter 4, where we show our implementation and methods used to achieve results. This includes how we implemented AI behavior, navigation, and how testing was executed. After methods we show results with implementation of the current system for navigation in the engine. The discussion is chapter 6, where we discuss the result, and evaluate our methods to others. Lastly in chapter 7 we have our conclusion, and discuss potential future work.

2 Background

There are many parts that needs to work together to get the best experience for the player from an AI. In shooting games one can simply make an AI unbeatable by always hitting the player perfectly the moment the player is visible to the AI, but it is not a good experience for the player if the game feels unfair. F.E.A.R.[24, 27] and Halo[11, 13] are two games that stand out for many when they think about good AI and a good experience when you play.

F.E.A.R. had the problem that the AI was too good and thinking too tactical, they solved much of the problem with this by making the AI shout what they were doing so the player felt that they could read the movement of the AI, and outmaneuver the AI agents.

Halo also had problems with the AI, but almost at the other scale. The players felt that the AI too easy. Through playtesting they found out that the player was killing the enemies too quickly. By simply making the AIs tougher, the amount of players that thought the AI was "Very Intelligent" increased from 8% to 43%.

2.1 Action space

For an AI to be able to do something in a game, it first needs a set of actions it is allowed to perform, also known as an action space. The action space will consist of all actions available to the AI, which will be used through methods such as GOAP or Behaviour Tree to determine its behaviour.

2.1.1 GOAP

Developed for F.E.A.R. in 2004 by Jeff Orkin [26, 27], and later used for other large game franchises as Tomb Raider and Middle-earth: Shadow of Mordor[12], Goal-Oriented Action Planning focuses on setting up goals for the AI in the game, with desired world states to aim for, using the actions available from the assigned action space. For example the goal could be to kill the enemy, therefore the desired world state would be "attacking target X", where X is the player. The actions used to complete this goal could be a melee or a ranged attack, which then again are dependent on certain world states. Both actions will satisfy the world state "attacking target X", with the melee attack action requiring other world states like "at target X" indicating that it is or is not at the location of the target and "equipped melee" checking if it has a viable weapon for melee combat, making a melee attack possible.

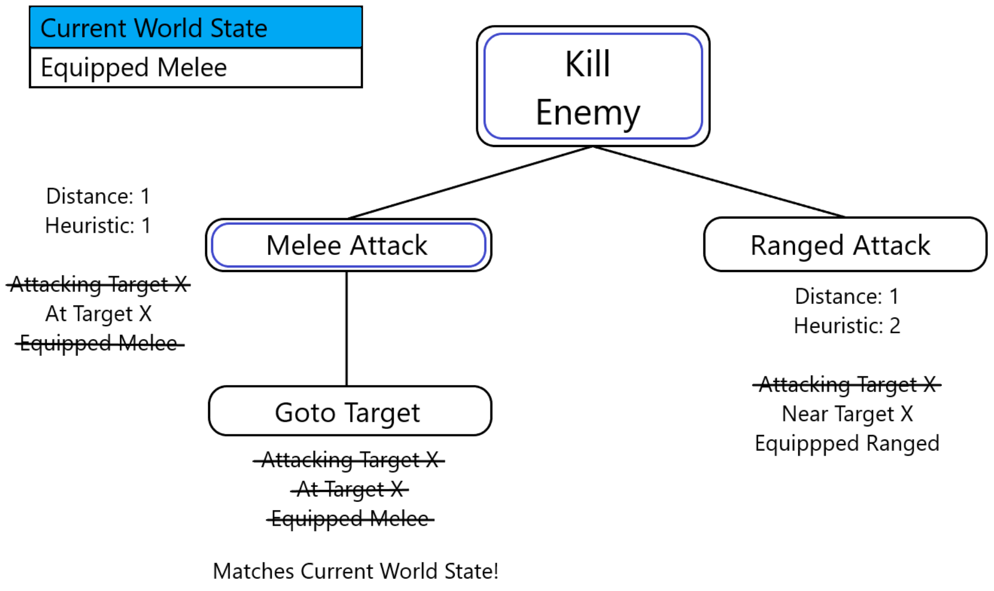


Figure 1: Choosing actions in GOAP[12]

From Figure 1 we can see an example of how different actions are chosen, based on world states, through the use of A-star. Based on the example situation, we see that not much is going on, except for the fact that the AI has a suitable melee weapon equipped. Running through a list of goals, the AI realizes it has an opportunity to kill an enemy, and starts to explore the opportunities. To be able to accomplish this the A-star algorithm can explore the two nodes "Melee Attack" and "Ranged Attack", both having a cost of one for keeping it simple, and both having required world states. Seeing that the "Melee Attack" node only need one more required world state, the A-star will prioritize exploring deeper that way first. Assuming we have another action such as "Goto Target" available in the action space, the A-star finds that node as a neighbour to "Melee Attack". Noticing that the "Goto Target" action will make the "At Target X" world state true, giving the AI a plan; using the action "Goto Target" and "Melee Attack" resulting in killing the enemy, assuming only one attack is necessary.

Applying this planning method for AIs in games, especially large ones as Tomb Raider and Middle-earth: Shadow of Mordor, can be quite performance heavy as both world states and plan lengths needs to constantly be calculated and changed.

2.1.2 Behavior Tree

Behaviour trees are excessively used in the game industry for determining behaviour for an AI. Games mentioned above like F.E.A.R and Halo are excellent examples of games utilizing this tool. The use of the behaviour trees are quite popular, due to the fact that they are easy to set up, debug and utilize, as well as giving the developer an easy way to view the AI's behaviour. The only thing required to use a behaviour tree is to first create an action space for the AI.

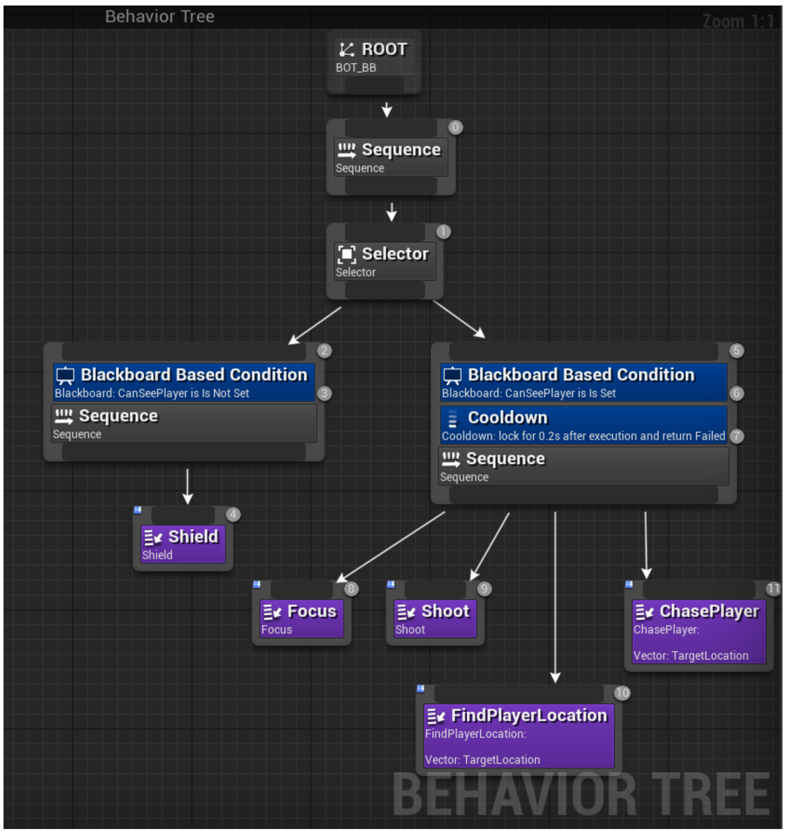


Figure 2: Behavior tree example from Unreal Engine

In Figure 2 we see an example of a behavior tree from UE4. The sequence is set to react to two conditions; either the "CanSeePlayer" variable is true or false. If it's false, the AI will hold up its shield. If it's true, the AI will focus its camera on the player, start shooting at it and follow the player as long as its within sight, following the actions from the set action space in order though the sequence.

2.2 Pathfinding

Without pathfinding an agent would be just moving in random directions and would have no information about the world that it is occupying. There are many different pathfinding techniques[1] and which pathfinding method that is used is often dependent on the problem that is in need of solving.

2.2.1 Dijkstra

Dijkstra's algorithm is an algorithm for finding the shortest path between nodes in a graph[34], used often to solve problems relating to network routing, navigation through networks of roads or in games. Using Figure 3 as an example, we see Dijkstra's algorithm in effect[44]. In this case the cost of every node is equal, and therefore all neighboring nodes will be explored in all directions until the goal is located, causing the "flooding" effect of Dijkstra's algorithm one can see from the figure. Observing Figure 3 it's worth noting that a large amount of the nodes explored are not in the right direction of the goal, and so variations to the algorithm has been made to solve this problem of having explored unnecessary nodes.

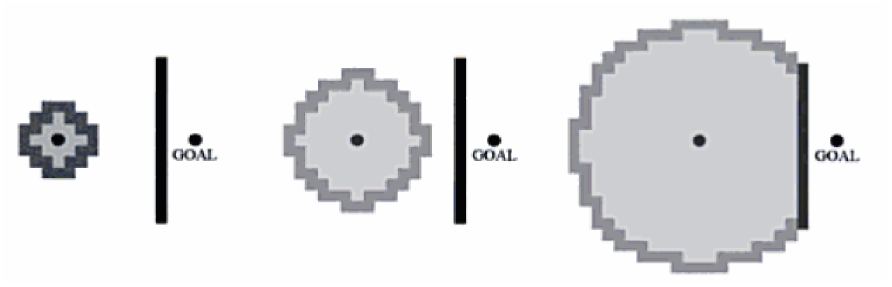


Figure 3: Dijkstra's algorithm (heuristic cost of zero)

2.2.2 A-star

A-star is a variant of Dijkstra's algorithm which uses the same logic[33], except it also takes into account a heuristic, often euclidean distance to the goal, to force the algorithm to prioritize nodes closer to the target. Observing Figure 4 we see that the algorithm immediately starts focusing on the nodes with a shorter euclidean distance to the goal. A-star substantially reduces the search area, resulting in finding the optimal path faster. Since the A-star algorithm takes an extra variable into account, its major drawback is the time used to explore each node is a little higher. However, due to the fact that the A-star algorithm is more efficient than Dijkstra's in terms of what nodes it explores by only exploring the promising ones, it's the one preferred to use in most cases[44].

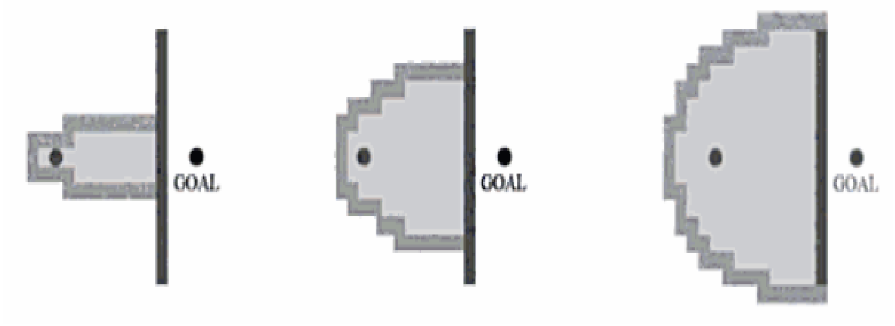


Figure 4: A* algorithm (heuristic cost using Euclidean distance)

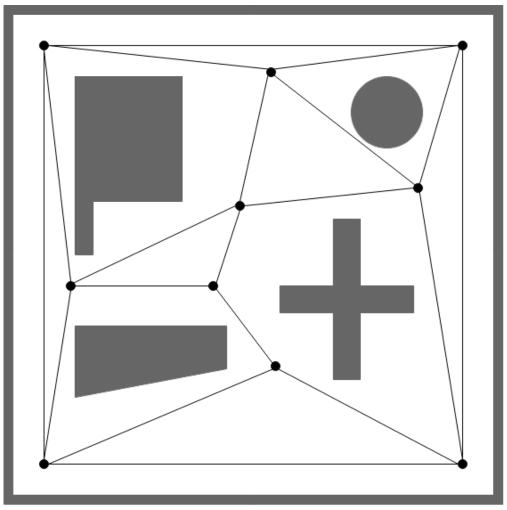
2.3 Discretization

This is one of the first problems of pathfinding[23], and it is how to represent the world in a way that keeps almost all the granularity of the world but represents it in the most lean way. The less data you keep of the world, the faster the resulting graph would be to traverse, but the pathfinding algorithm may find a sub-optimal path, as the graph is lacking about data about the world. On the other hand if the graph contains too much data about the world, the pathfinding algorithm may find the optimal path, but at the cost of having too high of a runtime.

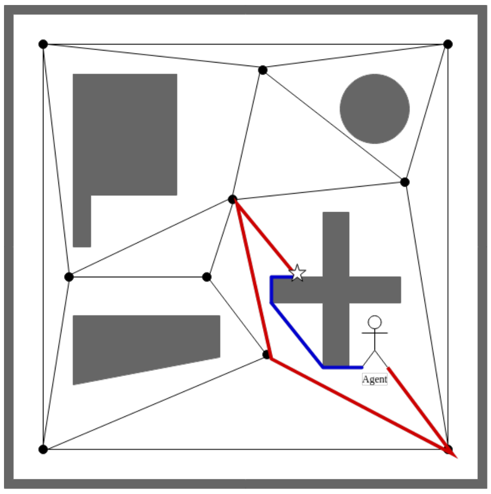
2.3.1 Waypoint graph

Waypoint graphs [3, 47] are created by placing graph nodes in the world. Each node in the graph should have no obstacle between it and at least one other node. One can choose which nodes are connected to each node to simplify the graph as much as possible. One of the largest benefits of using waypoints is that it is simple to select which nodes that give an advantage[16, 43] to the agent.

There are two problems that arise with these kinds of graphs. They require a lot of time to make, as each node and what node is its neighbors needs to be manual added, and they limit the paths that the agent may take. Figure 5 gives an example of the path the agent can take on a simple waypoint graph. One can increase the number nodes in the graph until you get a finely grained graph, but his will take even more time to make and will significantly slow down the search[23].



(a) A coarsly grained waypoint graph



(b) Waypoint path, red. True path, blue.

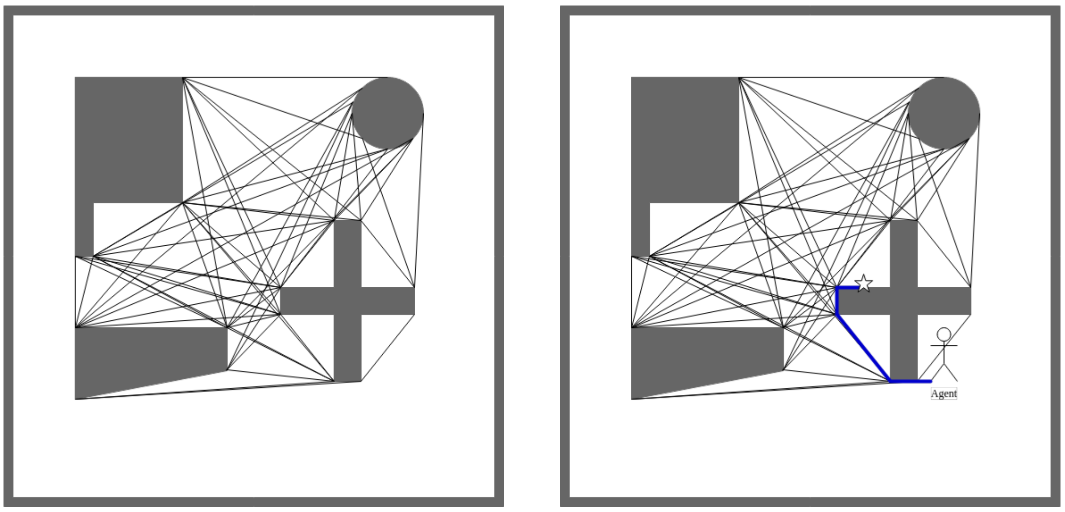
Figure 5: Waypoints graph

2.3.2 Visibility graphs

Visibility graphs[28, 9, 10] are graphs where each node in the graph is connected with all the other nodes that are visible to that node. More precisely this means that each node that can be reached from our node in a straight line that is not obstructed by any obstacle is visible to our node. An example of a visibility graph is in figure 6

When drawing the edge between our node and all the nodes visible to it, we compute the Euclidean distance between our nodes and the nodes visible. This lets us find the shortest Euclidean path through the graph.

The problem that arises with computing the edges and their weights is that the runtime of the algorithm suffers greatly[7, 17, 19].



(a) Visibility graph (b) Visibility and true path, blue

Figure 6: Visibility graph

2.3.3 Navigation mesh

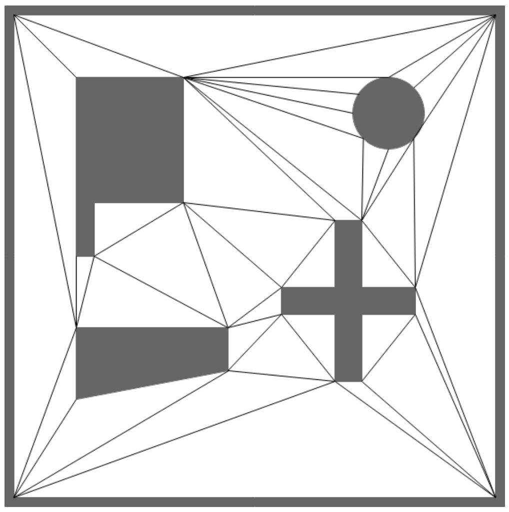
A navigation mesh[32, 5, 14], or navmesh, is a discretization of a 2D or 3D world, transformed to a 2D graph that consists entirely of convex polygons. By creating a graph made entirely of convex polygons, we are sure that we can draw a line across the polygon at any angle and not be blocked by any obstacle. Figure 7 shows an example of a navmesh in a 2D world, while Figure 8 and 9 shows a navmesh in Unreal Engine.

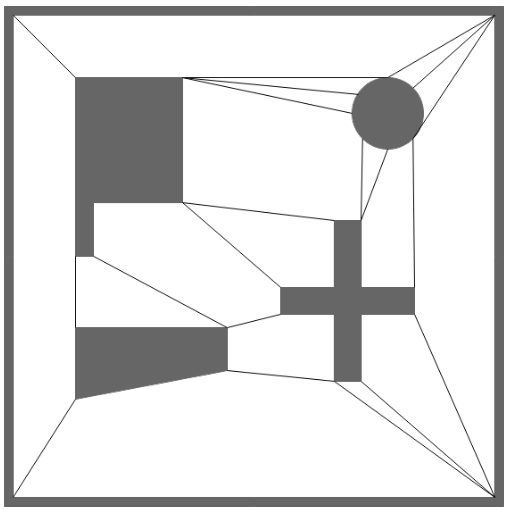
There are multiple ways to compute a navmesh[41]. One of the more widely used is Recast[22] and a new generator for near optimal generation of navigation meshes is NEOGEN[25]. With one of the main differnces between them being that NEOGEN manages to create navmeshes with fewer polygones, because when generating the navmesh one can relax the definition of what is a convex polygon.

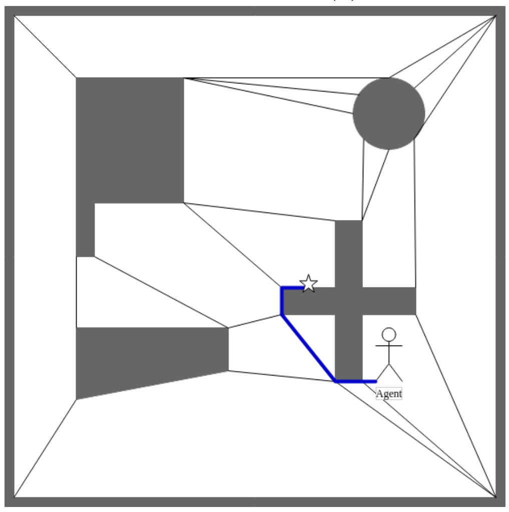
When generating a navmesh it is also important to take into account Agent height, so that the agent doesn't try to walk where the roof is too low, Agent step height, to determine how high of an object the agent can step on, and the walkable slope to differentiate between too steep and walkable surfaces. Only Recast also need to take into consideration Agent radius. This is because Recast shrinks the navmesh equal to agent radius, to hinder the agent from bouncing into walls.

A problem with navmeshes is what points are going to be used to generate the path. One can use the middle of the edges as the nodes, the corners, the center of the polygon, or all at the same time. If one uses all of these points one would produce a lot of additional nodes that the pathfinding would need to search through in addition to not producing the optimal path. Therefore often the center of the polygon is used, so that the graph is as lean as possible. Then after a path is found it is smoothed with SSF[21] or another string pulling algorithm.

The next problem for navmesh is that it is only a 2D structure describing 3D space, therefore it lacks the connection between areas that are disconnected by gaps in the ground or edges that are steeper than the given slope.


(a) Navmesh of triangles, 38 triangles


(b) Navmesh of polygons, 17 polygons



(c) Navmesh and true path, blue

Figure 7: Navmesh

Nav Link Proxy [8, 42] Figure 10, is the simplest solution to this problem of connecting these unconnected areas. Nav Link proxies are links that you can place by selecting the start and end point of the link, if it is going to be traversable both ways and if it shall trigger any special actions when the character enters one of the defined points. The Nav Link Proxy is treated by the pathfinding as a part of the navmesh, and the only difference is that the pathfinder has an additional node to check.

The Nav Link Proxy gives the option to limit the movement of the agent, by only adding the option to jump up or drop down where one might feel that it is correct. On the other hand this also has the drawback that it takes a lot of time and testing to get these links to seem natural and not too limited. As the agent in addition to only being able to follow the line that the Nav Link Proxy is created with. The agent will stop when it is over the end point of the link, instead

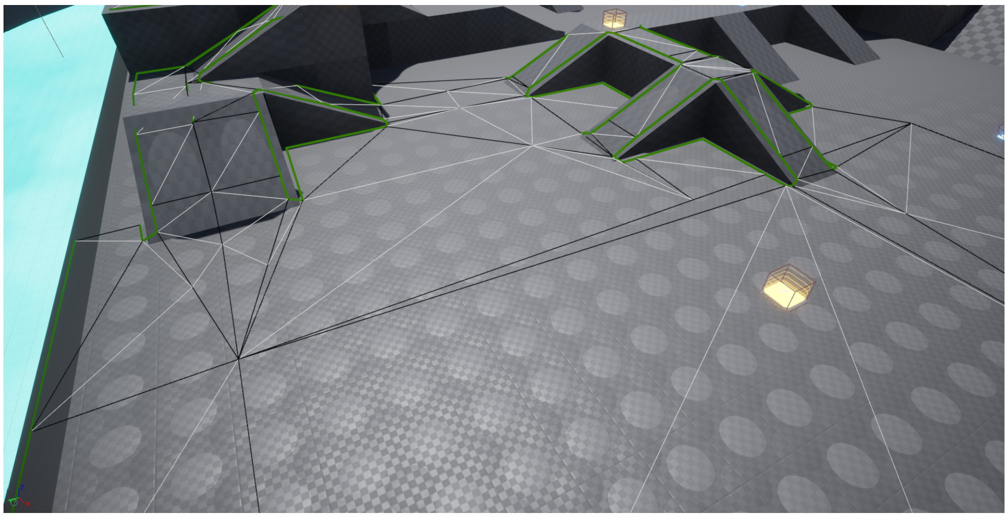


Figure 8: Navmesh in Unreal Engine 4. Green lines markes the edge of the navmesh, black lines mark the edge of the polygons and white lines markes the edge of the triangles the polygon is built from

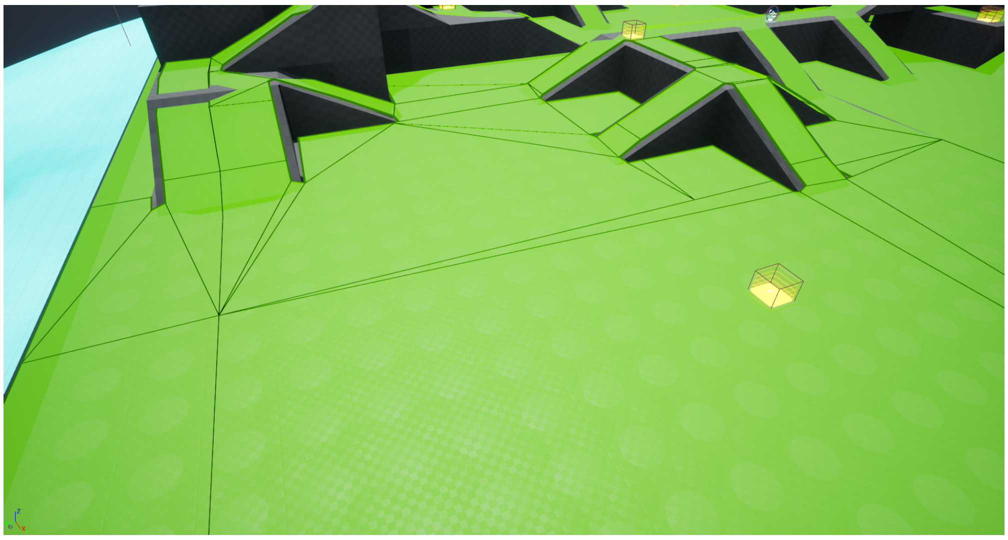


Figure 9: Navmesh in Unreal Engine 4 with filled polygons and polygon edges set to visible

of trying to move as far as possible in the air.

In Unity[39] these simple links are called Off-Mesh links, Figure 11 is an example. Unity also has the ability to automatically generate some types of Off-Mesh Links. These are jump across links and drop down links and can be turned on and off for each object in the environment. Unity also has what they call NavMesh Links, see Figure 12, these are somewhat more advanced way of connecting two navmeshes and creates an area that connect two navmeshes. You can determine the width of these areas and the agent may traverse the NavMesh Link diagonally. This is in contrast to the Nav Link Proxies and Off-Mesh Links, which only lets the agents traverse the

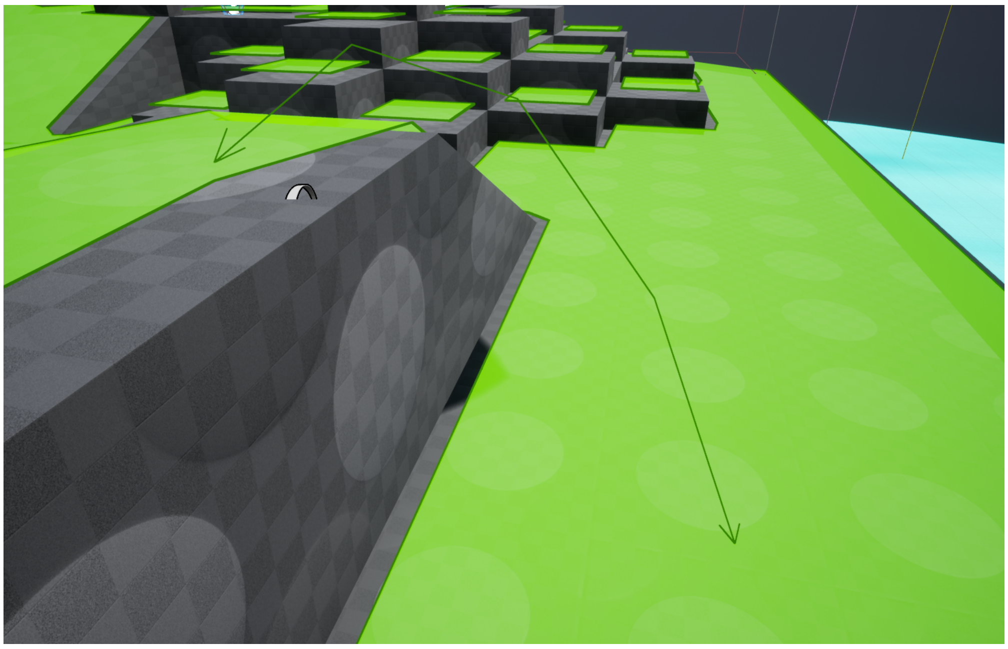


Figure 10: The Nav Link Proxy used in Unreal Engine 4

link in a single line.

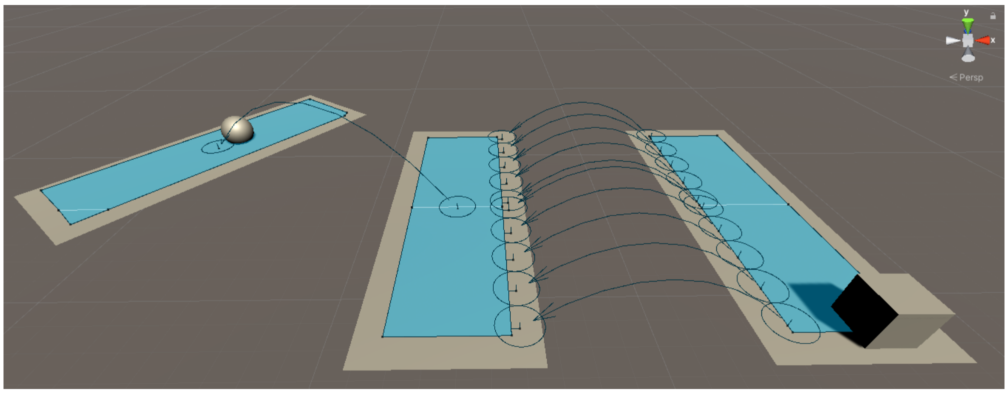


Figure 11: The Off-Mesh Link used in Unity. The multiple links to the right is the links generated by Unity, while the single link on the left is the manually added link

How the agent reacts to links in UE4 and Unity is a bit different. In UE4 the link is simply treated as a link and without any further action does nothing more than specify that the agent may walk between these formerly unconnected areas, but one can add events to the end points of the link, to make the agent jump or take other actions[2]. Unity on the other hand treats the link as a portal and simply transports the agent between the two nodes and if you want the agent to look like it is jumping you have to create a special animation for that. This applies to both the

[2]Nav Link Proxy video: https://youtu.be/iPPwf3aWFu4

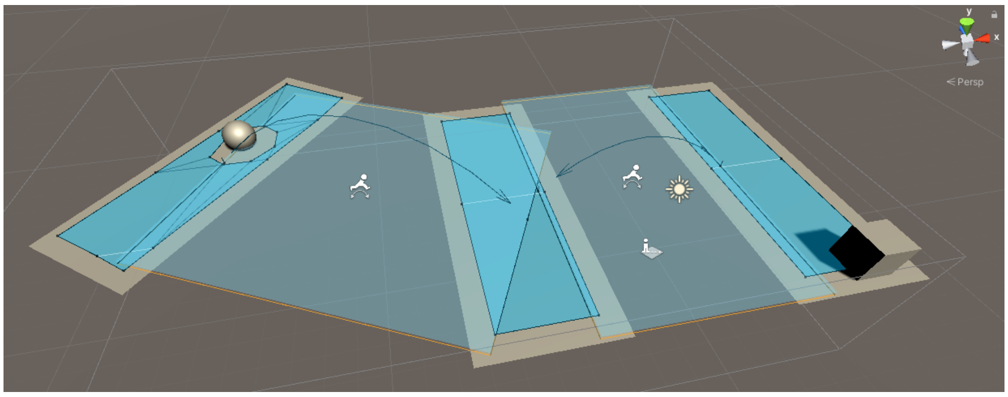
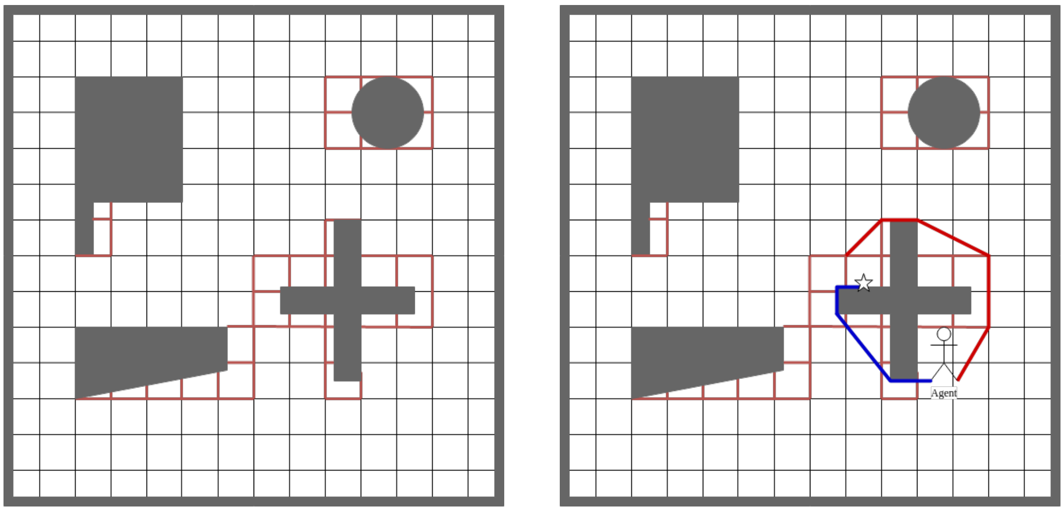


Figure 12: The NavMesh Link used in Unity

Off-Mesh Link[3] and the NavMesh Links[4].

2.3.4 Regular grid

A regular grid or a tile grid is used in many video games[46]. There are many ways to represent a regular grid, some of the more popular are tile, see Figure 13, hexagonal and octile. A tile grid is the normal checkerboard representation with squares laid all over the map. Hexagonal is the same, but you use hexagons instead of a square tile as it gives six degrees of movement instead of four. Octile is represented in the same way as tiles, but all the eight squares that are directly and diagonally next to a node is its neighbours.



(a) Regular grid

(b) Grid path, red. True path, blue.

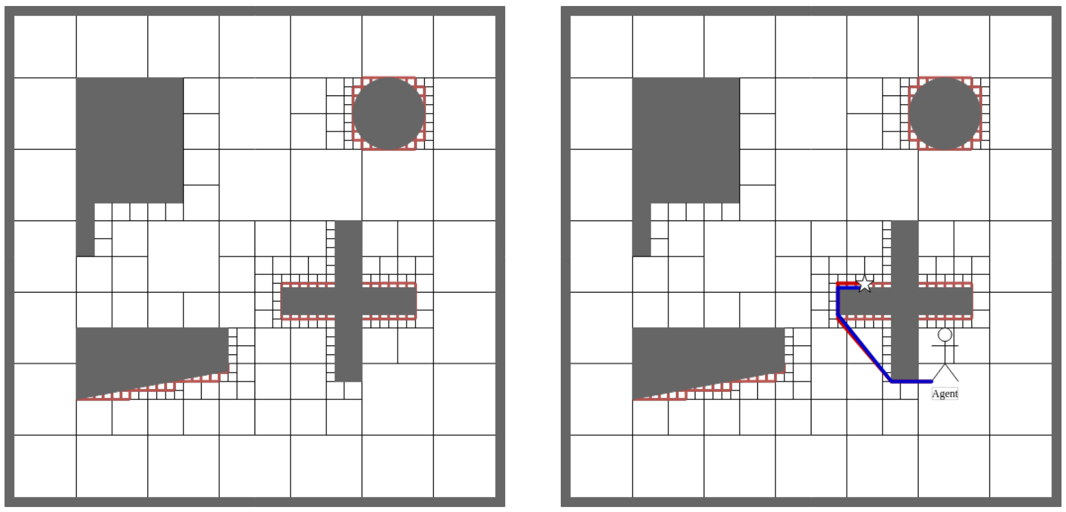
Figure 13: Regular grid, the blocked squares marked in red. Containing 183 tiles

There is a fourth method called a tex grid[45], which can be called a tile grid with the advantages

[3]Off-Mesh Link video: https://youtu.be/-aPSwrHybVk
[4]NavMesh Link video: https://youtu.be/Fyi7LjUJjYw

of a hex grid. To create a tex grid you can shift every second row a half a tile upwards so that each node has six neighbors.

An improvement to tile grids is quadtrees[30, 31], as seen in Figure 14. A quadtree can give additional granularity in a map without increasing the size of the graph quadratically and often it can even reduce the size of the graph. This is because by using a quad tree structure you can split every tile into four subtiles and each of those tile into four tiles and so on, to a specified max number of child nodes. You can therefore use a bigger standard tile size and still get the same, or higher, granularity of the map.



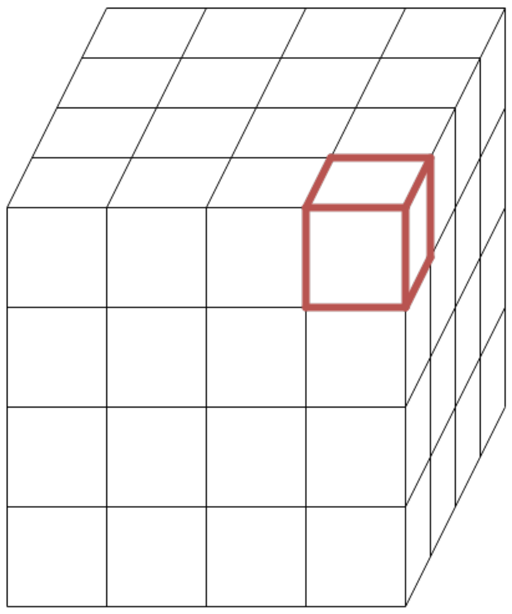
(a) Quadtree grid (b) Quadtree path, red. True path, blue.

Figure 14: Quadtree grid with 4 levels, the blocked squares marked in red. Containing 228 tiles, if we had only used 3 levels it would have contained 145 tiles

2.3.5 Voxel grid

A voxel grid is a 3D version of the 2D tile grid. Instead of being built by tiles, it is built by cubes. See Figure 15a for an example. An important difference between a tile grid and a voxel grid is that as you increase the size of the grid it increases by n^3 instead of n^2, but you map the whole 3d space instead of a representation of the walkable surfaces of the 3d space as a 2d graph.

There is a optimization of voxels grids, equals to the quadtrees of tile grids, these are called octrees. As each cube is divided into eight subcubes, as seen in Figure 15b, instead of the four that a tile is divided into. A memory optimized form of an octree is a sparse voxel octree(SVO)[2]. This lets us represent large open spaces with as little memory as possible in addition to reducing the search time. As the number of nodes we need to search can be significantly reduced.


(a) Voxel grid, single voxel in red

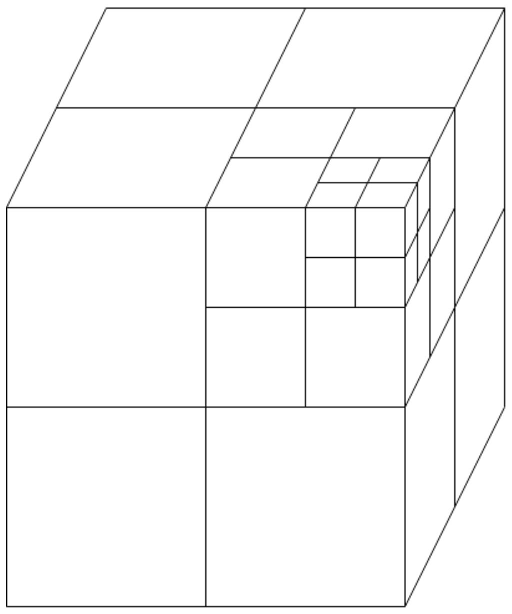

(b) Voxel octree, 3 levels

Figure 15: A voxel grid containing 64 cubes and a voxel octree containing 22 cubes

Leenknegt[15] proposed a solution for using a 3D navmesh instead of a voxel grid and shows that this can significantly reduce the search time of A*, but at the cost of more memory space, compared to a voxel grid.

3 Related work

What an agent is supposed to do is just as important as how an agent does it. An agent that randomly or in inopportune moments does an action not fitting to the situation that plays out in the game is just as jarring as an agent that can't complete an action that is expected of it.

An agent doing the correct actions correctly is an important part of an agent, the second part is how an agent moves. As modern game design have advanced, so has the demand for a functioning and efficient way to represent the environment. Through the last couple of decades there have been developed numerous ways of handling graph generation and algorithms for 2D/3D game environments [1], ergo the selection of tools at disposal is vast. The pathfinding for the agent is efficient enough that the agent does not need to stop randomly, and so the problem now is that the discretization of the environment contains too little information.

It should be noted that while this area has extensive research as video games' popularity have skyrocketed the last decades, papers on the subject rarely expose their method if they solved a problem related to game development. Potential implementation will require substantially more work, since the source code is rarely published. Most of these implementations also tightly integrates animation and navigation to give the best product[4].

3.1 AI behaviour

One of the cornerstone methods for determining AI behavior in video games today are Behavior trees, with games as Alien: Isolation[36] and Halo[35] being prime examples. With its streamlined logic and designer friendliness, its hard to pass up when deciding what sort of method you want to use to control the AI. Enemies in mentioned games are not required to come up with some intricate plan to do their job; in most cases, all they need to do is being able to react when the player does something and engage them in combat. The amount of enemies also sets a limit to what sort of method that can be applied, as planners implementing GOAP can be quite performance heavy depending on how many agents that are supposed to use it.

In 1998 Valve released their first game in the popular video game franchise "Half-Life"[38], using a Finite State Machine(FSM) to control the AI's behavior, which was the inspiration for Jeff Orkin's implementation of Goal-Oriented Action Planning(GOAP) for F.E.A.R. later in 2005[27]. The main difference between the two systems being that FSMs number of active states are not limited to one. Actions, or tasks as defined in the public SDK for Half-Life[40], may lead to several states, which can quickly affect performance considering there are over 80 different tasks for each enemy, with each task resulting in one or more of the many states possible within the game.

F.E.A.R.'s GOAP implementation on the other hand only have 3 states; Goto, Animate and Use Smart Object. Goto representing actions such as walking, climbing swimming, or any action really which is meant for travelling somewhere. Animate is for playing animations such as when taking damage or noticing a player, and Use Smart Object being when the AI interacts with objects like knocking a table over or throwing something. When executed in sequence, these 3 states will enable the AI so look intelligent to the player.

3.2 Runtime jump calculation

Dynamically planning the jumps for an agent may lead to a too high runtime[18], but gives the agent a good possibility to navigate a dynamic environment and using the dynamic objects in the environment to achieve the shortest path.

An UE4 plugin creator called Rama showcased[29] in 2015 a way to connect disconnected navmesh areas in runtime. He only published code for how to get all the navpolygons generated and did not publish anything about the performance cost of this method of getting jump links. Because of the lack of data it is an interesting solution, but it needs further investigation and verification.

3.3 Generating Jump Links

The method that seems most plausible at the moment is to make all the jump links in the navmesh when building the level, but at the same time we want to avoid the drawbacks of having to place navlinks. Sarah Budde[4] wrote a master thesis about jump link generation. One of the important aspects of Budde's thesis is that she talk about jump links being a single link that cover an area, in addition to having a jump trajectory lookup table, see figure 16. This means that each link can contain as little data as possible, reducing memory space and search time. The jump trajectory lookup table is then accessed during the path smoothing to be sure that the smoothed path is withing the jump limitations.

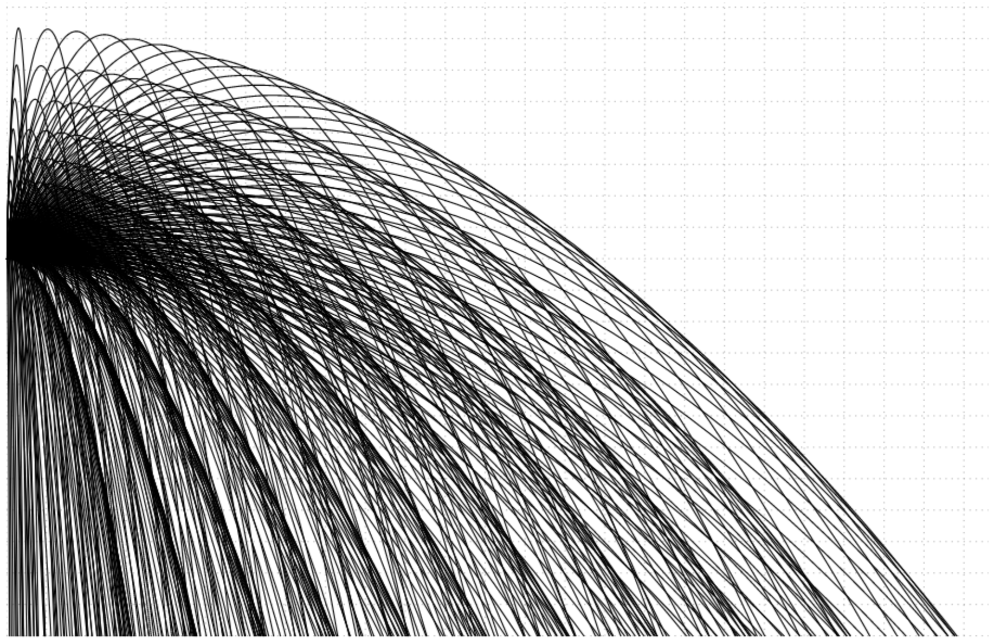


Figure 16: Jump trajectories drawn over a grid as visualization of the lookup table, Budde 2013[4]

Budde's generation process uses multiple steps. First it checks for locations that can be jumped down into, when these links have been found it checks if any of the links are low enough to be jumped up. The links that can be jumped up will be marked as bidirectional links, while the rest is just single direction downward links. The third step is for edge to edge jump links. Here all edges that are not facing each other is discarded, as they have either been found when searching for locations that can be jumped down into, or they will not be reachable from the edge.

The thesis shows that the generation of the jump link are good, but with higher density of obstacles there may be a redundancy of jump links generated. This redundancy leads to that in a lot of environments the amount of edges have doubled. Most of the redundant edges comes from the edge to edge generation of jump links. Her conclusion is that if one could set a way to classify all the jump links and then discard jump links using this classification could reduce the number drastically. The runtime for the link generation scales quadratically with the amount of edges in the environment and leads to the runtime in most of the test environments to average around 20s for jump link generation.

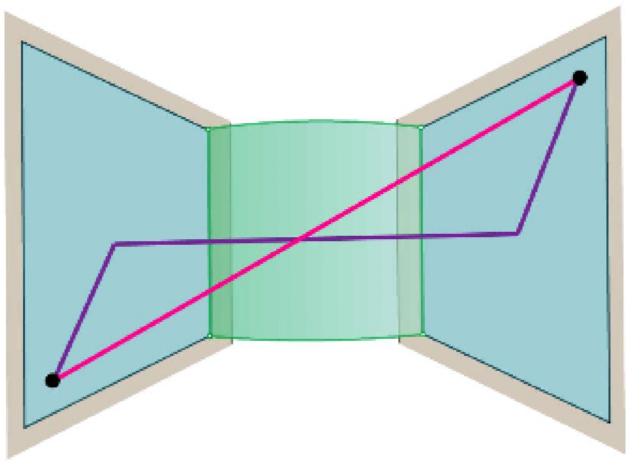


Figure 17: An unsmoothed path (purple) and a smoothed path (pink), both traveling through a jump link (green area). Budde 2013[4]

She show in the thesis not only that it is possible to generate jumps of arbitrary length, shown in Figure 16, but also lets the agent select the shortest path over the jump link, Figure 17. This is in stark contrast to Nav Link Proxies, Off-Mesh Links and NavMesh Links as the way to traverse these are created when building(UE4) or baking(Unity) the navmesh. When looking at a character performing the jumps, this greatly increases the fluidity of the movement.

Budde concluded that with her solution, she could generate complex jump possibilities in arbitrary and complex 3D environments. Her algorithm was proven to be fast enough as well, so performance was not an issue. If jumping was the only phenomenon present in Setback, this would have been the solution, but alas there are other complex ways to move in the air required by the game. A solution for taking into account events and movement while airborne would be desirable.

3.4 Different sized agents in a navmesh

The navmesh made with Recast is limited considering the agent size, as the mesh is generated with the consideration of the agent size. If you then have multiple agent sizes you need multiple navmeshes as the larger agents would crash into the walls if they are using the navmesh of smaller agents, Figure 18 is an example of this. This is because on the creation of the navmesh with Recast the navmesh shrinks the navigable area considering the radius of the agent.

NEOGEN does not have this limitation as the radius of the the agents is taken into consideration at runtime. Instead of shrinking the navmesh with half the size of the agent, NEOGEN creates portals at each edge of a polygon, the size of each portal is defined at runtime and is determined based on the radius of the agent, see Figure 19.

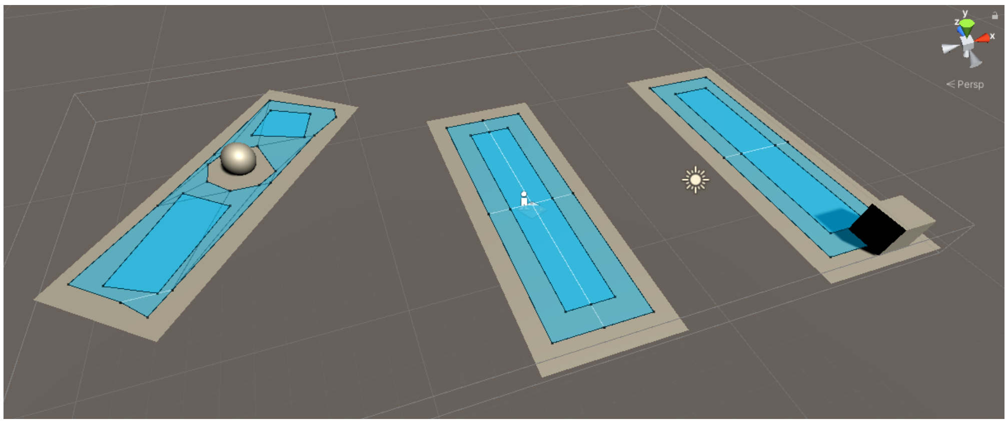


Figure 18: Multiple navmeshes to let agents of multiple sizes move around

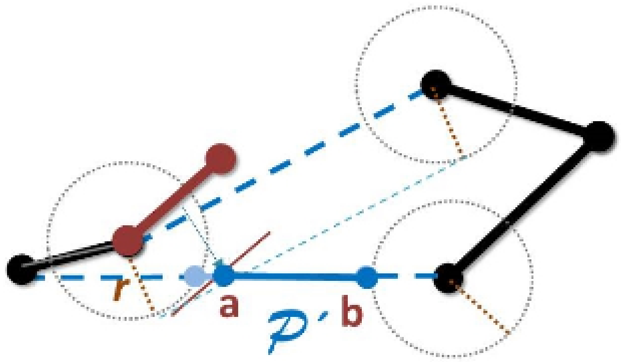


Figure 19: Example of a portal in NEOGEN[25]. The portal P is restricted by the agent radius r and is defined as the line between point a and b

3.5 Voxel Grid

In 2019 Gabriel Silva published paper on jumping mechanics for AI in UE4 [6]. In his paper he also delves into the theory of pathfinding techniques, specifically related to games where jumping is essential. He compares how Navigation links can be used to connect two point in the NavMesh in UE4 to enable agents in the environment to traverse between the points, with a voxel grid to be able to account for where the agent moves in the air. Setting up Navigation links in the NavMesh not only means you have to disconnect the agent form the NavMesh during, for example, a jump, but it also leads to more work for developers as the links has to be places manually throughout the environment. Furthermore, this implies that the agent will only have a set amount of spots to jump to or from, which may reduce players immersion in the game. Budde's solution to this would solve the problem of having to manually set up the links, but still not take into account events where the character is not in direct contact with the surface.

While testing his voxel-grid implementation, he concluded that not only does it make for an efficient pathfinding alternative, but it is achievable in runtime, which means it will not affect performance. Implementing a good pathfinding method accounts for little to nothing if the game needs to take breaks to calculate paths for agents. And considering the fact that we know runtime will not be a problem when using voxel-grids.

Daniel Brewer published a paper on how to solve aerial pathfinding with Sparse Voxel Octrees (SVO) for Warframe [2], which turns out to be an excellent solution for cases where most or all of the movement is executed in the air. With setback being a game with options like being able to grapple to certain points and swing yourself in the air, as well as a lot of jumping and aerial mechanics, this too is a game where a large part of the gameplay takes place in the air. Brewer's solution does not account for gravity, which is something absolutely present in setback.

One possible solution would be to try and integrate Silva's solution with a SVO to speed up the runtime of the pathfinding further. This would make it applicable to even larger environments and one could use the same representation of the environment for walking and flying creatures.

4 Method

4.1 Action Space

For controlling the AI's behaviour in Setback we used Behavior trees, due to the fact that it is the superior choice for a game like Setback. Behavior tree's will give the AI a more reactive play style, which is the most practical solution in cases where no intricate plans are needed. Implementing a GOAP technique for the AI is more time consuming compared to setting up a Behavior tree, especially since the game engine used for this paper offers Behavior trees as a standard solution for controlling the AI.

Debugging and understanding a Behavior tree is also substantially easier compared to a GOAP implementation. Since we are not technically directly writing the rules ourselves, it can at times seem like we lose control of the AI. For example, say if we used GOAP and wanted to AI to kill a player, the plan it devises might become more effective than what the developer originally meant for, making the game seem unfair. With the Behavior tree it's easy to oversee each action taken and when it will execute what. Generally, as long as the action space is not too large, and the number of rules required for each action is not too complicated, a Behaviour tree will be the easiest and most optimal choice.

We started with basic functions like recognizing when a player enters within sight range of the AI, aiming, shooting, shielding and chasing the player. With basic functions down, we could move on to actually making the AI navigate through the world.

Considering the fact that Setback is still a game in development at the time of writing this thesis, not all functions were implemented when we started using it as a test environment. All actions currently implemented, were only designed so that they would trigger when designated keys on keyboard and mouse were pressed, so for the AI to be able to reproduce all these actions, we had to replicate and modify them to be able to call the as functions for the AI to utilize. This was an easy fix for most of the actions, except for a couple like reloading. When reloading a gun in Setback, in one of the checks developers had coded they added a check for verifying if the current character had a the reticle UI element. And since none of the UI elements the player has are loaded when an AI controls the character in game, we had to temporarily modify weapons for the AI to have unlimited ammunition until a fix goes live.

To make and AI i UE4 be able to perceive other characters in game, we also had to modify the current character in game. This was done by adding two components to the AI; "AIPerception" and "AIPerceptionStimuliSource". The first one is a component for assigning the AI senses, and so the only one needed for us at the moment of writing was sight, so we added "AISense_Sight" to that component. The latter component "AIPerceptionStimuliSource" gives the character the trait of being able to be recognized by character with the "AIPerception" component. Again, in our case we are after sight so we also gave that component the "AISense_Sight" element. Now the AI can see and react with something when a player gets within range.

4.2 Navigation

The engine we are implementing the agent in is Unreal Engine 4. We are using a navmesh created by Recast to represent the environment, and Detour to implement the pathfinding with A*. Detour uses an extra step with SSF too smooth the path, this lets the agent take the shortest path through the nodes found by A*.

The navmesh is good at describing the walkable area of the map. In a normal environment, or map, the generator will only connect those areas that one can walk to and from. This means that if there are any cracks, pits or ledges this will produce a gap in the navmesh. It may even produce areas that are completely disconnected. This is why we need a method to link these gaps or unconnected areas. Most often a character has the ability to jump so that it can traverse such obstacles as a gap or pit by jumping over them. An agent can often also jump up or down a ledge, depending on the height of the ledge. We then need a way to add all of these undefined links to the navmesh, so that we may traverse the environment in the most optimal way and without adding to much data so that the pathfinding becomes too complicated.

4.2.1 Increasing step height

In UE4, when generating the navmesh one can decide on an important factor the step height. The step height is meant to represent the maximum height off a wall that an agent can just step on top of. This lets the generator treat stairs and small objects that may be spread around in the map as something the agent can walk over. By then increasing this to the height that the agent can jump, the navmesh gives navigable routes that include all the objects that the agent can jump up on. The problem that then arises is that the agent just know that it can jump up on these objects, but it has no trigger for a jump action. This is not a problem when the step height is set to the actual step height as the agent in UE4 have the ability to step up on low objects as a part of the walking system.

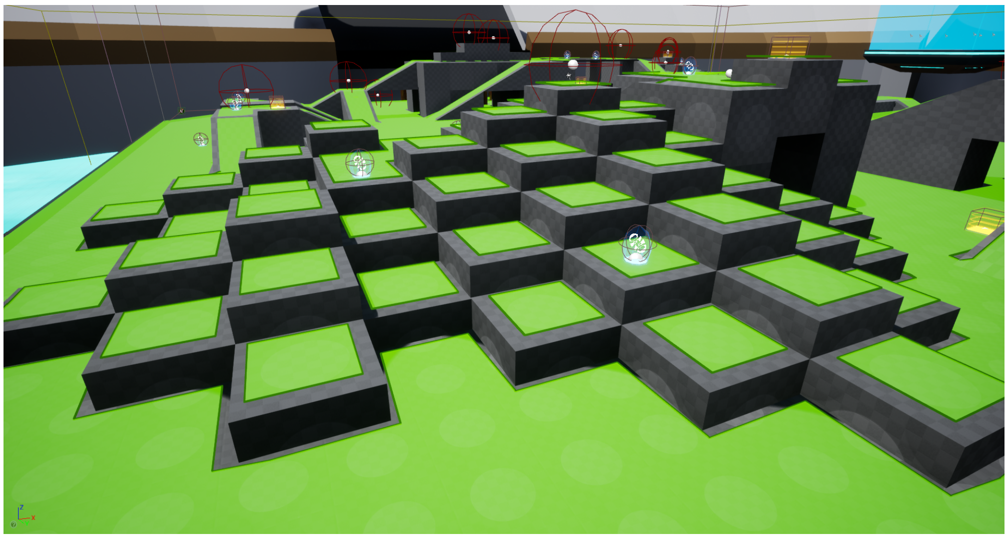


Figure 20: Standard step height in Unreal Engine 4

As the generator then sees all objects that are lower than the step height as flat terrain this gives the navmesh an unexpected representation when it is drawn. Since the navmesh generator tries to simplify the navmesh as much as possible it doesn't take the high step edges into any consideration and instead treats two flat surfaces that has some elevation between them as a slope. When the step height is low, Figure 20, this doesn't really stand out. When we instead use the step height in a way that is different from the expected use. The navmesh looks a lot more erratic, Figure 21, but it did not seem to negatively impact the pathfinding.

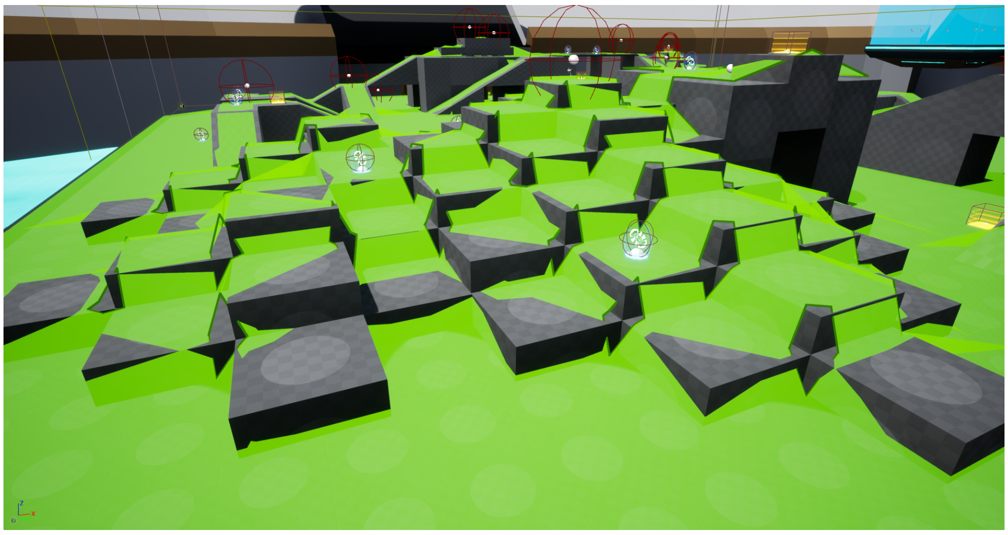


Figure 21: Increased step height in Unreal Engine 4

4.2.2 Jump detection

Detecting jump scenarios is then the next task. When doing this detection it is best to start with the basic and then build on that. First and foremost we need to detect if there is an object in front of the player. This can be done in UE4 by using Line Trace, a tool that traces a line between two vectors and determines if it hits anything. By doing this in the height that is the same as the height that the agent can step, we can say if the object is this high jump. An additional line trace at the same height as our max jump height, we can know if the object is to tall to jump over. There are multiple additional checks that is need or else the agent may jump when it is not wanted. See Figure 22 for a visualisation of the different line traces.

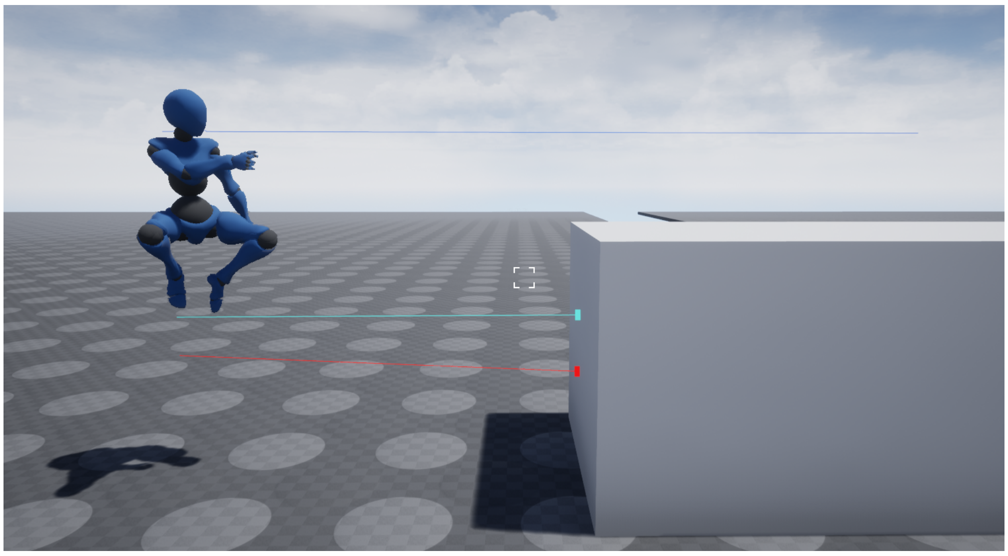


Figure 22: The line traces used to determine if a jump should be performed. The blue line is to check if the object is too tall, teal for checking that it is tall enough and the red line is to check if a edge should be jumped.

When increasing the step height to the maximum jump height, the agent now have a graph that includes all the routes the agent can take when running and jumping. We use the equations of motion to calculate the time before we hit the apex of the jump. Here v means reached velocity, u is the starting velocity for the jump upwards. a is the acceleration from gravity and t is the time. Equation 1 gives us the time before we hit the apex of the jump. We set v to 0 as we want to reach the top of the jump and that is when the agent stops having any upward speed.

$$v = u + at \Rightarrow t = -\frac{u}{a} \tag{1}$$

Changing this to UE4 variables, and adding that gravity is downwards, we get Equation 2

$$\text{time} = \frac{\text{jumpZVelocity}}{\text{gravity}} \tag{2}$$

We then calculate max jump height, Equation 3, here s is the jump height.

$$\begin{aligned} s =& ut + \frac{1}{2}at^2 \Rightarrow s = u(-\frac{u}{a}) + \frac{1}{2}a(-\frac{u}{a})^2 \\ \Rightarrow s =& -\frac{u^2}{a} + \frac{1}{2}\frac{u^2}{a} \Rightarrow s = -\frac{1}{2}\frac{u^2}{a} \end{aligned} \tag{3}$$

Then changing the name of our values to something more relevant for our problem. We get the max jump height, Equation 4

$$\text{maxJumpHeight} = \frac{1}{2}\frac{\text{jumpZVelocity}^2}{\text{gravity}} \tag{4}$$

Then at runtime, using line traces, we check if a jump is to be performed when following the path. An important factor when running the line trace is how long the line trace is going to be. We want the agent to be at the apex of the jump when it is above the edge that it is trying to jump on top of. We get the simple function, Equation 5

$$s = vt \Rightarrow s = v(-\frac{u}{a}) \tag{5}$$

Using more relevant variable names and changing the sign because of gravity, Equation 6

$$\text{lineTraceLength} = \text{forwardVelocity}\frac{\text{jumpZVelocity}}{\text{gravity}} \tag{6}$$

We use Function 6 to set the length of the line trace. The line trace we use is a bit longer than this as we run it angled towards and through the floor. This is to try and detect some of the scenarios that can arise when jumping between two platforms at the same height. We run a second trace in the max jump height to avoid trying to jump, when the object in front of the agent is too high. The third trace we need to run is a sphere trace. The sphere trace length is limited by the maximal slope we want to walk on and the step height. The longest the sphere trace can be away from the center of the character is defined in Equation 7

$$\text{maxShortSphereTraceLength} = \frac{\text{stepHeight} \times \cos(\text{maxSlopeAngle})}{\sin(\text{maxSlopeAngle})} \tag{7}$$

We call the SphereTraceLength max as this is the limit to how far we can put it in front of the agent, but the trace can safely be shorter. Pseudocode of the function can be seen in Listing 1.

```
JumpTest (agent)
  tooHigh = LineTrace(maxJumpHeight)
  if !tooHigh
    imediateWall = shortSphereTrace(maxStepHeight)
    if immediateWall
      agent.jump()
      return
    if moving
      hit, slope, relativeHeight, distance = \
          LineTrace(feetHeight)
      if hit && slope > 30
        if height < 0
          agent.jump()
          return
        else
          hit2, slope2, distance2 = \
              LineTrace(maxStepHeight)
          if hit2 && slope2 > 30 && \
              abs(distance - distance2) < 10
        agent.jump()
        return
```
Listing 1: Pseudocode of the test if a jump should be performed

A short explanation of the function: Every tick of game there is performed a line trace to check first if the object is too tall to be jumped. If true any further jump checks is stopped, because the wall is to high to jump. If the check doesn't hit anything we check if there is an wall or a platform right in front of the agent. If not we continue with the check for a standard jump. First line tracing if there is any object that in front of the agent that has a slope larger than 30 degrees. We then check if the object is higher than the step height of the agent with a second line trace. If this is also blocked by an object, with a slope steeper than 30 degrees and that the length of the first and second line trace is about as long. To also check if the two line traces is about as long is for avoiding jumping when walking up stairs.

There is an addition action that runs on every tick to improve the jumping of the agent, by giving the agent a small amount of forward momentum when it is in the air. It is able to get on top of a obstacle even if the jump is not performed at the perfect length.

We now look towards a second problem that occurs, because there is no damage to the agent from falling, it should always try to jump down from an obstacle. This means that we need to connect the navmesh of these obstacles to the navmesh on the ground beneath them. This can be performed with Nav Link Proxies, but as mentioned there are some significant drawbacks. Placing these links strategically in the path of the agent we can get a very different path than the one chosen if the agent has to never drop down from a high object. We can even use the links to make the agent use some of the jump pads that exist in the path. Some of these jump pads is even needed for the agent to be able to complete some maps.

4.2.3 Different sized agents

In most competitive shooters all the agents is of the same height and width. This is so the game is as fair as possible. The same is correct for Setback and we therefore only need to create one navmesh as the agents only has one size.

4.2.4 Navigation testing

The test method we will be using is with no change in the generation of the navmesh, increasing the step height to be the same as max jump height, just using Nav Link Proxies and using both, increasing the step height and Nav Link Proxies. In addition we will have test time with a human that is only allowed to use the standard movement, jumping and the elements in the map, and a second human time where the use of all the tools available in Setback is allowed, the most important additional tool is to look down and bash the shield into the ground to bounce.

We look at the time of completion, the number of checkpoints reached and the number of Nav Link Proxies that we needed to place. But we only use a minimal amount of Nav Link Proxies as a guideline for how many links is needed for the simplest solution with each method. This is important as each Nav Link Proxy represent a time investment.

The checkpoints the agent needs to pass through is marked 1-13. In the top down view in Figure 23 we can see the gray lines where the navmesh doesn't cover. This is edges that can either be jumped or dropped down from. As seen in Figure 24 two of the checkpoints is in the air. This is checkpoint 6 and 13, they are only reachable when using a jump pad. The jump pads are the yellow squares.

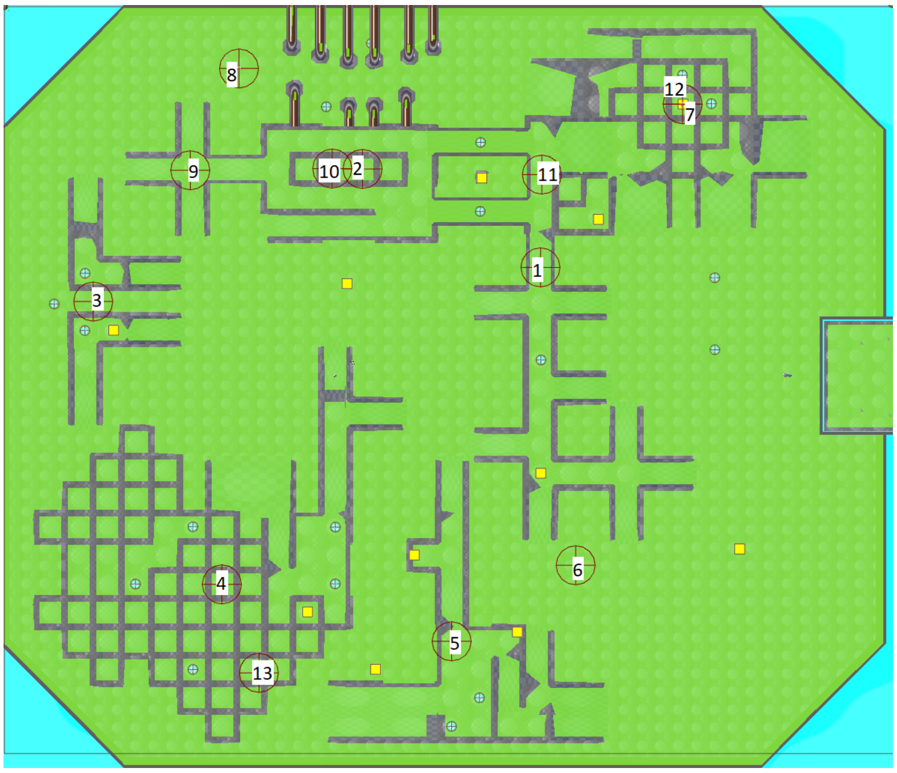


Figure 23: Our testing map as seen from the top

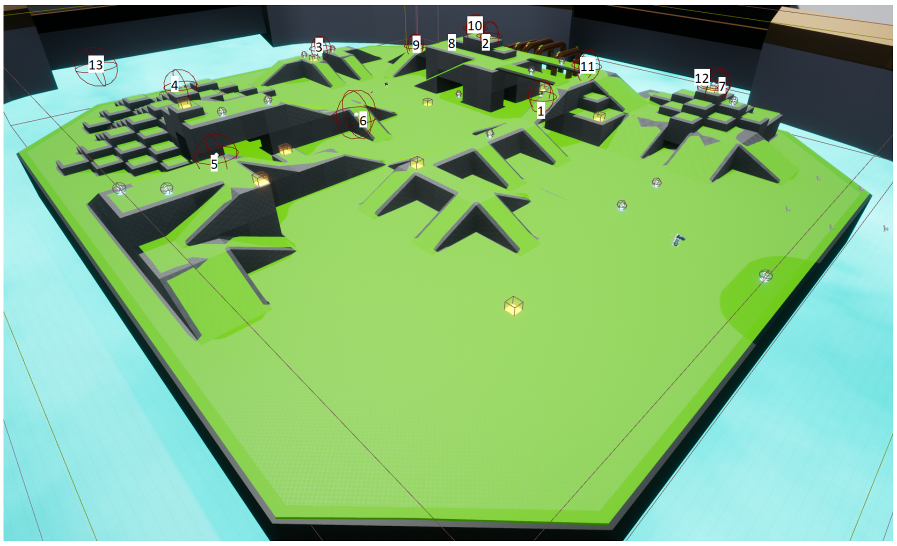


Figure 24: Our testing map as seen from the side

5 Results

5.1 Shooting

Shooting results					
Weapon name	Magazine size	Bullet Spread	Bullet velocity	Hits on still target	Hits on moving target
Handgun	15 bullets	None	Medium speed	15/15	3/15
Revolver	6 bullets	None	High speed	6/6	4/6
Machine pistol	80 bullets	Large	Medium speed	12/80	8/80

Table 1: Average results from shooting tests with different weapons over 5 tests per weapon

After having implemented the Behavior tree with the desired action space, testing performance was the next step. While testing shooting, the AI was set to notice the player and start shooting when within a 30m range. We tested for 3 weapons with different behavior; the handgun, the revolver and the machine pistol. With the handgun and revolver were semi-automatic so there was no bullet spread present, and the machine pistol being the only automatic weapon tested with bullet spread. Check out the video appendix located in the end of this paper for access to a video preview of the shooting.

5.2 Navigation

Navigation results			
Method	Time	Checkpoints Reached	Nav Links
Default	1 m 8 s	6/13	-
Step height	1 m 12 s	11/13	-
Nav Link	1 m 20 s	13/13	24
Both	1 m 9 s	13/13	4
Human only jumping	1 m 5 s	13/13	-
Human Record[20]	0 m 32 s	13/13	-

Table 2: Results from tests of navigating a test map with different methods

When testing we timed how long the agent used around the map by making the agent follow a set of waypoints placed at the same places as the checkpoints. When using the default settings the agent was unable to reach a lot of the checkpoints so the waypoints had to be moved to let the agent somewhat complete the map. All the other methods use the same placing of the waypoints. Some of the checkpoints are placed in the air, so when placing the waypoints for these checkpoints, we have to place the corresponding waypoint on the ground. This limitation is because the agent is moving on a navmesh. The different times, the checkpoints reached and the Nav Link Proxies used, are shown in table 2[5].

[5]Video of 10 agents: https://youtu.be/h4kGkTUewmw

Runtime				
Number of agents	Game Average Runtime	Game Max Runtime	Frame Average Runtime	Frame Max Runtime
0	7 ms	8 ms	17 ms	17 ms
1	9 ms	10 ms	17 ms	21 ms
5	10 ms	11 ms	17 ms	21 ms
10	13 ms	14 ms	17 ms	22 ms
15	15 ms	20 ms	17 ms	22 ms
20	17 ms	23 ms	17 ms	23 ms

Table 3: Results from tests of navigating a test map with different methods, was run on the laptop specified in table 4

An important note to the worst runtimes is that the first five seconds are excluded. This is because the max runtime is very high when the level is loaded for starting to play. The test was done on a laptop with the specifications listed in Table 4.

Specifications	
Component	Name
Laptop	MSI GT62VR
CPU	Intel i7 7700HQ
GPU	NVIDIA GeForce GTX 1060 6GB
Memory	16 GB 2400 MHz
Storage	Samsung SSD PM871a

Table 4: Specifications of the laptop the runtime test, Table 3, was run on

6 Discussion

In the following chapter we will discuss results on the implemented methods used for behavior and navigation, and evaluate potential flaws and improvements to be made. Lastly, we go over how teamwork between the two of us went over the course of writing the thesis.

6.1 Behavior

Observing the shooting results from Table 1 we can first take notice of the fact that while moving, the number of hits are generally lower than when standing still. Main reason for this, is that the bullets in Setback are not instant and have a certain velocity. The revolver is the only one still being able to hit the majority of bullets on a moving target, due to the fact that its bullets velocity is higher. While the machine pistol does not hit as near as good as the two others, it doesn't to that much worse on the moving target, because of the bullet spread the weapon has which makes some of the bullets hit even when not going directly in the direction of the reticle. Lastly, the handgun, was the one experiencing the highest difference in bullets hit when hitting a moving target compared to a still target. The handguns bullet velocity is not very high, and since the implemented shooting action does not try to make up for bullet velocity by aiming ahead of the target, it will miss a majority of the shots.

However, neither missing a lot or hitting a lot of the bullets incline that the current shooting implementation is desired. As mentioned before, both a too bad, and a too good AI, will decrease the players immersion in the game. You want the game to be challenging, but not unfair. Considering the fact that, in most cases, the target will be moving, the current AI will lean towards performing too bad. By implementing a function to take into account the enemy's current velocity, direction and the current weapons bullet speed and spread, one could make the AI make up for distance and speed by aiming ahead of the target.

So, let's say the latter function is implemented. Now we currently have an AI that hits all the shots no matter how far away you are and how fast you move, making it seem unfair to the player. This is where we need to implement measures to make the AI make mistakes sometimes, like players often do. One simple way is to add something called "fuzzing" to the function that decides where the AI aims. Fuzzing, or fuzz testing, is often used to generate unexpected or invalid inputs for checking that programs run as they should, but we can also use it to add an offset to the AI's aim. If difficulty were to be implemented, increasing or decreasing the amount of fuzzing would be one simple step to make the AI better or worse, depending on the skill level of the player.

6.2 Navigation

When looking at the navigation results one of the first things one might notice is that the method without any modifications is the best time for the agent. This is because the total path is much shorter as the agent doesn't reach a lot of the waypoints. To complete the level you have to get all the checkpoints even though you might somewhat get around the map. We also see that when only increasing the step height we are also missing two checkpoints. That is because these two checkpoints are in the air and only accessible by using jump pads that are spread around the map. There is no way to represent a jump pad with the current solution.

When using Nav Link Proxies we can get to all the checkpoints, this is because with Nav Link we can make a path that walks over the jump pads. This is the only way we found in UE4 that we could get the agent to use the jump pads. This is one of the biggest benefits of the Nav Link

Proxies as we can make paths that can go through checkpoints that we can't represent in the navmesh, since they are in the air. As we have now reached all the checkpoints we have finally completed the map. The problem now is that we can see that the time has suffered. The time can be improved by placing as many Nav Link Proxies as there are jumps and drop possibilities. Placing all these Nav Links would take a lot of time and is limited to the paths the developer see as they place the Nav Links.

By using both increased step height and Nav Link Proxies we cover all the possibilities of places that can be jumped up. The problem now is to add the places that can be jumped edge to edge, and dropped down. Doing this change we see that the minimal amount of Nav Link Proxies decreases drastically. In addition to the amount of Nav Link proxies decreasing the time to finish the route also decreases. This is the result of the fact that the pathfinding is better at finding an optimal paths than when guessing the optimal path when placing Nav Link Proxies.

We see that the the agent almost achieves the same time as a human player, under the same constraints. On the other hand when a human player have been able to train for an extended period of time and may use all the movement options, the humans time is halved in comparison to the agents best time.

6.2.1 Jumping

How well the jumping of the agent works was mostly as expected. There are some edge cases that still is not optimal, and because of this it would be better to let the map tell the agent to jump instead of the agent guessing that it should jump. One of the problems that arise is that if the agent jumps and misses where it is going, it may get stuck next to a wall. This is because the path isn't redrawn when the agent loses the path. The agent continues to follow the path for a while before it times out and redraws the path. It can also might get stuck in a loop of jumping over a little plateau, this is because we give the agent some forward momentum when it is in the air. When it then tries to jump on top of the plateau it may turn around and use the momentum in the wrong direction.

You can still avoid these lockups, but this is done by moving the waypoints around and sometimes placing volumes that removes the possibility of walking in an area to avoid it taking a shortcut where it hits an edge and gets knocked out of the path. We can add an event to the Nav Link Proxies to trigger jumps, avoiding having the agent guess when it should jump, but this would bring with it all the drawbacks of the Nav Link Proxies.

Floating platforms are a weak point. Most of this is solved with running sphere traces in the correct height. The problem is that these sphere traces need to be close to the agent or they might trigger on a slope. So the agent can't really jump long before it hit the floating platform.

The method proposed by Budde seems to be one of the best ways to implement navigation links in a game without using too much time and laying the burden of when a jump should be performed at the map instead of checking at runtime. The problem is that Budde‘s solution does not include a way to integrate jump pads and points that need to be reachable in the air. The solution that de Silva presented will gives us the possibility to have waypoints in the air, but then we meet the second problem of how should we plan a path that lets us get to the waypoint. In addition to the problems of actually planning a path that is possible, de Silva uses a voxel grid which may create too large graphs in addition to not representing the environment well enough.

6.2.2 Perfomance

The performance of one agent is well within the constraints of using less time than it takes to draw a single frame at 60 frames per second. The goal of all real time agents is to be finished with their planning within the time it takes to draw one frame at 60 frames per second, which is 17 ms. We see in Table 3 that we are within this margin with up to fifteen agents. With twenty agents, the time it takes the game to perform all its actions, is the same as the draw time of one frame. We start to be reliant on the speed of the game instead of how long it takes to draw each frame of the game. With up to fifteen agents without any loss to performance is within the bounds of the game, as most of the time you would not use more than 7 agents at the time.

We felt that the actual runtime of the game was a better performance metric and better showed the impact multiple agents had. In contrast to saying that a single agent on average uses 0.01 ms to search the behavior tree and then an average of 0.08 to find the path to the next checkpoint, the time it takes to find the path varies with 0.03 ms. This time is much lower than one agent actually uses as we are running multiple line traces, and the performance hit of the line traces is not represented in the time when just looking at the traversing of the behavior tree and the time it takes to find a path. We only need to traverse the behavior tree and find a path each time the agent has passed a checkpoint.

Most of all the extra runtime goes to running the line traces, without these we would decrease the runtime that the agent uses when it moves around. This gives an additional argument to generating the jump links before we start to play. As the in game performance hit would be lower from just using links instead of running multiple line traces per agent.

6.3 Cooperation

Finding a workflow between the two of us was fairly easy, as we have cooperated on other projects together before. In addition to having worked on a machine learning project together last semester, we also meet in real life rather frequently. Therefore because we know each other well, even if we disagree on certain aspects of a problem, we are quick to come to an agreement. We have focused on keeping the bar for letting the other person know when something is wrong low, so any disagreements or differences in view may be dealt with early for an optimal collaboration. As time was of the essence during the project cycle, we made sure to regularly stay in contact to keep communication up. We had status updates through voice chat daily, and at times met up in person to discuss larger problems. If one of us were having a problem, we made sure that asking for help or advice should be easy. Because of that, we were able to make up for each other weaknesses, and sustain a very efficient development process.

7 Conclusion

In this paper we have delved into theories behind behavior for AI's in games, and researched different ways to implement a working pathfinding method for AI's in a complex 3D environment, using Setback as an example. We used methods currently available from Unreal Engine 4, like Behavior trees and the Navmesh system. We then compared them to other current implementations with the same purpose in mind. Behavior trees are undoubtedly, from what we have learned throughout writing this thesis, the best current option for controlling AI Behavior for a game like Setback. With its easy of use and implementation making it the superior choice over methods as Goal-Oriented Action Planning.

The navigation gives us more problems. This is much because of the limitations of how a navmesh represents the environment. It only represents the walkable surfaces without any connection between areas that can be moved between by a jump or dropping down. We try to solve this with increasing the step height, but we still need to place additional Nav Link Proxies to drop down. To take a qualified guess when the agent need to jump by using line traces is not an optimal solution. Budde's[4] solution with generating jump links that can be traversed diagonally and let the agent jump different trajectories, would solve the problem of guessing when the agent need to jump by integrating this in the navmesh. Furthermore we need a way to represent interactable objects in the world that would help the agent, like jump pads and places where it can swing. In addition to this, we are being limited to only have checkpoints on the ground, as having them in the air disconnects them from the navmesh. We conclude that the current service for map representation from Unreal Engine 4 is not enough to satisfy a game such as Setback, and an alternative need to be developed to allow for taking events in the air into account.

7.1 Future work

Further improvement of the shooting can be done by implementing said functions in discussion to make up for slow bullet velocity and high bullet spread. Figuring out how well the AI needs to perform to give players the challenge they want will need substantial testing with users of different skill sets. The current behavior tree in itself could also be further improved upon to include more ways to interact with players in the world, which in turn will give players an impression of an even more intelligent opponent in the game. Looking at the Behavior tree from the AI of "Tom Clancy's The Division"[37] as seen in Figure 25, it goes to show that Behavior trees can achieve quite complex AI responses and behavior. It should be noted that that while the Behavior tree can be improved upon, it will not require near as much depth as the AI in The Division, as it in itself is a more complex game with a large number of mechanics to take into consideration. It would also be interesting to send the action space through a machine learning algorithm to observe how the behavior changes, which is something Unreal Engine 4 at the time of writing this is not a function it offers. Tactics are a big part of games, and letting machine learning have a go at it and observing the results may yield in new ideas one wouldn't have thought of to implement originally as an event or trigger for an action in the behavior tree.

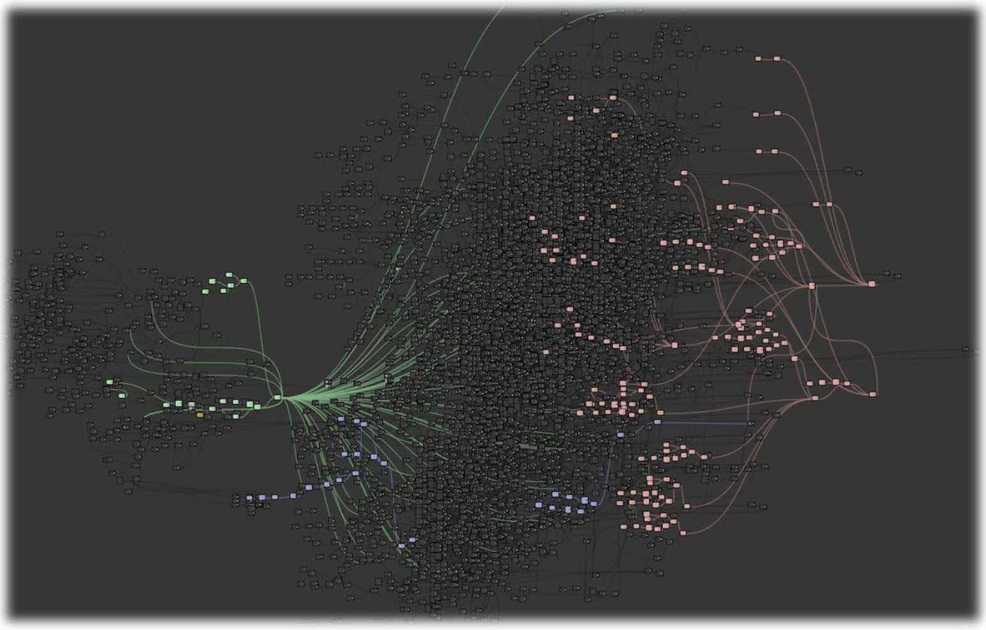


Figure 25: Behavior tree for AI behavior in Tom Clancy's The Division

Implementing and testing Budde's[4] thesis in Setback would be the next step, then trying to add the generation of jump pad links and swinging links using the same idea Budde presented and test if the graph is still lean enough. It would also be interesting to see if it would be possible to create links that would let the agent find paths to find reachable points in the air. The second part of this would be to try and implement de Silva's[6] solution with voxels. Then comparing navmesh with Budde's links with de Silva's solution. Which would make it possible, and with the lowest runtime when implementing jump pads, swinging, and navigating to checkpoints in the air.

References

[1] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *Int. J. Comput. Games Technol.*, 2015, January 2015.

[2] Daniel Brewer. Simplified 3d movement and pathfinding using navigation meshes. *Game Ai Pro 3*, page 21:265–274, 2017.

[3] Mat Buckland. *Programming Game AI by Example*, pages 333–340. Wordware, Sudbury, 2004.

[4] Sara Budde. Automatic Generation of JumpLinks in Arbitrary 3DEnvironments for NavigationMeshes. "`http://www.shock-tactics-game.com/theses/Sara_Budde_Thesis.pdf`", 2013.

[5] Xiao Yan Cui and Hao Shi. An overview of pathfinding in navigation mesh, 2012.

[6] Gabriel Quaresma Moreira da Silva. Jumping AI for Unreal Engine. "`https://www.iconline.ipleiria.pt/bitstream/10400.8/4545/1/gabriel_silva_2162071_Jumping_AI_for_Unreal_Engine.pdf`", 2019.

[7] K. Daniel, A. Nash, S. Koenig, and A. Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, Oct 2010.

[8] Epic Games. Nav link proxies. "`https://docs.unrealengine.com/en-US/Resources/ContentExamples/NavMesh/1_2/index.html`".

[9] Subir Ghosh. Visibility algorithms in the plane. *Visibility Algorithms in the Plane*, 01 2007.

[10] Subir K. Ghosh and Partha P. Goswami. Unsolved problems in visibility graphs of points, segments, and polygons. *ACM Comput. Surv.*, 46(2), December 2013.

[11] J. Greisemer and C. Butcher. Creating the illusion of intelligence: The integration of ai and level design in halo. In *In the Proceedings of the Game Developers Conference*, 2002.

[12] Peter Higley. Goal-oriented action planning: Ten years old and no fear! "`https://www.youtube.com/watch?v=gm7K68663rA`", 2017.

[13] D. Islam. Handling complexity in the halo 2 ai. In *GDC 2005 Proceedings*, 2005.

[14] Marcelo Kallmann and Mubbasir Kapadia. Navigation meshes and real-time dynamic planning for virtual worlds, 2014.

[15] Tijs Leenknegt. Three-dimensional path planning in complex enviroments. "`https://lib.ugent.be/fulltxt/RUG01/002/033/150/RUG01-002033150_2013_0001_AC.pdf`", 2013.

[16] Lars Lidén and Valve Software. Strategic and tactical reasoning with waypoints. *AI Game Programming Wisdom, Charles River Media*, pages 211–220, 2002.

[17] Yun-Hui Liu and Suguru Arimoto. Path planning using a tangent graph for mobile robots among polygonal and curved obstacles: Communication. *The International Journal of Robotics Research*, 11(4):376–382, 1992.

[18] Thomas Lopez, Fabrice Lamarche, and Tsai-Yen Li. Space-time planning in changing environments : using dynamic objects for accessibility. *Computer Animation and Virtual Worlds*, 23(2):87–99, 2012.

[19] Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22(10):560–570, October 1979.

[20] Mew. Setback - 32:23 run. "https://www.youtube.com/watch?v=2jxVfvGNoy0".

[21] Mikko Mononen. Simple stupid funnel algorithm. "http://digestingduck.blogspot.com/2010/03/simple-stupid-funnel-algorithm.html", 2010.

[22] Mikko Mononen. Recast navigation. "https://github.com/recastnavigation/recastnavigation", 2019.

[23] Christoph Niederberger, Dejan Radovic, and Markus Gross. Generic path planning for real-time applications. In *Proceedings of the Computer Graphics International*, CGI '04, page 299–306, USA, 2004. IEEE Computer Society.

[24] Christian Nutt. The technology of f.e.a.r. 2: An interview on engine and ai development. "https://www.gamasutra.com/view/feature/132280/the_technology_of_fear_2_an_.php", 2008.

[25] Ramon Oliva Martínez. A framework for navigation of autonomous characters in complex virtual environments, 2016.

[26] Jeff Orkin. Applying goal-oriented action planning to games. "https://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf", 2003.

[27] Jeff Orkin. Three states and a plan: The a.i. of f.e.a.r.1 game developers conference 2006. In *GDC 2006 Proceedings*, 2006.

[28] Joseph O'Rourke. *Art Gallery Theorems and Algorithms.* Oxford University Press, New York, 1987.

[29] Rama. Rama custom pathfinding. "https://forums.unrealengine.com/development-discussion/c-gameplay-programming/34485".

[30] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, June 1984.

[31] Ivan Simecek, Daniel Langr, and Pavel Tvrdik. Minimal quadtree format for compression of sparse matrices storage. In *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 359–364, 09 2012.

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

[33] Peter Norvig Stuart Russel. *Artificial Intelligence, A Modern Approach*, page 93. Pearson Education Inc., 2010.

[34] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introductions to Algorithms*, page 658. MIT Press and McGraw-Hill, 2001.

[35] Tommy Thompson. Outsmarting the covenant: The ai of halo 2. "https://medium.com/the-cube/theaiofhalo2-33e824209a4c", 2017.

[36] Tommy Thompson. The perfect organism: The ai of alien: Isolation. "https://www.gamasutra.com/blogs/TommyThompson/20171031/308027/The_Perfect_Organism_The_AI_of_Alien_Isolation.php", 2017.

[37] Tommy Thompson. Enemy ai design in tom clancy's the division. "https://www.gamasutra.com/blogs/TommyThompson/20181203/331725/Enemy_AI_Design_in_Tom_Clancys_The_Division.php", 2018.

[38] Tommy Thompson. The ai of half-life: Finite state machines. "https://www.youtube.com/watch?v=JyF0oyarz4U", 2019.

[39] Unity. Nav link proxies. "https://docs.unity3d.com/Manual/nav-Overview.html".

[40] Valve. Half-life source code. "http://metamod.sourceforge.net/files/sdk/", 2013.

[41] Wouter van Toll, Roy Triesscheijn, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, Julien Pettré, and Roland Geraerts. A comparative study of navigation meshes. In *Proceedings of the 9th International Conference on Motion in Games*, MIG '16, page 91–100, New York, NY, USA, 2016. Association for Computing Machinery.

[42] Walldiv. Nav link proxies. "https://forums.unrealengine.com/unreal-engine/feedback-for-epic/109539".

[43] Dustin White. Clarifications and extensions to tactical waypoint graph algorithms for video games. In *Proceedings of the 45th Annual Southeast Regional Conference*, ACM-SE 45, page 316–320, New York, NY, USA, 2007. Association for Computing Machinery.

[44] Hao Shi Xiao Cui. Direction oriented pathfinding in video games. "https://www.researchgate.net/publication/267405818_Direction_Oriented_Pathfinding_In_Video_Games", 2011.

[45] Peter Yap. Grid-based path-finding. In *Proceedings of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, AI '02, page 44–55, Berlin, Heidelberg, 2002. Springer-Verlag.

[46] Peter Yap. New ideas in pathfinding. "https://www.aaai.org/Papers/Symposia/Spring/2002/SS-02-01/SS02-01-018.pdf", 2002.

[47] Weiping Zhu, Daoyuan Jia, Hongyu Wan, Tuo Yang, Cheng Hu, Kechen Qin, and Xiaohui Cui. Waypoint graph based fast pathfinding in dynamic environment. *International Journal of Distributed Sensor Networks*, 11(8):238727, 2015.

A Blueprint of the jump function

This is the jump appendix, with blueprint of the jump function needed for the AI to jump over obstacles in the game.

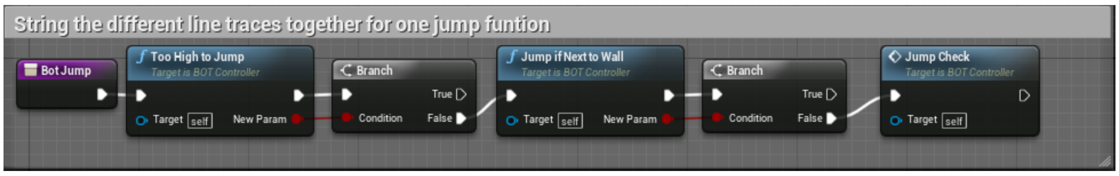


Figure 26: Combination of all the jump functions

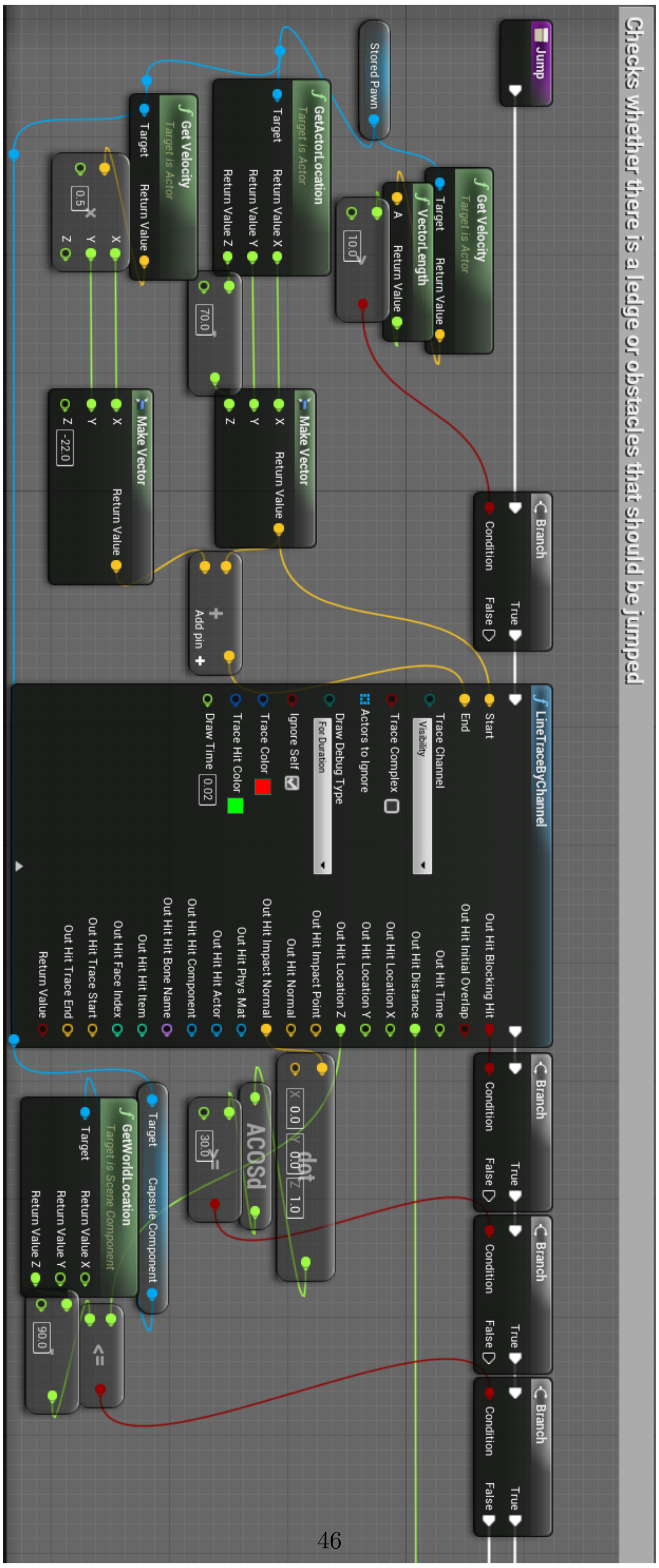


Figure 27: First part of the Jump Check Blueprint

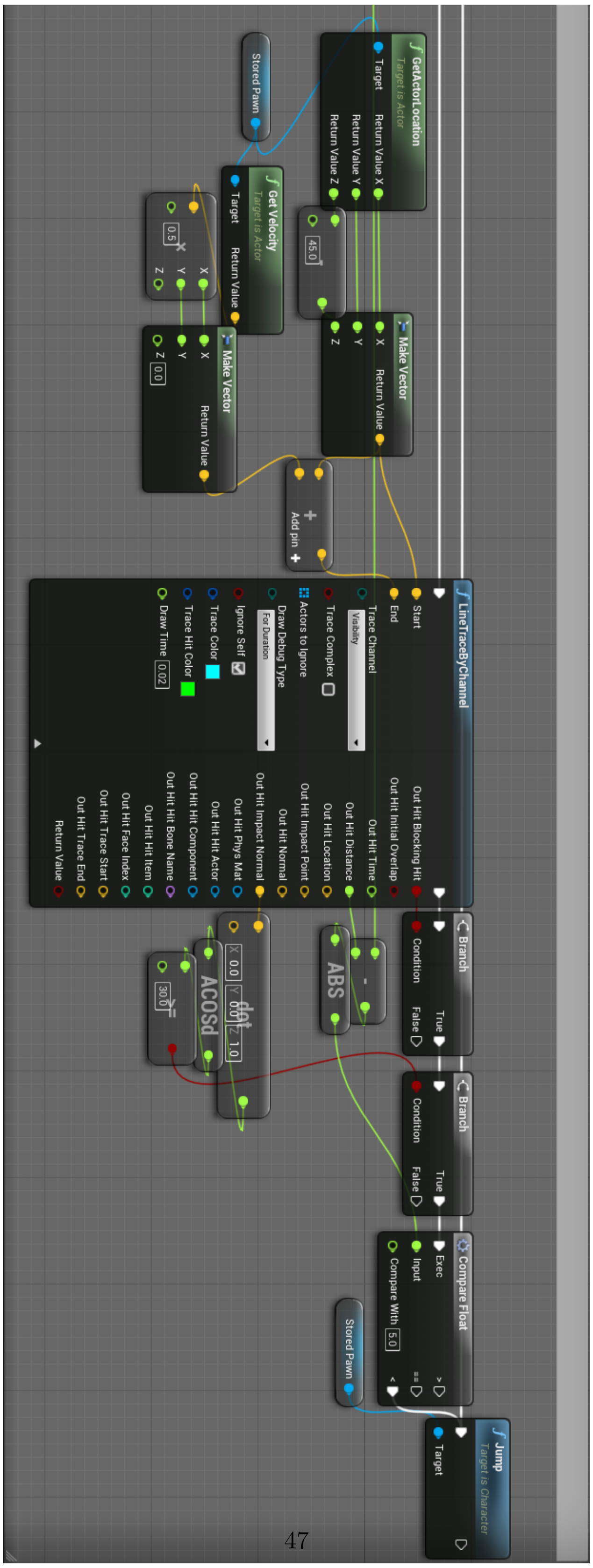


Figure 28: Second part of the Jump Check Blueprint

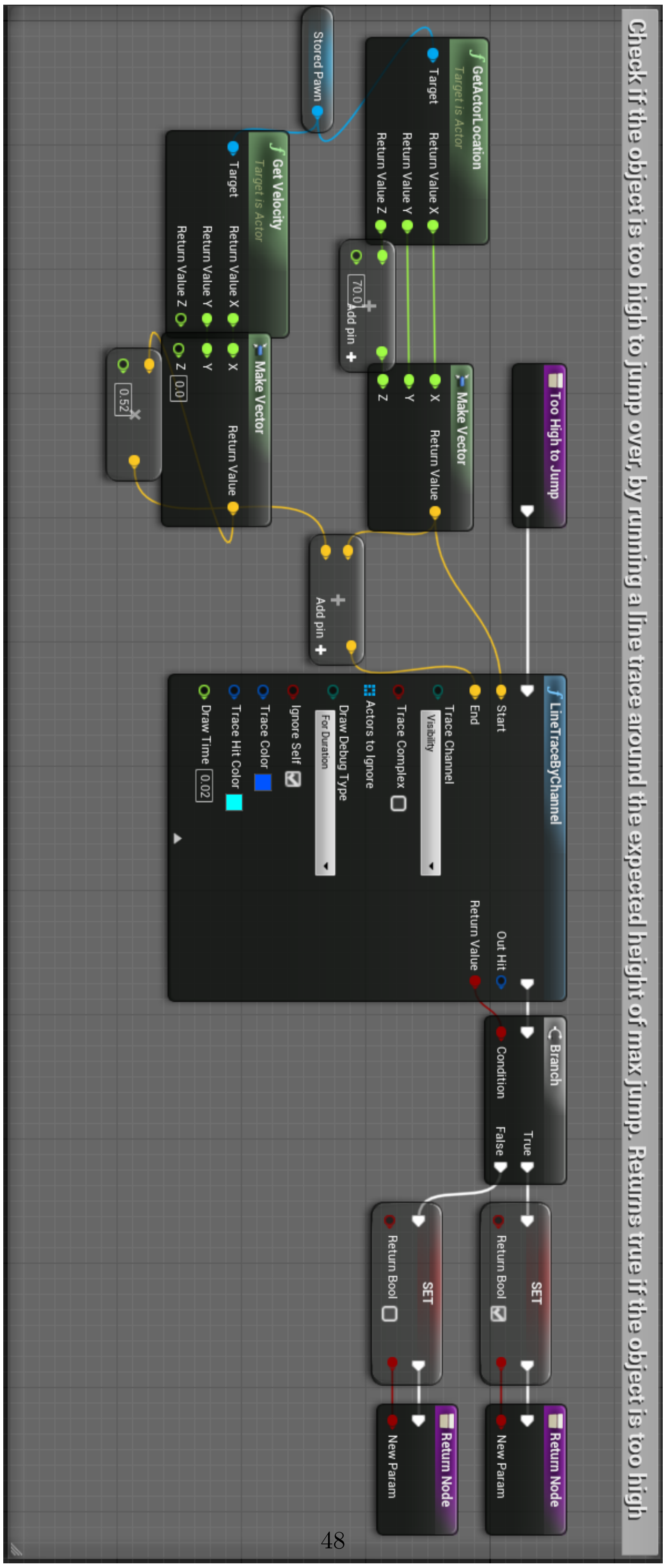

Check if the object is too high to jump over, by running a line trace around the expected height of max jump. Returns true if the object is too high

Figure 29: This is run to check if obstacle is too high

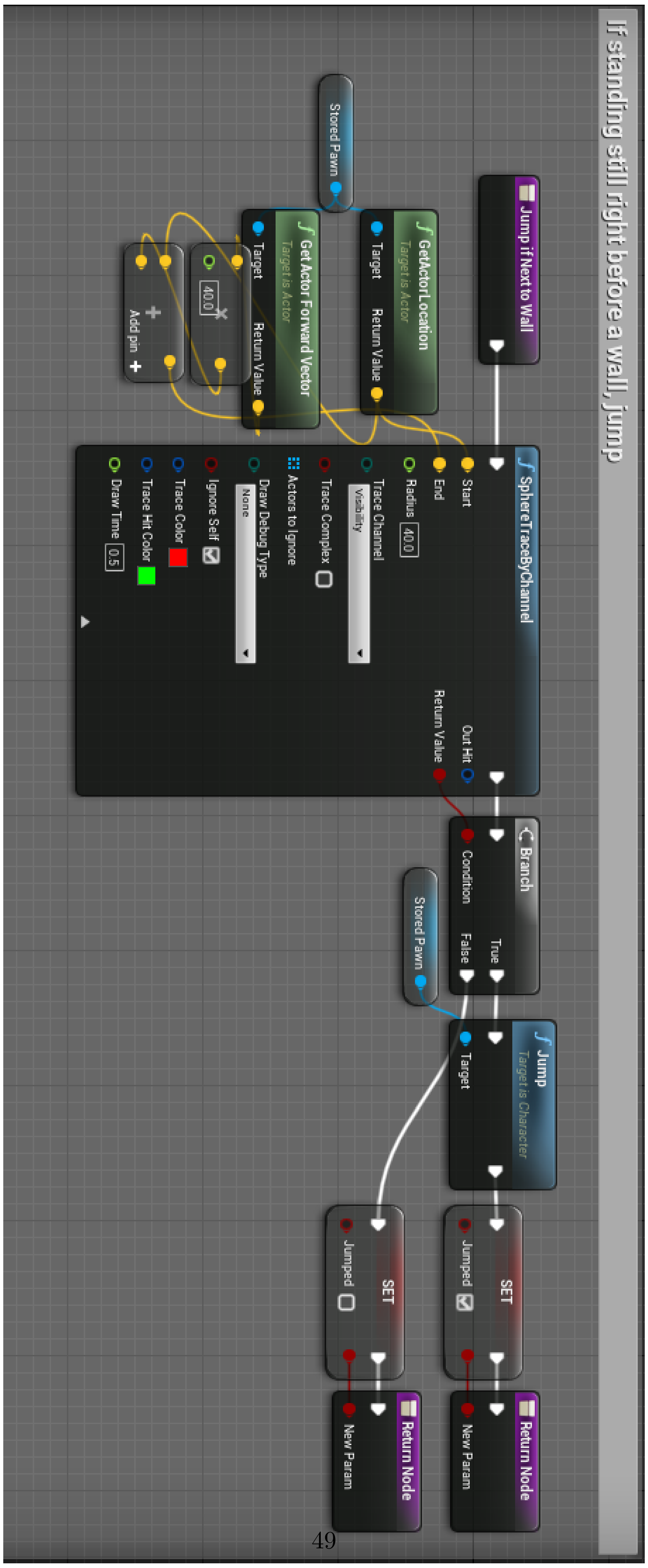


Figure 30: This runs to make it jump if its right in front of a wall

B Resource Appendix

Should you want to see some of the implemented functions in actions, check out the following links for youtube videos with a preview.
Machine pistol shooting: https://youtu.be/LyFaUFn3WK8
Revolver shooting: https://youtu.be/CzNIN959LRs
Shield-bashing: https://youtu.be/BxLzeKcCIL8
Nav Link Proxy: https://youtu.be/iPPwf3aWFu4
Off-Mesh Link: https://youtu.be/-aPSwrHybVk
NavMesh Link: https://youtu.be/Fyi7LjUJjYw
Agents completing map: https://youtu.be/h4kGkTUewmw

To utilize all actions currently implemented for a player in Setback, these functions need to be added to the "BP_SBPlayer" blueprint in the Setback project files if the bot is to gain access to them, as seen in Figure 31.

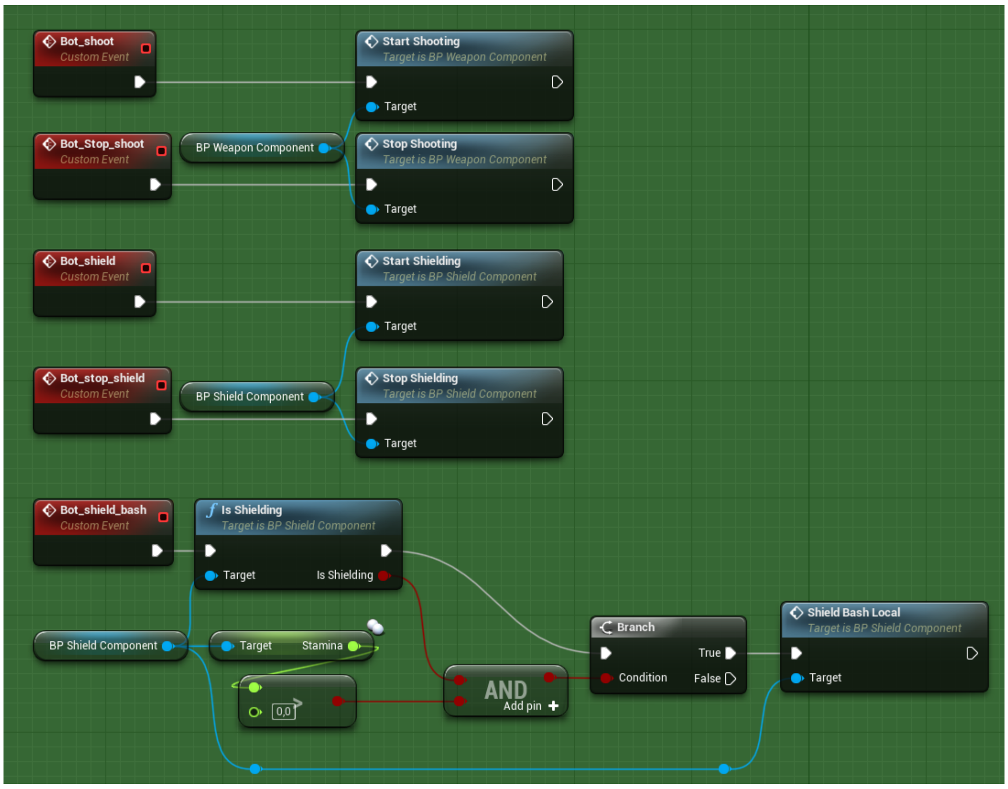


Figure 31: Functions to add for bot in BP_SBPlayer