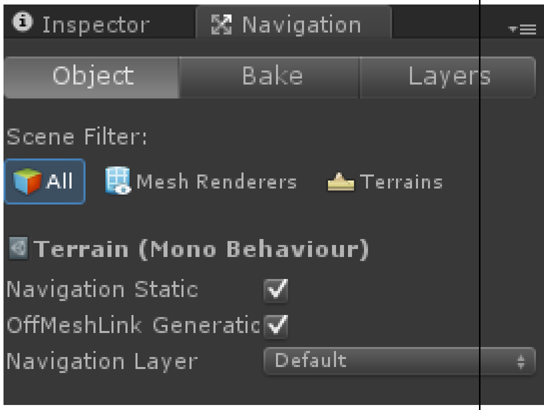
Navmeshes

This tutorial will need you to have completed the prior ones, particularly the setting up of the enemies. At the moment the script we're using isn't great, it's pretty bad in fact, we'll improve it by adding and using the Navmesh system built into Unity, which whilst not great, is a damn sight better.

What is a Navmesh?

The built in navigation system, uses something called a NavMesh that makes an agent intelligently walk through it to reach different targets.

NavMesh is a commonly-used technique in Game AI splitting the map into a walkable area and non-walkable. It's also used to calculate a path between two points, enabling the AI to travel from its position to a goal, let an enemy reach a player or move the player to a desired destination. The drawback of this in Unity is that it only works on the X and Z planes, it doesn't let you completely navigate in 3D space, it also can't be updated at runtime.



Terrain (Mono Behaviour)
Navigation Static ✓
OffMeshLink Generatic ✓
Navigation Layer: Default

You can access the navigation system through the Navigation panel using Window – Navigation, and it will open next to the Inspector.

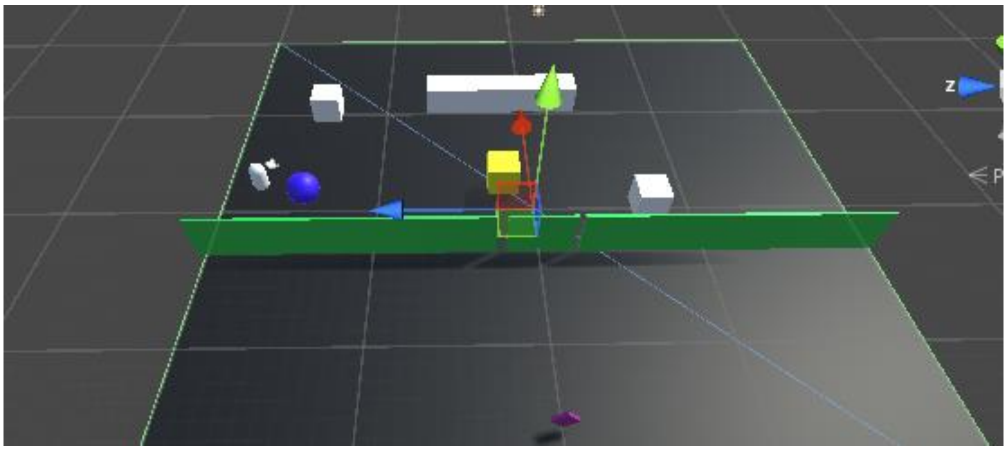
Marking Scene objects and baking

First in the hierarchy select your ground object, then with the Navigation window open, go to the object tab and define whether the selected object is Navigation static or not; walls, floors, most obstacles and platforms are all static objects- they don't move, they don't change.

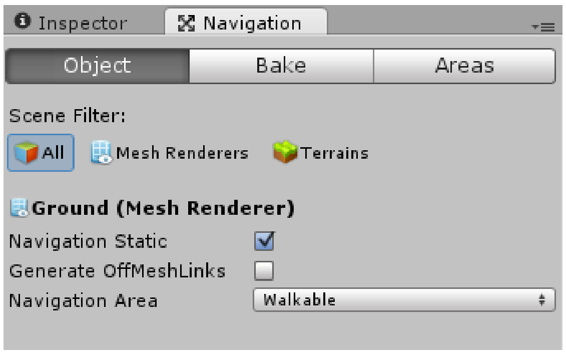
Make sure to go through all the objects in the hierarchy and set them appropriately. You can use the Scene filter in the Navigation panel to show only Mesh Renderers or Terrains. When doing this, make sure to select the actual object with a mesh, not the parent, the filters might help. Don't select any objects that are going to be moving!

Next, define the Navigation Layer for the object using the related drop-down. You can choose between Default (walkable), Not Walkable and Jump. You can also define custom layers but we won't be covering this just now

In the below screenshot you can see my setup, the white boxes are the obstacles, the yellow is a pickup, the green are walls and a door and the purple is the camera switcher. All these are covered in the former tutorials.

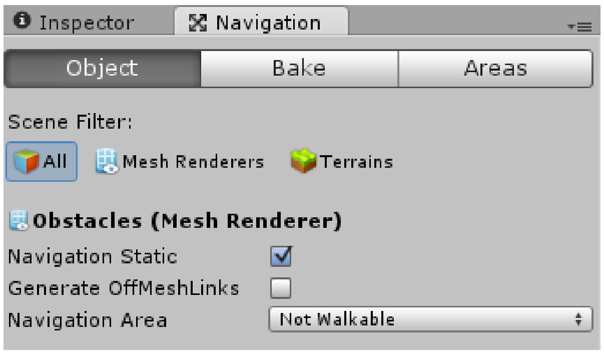


Here, the white blocks, the black ground, the purple switcher and the green walls are all static, they don't move. Yours doesn't have to look exactly like this but should contain roughly the same items. The ground is obviously marked as walkable; the rest is not, the door we will set later, leave that for the time being.



Ground (Mesh Renderer)

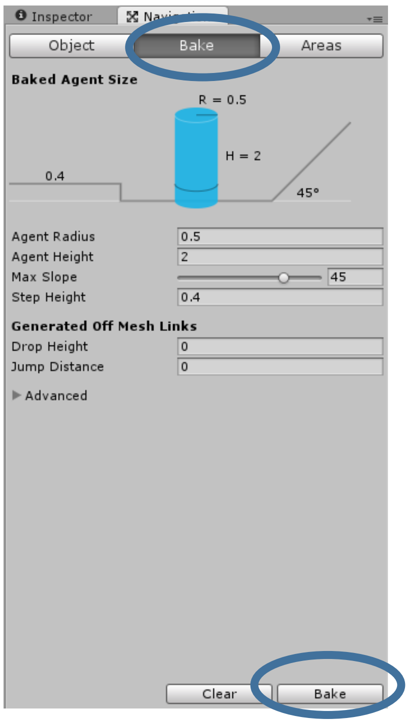
Navigation Static	✓
Generate OffMeshLinks	
Navigation Area	Walkable



Obstacles (Mesh Renderer)

Navigation Static	✓
Generate OffMeshLinks	
Navigation Area	Not Walkable

Now you're ready to try baking the navmesh, click the bake button towards the bottom you should see your scene change, with the navmesh "baked" on, it's visible blue when you have the navigation tab open.

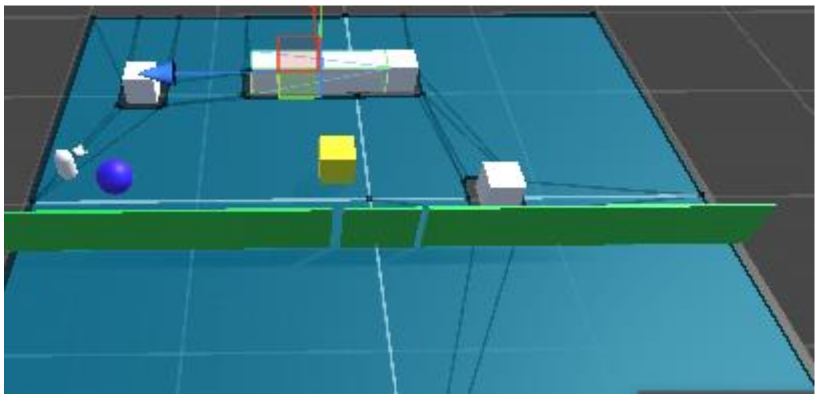


Baked Agent Size

Agent Radius	0.5
Agent Height	2
Max Slope	45
Step Height	0.4

Generated Off Mesh Links

Drop Height	0
Jump Distance	0



AI will be able to walk only on the blue area of the map, you can see that some areas aren't coloured, that's the bits the AI won't be able to use make sure you remember though, every time you make a modification to placement you need to rebake your scene.

Setting NavMesh properties

You can customize your NavMesh in the Bake Tab seen in the screenshot above, there's a number of settings that you can change.

Radius is the distance between the walls and the navmesh, and represents the agent's radius. If you feel that your agent keeps bumping on walls or objects while moving, increase the radius to make it smoother.

Height represents the agent's height and specifies the minimum height of areas where the agent will be able to venture into.

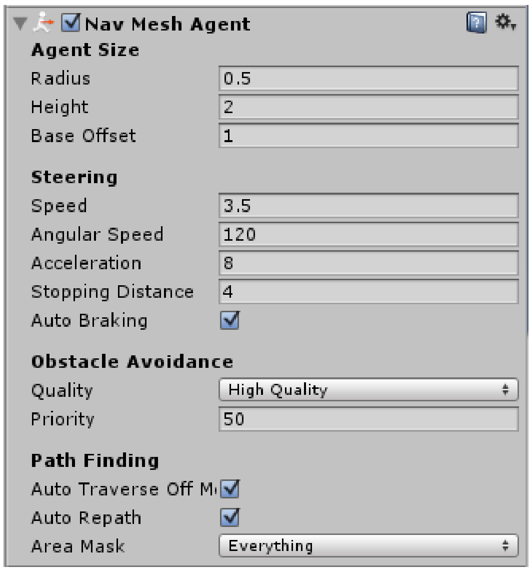
Max slope specifies the maximum slope for a surface to be considered walkable, while Step Height is the height difference between two surfaces to be considered connected.

Under the Advanced group, you can set the width and height inaccuracy, which specify the approximation allowed when generating the NavMesh. Lower values will give a higher quality NavMesh, but are more computationally expensive. Generating a more accurate NavMesh will also take longer. In this scene there is no need for slopes yet, but there may be later. If you’ve done the challenge to replace the ground with a Unity terrain, you may need to change some of these settings to suit the terrain.

The NavMesh Agent

Now that we've got a NavMesh in our scene, we need a way to tell our character to walk through it. This is done using the NavMeshAgent component. You'll find it in the Add Component Menu, under Navigation.

Add a NavMeshAgent component to your player game object, this component is responsible for the agent pathfinding and its actual movement control.



Nav Mesh Agent

Agent Size

Radius	0.5
Height	2
Base Offset	1

Steering

Speed	3.5
Angular Speed	120
Acceleration	8
Stopping Distance	4
Auto Braking	✓

Obstacle Avoidance

Quality	High Quality
Priority	50

Path Finding

Auto Traverse Off Mı	✓
Auto Repath	✓
Area Mask	Everything

There are many properties that you can set in the component. Again, Radius specifies the radius of the agent (and defines whether they can go through a narrow path), while Height is the agent's height and defines whether they can pass under obstacles.

Speed, Acceleration and Angular Speed are self-explanatory. They define how your agent will move, while Stopping Distance defines how close the agent will get to the target position before stopping and auto braking lets the agent brake in time to stop in good time.

We now need to tell the agent what to do via scripting. We’re going to create a new script named “EnemyNavMovement” and copy the following;

We need a variable to hold the NavMeshAgent the script is going to control and the target of the enemy.

```
NavMeshAgent agent;
public Transform target;

void Start()
{
    target = GameObject.FindGameObjectWithTag("Player").transform;
}

  // Update is called once per frame
  void Update () {
      agent = GetComponent<NavMeshAgent> ();
      agent.SetDestination (target.position);
  }
```

Then, in update we first set the agent value to the NavMeshAgent attached to the same object then we call “SetDestination” which does as you might expect, tells the agent to make a path towards the position we set in between the brackets.

This is just a very basic navScript, there’s nothing complicated. Attach it to the enemy (making sure to disable the enemyMovement script) and drag the player onto the target variable in the inspector. The enemy should better follow the player, avoiding obstacles littered around.

NavMesh Obstacles

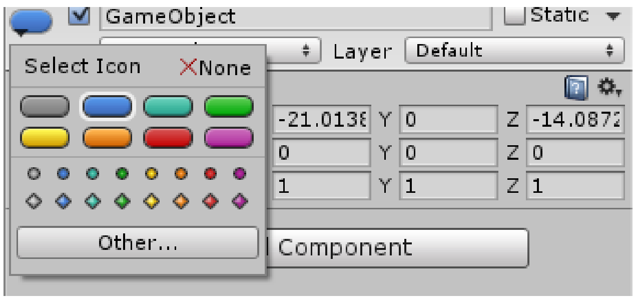
Select the door or another object that can move, instead of setting it as a navigation static we're going to use something else. Statics are things that don't move, don't change, if a door opens it moves, therefore it must change. What we use instead is a NavMesh Obstacle, added in a similar way to NavMeshAgents they are a bit like colliders for a navmesh, an agent can't path through them and they will avoid them in their pathing. Try adding one.

Waypoints

Sometimes what you want is for your enemy to patrol back and forth; this can be done with waypoints which are much easier with the navmesh system.

Waypoints allow travel between numerous points, let's set those points up first, we'll use a series of empty gameObjects as place holders, much easier than just saving coordinates.

Create an Empty GameObject (GameObject->Create Empty) which we'll use as a holder, inside this one, right click and add a four or five more empty waypoints. Make sure each of them has a Y value of 0- in other words is level with the floor, too far up or down and you'll not be able to get to it. Scatter them around the map, in order to see them better you can add an editor only marker to show the position. In Inspector, next to the name is a little icon, click it and you can select a whole bunch of icons that show in the inspector. The screenshot right may help.



Once you have a list of waypoints, we can proceed.

Create a new script WayPoints and copy the below, it's a little more advanced than some of what we've seen before.

```
public class Waypoints : MonoBehaviour {

    NavMeshAgent agent;
    // List or Array of points- starts at number 0
    public GameObject waypoints;
    Transform[] points;
    public int destPoint=0;

    // Use this for initialization
    void Start () {
        points = waypoints.GetComponentsInChildren<Transform>();
        agent = GetComponent<NavMeshAgent>();
        //Can turn off autobraking to not stop between waypoints
        agent.autoBraking = false;
        GotoNextPoint ();
    }
```

Again, we need to get the navmeshagent, we also have variabes to store the waypoints collection, the individual points (stored in an array, a collection of things, like a List) and the point we're going to head to (starting off at 0, because that's what an array starts from).

In start we mostly set this up, we set points to be equal to the result of calling "GetComponentsInChildren<Transform>" on waypoints. This gets all the transforms in the waypoint collection, including the top level one we created. This is a quick way of getting all the transforms, we could make the array public but that would mean dragging each one in one by one, if there's lots of them it's slow.

We also turn off autobraking, this stops the navmeshagent slowing and stopping between each waypoint, giving a smooth transition. Finally in start we call "GotoNextPoint()" a function we're about to create that we use to travel between points.

```
    void GotoNextPoint()
    {
        if (points.Length == 0) {
            return;     }

        agent.SetDestination (points [destPoint].position);
        //if goes higher than the total number of waypoints -> go back to start of array
        destPoint = (destPoint + 1) % points.Length;
    }

    // Update is called once per frame
    void Update () {
        if (agent.remainingDistance < 0.5f) {
            GotoNextPoint ();
        }
    }
}
```

Underneath the closing bracket of start (}) you need to include the above. GotoNextPoint is called every time we get to a waypoint, it checks if there are waypoints before sending the agent to the point in the list tracked by the destPoint variable. It then updates the destPoint variable to ensure that if it gets higher than the number of destination points it goes back to start. % here is modulo, this means that it finds the remainder after dividing (destPoint+1) and points.length. If they're equal it'll be 0, otherwise it'll be destPoint+1, a neat little bit of maths.

Finally, in update, we check the distance to the point we're going and if it's less than 0.5 we call goToNextPoint. The 0.5 should be increased to whatever the stopping distance is, otherwise you'll never get to the waypoints.

RTS Style click to move

This one is a little different and probably not needed in this project but worth looking at, it's fairly self-explanatory.

```
public class RTSStyle : MonoBehaviour {

    public Transform target;
    NavMeshAgent agent;

    // Use this for initialization
    void Start () {
        agent = GetComponent<NavMeshAgent> ();
    }

    // Update is called once per frame
    void Update () {

        if (Input.GetMouseButton (0)) {
            //holds the info about raycast
            RaycastHit hit;

            //launch a raycast from the middle of the camera to a point
            if(Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition),out hit,100))
            {
                agent.destination = hit.point;
            }
        }
    }
}
```

NavMeshes have some drawbacks, one of which is only being able to move in the X and Z axis, in other words they can't go up, difficult for flying, space or underwater games. Another is that they can't be updated at runtime, meaning it needs to be baked. Alternatives are available on the asset store or via the web, many of them better but they all use similar techniques. Splitting the map up into segments and using an algorithm to work out the best route between those segments.

Challenges

Challenge 1

Add a new enemy type prefab that is faster but less attack (make sure it's still not faster than the player) and one that's slower but hits harder. Make sure both of these use the Navmesh.

Challenge 2

Implement a system with a stationary or patrolling guard, when you pass through a trigger area somewhere on the ground have the AI chase you. This simulates an alarm.

Challenge 3

This combines a number of elements, your patrolling enemies should react to the presence of the player within a certain distance. You'll need to work out the distance between player and enemy, if the distance is above a certain amount, have the enemy chase the player.

Challenge 4

Add a randomness to it, make it a random chance that the enemies will react to you getting close.