

Lab 11 - Path Finding - Unity's NavMesh

COMP396-Game Programming 2

[Introduction](#) · [Navigation Mesh \(NavMesh\)](#) · [Introduction](#) · [Setting up the map](#) · [Summary](#) · [Deliverables](#)

Purpose: To **study** and **implement** Path Finding via:

- ~~BFS Algorithm (Breadth First Search)~~ — DONE
- ~~DFS Algorithm (Depth First Search)~~ — DONE
- ~~Dijkstra Algorithm~~ — DONE
- ~~A* Algorithm (not grid-based)~~ — DONE
- ~~A* Algorithm (grid-based)~~ — DONE
- Unity's NavMesh and NavMeshAgent.

1. Introduction

We will deal hereby:

- Using Unity's NavMesh and NavMesh agent.

There is a new[er] package named **AI Navigation** available for new[er] Unity versions. See also the following youtube references:

- Ref: <https://www.youtube.com/watch?v=K6bBC0qkImI>
 - Duration 21':40"
 - Features Covered:
 - agent(s) going to a destination
 -
- Ref: <https://www.youtube.com/watch?v=hOdLLaigcHs> (3':37" - agent(s) following/chasing the player)

2. Navigation Mesh (NavMesh)

- Open the last project
- Create a folder under Scenes folder named **Lab11_PathFinding_WithNavMesh_{YourInitials}**

3. Introduction


We saw in previous labs about A* Pathfinding, that a critical decision is how to represent the scene's geometry. The AI agents need to know where the obstacles are, and it is our job as AI designers to provide the best representation we can to the pathfinding algorithm. Previously, we created a custom representation by dividing the map into a 2D grid, and then we implemented a custom pathfinding algorithm by implementing A* using that representation. In Unity we can use Navigation Meshes (NavMeshes) to automate this process. While in the previous 2D representation, we divided the world into perfect squares, NavMesh, divides the world using convex polygons, thereby resembling our previous non-grid-based A* implementation under the hood.

This representation has two advantages:

- first, every polygon can be different, so we can use:
 - a small number of big polygons for vast open areas, and
 - many smaller polygons for very crowded spaces;

- second, we do not need to lock the Agent on a grid anymore, and so the pathfinding produces more natural paths.

In this lab we'll use Unity's built-in NavMesh generator to make pathfinding for AI agents much easier and more performant.

 Some years ago, NavMeshes were an exclusive Unity Pro feature. Fortunately, this is not true anymore; NavMeshes are available in the free version of Unity for everyone!

In this lab, we will do the following:

- Setting up the map
- Building the scene with slopes
- Creating navigation areas
- An overview of Off Mesh Links

4. Setting up the map

Let's set up a scene:


- Set up a demo scene, named **Lab11_PathFinding_WithNavMesh_{YourInitials}**, with the following game objects:

Game Object	Type	Tag	Parent	P(x,y,z)	R(x,y,z)	S(x,y,z)	Color/Texture
Floor	Plane	Floor	Root	(0,0,0)	(0,0,0)	(10,1,10)	Gray
Start	Cube	Start	Root	Random	Random	Random	Green
End	Cube	End	Root	Random	Random	Random	Red
Obstacles	Empty	-	Root	(0,0,0)	(0,0,0)	(1,1,1)	-
Obstacle1-9	Cube	Obstacle	Root	Random	Random	Random	Gray/Wall

- *Take Snapshot*


4.1. NavMeshSurface

- Select the Floor
- Add the **NavMeshSurface** component

 All the properties in the NavMeshSurface are pretty self-explanatory:

- **Agent Radius** and **Agent Height** represent the size of the virtual agent used by Unity to bake the NavMesh,

Max Slope is the value in degrees of the sharpest incline the character can walk up, and so on.

 If we have multiple AI agents with different properties, we can define an appropriate NavMeshSurface for each of them with the respective appropriate properties.

- Explore the properties
- Click **Bake**
- Notice the bluish NavMesh created; notice the convex areas and the irregular triangles that mae up each of them.

4.2. NavMesh agent

At this point, we have completed the super-simple scene setup. Now, let's add some AI agents to see if it works:

1. As a character, we use a capsule for now (you can use any character you like).
2. Add the **NavMeshAgent** component to it.



This component makes pathfinding easy. We do not need to implement pathfinding algorithms such as A* anymore. Instead, we only need to set the **destination** property of the component at runtime, and the component will compute the path using Unity's internal pathfinding algorithm.

4.3. Updating an agent's destinations

Now that we have set up our AI agent, we need a way to tell it where to go and update the destination of the tank to the mouse click position. So, let's add a **sphere** entity, which we use as a marker object, and then attach the Target.cs script to an empty game object. Then, drag and drop this sphere entity onto this script's **targetMarker** transform property in the Inspector.

4.4. The Target.cs class

This script contains a simple class that does three things:

- Gets the mouse click position using a ray
- Updates the marker position
- Updates the destination property of all the NavMesh agents

The following lines show the Target class's code:

```
using UnityEngine;
using System.Collections;
using UnityEngine.AI; //for NavMeshAgent
public class Target : MonoBehaviour {
    private NavMeshAgent[] navAgents;
    public Transform targetMarker;
    public float verticalOffset = 10.0f;
    void Start() {
        navAgents = FindObjectsOfType(typeof(NavMeshAgent)) as NavMeshAgent[];
    }
    void UpdateTargets(Vector3 targetPosition) {
        foreach (NavMeshAgent agent in navAgents) {
            agent.destination = targetPosition;
        }
    }
    void Update() {
        // Get the point of the hit position when the mouse is being clicked
        if(Input.GetMouseButtonDown(0)) {
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            if (Physics.Raycast(ray.origin, ray.direction, out var hitInfo)) {
                Vector3 targetPosition = hitInfo.point;
                UpdateTargets(targetPosition);
                targetMarker.position = targetPosition + new Vector3(0, verticalOffset, 0);
            }
        }
    }
}
```

At the start of the game, we look for all the NavMeshAgent type entities in our game and store them in our referenced NavMeshAgent array (note that if you want to spawn new agents at runtime, you need to update the navAgents list). Then, whenever there's a mouse click event, we do a simple raycast to determine the first object colliding with the ray. If the beam hits an object, we update the position of our marker and update each NavMesh agent's destination by setting the destination property with the new position. We will be using this script throughout this chapter to tell the destination position for our AI agents.

- Test the scene, and click on a point that you want your tanks to go to. The NPC character(s) should move as close as possible to that point while avoiding every static obstacle (in this case, the walls).

4.5. Setting up a scene with slopes

One important thing to note is that the slopes and the wall should be in contact. If we want to use NavMeshes, objects need to be perfectly connected. Otherwise, there'll be gaps in the NavMesh, and the Agents will not be able to find the path anymore. There's a feature called Off Mesh Link generation to solve similar problems, but we will look at Off Mesh Links in the Using Off Mesh Links section later in this chapter. For now, let's concentrate on building the slope:

1. Make sure to connect the slope properly
2. We can adjust the **Max Slope** property in the Navigation window's Bake tab according to the level of slope in our scenes that we want to allow the Agents to travel. We'll use 45° here. If your slopes are steeper than this, you can use a higher Max Slope value.
3. Bake the scene, and you should have generated a NavMesh
4. We will place some NPCs with the Nav Mesh Agent component.
5. Create a new cube object and use it as the target reference position.
6. We will be using our previous **Target.cs** script to update the destination property of the AI agent.
7. Test run the scene, and you should see the AI agent crossing the slopes to reach the target.

Congratulation, you have implemented your first basic NavMesh-powered AI.

4.6. Challenge

- Implement the additional tasks seen in the youtube video:
 - Dynamic Obstacles
 - Multiple NavMeshSurface(s) for more than one type of agent(s).
 - Using OffMeshLinks
- Explore the examples coming with **AI Navigation** package.

5. Summary

- We used NavMeshSurface and NavMeshAgent to implement pathfinding in Unity.

6. Deliverables

- Zip into a file named **Lab11_{YourInitials}.zip** the following content:
 - the solution (you may create a unitypackage but make sure you have all is needed first).
 - the snapshots document
 - a short (~1 min) video with playtesting of the solution
- Upload in eCentennial Lab11 folder.