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.

We would like to thank Mr. Yogesh Dasgaonkar for helping us by providing the model for the building to implement on and to also help us on each and every step of our project.

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. We have also implemented the NavMesh on the building structure and have tested various possibilities of doing so like dynamic baking, baking on click, improving the accuracy of the baked NavMesh etc.

3) Solution and Its Evaluation

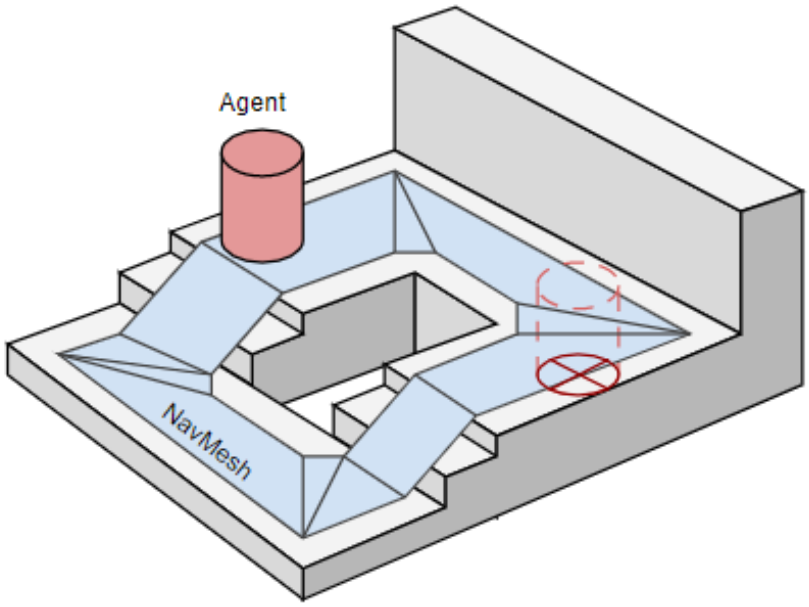
3.1) NavMesh

3.1.1) Introduction

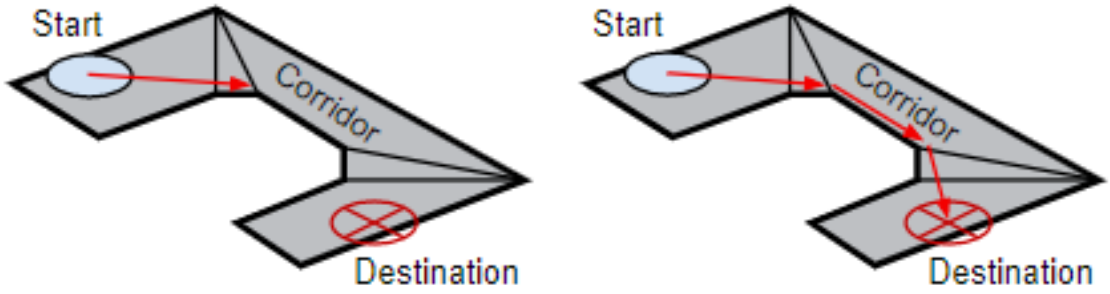
The NavMesh is a class that can be used to do spatial queries, like pathfinding and walkability tests, set the pathfinding cost for specific area types, and to tweak the global behavior of pathfinding and avoidance. It is implemented inside **UnityEngine.AIModule** package and it internally uses **A*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

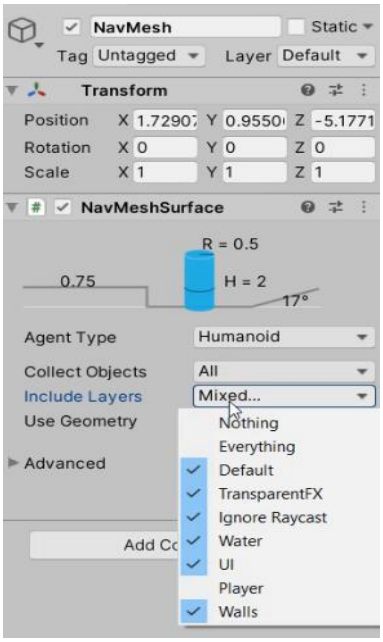


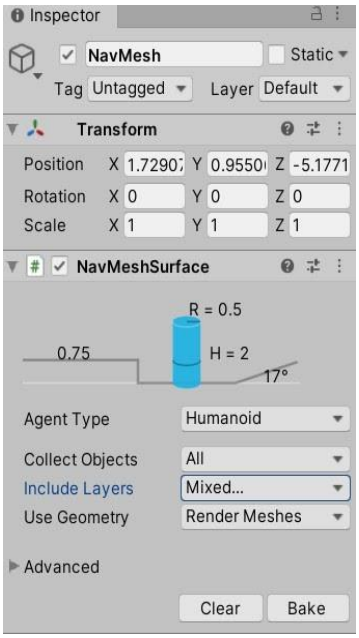
Now, after defining the walkable area as a set of polygons better known as the ***corridor***, the agent needs to be moved from the start to the destination. The agent simply will reach the destination by always steering towards the next visible/nearest corner of the corridor.



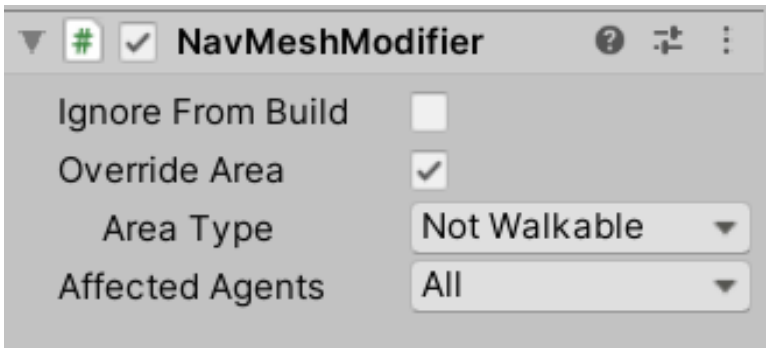
3.1.3) **Setting Up Navigation Map**

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.

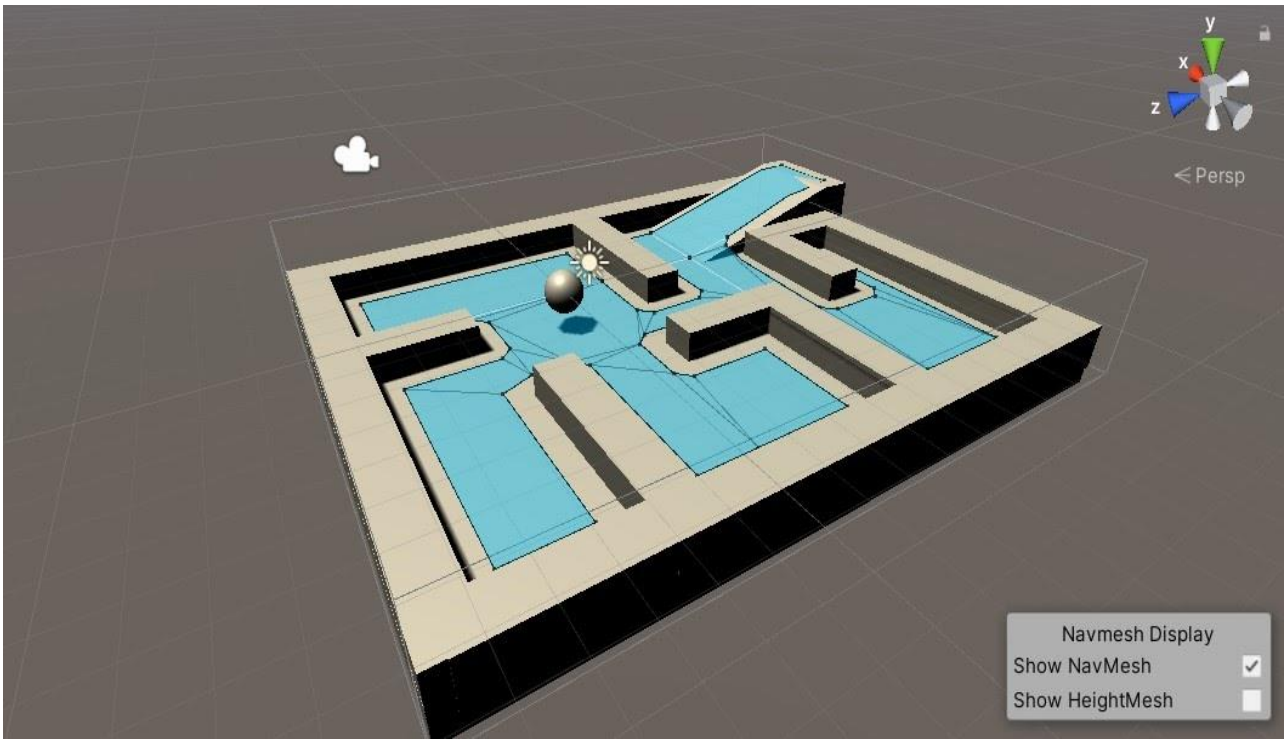




The game objects individually can be specified as either walkable or non-walkable and this can be done by adding the **NavMeshModifier** component to that object and specifying the either type.



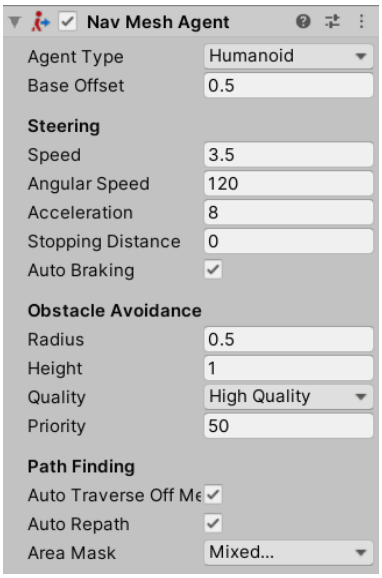
After setting up the game scene and baking the navigation mesh, this is the resultant mesh.



3.1.4) Moving the Navigation Agent

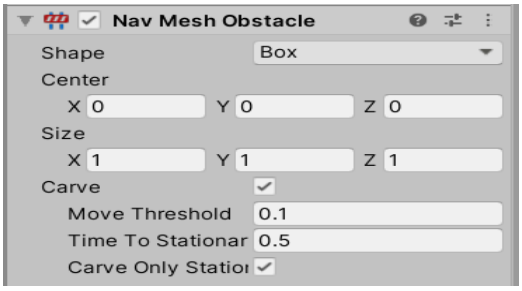
The actor in our game scene is a simple 3D sphere game object. For the object to move along the decided path, the **Nav Mesh Agent** component was added to it and a custom script to make it respond to user clicks (left mouse click in desktops & simple touch in Android devices).

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.



3.1.5) NavMesh Obstacle

The **NavMesh Obstacle** component allows developers to describe obstacles that NavMesh Agents should avoid while navigating the mesh. To use NavMesh Obstacle, simply add this as a component on the game object which is to be treated as an obstacle.



Carve

The obstacle component also provides an option "**carve**" which basically decides whether to ignore the obstacle (*when unticked*) while baking the mesh, or to exclude it right-ahead while baking (*when ticked*).

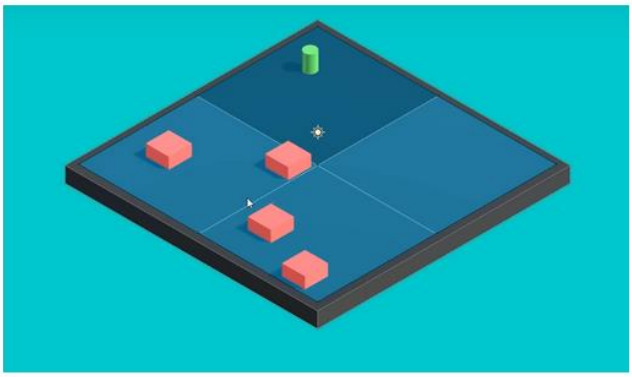


Fig. When "carve" is not being used

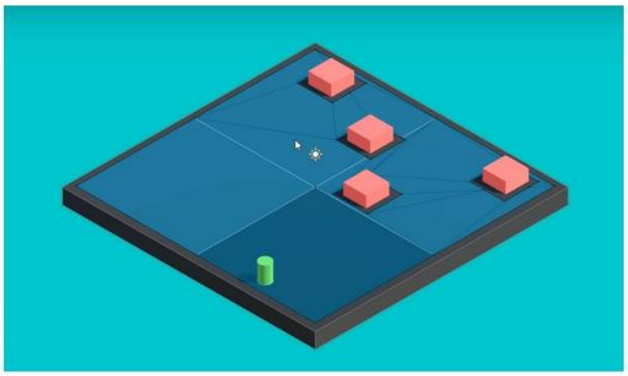


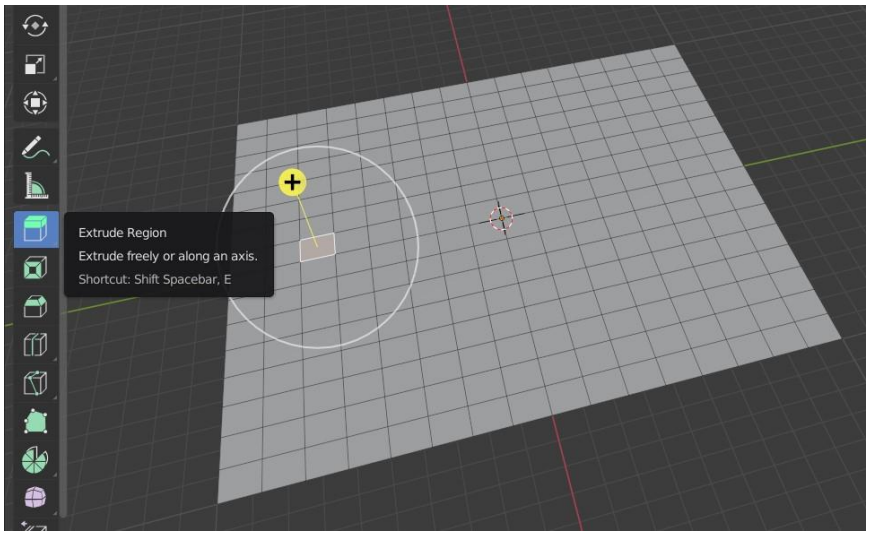
Fig. When "carve" is being used

In the first case, the agent moves on the mesh as usual and only when it comes near to the obstacle, it makes an effort to avoid it. This approach is generally not recommended as the agent may not be always able to avoid the obstacle and this may lead to unsatisfactory results but on the other hand simply ignoring the obstacles while baking also makes it easy/efficient to bake the mesh as now while baking Unity doesn't have to deal with excluding the obstacles. Hence for fairly large models, one may opt to not use the "carve" option.

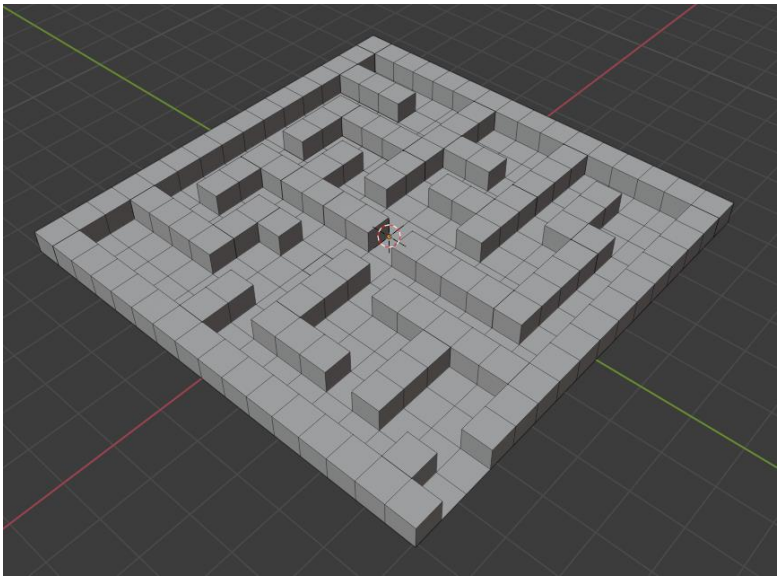
3.2) Blender

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.

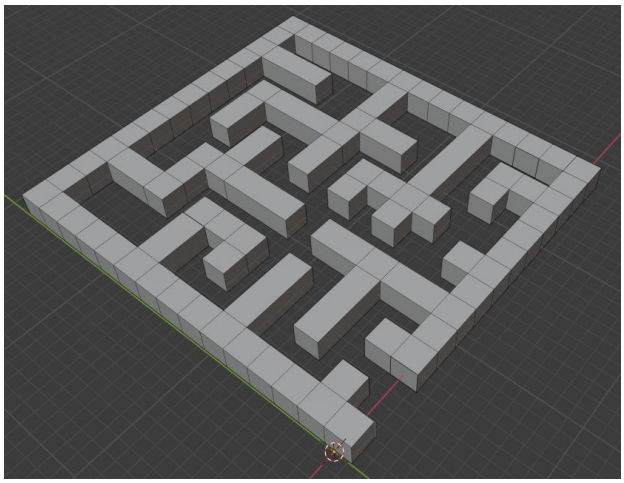
In order to make the 3D models of the maze, we first tried to use a normal Mesh in Blender to make the maze by subdividing it into smaller grids so each grid can represent a block on which either the path or the walls of the maze will be made.

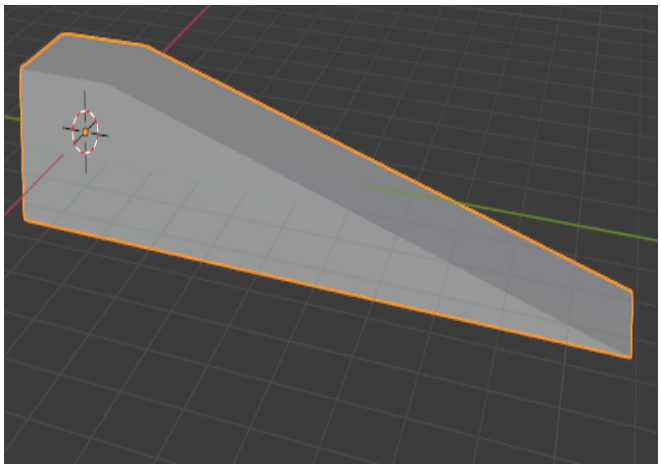


The resulting maze looked like this:

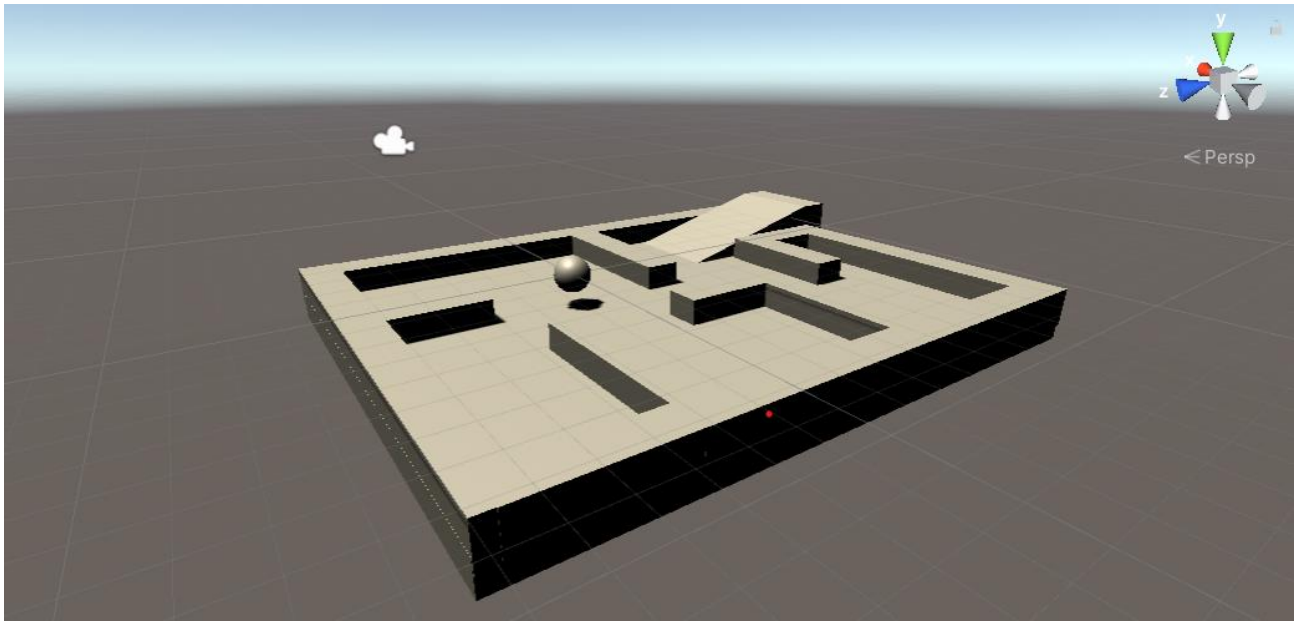


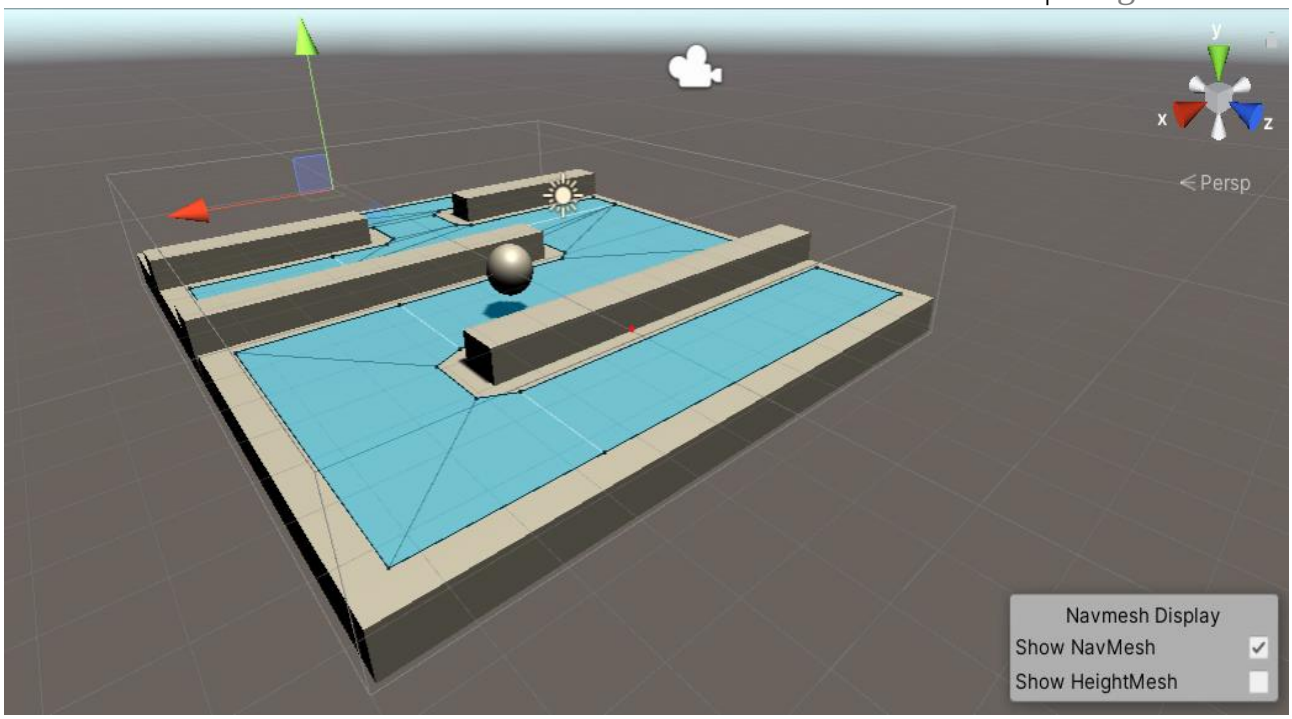
NavMesh classifies the paths as walkable and non-walkable to find the obstacles and the available paths present in the scenario. But the problem with this asset was when we were importing this maze in Unity 3D, it was being imported as a single entity instead of a collection of grids in a Mesh. Because 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.

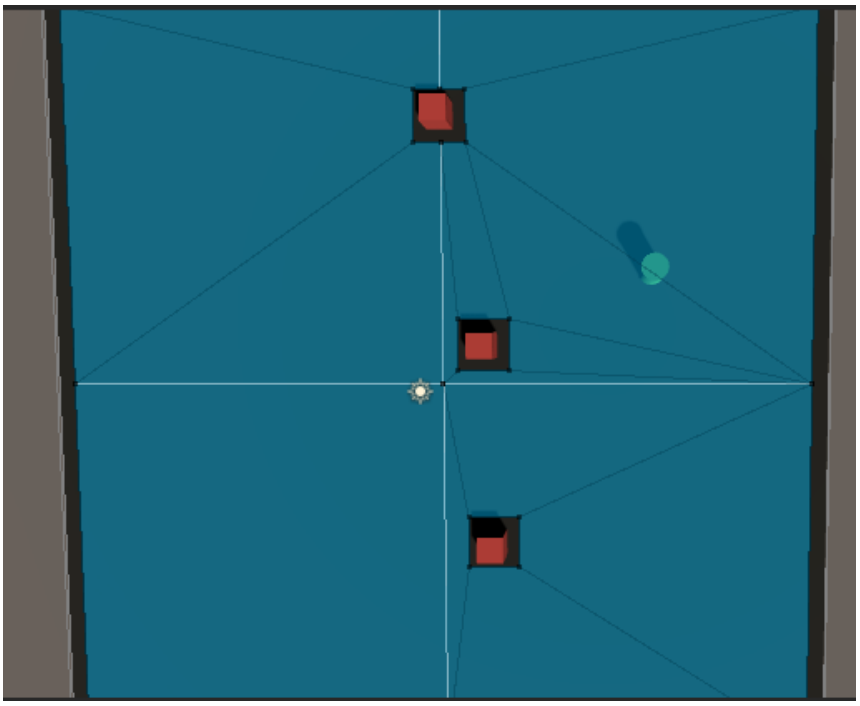


Fig: Statically Baked NavMesh with Red Cube Obstacles and different polygons

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 another problem in terms of the ram of the smartphone. If we are already baking the NavMesh for the entire model, then we will have to store that information somewhere which first will increase the size of the overall application (*not our primary concern*) but also needs to be loaded into the ram as soon as the game starts. For bigger models, loading all the baked information at once may use up an excessive amount of the device's memory and can cause unexpected behaviors.

If the obstacles were 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.

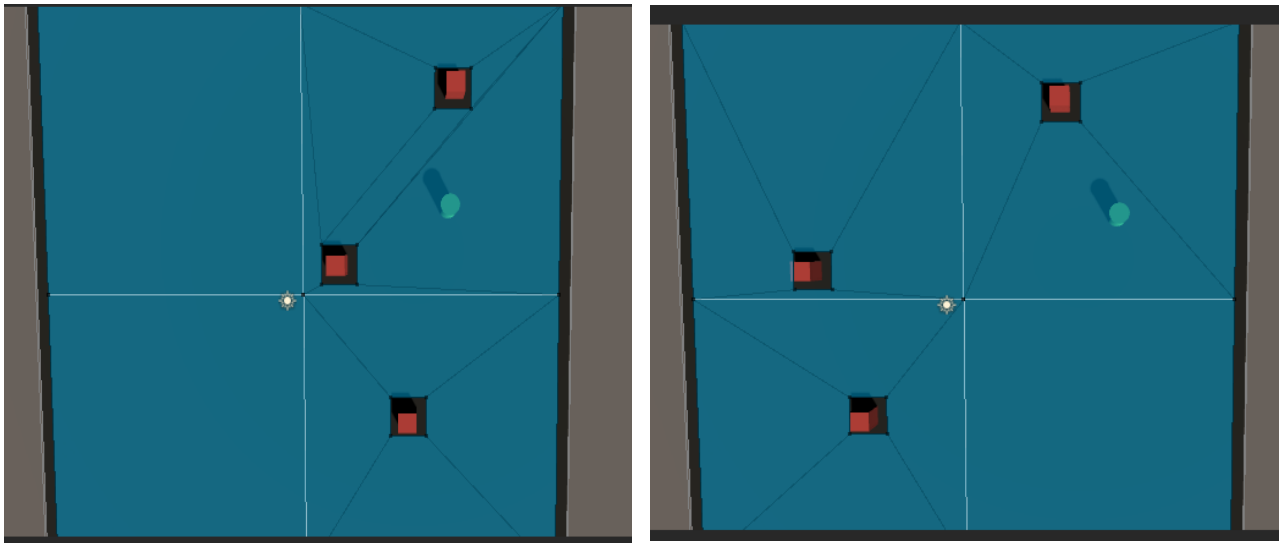


Fig: Dynamically Baked NavMesh with Moving Red Cube Obstacles. Size of the polygons can be seen changing.

If our model is very large, the NavMesh will be baked over a large region resulting in the increase in size of data to be stored. Since smartphones have limited computation power and memory, we need to come up with a way of reducing this data and still managing to get the best out of our model. We thought of an idea of baking the Mesh of only that floor in which NavMeshAgent is currently in and leave the other floors of the model unbaked so as to reduce the size of the data to be processed and loaded into

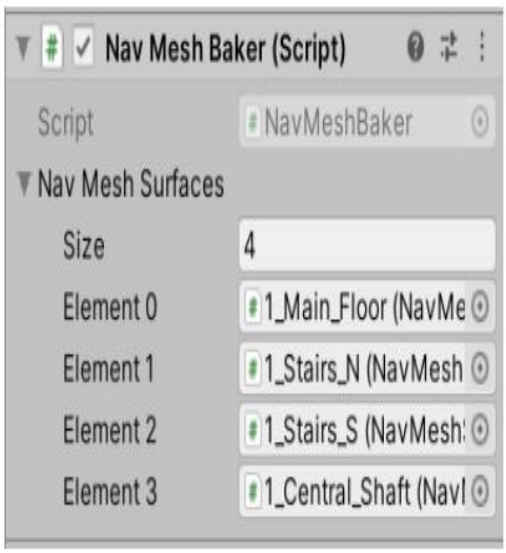
the memory at a time. When we want to go to a different floor in the model where the NavMesh is not baked yet, we will simply call our method to bake the Mesh of that particular region by simply modifying the NavMeshSurface array that we are using (*explained later*). This will result in the dynamic 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.

3.4) Dynamic Baking?

As mentioned earlier, the main idea behind baking the mesh dynamically was to take advantage of the fact that we can break a huge model into small chunks of objects and bake these at a time only when needed. This approach initially seemed to be very appealing as our primary target for this project are mobile devices which with their limited computation power and main memory may show unexpected behaviour due to excessive ram usage if implemented otherwise.

To dynamically bake building model that we were provided with, we simply created an array of "NavMeshSurface" and appended surfaces which are to be baked at that moment.
For demonstration purposes, below are the screenshots of the code base and the Unity editor space where we added the elements of the 1st floor to our array and on run-time we are calling the navMeshBaker method to bake only these specified objects dynamically.

```
public NavMeshSurface[] surfaces;

public void navMeshBaker() {
    for (int i = 0; i < surfaces.Length; i++) {
        surfaces [i].BuildNavMesh ();
    }
}
```



Dynamic baking worked perfectly fine while we were testing, and it seemed the way to go but later as we explored, we found that dynamic baking is not supported outside the Unity editor space yet. Hence, we cannot take advantage of it in mobile devices which were the main motivation behind moving to dynamic baking. When simply trying to bake the mesh dynamically on a mobile device, nothing would happen and as a result our agent wasn't moving at all. There are very few posts and threads related to this over the internet, and to confirm our findings we could only find one convincing article that explains why dynamic baking is not available in Android devices, the link to the post is here.

3.4.1) Why Dynamic Baking?

Initially we were using pre-baked models, but the thing with that is with bigger models loading the baked information all at once into the main memory may cause system crashes or other unexpected behaviors due to excessive ram usage. For smaller models this may not be the case but for bigger models such as our building model it may cause issues in some mobile devices.
Hence, we pivoted to the dynamic baking approach where the main idea was to bake a floor at a time dynamically and load others only when needed. As mentioned earlier, dynamic baking is supported in Android devices yet, hence we weren't able to gauge its significance yet and it is something we would like to work upon in the future.

3.4.2) Overheard caused by Dynamic Baking

Baking mesh dynamically does add an overhead and for certain situations it may not be ideal at all to bake the mesh dynamically, also baking mesh dynamically is *not possible in Android* devices yet, and as mentioned on the Unity forum till now this feature is only available in the Unity editor only. The link for this can be found here .

Hence to help developers decide whether to opt for run-time baking or to use the pre-baked meshes, we have tried to measure the overhead by using three parameters.

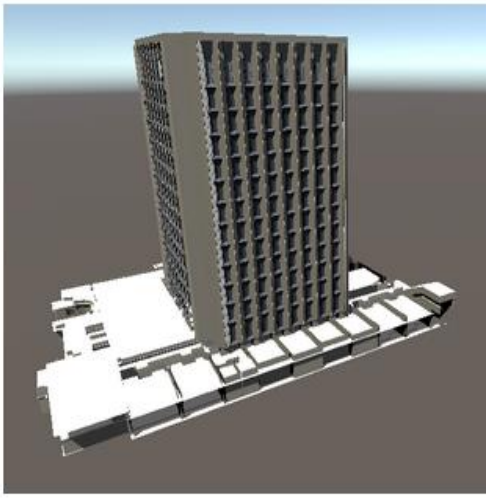
- Baking Time
- Application Size: Baked vs Pre-baked
- Resource Utilization

Run-time Baking Time

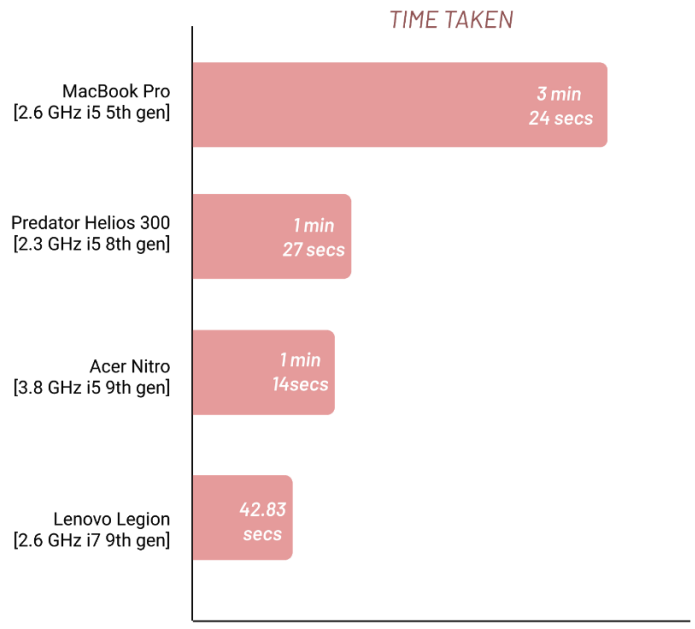
Baking the Navigation map is primarily a CPU intensive task, and it's overhead varies significantly with the baking settings, especially the *voxel size*. In order to get a proper idea of overhead, we measured the time taken for baking the building model that we have under the below specified settings on four devices and here are our findings:

Field	Value
Voxel Size	0.01 *(minimum value allowed by Unity)*
Agent Type	Humanoid
Obstacle Avoidance: Agent Radius	0.5
Obstacle Avoidance: Agent Height	1

Baking Settings



The Building Model



The above graph shows the time taken by different computers with different hardware to bake the model dynamically on the start of the application. We can clearly see that the better the processor of the laptop, lesser will be the baking time. We can clearly observe that even the top-of-the-line laptop is taking at least 40 seconds to bake the model dynamically and since we need this application for indoor localization, we need this to run on a portable device which does not have this much computation power.

Application Size: Pre-Baked Vs Unbaked

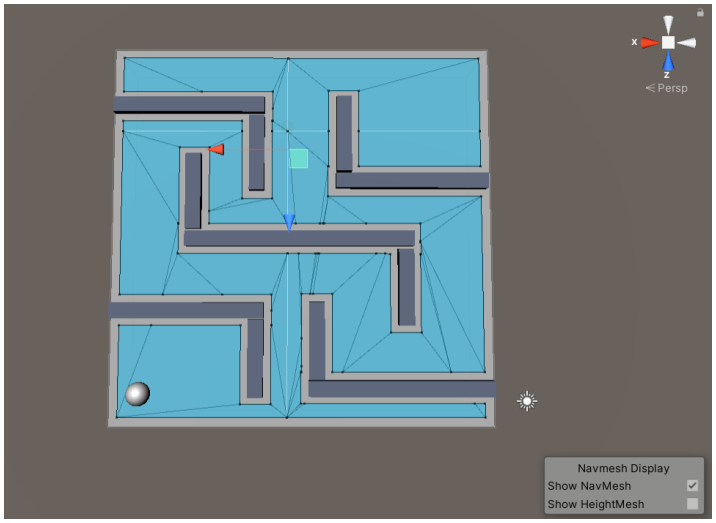
A pre-baked game scene stores the baked information along with the project data and is loaded into the main memory when needed. This information is generally in order of a few Kilobytes. When we tried to compare the size of the baked information for the entire building model that we were provided by Yogesh sir, we originally expected it to be little larger than the expected value, but because of the massive size of the model and because of all its details, the size of the compiled applications was as follows:

Application Type	Apk size (in MB)
Pre-Baked	18.407
Unbaked	19.124

This clearly shows that there was no drastic change in the size of the application with respect to the pre-baked and the unbaked state of the project.

Resource Utilisation

In order to gauge the overhead caused by the dynamic baking, we compared the resource utilization of baking a simple Unity scene “only once” with the once forcing it to “rebake every frame”. The idea behind this was to test whether it is possible to bake smaller parts of a big game scene simultaneously, and if possible, how much overhead will it add. For demonstrating this, we set up a simple game scene and created a class where a single call to the BuildNavMesh method bakes the surface dynamically.



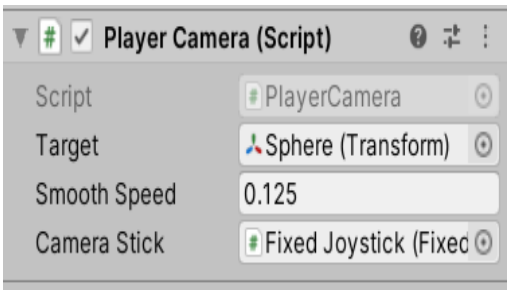

```
public class BakeDynamically : MonoBehaviour {

    public NavMeshSurface navMeshSurface;

    /**
        In order to bake the mesh only once i.e. when starting the game,
        un-comment the below lines and comment out the `Update` method's body
    */

    /*
        void Start() {
            navMeshSurface.BuildNavMesh();
            Debug.Log("Start Method: baked surface dynamically but only once");
        }
    */

    // Update is called once per frame
    void Update() {
        navMeshSurface.BuildNavMesh();
        Debug.Log("Update Method: baked surface dynamically");
    }
}
```

While monitoring, we found that for a relatively small scene, the overhead is approximately insignificant and hence if needed one can bake smaller parts of a big game scene simultaneously without affecting the user-experience. As mentioned earlier, since dynamic baking is not available outside the Unity editor, one cannot take advantage of this approach on mobile devices.

3.5) Extra Functionalities

In order to improve the testability and user-experience, we added a few extra features to the project. These include:

- Virtual Joystick for player camera *(code inside Player Camera Script)*
- Player following camera *(code inside Player Camera Script)*



Specifying Camera's Parameters [described below]

3.5.1) **Virtual Joystick**

For adding the virtual joystick in the game scene, we used the "Joystick Pack" from the Unity asset store, and the link to the asset is here. This asset simply needs to be imported into the Unity project and it contains multiple forms of virtual joysticks. In our project, we used the "Fixed Joystick" prefab and added it to the Canvas (*Unity's implementation for on-screen static menus*). After this the joystick actions/movements are simply needed to be handled inside the targeted camera's script. The code for the same is:

```
private void handleJoyStickAction() {
    float verticalMovement = cameraStick.input.y * cameraAngleSpeed;
    float horizontalMovement = cameraStick.input.x * cameraAngleSpeed;
    Camera.main.transform.Rotate (-verticalMovement, horizontalMovement, 0);
}


public void zoomCamera() {
    offset.x = 0;
    offset.y = 0;
    offset.z = 0;
    Debug.Log("Player Camera: camera zoomed");
}

public void farCamera() {
    offset.x = 0;
    offset.y = 1;
    offset.z = 1;
    Debug.Log("Player Camera: camera moved backwards!");
}
```

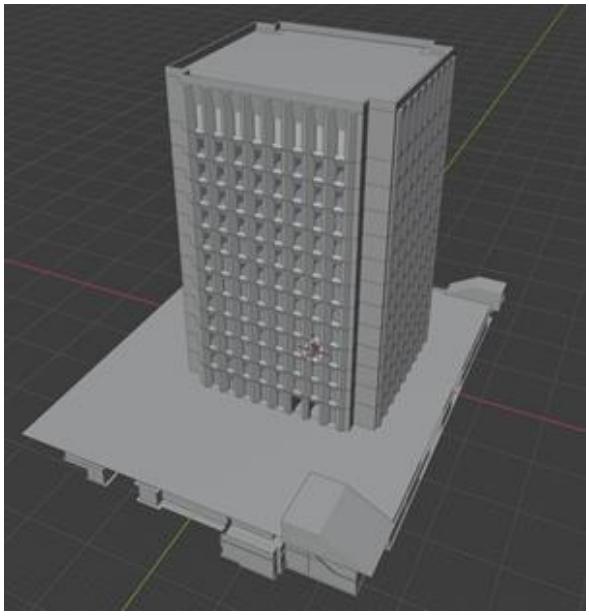
3.5.2) **Player Following Camera**

In order to make the camera follow the sphere (*the moving agent in our game*), the camera's position needs to be updated relative to the sphere in every frame. This can be achieved by first specifying the target (sphere in our case) and then modifying the camera's position relative to it. The code snippet for this is attached below

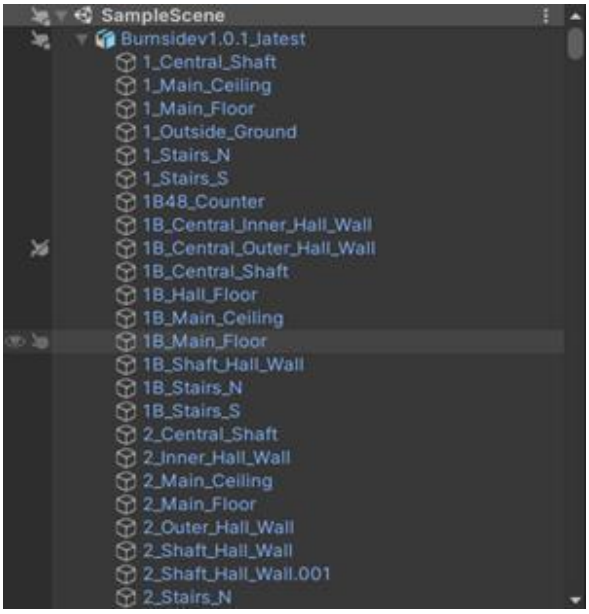
```
public Transform target;
public float smoothSpeed = 0.125f;
Vector3 offset;
public FixedJoystick cameraStick;

protected float cameraAngleSpeed = 2f;

void Start() {
    // reseting value on starting, can be modified using the Unity's editor space
    offset = new Vector3(0, 0, 0);
}

private void handlePlayerDrivenCamera() {
    Vector3 desiredPosition = target.position + offset;
    Vector3 smoothedPosition = Vector3.Lerp(transform.position, desiredPosition, smoothSpeed);
    transform.position = smoothedPosition;
}
```

3.6) Blender Model of a building

The final step of our project was to implement the shortest path functionality onto the large 3-D building model made in Blender which was given to us by Dr. Vinayak Naik. The Large scale 3-D model of the building on which we had to implement the NavMesh (A Star Routing Algorithm) was made from different components all combined together. For eg. every staircase, floor, char, table, door, window, wall etc. was an entirely separate object which helped us a lot as we could implement the NavMesh and the Navmesh would itself realize that the model consisted of smaller parts and could differentiate between the walkable and the non-walkable areas present in the building and lay a mesh network there.



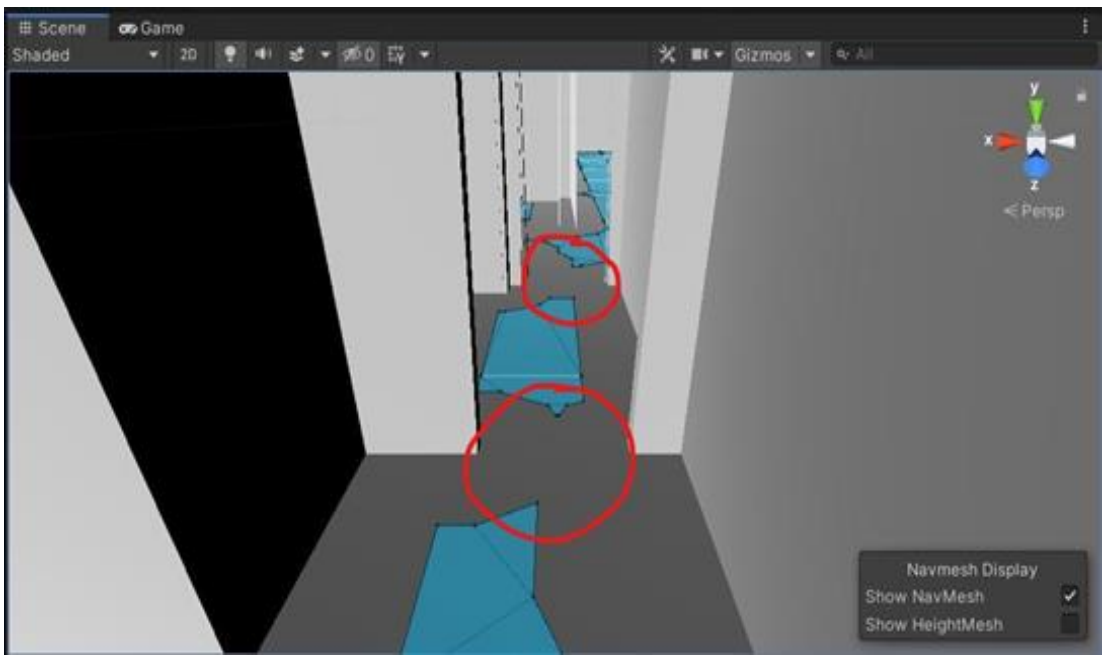
Actual Model of the Building



Components of the Building

3.7) Perfecting the NavMesh on 3-D Model

We implemented the NavMesh Surface on our 3-D model and baked the model initially. After baking the model for the first time, we found out that the NavMesh Surface inside the model was broken into different unconnected parts. As a result of this, our NavMeshAgent was not able to reach every area in the building because there were scattered navMesh components inside the building and few were disconnected also. The following images show the broken paths inside the building.



We came across 2 parameters which were used in order to fix this problem.

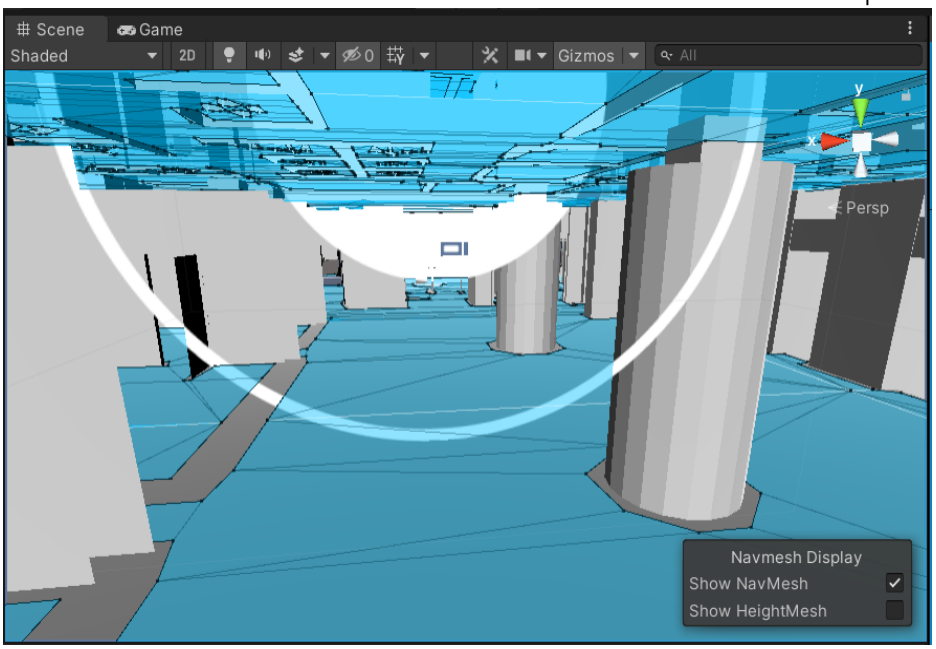
3.7.1) **Voxel Value**

The NavMesh is built by first voxelizing the Scene, and then figuring out walkable spaces from the voxelized representation of the Scene. The voxel size controls how closely the NavMesh fits the geometry of your Scene, and is defined in world units.

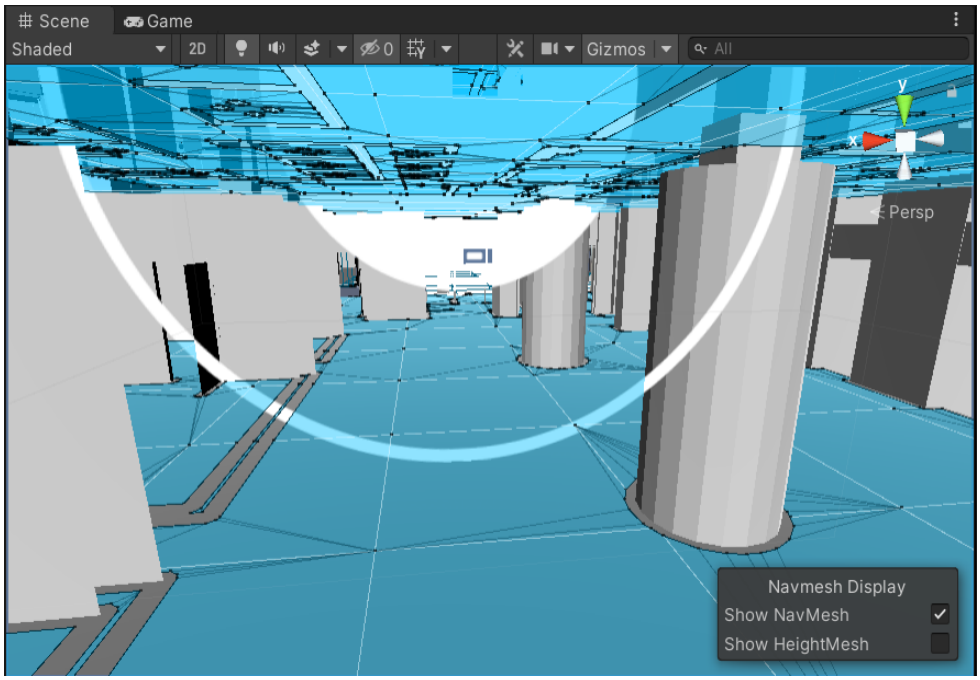
If we want our NavMesh to more closely fit our Scene's geometry so that we get more details, we will need to decrease the voxel size that will be used to bake our NavMesh finally. The game will consume more memory and will also and will take more time for the calculation of the NavMesh Data if we decrease the voxel size as there will be an increase in detailing on the whole model on which we are baking our Navmesh. The scaling used here is generally quadratic so if we double the resolution, it will basically quadruple the building time that will be used to bake the NavMesh. The best example of this is if we have a Scene with characters that have a radius of 0.3, a good voxel size would be 0.1. The default value of the voxel size is generally set to a third of the agent Radius. In order to change these values, we need to check the **overrideVoxelSize** setting which enables us to override the voxel value.

The voxel value we are using in this project for baking the NavMesh on the model is 0.01. We tried to change the voxel value to 0.001 instead of 0.01 but the minimum voxel value that can be used in Unity is 0.01 as decreasing the voxel value increases the building time and also consumes more memory. By reducing the voxel value to 0.01, we were able to reduce the non-walkable area around the obstacle, our Navmesh inside the building was better than what we had observed initially.

The image below shows a part of the building when the voxel value was set to 0.1 while baking the NavMesh. We can see that the precision with which the mesh is baked is not very ideal (like the non-walkable area around the pillar).



The image below shows a part of the building when the voxel value was set to 0.01 meaning the resolution is increased while baking the NavMesh. We can see that the precision with which the mesh baked is better compared to the previous case. Therefore, lower voxel value is preferred but that comes at the price of more building time and more memory consumption.



3.7.2) **Agent Radius**

The Agent Radius specifies the avoidance radius of the agent. This means that this value is used to set the Agent's "personal space" within which any other obstacle or agent won't be able to enter. This value is a very important factor in our model not getting connected components of the NavMesh inside the building.The explanation for the same is given below.

During the baking of the NavMesh, basically all the obstacles are discarded and a non-walkable area forms surrounding them. We can control how big we want this non-walkable area to be using Agent Radius. For e.g. if there is a pillar in the building, a circular non walkable area will be formed around it. Using the Agent Radius value, we can control the radius of this non walkable circle. If we set a higher Agent Radius value, this means that the non-walkable area around that pillar will be of larger radius thus reducing the walkable space near it. Now if we reduce this value, the radius of that non-walkable circle decreases and we will get better connected NavMesh components inside our building.

In the picture given below, we can see that the meshes are in fact connected and in better shape if we modify the Agent Radius value. But still we are able to observe that the non-walkable area surrounding the obstacles like the wall or the entrance of the door is very large and because of this our agent will have problems to access the areas smoothly. In order to fix this problem, we need to maintain a balance between the voxel value and the agent radius such that we will be able to obtain a perfectly baked mesh inside the building so that our Agent can move freely and access more part of the building.

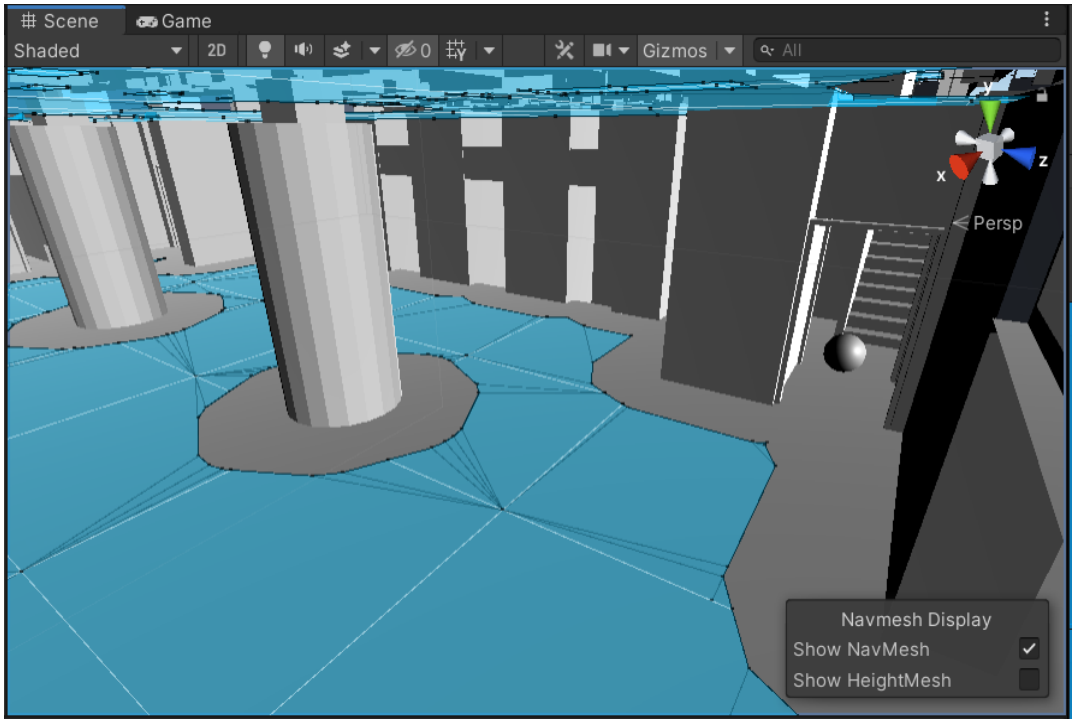
3.8) Creating More Accurate NavMesh

In order for the smooth movement and maximum accessibility of our Agent inside the building, we decided to test the baking process for different voxel values and Agent Radius.

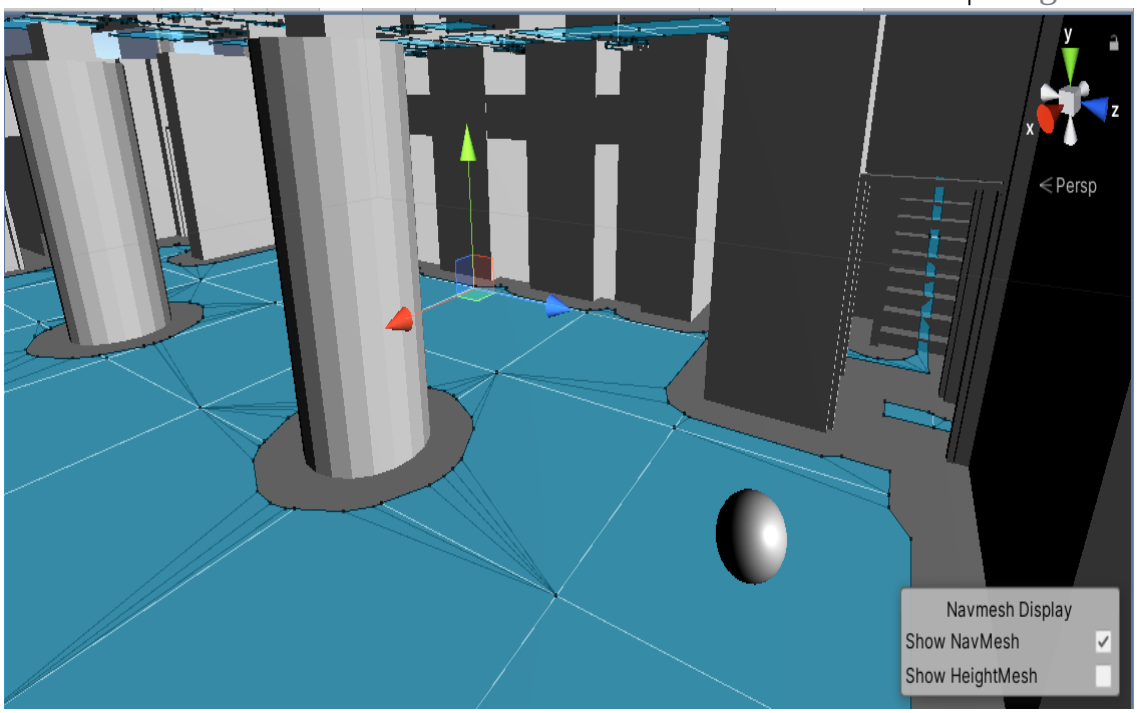
In order to reach all the tight spots in our model, we needed to increase the accuracy by making the voxel value smaller. We also need to work on the value of the Agent Radius in order to reduce the non-walkable area around the obstacles so our agent can move freely. Thus, we needed to find a relation between the voxel value and the Agent Radius in order to get the best model with connected NavMesh inside it.

A good estimation is that the Agent Radius needs to be approximately 2-8 times the voxel value which we have to use in order to achieve a good result. Moreover, the tighter corridors in the building we should leave at least ***4 * voxelSize*** clearance in addition to the agent radius, especially if the corridors are at angles. Upon testing for different values, we came across the following observations.

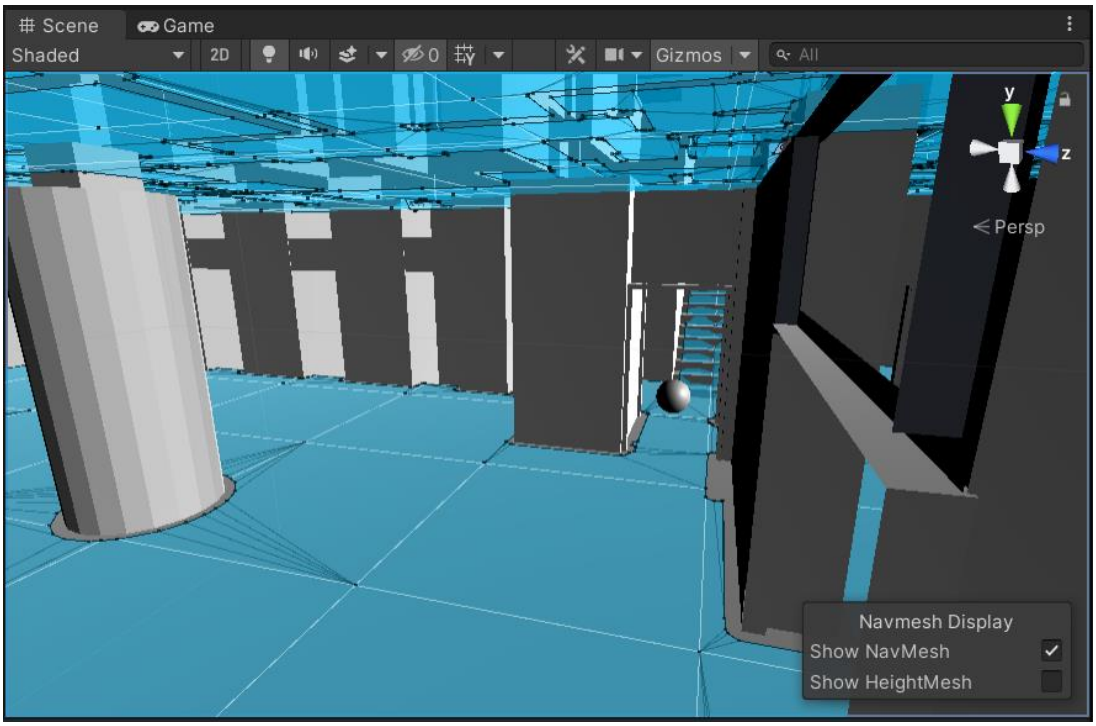
The following image shows the NavMesh after baking for an Agent Radius of 2. We can clearly see that the non-walkable area surrounding the obstacles and in between two obstacles is very large and should be taken care of in order to get a connected NavMesh. The disconnected part is where the ball is present. Due to the large Agent Radius, that part of the building with cluttered obstacles like walls, doors etc doesn't have a walkable path and is thus disconnected.



The following image shows the NavMesh after baking for an Agent Radius of 1.5. Here also we can see that the non-walkable area surrounding the obstacles and in between two obstacles is large but is better than the case before. We can see a part of the NavMesh present in the area where the mesh was absent earlier. This clearly shows the impact of Agent Radius on the baking of the NavMesh.



The following image shows the NavMesh after baking for an Agent Radius of 0.4. Here also we can see that the non-walkable area surrounding the obstacles and in between two obstacles is very small and each and every part of the building is connected by a single NavMesh which allows our agent to reach every part of the building. We can clearly see that the area which was before either not present or disconnected is now well connected to the main NavMesh in the hall. Due to this our agent can easily move inside the building and not get stuck anywhere.



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.

For making our project playable, we also added a few extra camera features as mentioned above in our report. The complete script for those features is present at https://pastebin.pl/view/ff4c05f7. This script allows users to change the camera angle by using a virtual joystick and also makes the camera follow the agent so that users can look inside building wherever the agent is going.

5) Conclusion

In our project, we have focused on developing a resource efficient Android application using Unity 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 to continue with the pre-baked models for now as the memory overhead even for relatively bigger models probably may not be an issue for modern day smartphones. Still we intend to keep our minds open to the idea of dynamic baking in Android devices and re-visit and evaluate it when this feature becomes available on them.

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

6) Future Work

After having the Unity's NavMesh tested under different use case scenarios including the final 3-D Model of the Building, the future steps for us to achieve a fully functional Indoor GIS System would include incorporation of the WiFi-signals to get the exact location of our Agent inside the building using the Access Points signals strength. This will be very useful in pin-pointing the exact floor and the exact room our Agent is in so we can give appropriate instructions to it for calculation of the shortest path between the Agent and its destination.

The large-scale dynamic baking of the model at runtime is also one of the things which can be explored in the near future when we get all the functionalities of this in Unity. The dynamic baking has been only tested on the PC because of its limited functionality as of now, but when we get the functionality on the mobile phone, we will need to test this project on the mobile phone and check whether the memory consumption, limited computational power etc. possess problems to this method.

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. Github, https://github.com/Brackeys/NavMesh-Tutorial/tree/master/NavMesh%20Example%20Project/Assets/NavMeshComponents
7. Documentation Unity, https://docs.unity3d.com/ScriptReference/AI.NavMeshAgent.SetDestination.html
8. Maze Creation Blender, https://www.youtube.com/watch?v=09ANJzy4dQY&ab_channel=AtMind
9. Advanced NavMesh Baked Settins, https://docs.unity3d.com/Manual/nav-AdvancedSettings.html
10. Unity Discussions Thread, https://answers.unity.com/questions/782693/finding-navmesh-layer-dynamically-during-runtime-o.html
11. NavMesh Build Settings, https://docs.unity3d.com/ScriptReference/AI.NavMeshBuildSettings.html
12. Virtual Joystick Pack, https://assetstore.unity.com/packages/tools/input-management/joystick-pack-107631