A REPORT
ON

INDOOR LOCALIZATION USING WiFi BASED NETWORKS

By

Name of the Student:

Vaibhav Chaudhari (**2017B5A70834G**): f20170834@goa.bits-pilani.ac.in
Devansh Aggarwal (**2018A7PS0131G**): f20180131@goa.bits-pilani.ac.in

Prepared under the fulfilment of Course CS F266/F372

Birla Institute of Technology & Science, Pilani
Goa Campus

Acknowledgments

We are very thankful to Dr. Vinayak Naik (HOD, Department of Computer Sciences and Information Systems, BITS Goa) for guiding us through this project which has led us to get a wider understanding of how Indoor Localization can be used for multiple purposes.

We also thank him for helping us and giving advice on this project without which this project could not have been completed.

Table of Contents

Problem Statement

With the boom in the field of augmented reality, social networking and the retail shopping applications which can all be easily manages by our handheld devices, there is a need for an accurate and fast location technology which will be able to give us a better user experience. We all know that **the Global Positioning System (GPS)** is a very reliable source for getting an accurate location outdoors but there is a need for indoor localization also which is not achieved with accuracy using the GPS technology. Therefore, we need some idea on how to accurate locate one's location indoors where the GPS signal is not that reliable.

We can approach this problem by using **localization techniques** which use the **WiFi networks strengths** inside the building to accurately predict the location of one's cellphone which is connected to the access points. The most common and widespread localization technique used for positioning with wireless access points is based on measuring the intensity of the received signal (received signal strength indication or RSSI) and the method of **fingerprinting**.

The proposed solution performs the indoor localization in which we can guide/track a user with a smartphone using the WiFi based networks by using the Received Signal Strength (RSS) and also predicting the motion of the user with respect to the Access Points in real-time using a Unity 3D application for smartphones. We will be developing an android based application which will use a routing algorithm and the WiFi signal strengths inside the building and guide the user to his/her destination. The main functionality of this is that it can be used to find the shortest distance between two locations inside the building. This will provide a real-world use of the WiFi signal strengths for the localization indoors.

The main obstacle of this project is the limited computational capacity and the storage size of the smartphones. We will need to make out project very space and computational inexpensive so that this can be used for a normal smartphone. Then there is also a case when we are on a particular floor in the building and we want to go to a place on a different floor inside the building.

1) Related Work

We took inspiration and ideas from the work of Mr. Dhairya Parikh and Mr. Nand Parikh, two Higher Degree Students of BITS Pilani K.K. Birla Goa Campus. Their work has helped us a lot in understanding the need for such a solution and also the way to approach such a problem. Their problem statement was very similar to ours and also used the same techniques and ideology to find a solution. They proposed building a Unity 3D application for the indoor localization. They created a 3D model of the new Computer Science Building at BITS Pilani K.K. Birla Goa Campus and used the A Star Routing Algorithm along with RSS to find the shortest paths between different sections of the building. They also analyzed the computational power used by their application in order to minimize the load put on the smartphone.

SOURCE CODE:
https://drive.google.com/drive/u/1/folders/1jdb7CXZL9CUMaZAGAsEdoFkmQVKYO568

In order to understand the internal working of the A Star Routing algorithm that is being used in this project, we used the articles published by Mr. Amit Patel from Stanford University. He has researched in great detail about the working of the Routing Algorithms and their heuristics specially the most talked A Star Routing algorithm. He has also explained about the different variations of the A Star Algorithm that can be used in specific conditions. The article can be found here.

2) Approach

The approach opted by us in this project was to first test out the Algorithm that will be used for the Routing inside the buildings in a demo 3-D building which will be created by us in Unity 3D. So the reason of us choosing to work on the Routing Algorithm first was due to the fact that we wanted to perfect the working of the Algorithm inside a building by eliminating the edge cases and the bottlenecks we might face in the actual scenario when our project will be deployed on a larger scale. Once we perfect this, we will start with the WiFi signals and the RSS part and try to integrate it with the already created Routing Algorithm.

In order to test the working of the Algorithm, we first had to read about the working of the algorithm. We initially began with our own implementation of A-Star Algorithm in a simple Unity project. We referenced the algorithm's code from here. Later, we got to know that there was already an implementation of the A Star Algorithm present in Unity by the name of NavMesh. We decided to use this implementation to test whether we were able to find the shortest path between the start and the end points in a 3-D structure.

In order to test out the algorithm, we planned to make a 3D maze structure and then used this maze to find whether the NavMesh was able to find the shortest path between two points. Since we were not able to make a complex and aesthetic Mesh in Unity 3D, we decided to make the Maze on an open-source 3D Computer Graphics Software known as **Blender**. This enabled us to make complex meshes with different features. All these assets created were then imported to Unity 3D and were used to test the working of the algorithm.

3) Solution and Its Evaluation

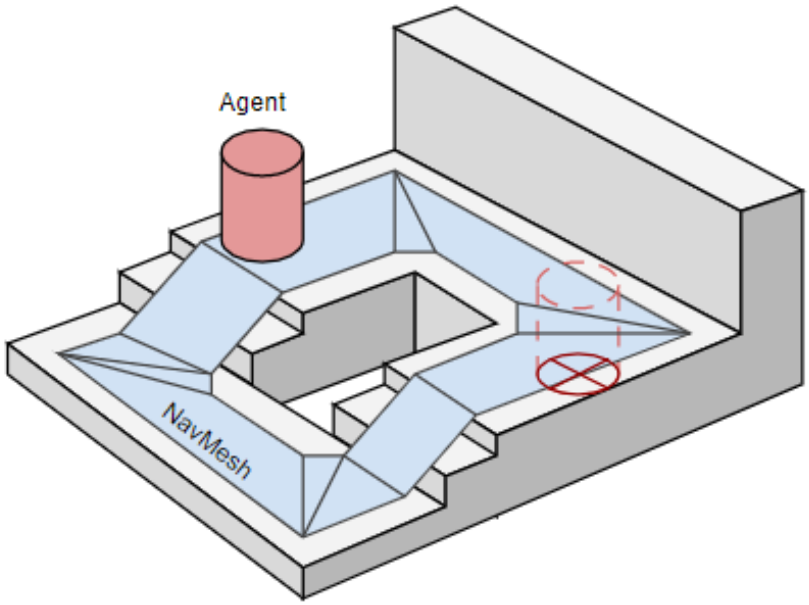
3.1) NavMesh

3.1.1) **Introduction**

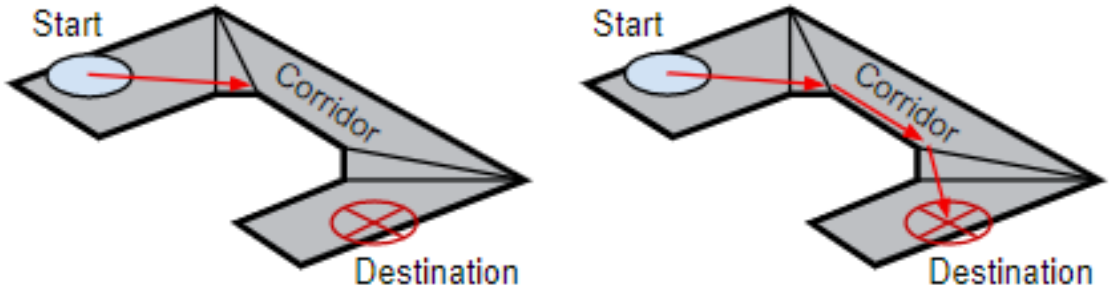
The NavMesh is a class that can be used to do spatial queries, like pathfinding and walkability tests, set the pathfinding cost for specific area types, and to tweak the global behavior of pathfinding and avoidance. It is implemented inside **UnityEngine.AIModule** package and it internally uses **A* Algorithm**.

3.1.2) **Working**

Unity navigation representation is mesh of polygons. First thing it does is to place a point on each of the polygons which basically represents the location of that particular node and then it finds the shortest path between different nodes. For NavMesh to work as desired, the navigation system needs its own data to represent the walkable areas in a game scene. The walkable areas define the places in the scene where the agent can stand and move and combined together these areas form a continuous surface where the agent can move. This surface is known as **Navigation Mesh** and these surfaces are stored as a combination of convex polygons.

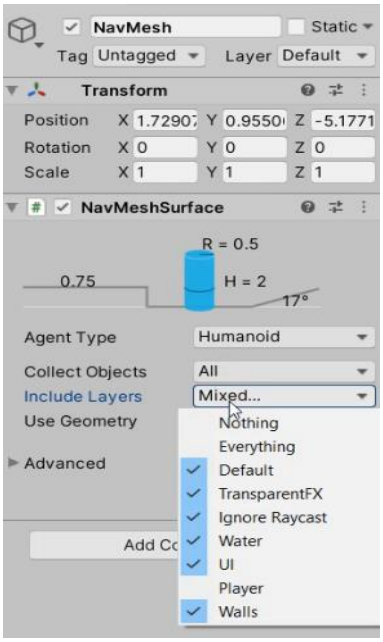


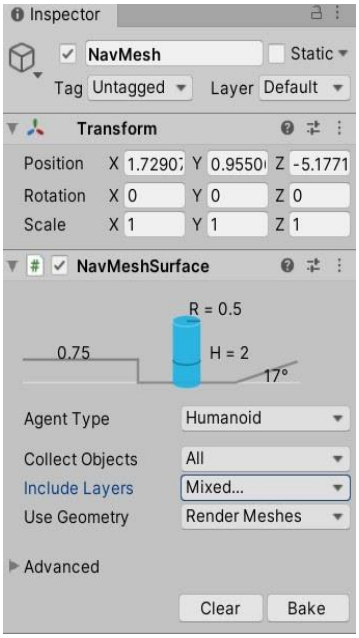
Now, after defining the walkable area as a set of polygons better known as the ***corridor***, the agent needs to be moved from the start to the destination. The agent simply will reach the destination by always steering towards the next visible/nearest corner of the corridor.



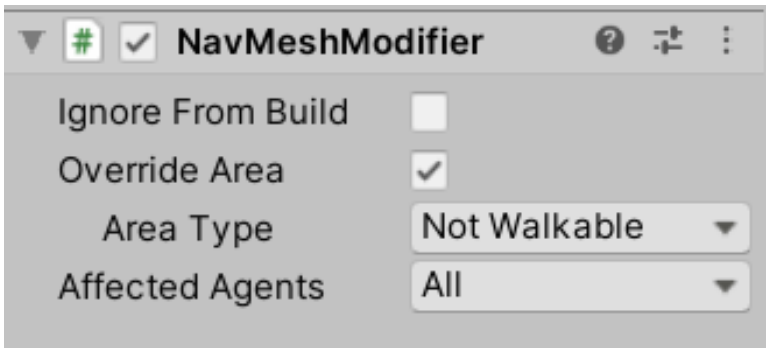
3.1.3) **Setting Up Navigation Map**

Setting up the navigation map involves creating a navigation surface game entity and deciding upon the map parameters like *walkable* and *non-walkable* surfaces, layers to be included while baking the mesh, and agent's maximum walkable inclination angle and step height.
The below screenshots show how we set up our NavMesh surface object, and while doing so we excluded the ***Player*** layer from being taken into consideration at all for baking. This was done to ignore the agent itself to be taken into consideration while baking the mesh.

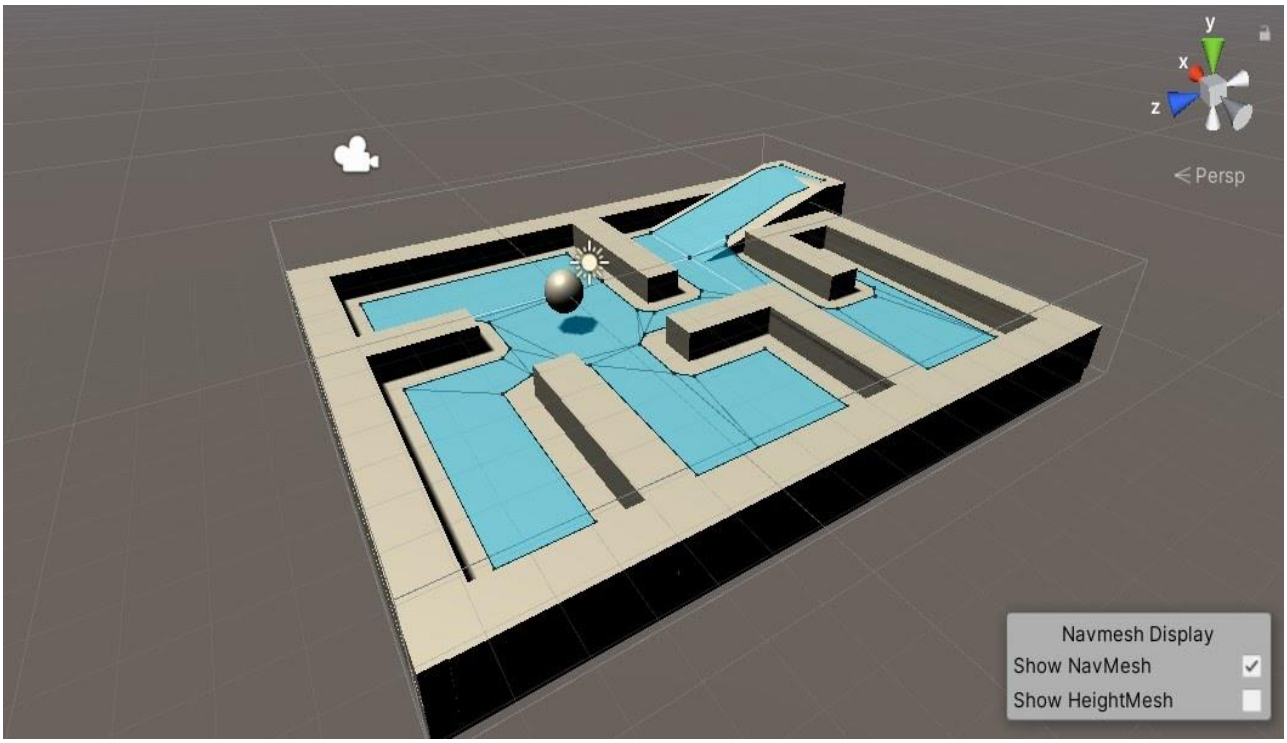




The game objects individually can be specified as either walkable or non-walkable and this can be done by adding the **NavMeshModifier** component to that object and specifying the either type.



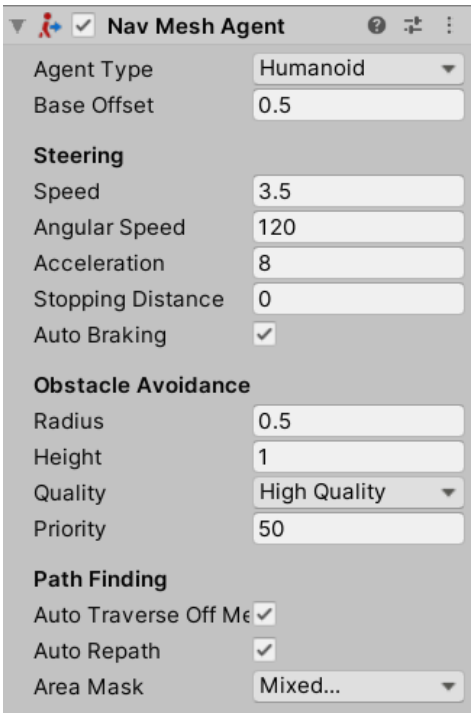
After setting up the game scene and baking the navigation mesh, this is the resultant mesh.



3.1.4) **Moving the Navigation Agent**

The actor in our game scene is a simple 3D sphere game object. For the object to move along the decided path, the **Nav Mesh Agent** component was added to it and a custom script to make it respond to user clicks (left mouse click in desktops & simple touch in Android devices).

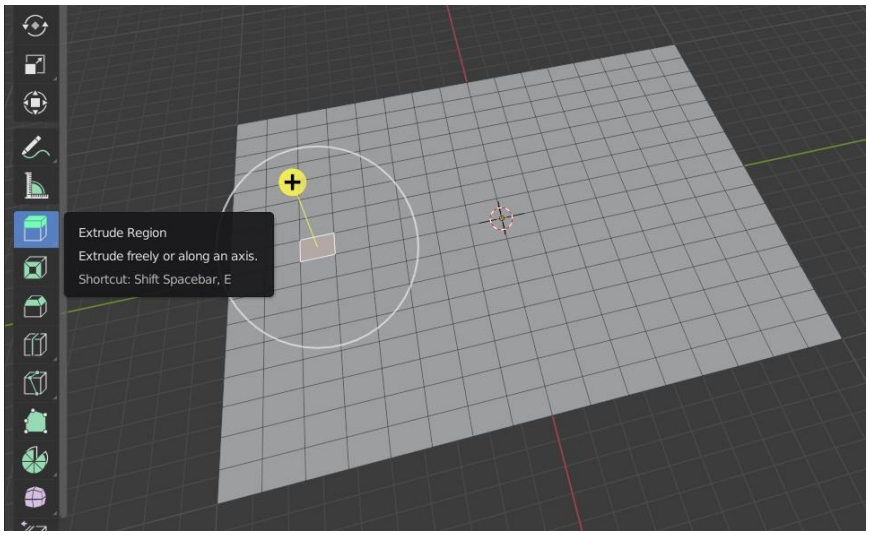
Inside the Nav Mesh Agent component, we can specify the agent's speed, stopping distance from the destination, and obstacle avoidance radius, and other layers to consider for moving the agent.



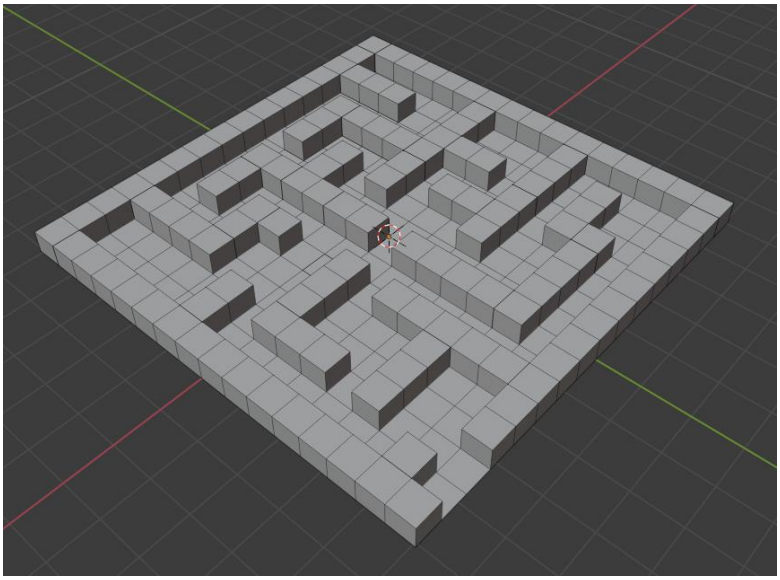
3.2) Blender

In order to make our Algorithm testing rigorous, we made the use of Blender. Blender is the free and open source 3D creation suite. It supports the entirety of the 3D pipeline—modeling, rigging, animation, simulation, rendering, compositing and motion tracking, video editing and 2D animation pipeline.

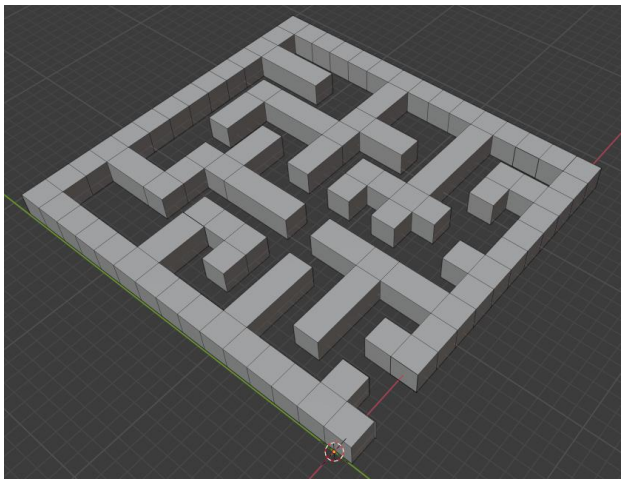
In order to make the 3D models of the maze, we first tried to use a normal Mesh in Blender to make the maze by subdividing it into smaller grids so each grid can represent a block on which either the path or the walls of the maze will be made.

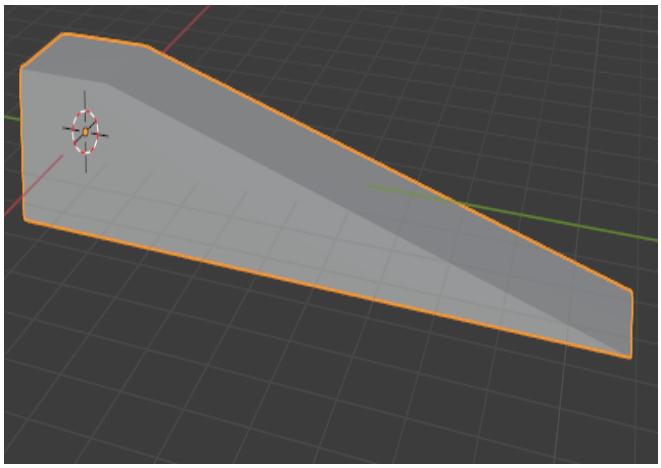


The resulting maze looked like this:

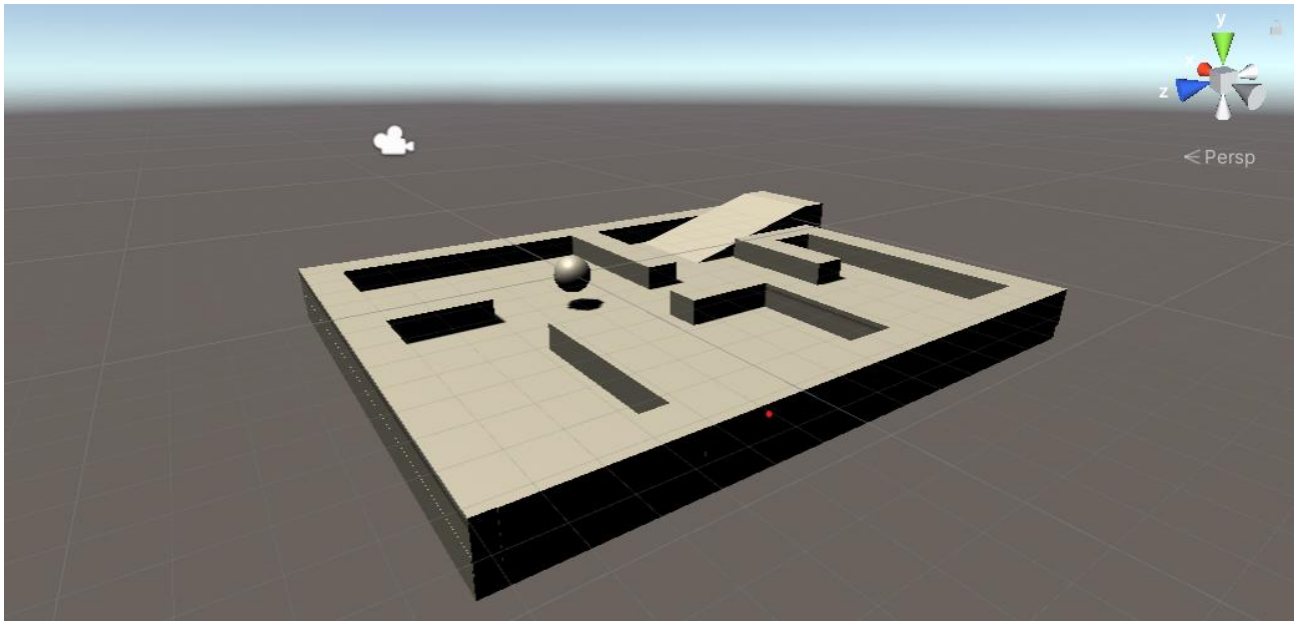


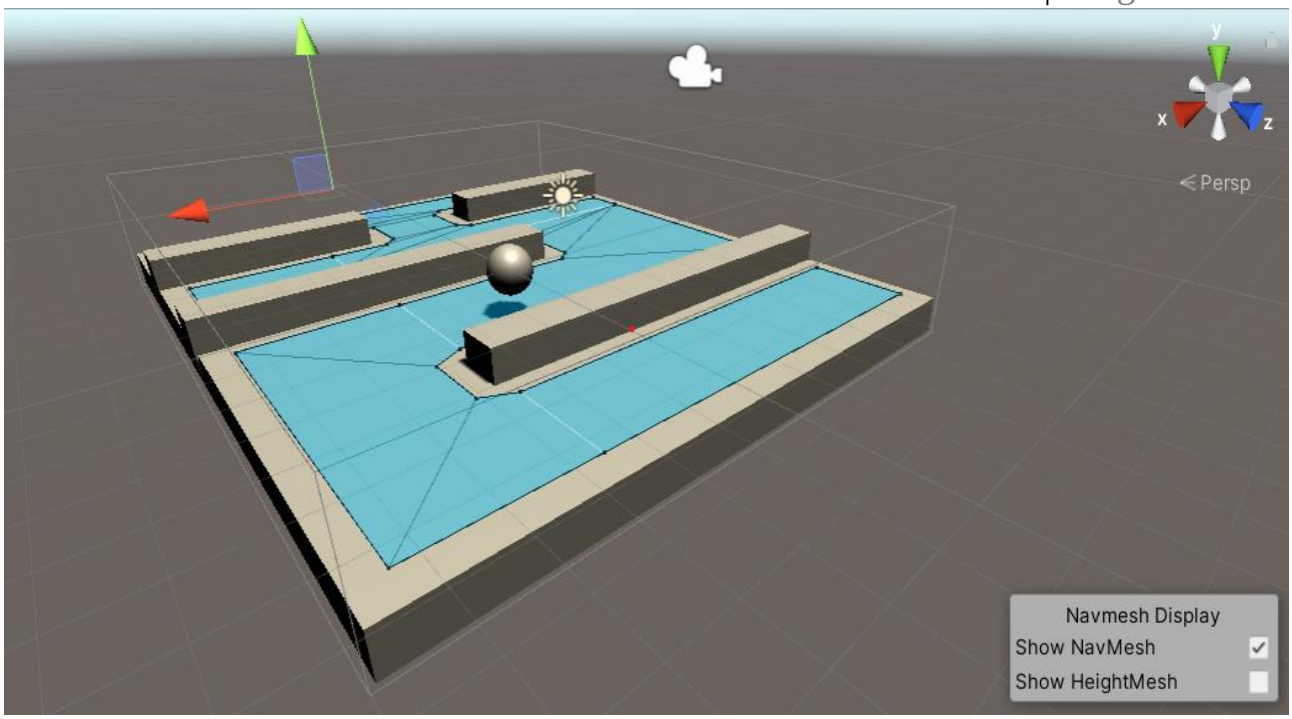
NavMesh classifies the paths as walkable and non-walkable to find the obstacles and the available paths present in the scenario. But the problem with this asset was when we were importing this maze in Unity 3D, it was being imported as a single entity instead of a collection of grids in a Mesh. Because of this reason, we were not able to mark the walkable and non-walkable paths for the NavMesh Implementation to work. We decided to make only the walls of the maze and import it as a single entity which was non-walkable which could be then placed on a walkable mesh. We thus made the maze using the cubes in blender. We also made the slope element which could be thought of as a stairway in a building.





The final model in Unity 3D looked something like this:





3.3) Is Optimization Needed?

3.3.1) **NavMesh Precomputation Problem**

When we were working with the NavMesh in Unity 3D, we came across the question of whether the computation of the path in the scenario happens at the time of running or at the time of baking the Mesh. This would have a significant outcome on the performance when running the application on smartphones. And since our goal is to make the Unity Application as inexpensive as possible in terms of computation due to the limited computational capabilities of the smartphone, we needed to find the answer to this question so we can take appropriate measures in order to maximize efficiency.

In order to analyze whether the NavMesh precomputes the shortest path between the two points, we first tried to see whether there was a time difference when we ran the algorithm inn a smaller maze as compared to a bigger maze but the thing was that in order to observe a significant difference in the time of execution of the application, the size of the maze or the area in which the NavMesh had to bake and identify the polygons had to be very very large as compared to the normal maze that we had.

So we tried to got through the documentation of the NavMesh to get a closer look at the inner working of this A Star implementation. We came across a very interesting function for the NavMeshAgent which clearly states that upon the updating the destination, the NavMeshAgent triggers the calculation of a new

path during the runtime of the application which clearly tells us that the NavMesh does not do the precomputation of the distances between two points, rather it computes the path dynamically when the application is actually running on the smartphone.

NavMeshAgent.SetDestination

Leave feedback

public bool **SetDestination**(Vector3 **target**);

Parameters

target	The target point to navigate to.

Returns

bool True if the destination was requested successfully, otherwise false.

Description

Sets or updates the destination thus triggering the calculation for a new path.

Note that the path may not become available until after a few frames later. While the path is being computed, pathPending will be true. If a valid path becomes available then the agent will resume movement.

We came to the conclusion that the NavMesh bakes the scenario before we start the execution of the application. This pre-execution baking helps the NavMesh to identify the areas where the NavMeshAgent will be able to move around and break them into polygons and also determines the non-walkable areas which the agent has to avoid.

The actual calculation for the shortest path between two points is being done as soon as the destination point is selected. Since each polygon is considered as a node in a graph, the problem reduces to finding the shortest path between two nodes of a graph.

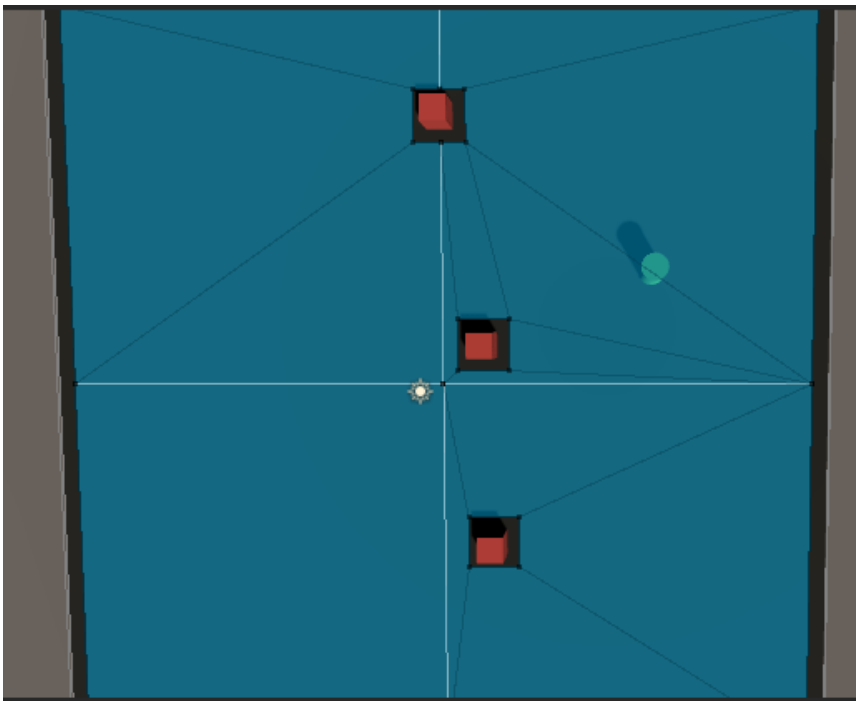


Fig: Statically Baked NavMesh with Red Cube Obstacles and different polygons

3.3.2) Dynamic Mesh Baking Possible?

The baking of the Mesh before the execution was a plus point for us as we did not need to invest the computational power in baking the Mesh at the time of execution. But this created a problem in terms of storage in the smartphone. If we are already baking the NavMesh for the entire model, then we will have to store that information somewhere so it can be later used to computer the path.

If the obstacles were also considered to be moving then the baking of the NavMesh will happen dynamically during the execution of our application in order to adjust the walkable areas and the sizes of the polygons with respect to the moving objects. This will be a bit more computationally expensive as compared to the case where the obstacles were stationary and the NavMesh was being baked statically before the start of execution of our program. We can see the shapes of the polygons changing in the following images because of the movement of obstacles as our NavMeshAgent will need the latest configuration of the objects within the area.

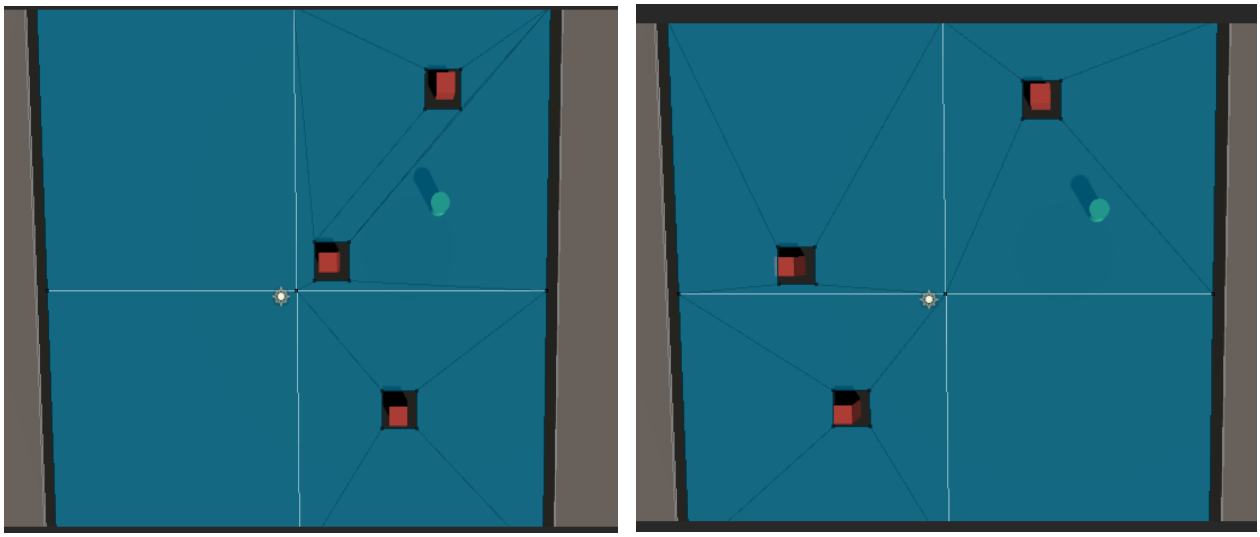


Fig: Dynamically Baked NavMesh with Moving Red Cube Obstacles. Size of the polygons can be seen changing.

If our model if very large, the NavMesh will be baked over a large region resulting in the increase in size of data to be stored. Since smartphones don't have a lot of storage capacity, we need to come up with a way of reducing this storage space and still managing to get the best out of our model. We thought of an idea of baking the Mesh of only that room in which are NavMeshAgent is currently in and leave the other rooms/areas of the model unbaked so as to reduce the size of the project. When we want to go to a different place in the model where the NavMesh is not baked yet, we will send a signal to bake the Mesh of that particular region as soon as we select it as out destination. This will result in the dynamic baking of the NavMesh of only those regions which are the regions of our interest. This will reduce the storage problem at the cost of a bit of computational expensiveness. We will try to implement this approach in the future to get a clear understanding of what can be done for further optimization.

4) Source Code

Below is the code for the NavMeshAgent so that it can move to the destination. We have taken the input in the form of a mouse click which tells us the destination we have to reach. We then set the destination of the click and compute the shortest path from the starting point.

```
// Update is called once per frame
void Update() {
    /*
        0 -> for Left Mouse button
        1 -> for Right Mouse button
    */
    if (Input.GetMouseButtonDown(0)) {
        Debug.Log("Mouse Clicked");
        // to make player react to user clicks
        Ray ray = mCamera.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;

        if(Physics.Raycast(ray, out hit)) {
            Debug.Log("Valid");
            // set agent's destination
            player.SetDestination(hit.point);
        }
    }
}
```

Below is the script that was used for making the obstacles move during the execution of the program for inspecting the dynamically baked NavMesh.

ObstacleAnimation.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ObstacleAnimation : MonoBehaviour {

    public float speed = .2f;
    public float strength = 9f;

    private float randomOffset;

    // Use this for initialization
    void Start () {
        randomOffset = Random.Range(0f, 2f);
    }

    // Update is called once per frame
    void Update () {
        Vector3 pos = transform.position;
        pos.x = Mathf.Sin(Time.time * speed + randomOffset) * strength;
        transform.position = pos;
    }
}
```

We also experimented with the actual implementation of the A Star Algorithm and tested it before moving on to the NavMesh in Unity 3D. Below is the code for the same. This code which was used is present at https://pastebin.pl/view/cb09168d and was taken from here.

5) Conclusion

In our project, we have focused on developing resource efficient Android application using Unity 3D game engine which determines the shortest path to a destination from the user's current location using Unity's NavMesh components which internally uses A-Star Algorithm. Our project can be extensively used inside universities, large hospitals to efficiently commute from one location to another in case of emergencies. We have tested Unity's implementation of A-Star Algorithm**:**
NavMesh, under different circumstances and based on our testing we have decided upon the final approach to handle the multi floor use case.

All the assets used and the resources can be found here.

6) Future Work

After having Unity' NavMesh tested under different use case scenarios, the future steps for us in achieving our goal is to work on developing a Unity project consisting of multiple floors each with a Navigation mesh baked **only when needed**. For this, we are planning on using a drop-down menu which asks the user to select the floor and after selecting the floor only we shall bake the Navigation mesh for that floor. We came up with this enhancement after being guided by Vinayak sir during our recent meeting. The main reasoning behind doing this is to reduce the application size and reduce the main memory usage.

One of the major tasks of this project was to measure and profile the load on mobile devices while adopting different approaches. After having tested NavMesh under varied scenarios, the next task for us to prepare the detailed report of resource consumption in all these cases.

Last but not the least, in the upcoming days, we shall also focus on WiFi network's strength-based path finding. At the moment, we haven't completely figured out how to proceed with this and hence may need some guidance.

7) References

1. Indoor Localization Using WiFi Based Networks, Dhairya Parikh and Nand Parikh(2019), https://drive.google.com/drive/u/1/folders/1Q9rWfIanFXMtNTG9aTAgy_A-oRRIRv9H
2. A Star Pages, Red Blob Games, Amit Patel, http://theory.stanford.edu/~amitp/GameProgramming/
3. Unity, https://unity.com/learn/get-started
4. Github, https://github.com/LifeIsGoodMI/Unity-A-Star
5. Github, https://github.com/Brackeys/NavMesh-Tutorial
6. Documentation Unity, https://docs.unity3d.com/ScriptReference/AI.NavMeshAgent.SetDestination.html
7. Maze Creation Blender, https://www.youtube.com/watch?v=09ANJzy4dQY&ab_channel=AtMind