International Journal of New Technology and Research (IJNTR)
ISSN:2454-4116, Volume-4, Issue-3, March 2018 Pages 21-25

# Traversal of 3D AI Objects in Virtual Game Worlds

**Syed Waqas Ahmed**

***Abstract*- This paper aims to highlight the underlying problem of AI traversal in virtual game worlds. AI's self-awareness as an entity, solely depends on the programmer and hence can run into problems if it's not coded properly. The purpose of this research is to eliminate the various factors involved in the traversal of AI and provide a simple solution to the problem that even a novice would be able to code their AI to move around the terrain of the virtual world or in-game world. By using collisions, objective markers and virtual invisible points, the AI can not only move in a 360-degree direction, but cover sufficient in-game mileage and abstain from traversing in to the 'danger zones' which are aptly titled due to them being inaccessible or missing the collision feature.**

**The medium of choice to depict the virtual world is Unreal Engine 4. The reason for choosing UE4 is because it has a similar layout to other game engines like Unity and hence migration of code is possible though not without its own trials.**

***Index Terms*- video games, education, simulation, artificial intelligence.**

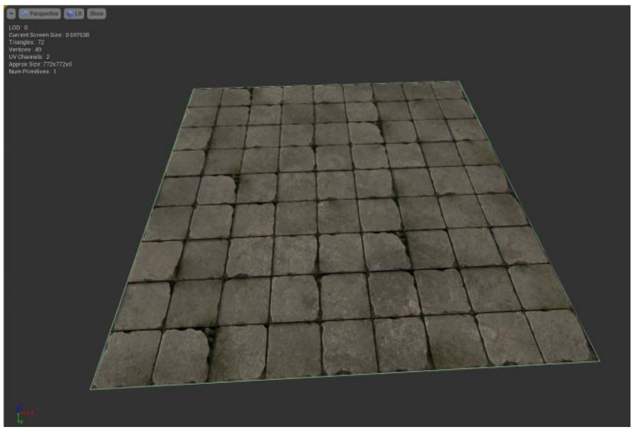
## I. INTRODUCTION

Video gaming has currently taken the market by storm. In the last 30 years the industry has gone from simple 2D platformers like Mario to full blown 3D games with huge open worlds and stunningly beautiful visuals. While an open world game certainly looks alluring, the AI at work behind the several NPCs (non-playable characters) and enemies in these games is making sure they are a contributing factor in making the game look 'real'. As such, one of the most fundamental functions of the AI characters is to move. Simply walk around or run. While in theory it may sound simple enough at first glance but the underlying problem this presents is far greater.

The platforms that an AI object moves on has its very own collision mesh. As do every other object that it interacts with. To better understand what a collision mesh is, imagine a thin piece of layer surrounding the object. This solidifies the game object preventing any other object to move through it. When it comes to platforms, it prevents the AI object to 'fall through' the platform, providing it a solid path to move on. Collision mesh on other game objects allow the AI to recognize the object as an obstacle hence preventing direct traversal through it.

An example is taken of an in-game object of a boulder. If the boulder doesn't have a collision mesh. The AI can simply walk right through it since at this moment, it's just a 3D model. However, if collision is applied to it. The AI's own collision mesh will collide with that of the boulder and prevent movement through it.

To sum it up every in-game solid object needs to have a collision mesh and that also goes for the platform on which the AI is moving.



LOD: 0
Current Screen Size: 0.697638
Triangles: 72
Vertices: 49
UV Channels: 2
Approx Size: 772x772x0
Num Primitives: 1

FIG 1: A PLATFORM WITH A COLLISION MESH AS DEPICTED BY THE BARELY VISIBLE GREEN OUTLINE

Once this has been achieved the real problem on how to get AI to move to a specific point or multiple different points on that platform arises. Due to diversity in video games, these platforms can be in many shapes and sizes, the destination point of the AI can either be 1m away or 100m away. It can even be on the same platform or another platform divided by a gap in between. There are hundreds of different situations depending on the programmer.

For the sake of simplicity, the graphical assets used in this research were created in both Blender and 3DS Max, after which they were imported into Unreal Engine 4 and given a collision mesh. Since both tools are quite similar, this allows for this research to be effective even in the case where other graphical tools are being used.

## II. RESEARCH ELABORATIONS

The figures below show the overhead map of the environments in UE4, i.e. the terrain that was used for the AI traversal. Multiple environments were created in order to receive data from multiple situations.

Care was taken to diversify the environments as much as possible so the results can cover the maximum amount of ground and leave no variable unchecked.

**Syed Waqas Ahmed**, Department of Software Engineering, Sir Syed University of Engineering and Technology, Karachi, Pakistan

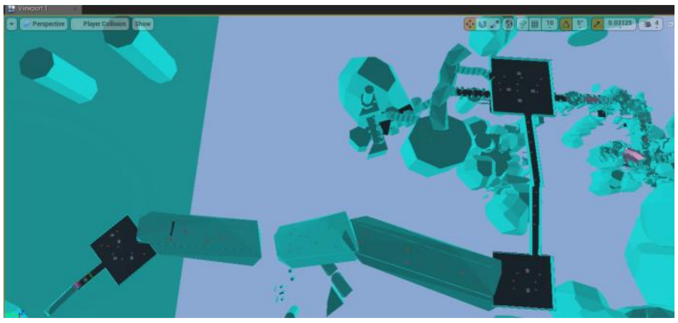


FIG 2: TESTING ENVIRONMENT A

Environment A is a vast open environment with vastly wide and long platforms. There is a narrow pathway between 2 platforms as well.

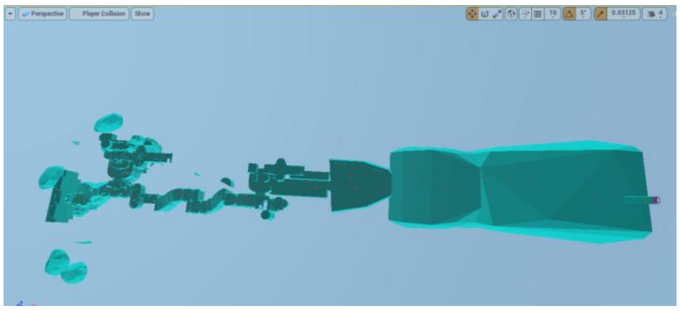


FIG 3: TESTING ENVIRONMENT B

Environment B starts off with a narrow conjoined pathway and opens up as you go forward.

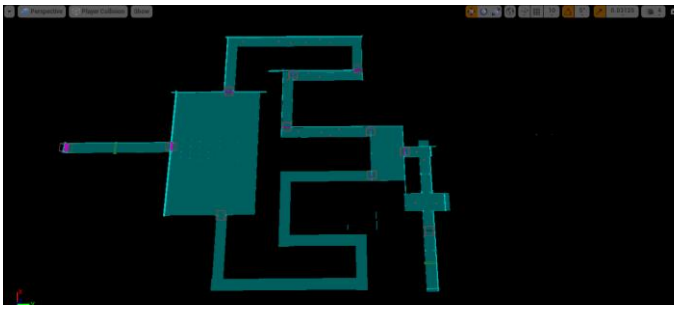


FIG 4: TESTING ENVIRONMENT C

Environment C is created in the form of a simple and narrow maze.

Care was taken that each environment was as diverse as possible in order to maximize efficiency.

The AI object used for this study was the default Unreal Engine 4 mannequin.

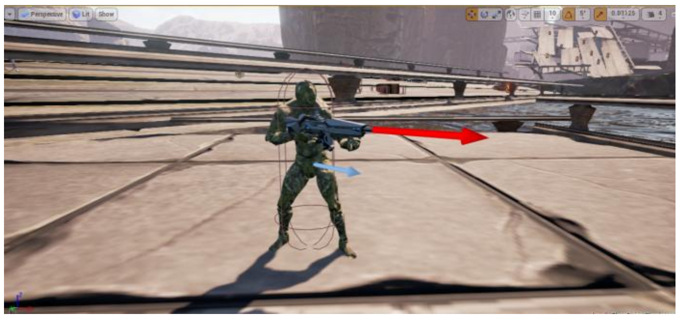


FIG 5: AI OBJECT

The objective is to make this AI move across the environments on its own. The destination points will be set by the programmer. Different percepts will trigger different responses.

First order of the matter is to provide the AI with virtual boundaries. If the player of the game crosses these boundaries the AI will move towards the player.

To do so, a PawnSensing component must be added to the AI mesh. This allows for the manipulation of the virtual boundaries which comprises of Hearing Threshold, Sight Radius, Peripheral vision angle among others. They can be manipulated to hold any value as long as the scene being created in the game is big enough to accommodate them. The invasion of these boundaries will act as a percept to trigger the AI response.

Example: The player character walks into the line of sight of the AI triggering the AI to 'see' the player and move towards them.

Example 2: The player causes a sound effect while inside the AI's virtual boundaries causing the AI to 'hear' it and move towards the player

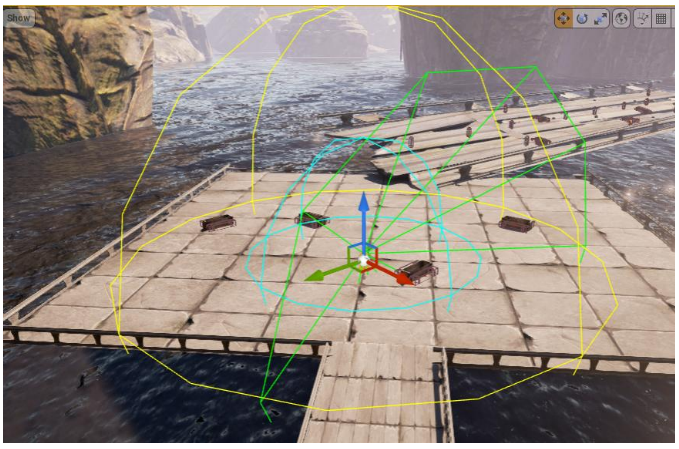


FIG 6: VIRTUAL BOUNDARIES OF THE AI AS DEPICT ED BY THE COLOURED LINES

In order to code the AI movement behavior, a Behavior Tree is used. This tree stores every trigger response to specific percepts. To simply move the AI character towards the player, you get the player co-ordinates in the game world using the 'GetWorldLocation' function and set those coordinates as the destination co-ordinates for the AI. The AI will navigate the game world to reach that point in the shortest distance possible.

By default, the path finding algorithm is already integrated in UE4 so the AI can find its own path and navigate through obstacles but what if the programmer wants to provide a specific path to the AI. A path of their choosing.

To create a custom path for the AI, it is necessary to create specific points on the game map. While these points are going to be visible in developer mode, they will not be visible to the player in the executable file. This paper will refer to these points as 'waypoints'

To create a 'waypoint', a 'billboard' component is needed to be attached to the AI mesh. This billboard can be given visible form by providing it with a sprite. In this experiment the following sprite was used to give the billboard a visible form.

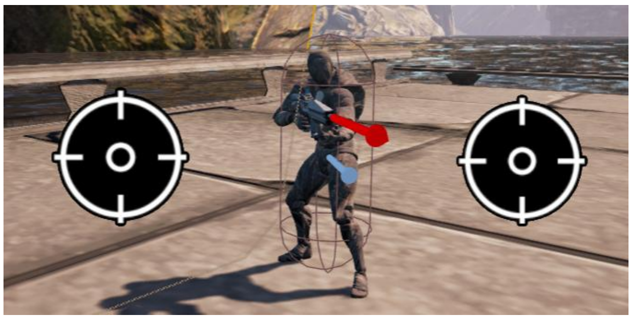


FIG 7: 'TARGET' SPRITE ADDED TO BILLBOARD TO FORM A 'WAYPOINT'

Referencing these waypoints in the code will allow to manipulate how the AI will see them as. For the sake of this paper the AI needs to see them as destination points.

Once the waypoints are created and attached to the mesh, the function 'GetWorldLocation' will once again allow to obtain the coordinates of the placement of the waypoints in the world and after that these coordinates will need to be promoted to a variable. By creating variables of the waypoint coordinates, the location value of the waypoints will always be stored thus making it easier to assign them for use in different functions.

The next step is to set these values as vectors. Simply calling the function of 'Set Blackboard value as Vector' and assigning it, the waypoint variables will accomplish this task. These vector values will then further be cast on the AI Object's blueprint providing them with the aforementioned 'destination coordinates'

The programmer can create several waypoints like this. This experiment was conducted using one, two and three waypoints each attached to a separate AI object mesh.

Once these steps are performed, these waypoints will show up on the viewport in the game world as 'mesh components', whenever the programmer places the AI objects on the map. By clicking on these waypoints or selecting them from AI mesh properties their position in the world can be manipulated using simple translation.

However extreme care must be taken when appointing a destination to the AI using these waypoints. These waypoints must touch a collision mesh on the platform for it to even register as a destination point, otherwise it will fail to register,

as the coordinates received by these waypoints would be pointing towards a destination where traversal is impossible. For this very reason this paper emphasized the importance of appointing collision meshes to the platforms early on.

The figure below shows the correct and incorrect ways of waypoint placements.

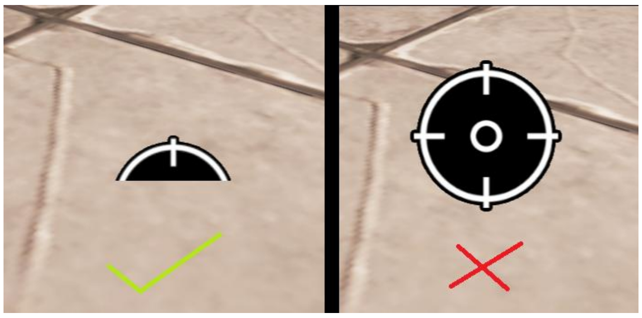


FIG 8: FROM LEFT TO RIGHT, CORRECT AND INCORRECT METHODS OF WAYPOINT PLACEMENT

Once these waypoints are aptly placed, the behavior tree is used to direct the AI towards these points. The process involves calling AI reference and giving it a 'move' function. This 'move' function is a pre-built function that appoints a destination and moves the AI towards it. The destination node is provided with the waypoint variable, and the waypoint itself is placed anywhere on the map and provided the conditions above are met, the AI will move towards the waypoint.

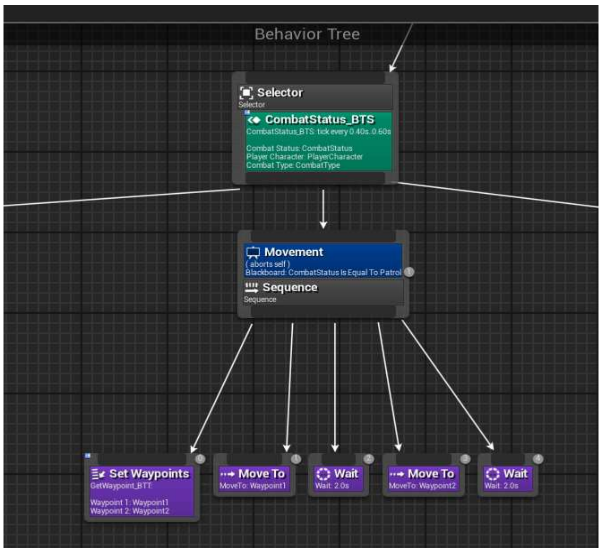


FIG 9: SETTING UP AI MOVEMENT VIA BEHAVIOUR TREE

The above figure shows a sample behavior tree used to move the AI back and forth between 2 way points. The 'move' and 'wait' functions are pre-built functions, while the move functions requires a vector variable as input, the wait variable requires float and integers which here counts as the SI unit 'second' for which the AI will stop moving.

One functionality of the behavior tree is by default; the tree functions will keep on looping. Once the process reaches the end, it will restart once again. So by that logic, what we have in our hands is an infinite loop of the AI moving back and forth between the two waypoints.
Of coursed this is subject to change because at this moment it is performing a very primitive function and a programmer wants their AI to be more diverse and execute more actions like shooting or jumping. But since that would be deviating from the topic of this paper, it will be disregarded.
There are some other factors that need to be taken note of. One of them is to assign a traversable area to the AI. It's all well and good when your AI can walk but there needs to be certain limitation of the traversable area itself. Some areas which are blocked off, some which just exist as 3D models like bodies of water without any collision mesh applied to them. It wouldn't be very realistic after all if the AI starts walking on water or starts to move through blockades into 'danger zones' where there's a substantial risk that the AI might fall of the game map completely.
In order to define a strict area for AI traversal, a component called 'Navigation Mesh Bounds Volume' is used, also known more commonly as 'NavMesh'.
The 'NavMesh' can simply be selected from the Top Down template tab and dragged onto the scene. It is by default a transparent cube. By usage of scaling and translation, one is supposed to make it encapsulate the entire scene or at least the area which one wants the AI to move in. Then, on the viewport, pressing the 'P' key on the keyboard highlights the AI's traversable area. Of course it is subject to change as the programmer wills, but even by default it provides a very accurate calculation of the traversal area.
The area however does not limit the player as the player character's control is dependent on the player himself rather than the AI so in a particular scenario the AI can be 'trapped' by the limitation of the NavMesh while the player is able to move freely even outside the NavMesh bounds.

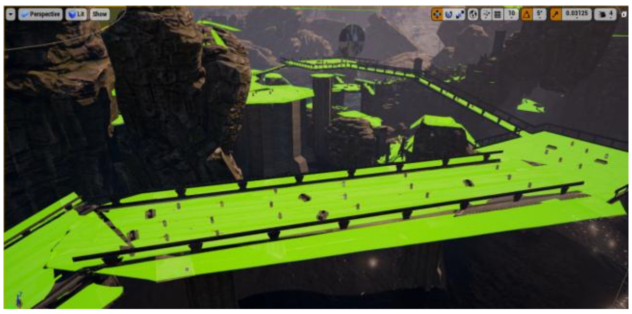


FIG 10: NAVMESH SURROUNDING THE SCENE AND SHOWING THE AI TRAVERSAL AREA HIGHLIGHTED IN GREEN

The above picture provides a graphical view of how the NavMesh surrounding the scene, functions. It automatically filters out objects with collision meshes that are in its boundaries so the AI would avoid them in their pursuit of the waypoints. Thanks to this the AI doesn't collide with static objects and is able to navigate through the green traversable field to find its destination.
There are other instances where the NavMesh turns red, but it is hardly a considerable problem, since it only occurs when you have moved around, one or more objects in the scene so as a result, it recalculates the entire navigational area again, hence the red highlights.
This overall AI navigation method can work in various different ways not just limited to creating waypoints and having the AI move towards it.
Take a specific game scenario where an enemy upon the noticing the player, moves towards the player trying to attack him and wherever the player runs, the AI enemy object follows. To implement such a scenario, instead of obtaining the waypoint coordinates and providing it as a vector destination to the AI, the player's world location co-ordinates are taken instead, via 'GetWorldLocation', then promoted to vector variables and appointed as destination nodes to the AI, this way, instead of moving towards a waypoint, the AI will now move towards the player, following the player wherever he goes.
The above function can also be triggered via a percept. Early on in the paper, virtual boundaries of the AI were explained which can be used as percepts to trigger AI responses. In the experiment performed, during the AI's traversal between two points, if a player invades its boundaries, the AI will abandon its movements towards waypoints and instead follow the player instead, whilst attacking him.
This scenario was also documented in a video to help readers better visualize the context, the link of which will be provided in the 'References' section of this paper [1].

## III. RESULTS AND FINDINGS

As mentioned above this experiment was performed in 3 different virtual environments. Each environment was vastly different in terms of terrain quality, texture and size.
Once the traversal feature had been coded in, an AI object was placed in each environment and it was given 2 waypoints. The distance from the AI to each waypoint was kept constant, as was the distance between the 2 waypoints. The experiment was to check whether the AI would travel from its initial position to Waypoint A and from there to Waypoint B then continue moving back and forth between the two points.
Below are the results shown for each of the 3 environments that were used (pictures of the environments can be found above)

CONSTANTS:

1) AI Object (UE4 default mannequin)
2) Straight-Line Distance between AI object origin and Waypoint A (variable value integer: 200)
3) Straight-Line Distance between Waypoint A and Waypoint B (variable value integer: 200)

4) AI movement speed (variable value: 500)

RESULTS:

TESTING ENVIRONMENT A:

| | |
|---|---|
| AI traversal from origin to point A | Success |
| AI traversal from point A to point B | Success |

TESTING ENVIRONMENT B:

| | |
|---|---|
| AI traversal from origin to point A | Success |
| AI traversal from point A to point B | Success |

TESTING ENVIRONMENT C:

| | |
|---|---|
| AI traversal from origin to point A | Success |
| AI traversal from point A to point B | Success |

While the AI movements were largely similar to the proposed hypothesis, there was some discrepancy in Environment C since unlike A and B which were vastly open areas, Environment C was more similar to a maze, so the AI had to navigate between different paths (using NavMesh to identify traversable platforms) to reach the goal. While the straight line distance between the AI and each of the waypoints was kept constant it was the narrow maze-like environment itself that turned out to be an obstacle thus delaying the AI to reach the waypoints.

## IV. CONCLUSION

Using Waypoints, one can easily give as many destination points to their AI objects and the procedure is so simple that any unexperienced programmer can use it to create small fun games on their own.

Pathfinding have always been associated with AI modules and implementing it in a virtual world is a fundamental aspect in video game development.

Waypoint creation and coordinate assignment makes the task much easier and allow developers to save time rather than look for complicated algorithms to get their AI navigation perfect.

Of course, waypoint creation is one of the many ways to assign navigation to AI but compared to other processes it is relatively simple and easy to manage and integrate.

## REFERENCES

[1] Demonstration Video to document the procedure: https://www.youtube.com/watch?v=xF7WMBtn2uA&index=11&list=PL5o5c7sLA1Cg-JbvKfePnvIjMX8lHd0SX

[2] Kevin Smith and Anderson, “Blue print technology in Unreal Engine, 2015.”

[3] Greg Penninck “Modularity for Next Gen Games Designers”

[4] Hans Ferchland “Game UI Prototyping with Unreal Engine 4”

[5] Damien Vurpillot “Aspectus operis and visual attention: from Vitruvius to Virtual Reality”

[6] Rowan Wilson “Visualizing research data in 3D with Blender”

[7] Ashwin Ram, Santiago Ontanon, and Manish Mehta “Artificial Intelligence for Adaptive Computer Games”

[8] Firas Safadi, Raphael Fonteneau, and Damien Ernst “Artificial Intelligence in Video Games: Towards a Unified Framework”

[9] Daniel Johnson and Janet Wiles “Computer Games with Intelligence”

[10] Simon M. Lucas and Diego Perez-Liebana “General Video Game AI: Learning from Screen Capture”

[11] Aliza Gold “Academic AI and Video games: a case study of incorporating innovative academic research into a video game prototype”