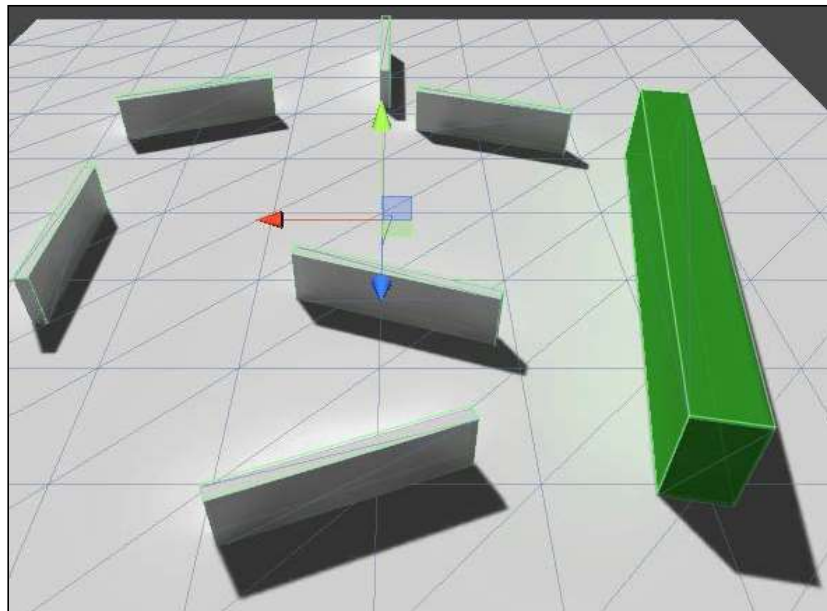


Navigation mesh

Next, we'll learn how to use Unity's built-in navigation mesh generator that can make pathfinding for AI agents a lot easier. As of Unity 5, NavMesh is available to all the users. Previously a Unity Pro-only feature, NavMesh is now a part of the Personal Edition of Unity. We were briefly exposed to Unity's NavMesh in *Chapter 2, Finite State Machines and You*, which relied on a NavMesh agent for movement in testing our state machine. Now, we will finally dive in and explore all that this system has to offer. AI pathfinding needs representation of the scene in a particular format. We've seen that using a 2D grid (array) for A* Pathfinding on a 2D map. AI agents need to know where the obstacles are, especially the static obstacles. Dealing with collision avoidance between dynamically moving objects is another subject, primarily known as steering behaviors. Unity has a built-in navigation feature to generate a NavMesh that represents the scene in a context that makes sense for our AI agents to find the optimum path to the target. This chapter comes with a Unity project that has four scenes in it. You should open it in Unity and see how it works to get a feeling of what we are going to build. Using this sample project, we'll study how to create a NavMesh and use it with AI agents inside our own scenes.

Setting up the map

To get started, we'll build a simple scene, as shown in the following screenshot:

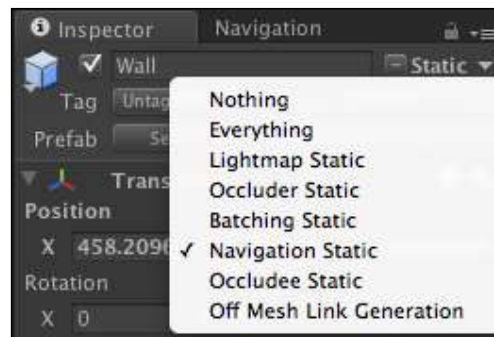


A scene with obstacles

This is the first scene in our sample project called `NavMesh01-Simple.scene`. You can use a plane as a ground object and several cube entities as the wall objects. Later, we'll put in some AI agents (we'll be turning to our trusted tank for this example as well) to go to the mouse-clicked position, as in an **RTS (real-time strategy)** game.

Navigation Static

Once we've added the walls and ground, it's important to mark them as **Navigation Static** so that the NavMesh generator knows that these are the static obstacle objects to avoid. Only game objects marked as navigation static will be taken into account when building the NavMesh, so be sure to mark any environment elements accordingly. To do this, select all those objects, click on the **Static** dropdown, and choose **Navigation Static**, as shown in the following screenshot:



The Navigation Static property

Baking the navigation mesh

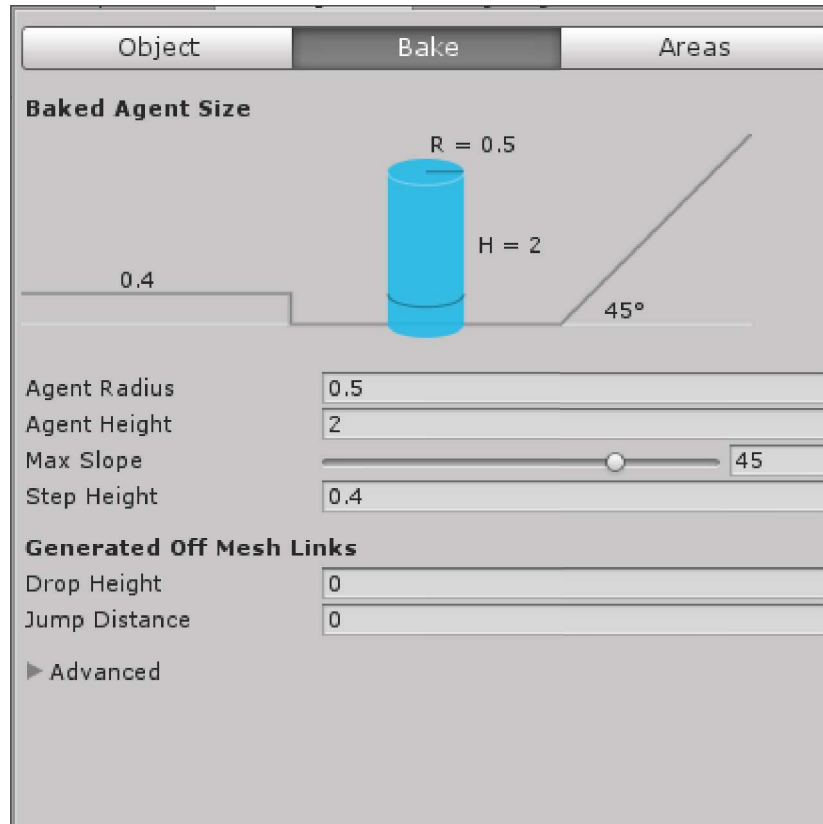
Now we're done with our scene. Let's bake the NavMesh. Firstly, we need to open the navigation window. Navigate to **Window | Navigation**. The navigation window is broken up into three different sections. The first, **Object**, looks similar to the following screenshot:



The navigation object window

The **Object** tab of the navigation window is simply a shortcut to selecting objects and modifying their navigation-related attributes. Toggling between the **Scene Filter** options, **All**, **Mesh Renderers**, and **Terrains**, will filter out objects in your hierarchy accordingly so that you can easily select objects and change their **Navigation Static** and **Generate OffMeshLinks** flags as well as set their **Navigation Area**.

The second tab is the **Bake** tab. It looks similar to the following screenshot:



If you've ever stumbled across this tab prior to Unity 5, you may notice that it now looks a bit different. Unity 5 added a visualizer to see exactly what each setting does. Let's take a look at what each of these settings does:

- **Agent Radius:** The Unity documentation describes it best as the NavMesh agent's "personal space". The agent will use this radius when it needs to avoid other objects.
- **Agent Height:** This is similar to radius, except for the fact that it designates the height of the agent that determines if it can pass under obstacles, and so on.
- **Max Slope:** This is the max angle that the agent can walk up to. The agent will not be able to walk up the slopes that are steeper than this value.
- **Step Height:** Agents can step or climb over obstacles of this value or less.

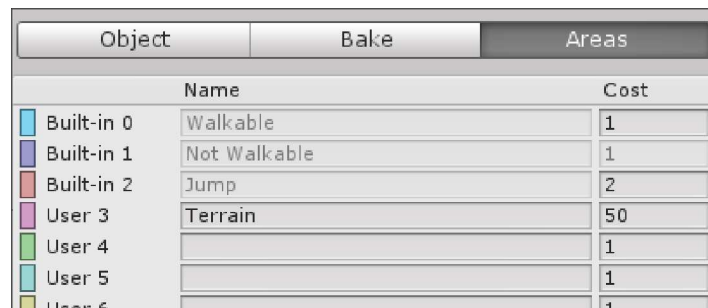
The second category of values only applies when you checked **Generate OffMeshLinks** when building your NavMesh. This simply means that the agent will be able to potentially navigate the NavMesh even when gaps are present due to physical distance:

- **Drop Height:** Fairly straightforward, this is the distance an agent can jump down. For example, the height of a cliff from which an agent will be "brave enough" to jump down.
- **Jump Distance:** This is the distance an agent will jump between offmesh links.

The third and final set of parameters is not the one that you would generally need to change:

- **Manual Voxel Size:** Unity's NavMesh implementation relies on voxels. This setting lets you increase the accuracy of the NavMesh generation. A lower number is more accurate, while a larger number is less accurate, but faster.
- **Min Region Area:** Areas smaller than this will simply be culled away, and ignored.
- **Height Mesh:** It gives you a higher level of detail in vertical placement of your agent at the cost of speed at runtime.

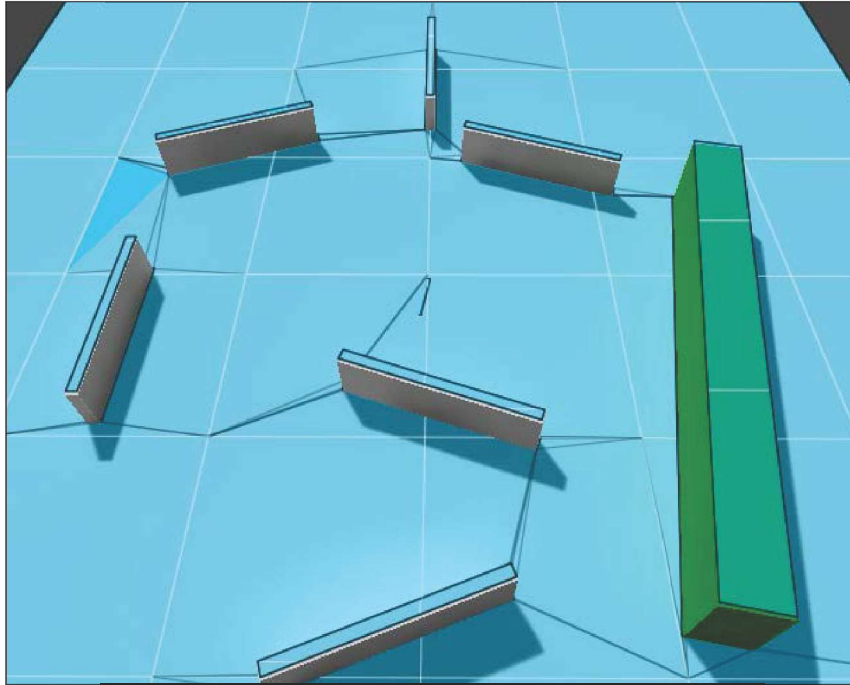
The third and last tab is the **Areas** tab, which looks similar to the following screenshot:



Object			Bake			Areas		
						Name	Cost	
	Built-in 0					Walkable	1	
	Built-in 1					Not Walkable	1	
	Built-in 2					Jump	2	
	User 3					Terrain	50	
	User 4						1	
	User 5						1	
	User 6						1	

If you recall, the **Object** tab allows you to assign the specific objects to certain areas, for example, grass, sand, water, and so on. You can then assign an area mask to an agent, which allows you to pick areas agents can or cannot walk through. The cost parameter affects the likeliness of an agent to attempt to traverse that area. Agents will prefer lower-cost paths when possible.

We will keep our example simple, but feel free to experiment with the various settings. For now, we'll leave the default values and just click on **Bake** at the bottom of the window. You should see a progress bar baking the NavMesh for your scene, and after a while, you'll see your NavMesh in your scene, as shown in following diagram:



The navigation mesh baked

Using the NavMesh agent

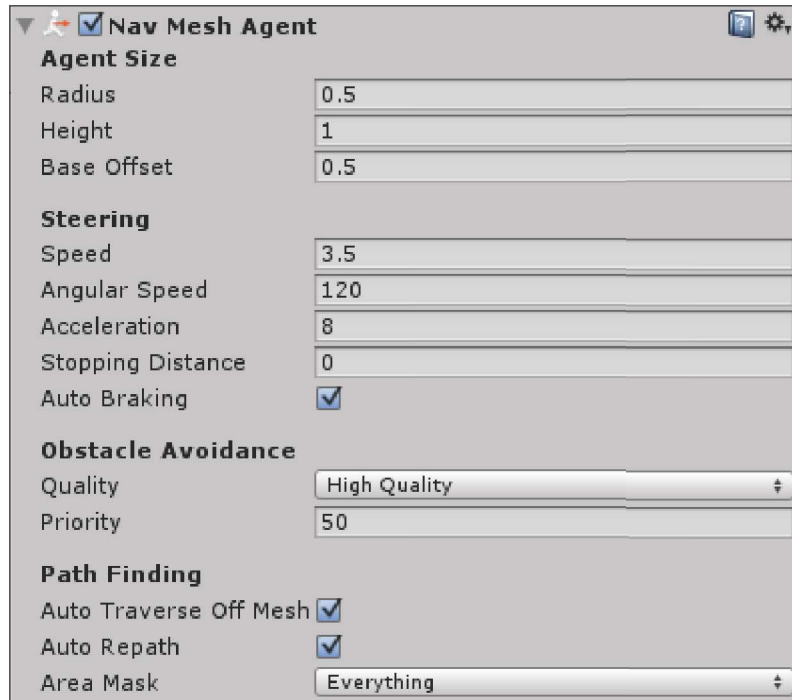
We're pretty much done with setting up our super simple scene. Now, let's add some AI agents to see if it works. We'll use our tank model here, but if you're working with your own scene and don't have this model, you can just put a cube or a sphere entity as an agent. It'll work the same way.




The tank entity

The next step is to add the **NavMesh Agent** component to our tank entity. This component makes pathfinding really easy. We don't need to deal with pathfinding algorithms directly anymore as Unity handles this for us in the background. By just setting the `destination` property of the component during runtime, our AI agent will automatically find the path itself.

Navigate to **Component | Navigation | Nav Mesh Agent** to add this component.



The Nav Mesh Agent properties

[ Unity reference for the **NavMesh Agent** component can be found at <http://docs.unity3d.com/Documentation/Components/class-NavMeshAgent.html>.]

Setting a destination

Now that we've set up our AI agent, we need a way to tell this agent where to go and update the destination of our tanks to the mouse-click position.

So, let's add a sphere entity to be used as a marker object and then attach the following `Target.cs` script to an empty game object. Drag-and-drop this sphere entity onto this script's `targetMarker` transform property in the inspector.

The Target class

This is 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 code present in this class:

```
using UnityEngine;
using System.Collections;

public class Target : MonoBehaviour {
    private NavMeshAgent[] navAgents;
    public Transform targetMarker;

    void Start() {
        navAgents = FindObjectsOfType(typeof(NavMeshAgent)) as
            NavMeshAgent[];
    }

    void UpdateTargets(Vector3 targetPosition) {
        foreach (NavMeshAgent agent in navAgents) {
            agent.destination = targetPosition;
        }
    }

    void Update() {
        int button = 0;

        //Get the point of the hit position when the mouse is
        //being clicked
        if (Input.GetMouseButtonDown(button)) {
            Ray ray = Camera.main.ScreenPointToRay(
                Input.mousePosition);

            RaycastHit hitInfo;

            if (Physics.Raycast(ray.origin, ray.direction,
                out hitInfo)) {
                Vector3 targetPosition = hitInfo.point;
                UpdateTargets(targetPosition);
                targetMarker.position = targetPosition +
```

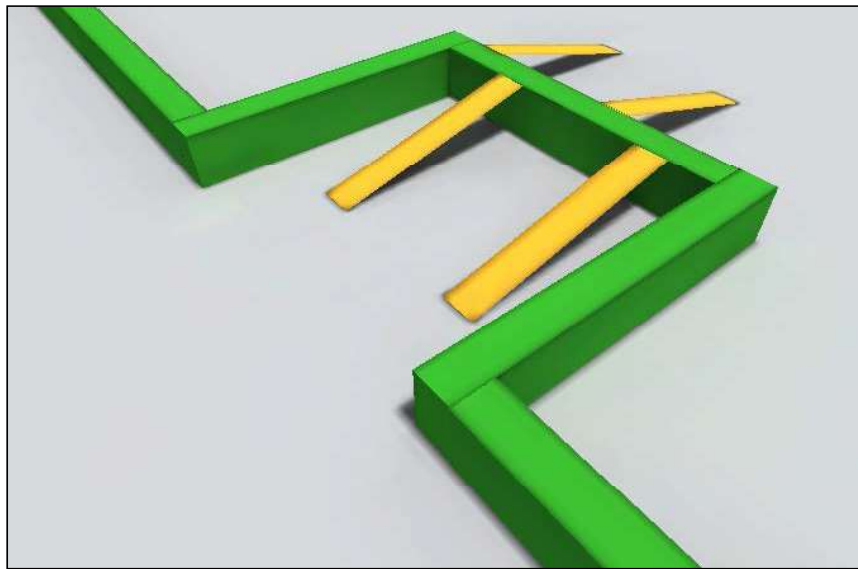
```
        new Vector3(0,5,0);  
    }  
}  
}
```

At the start of the game, we look for all the `NavMeshAgent` type entities in our game and store them in our reference `NavMeshAgent` array. Whenever there's a mouse-click event, we do a simple raycast to determine the first objects that collide with our ray. If the ray hits any object, we update the position of our marker and update each `NavMeshAgent`'s destination by setting the destination property with the new position. We'll be using this script throughout this chapter to tell the destination position for our AI agents.

Now, test run the scene and click on a point where you want your tanks to go. The tanks should come as close as possible to that point while avoiding the static obstacles like walls.

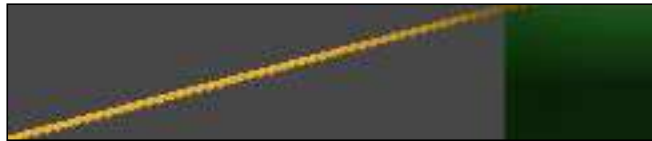
Testing slopes

Let's build a scene with some slopes like this:



Scene with slopes

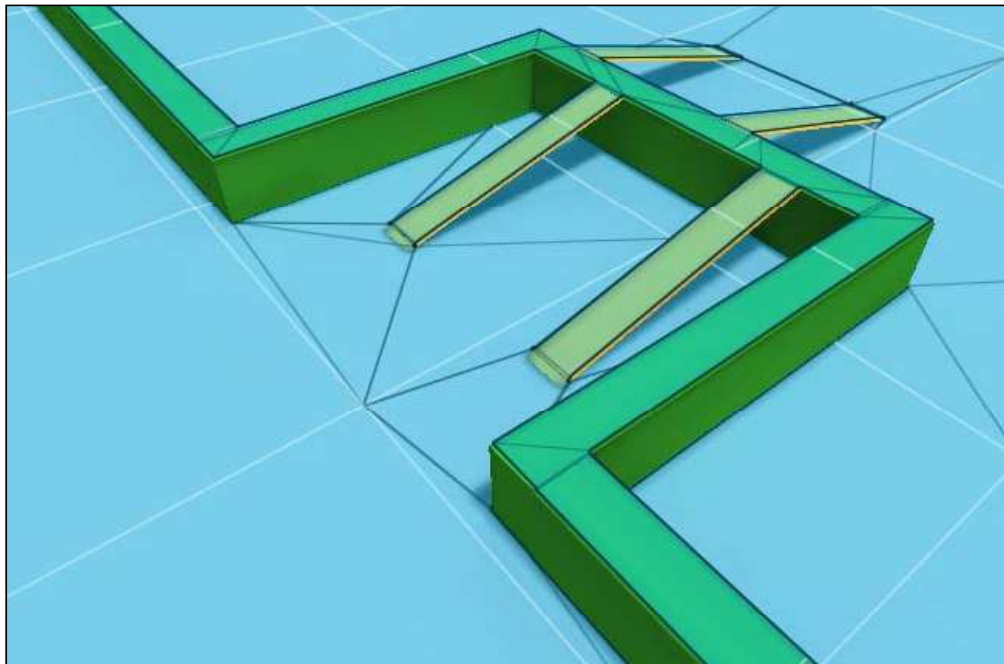
One important thing to note is that the slopes and the wall should be in contact with each other. Objects need to be perfectly connected when creating such joints in the scene with the purpose of generating a NavMesh later, otherwise, there'll be gaps in NavMesh and the agents will not be able to find the path anymore. For now, make sure to connect the slope properly.



A well-connected slope

Next, 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 agents to travel. We'll use 45 degrees here. If your slopes are steeper than this, you can use a higher `Max Slope` value.

Bake the scene, and you should have a NavMesh generated like this:



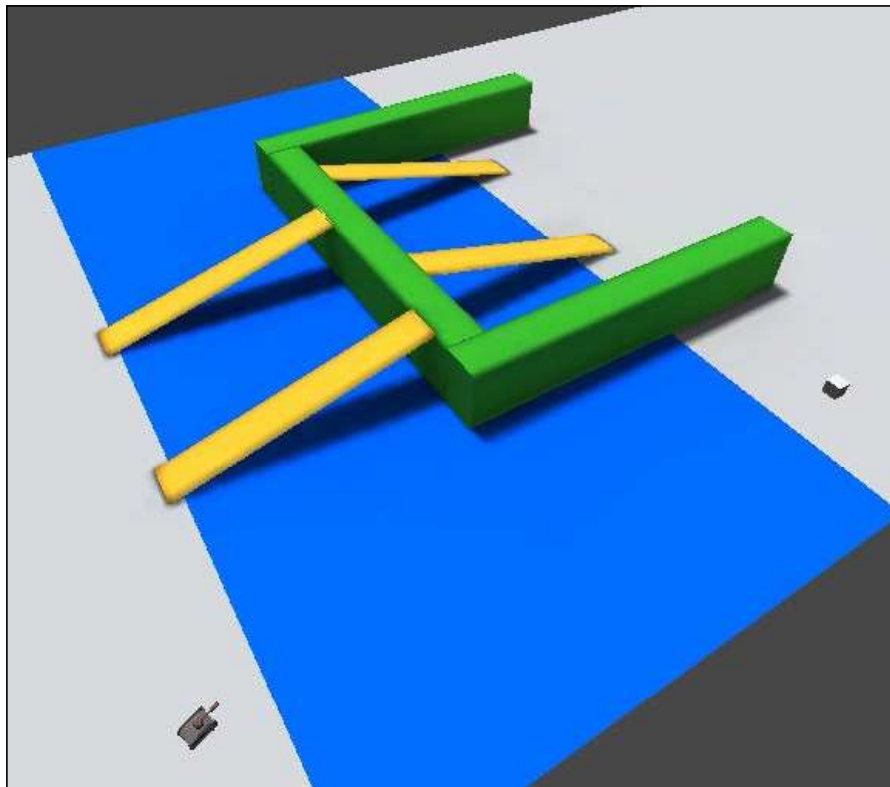
NavMesh generated

Next, we'll place some tanks with the **NavMesh Agent** component. Create a new cube object to be used as a target reference position. We'll be using our previous `Target.cs` script to update the destination property of our AI agent. Test run the scene, and you should have your AI agents crossing the slopes to reach the target.

Exploring areas

In games with complex environments, we usually have some areas that are harder to travel in than others, such as a pond or lake compared to crossing a bridge. Even though it could be the shortest path to target by crossing the pond directly, we would want our agents to choose the bridge as it makes more sense. In other words, we want to make crossing the pond to be more navigationally expensive than using the bridge. In this section, we'll look at NavMesh areas, a way to define different layers with different navigation cost values.

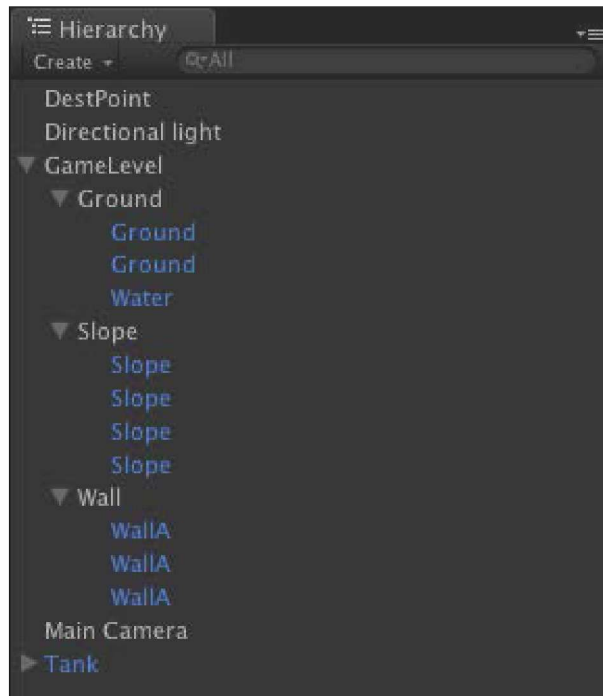
We're going to build a scene, as shown in the following screenshot:



Scene with layers

There'll be three planes to represent two ground planes connected with a bridge-like structure and a water plane between them. As you can see, it's the shortest path for our tank to cross over the water plane to reach our cube target, but we want our AI agents to choose the bridge if possible and to cross the water plane only if absolutely necessary, such as when the target object is on the water plane.

The scene hierarchy can be seen in the following screenshot. Our game level is composed of planes, slopes, and walls. We've a tank entity and a destination cube with the `Target.cs` script attached.



The Scene Hierarchy

As we saw earlier, NavMesh areas can be edited in the **Areas** tab of the **Navigation** window.

Unity comes with three default layers—Default, Not Walkable, and Jump—each with potentially different cost values. Let's add a new layer called Water and give it a cost of 5.

Next, select the water plane. Go to the **Navigation** window and under the **Object** tab, set **Navigation Area** to **Water**.



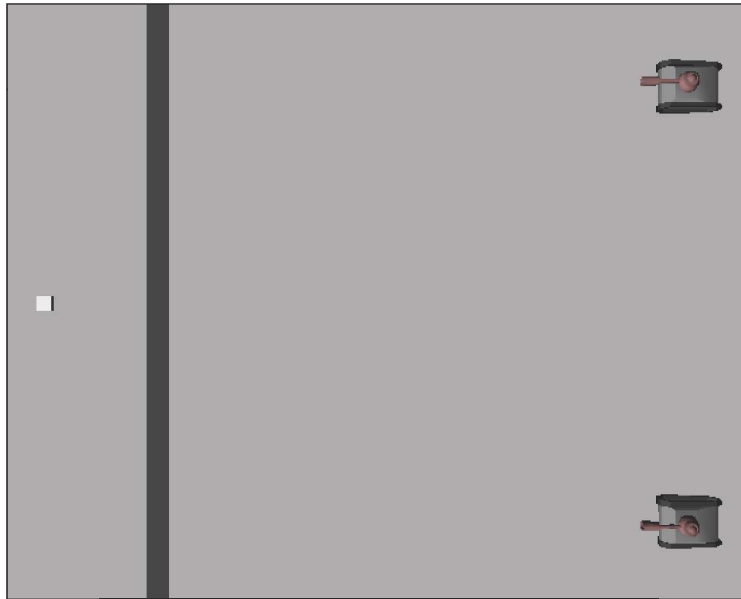
The Water area

Bake the NavMesh for the scene and run it to test it. You should see that the AI agents now choose the slope rather than going through the plane marked as the water layer because it's more expensive to choose this path. Try experimenting with placing the target object at different points in the water plane. You will see that the AI agents will sometimes swim back to the shore and take the bridge rather than trying to swim all the way across the water.

Making sense of Off Mesh Links

Sometimes, there could be some gaps inside the scene that can make the navigation meshes disconnected. For example, our agents will not be able to find the path if our slopes are not connected to the walls in our previous examples. Or, we could have set up points where our agents could jump off the wall and onto the plane below. Unity has a feature called **Off Mesh Links** to connect such gaps. Off Mesh Links can either be set up manually or generated automatically by Unity's NavMesh generator.

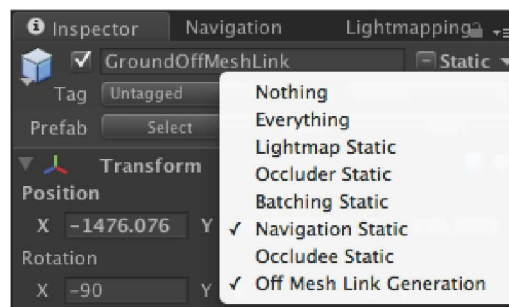
Here's the example scene that we're going to build in this example. As you can see, there's a small gap between the two planes. Let's see how to connect these two planes using Off Mesh Links.



Scene with Off Mesh Links

Using the generated Off Mesh Links

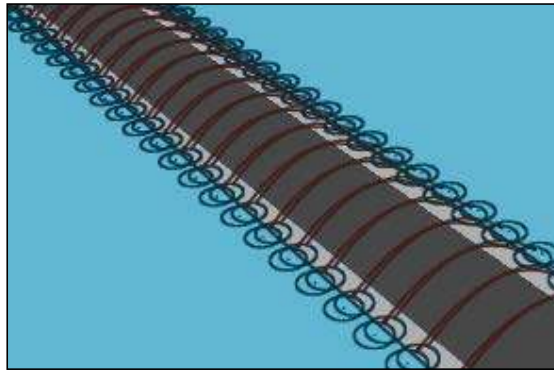
Firstly, we'll use the autogenerated Off Mesh Links to connect the two planes. The first thing to do is to mark these two planes as the **Off Mesh Link Generation** static in the property inspector, as shown in the following screenshot:



Off Mesh Link Generation static

You can set the distance threshold to autogenerate Off Mesh Links in the **Bake** tab of the **Navigation** window as seen earlier.

Click on **Bake**, and you should have Off Mesh Links connecting two planes like this:

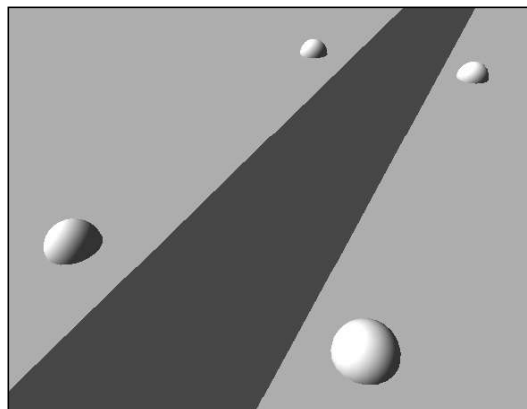


Generated Off Mesh Links

Now our AI agents can traverse and find the path across both planes. Agents will be essentially teleported to the other plane once they have reached the edge of the plane and found the Off Mesh Link. Unless having a teleporting agent is what you want, it might be a good idea to place a bridge to allow the agent to cross.

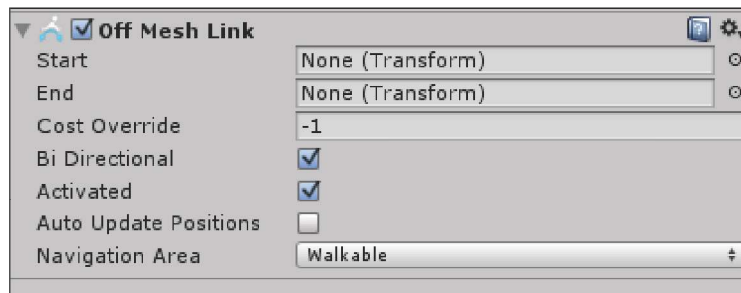
Setting the manual Off Mesh Links

If we don't want to generate Off Mesh Links along the edge, and want to force the agents to come to a certain point to be teleported to another plane, we can also manually set up the Off Mesh Links. Here's how:

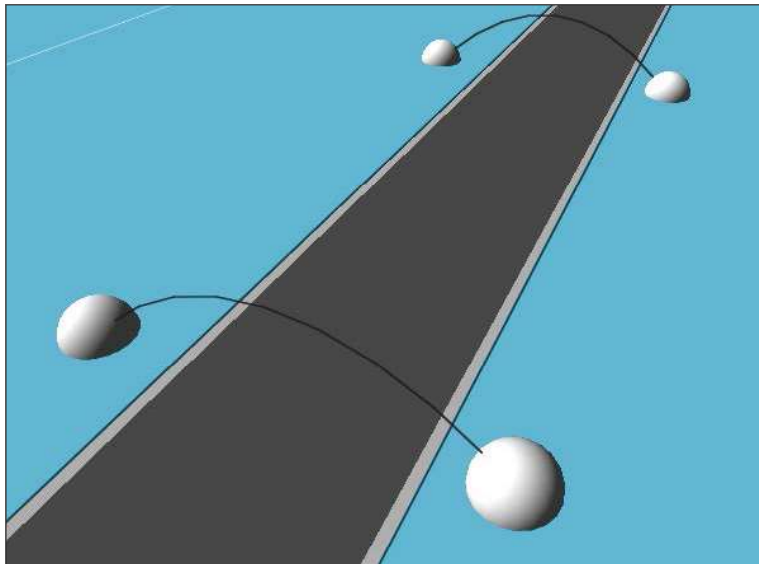


The manual Off Mesh Links setup

This is our scene with a significant gap between two planes. We placed two pairs of sphere entities on both sides of the plane. Choose a sphere, and add an Off Mesh Link by navigating to **Component | Navigation | Off Mesh Link**. We only need to add this component on one sphere. Next, drag-and-drop the first sphere to the **Start** property, and the other sphere to the **End** property.



The Off Mesh Link component



The manual Off Mesh Links generated

Go to the **Navigation** window and bake the scene. The planes are now connected with the manual Off Mesh Links that can be used by AI agents to traverse even though there's a gap.

Summary

You could say we navigated through quite a bit of content in this chapter. We started with a basic waypoint-based system, then learned how to implement our own simple A* Pathfinding system, and finally moved onto Unity's built-in navigation system. While many would opt to go with the simplicity of Unity's NavMesh system, others may find the granular control of a custom A* implementation more appealing. What is most important, however, is understanding when and how to use these different systems.

Furthermore, without even realizing it, we saw how these systems can integrate with other concepts we learned earlier in the book.

In the next chapter, *Flocks and Crowds*, we'll expand on these concepts and learn how we can simulate entire groups of agents moving in unison in a believable and performant fashion.

