11
Advanced NavMesh Generation

Navigation mesh generation is one of the most important topics in game AI. We have been using navigation meshes in almost all the chapters in this book, but haven't looked at them in detail. In this chapter, we will provide a more detailed overview of navigation meshes and look at the algorithm used to generate them. Then, we'll look at different options of customizing our navigation meshes better.

In this chapter, you will learn about:

- The working of navigation mesh generation and the algorithm behind it
- Advanced options for customizing navigation meshes
- Creating advanced navigation meshes with RAIN

An overview of a NavMesh

To use navigation meshes effectively, also referred to as **NavMeshes**, the first things we need to know are what exactly navigation meshes are and how they are created. A navigation mesh is a definition of the area an AI character can travel to in a level. It is a mesh, but it is not intended to be rendered or seen by the player; instead, it is used by the AI system. A NavMesh usually does not cover all the area in a level (if it did, we wouldn't need one) as it's just the area a character can walk. The mesh is also almost always a simplified version of the geometry. For instance, you could have a cave floor in a game with thousands of polygons along the bottom that show different details in the rock; however, for the navigation mesh, the areas would just be a handful of very large polygons that give a simplified view of the level. The purpose of a navigation mesh is to provide this simplified representation to the rest of the AI system as a way to find a path between two points on a level for a character. This is its purpose; let's discuss how they are created.

It used to be a common practice in the games industry to create navigation meshes manually. A designer or artist would take the completed level geometry and create one using standard polygon mesh modeling tools and save it. As you might imagine, this allowed for nice, custom, efficient meshes, was also a time sink, as every time the level changed, the navigation mesh would need to be manually edited and updated. In recent years, there has been more research in automatic navigation mesh generation.

There are many approaches to automatic navigation mesh generation, but the most popular is **Recast**, originally developed and designed by Mikko Monomen. Recast takes in level geometry and a set of parameters that define the character, such as the size of the character and how big of steps it can take, and then does a multipass approach to filter and creates the final NavMesh. The most important phase of this is **voxelizing** the level based on an inputted cell size. This means the level geometry is divided into voxels (cubes), creating a version of the level geometry where everything is partitioned into different boxes called cells. Then, the geometry in each of these cells is analyzed and simplified based on its intersection with the sides of the boxes and is culled based on things such as the slope of the geometry or how big a step height is between geometry. This simplified geometry is then merged and triangulated to make a final navigation mesh that can be used by the AI system.

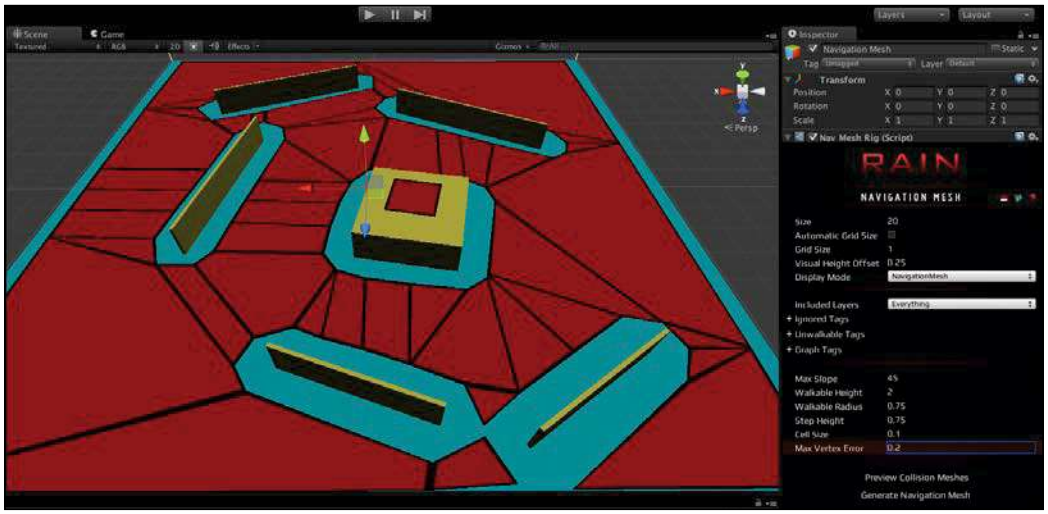
The source code and more information on the original C++ implementation of Recast is available at `https://github.com/memononen/recastnavigation`.

Advanced NavMesh parameters

Now that we know how navigation mesh generations works, let's look at the different parameters you can set to generate them in more detail.

We'll look at how to do these parameters with RAIN using the following steps:

1. Open one of our previous scenes or create a new one with a floor and some blocks for walls.
2. Then, go to **RAIN** | **Create NavMesh**. Also, right-click on the **RAIN** menu and choose **Show Advanced Settings**. The setup should look something like the following screenshot:

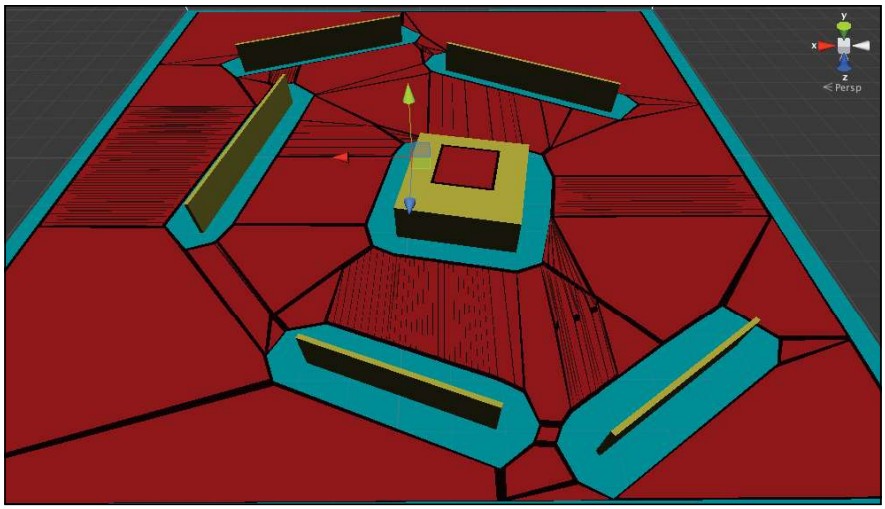


The NavMesh setup

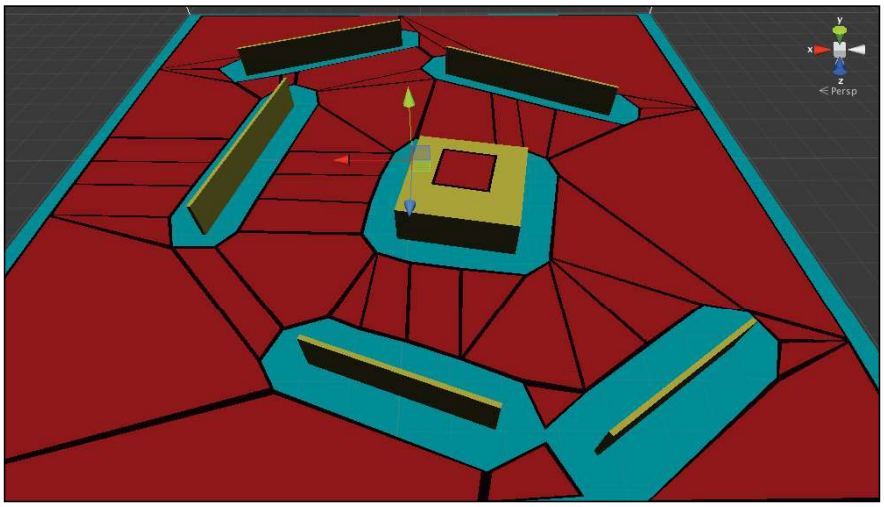
Now let's look at some of the important parameters:

- **Size**: This is the overall size of the navigation mesh. You'll want the navigation mesh to cover your entire level and use this parameter instead of trying to scale up the navigation mesh through the **Scale** transform in the **Inspector** window. For our demo here, set the **Size** parameter to **20**.
- **Walkable Radius**: This is an important parameter to define the character size of the mesh. Remember, each mesh will be matched to the size of a particular character, and this is the radius of the character. You can visualize the radius for a character by adding a Unity **Sphere Collider** script to your object (by going to **Component | Physics | Sphere Collider**) and adjusting the radius of the collider.
- **Cell Size**: This is also a very important parameter. During the voxel step of the Recast algorithm, this sets the size of the cubes to inspect the geometry. The smaller the size, the more detailed and finer the mesh, but the longer the processing time for Recast. A large cell size makes computation fast but loses detail.

For example, here is a NavMesh from our demo with a cell size of **0.01**:



You can see the finer detail here. The following is the navigation mesh generated with a cell size of **0.1**:



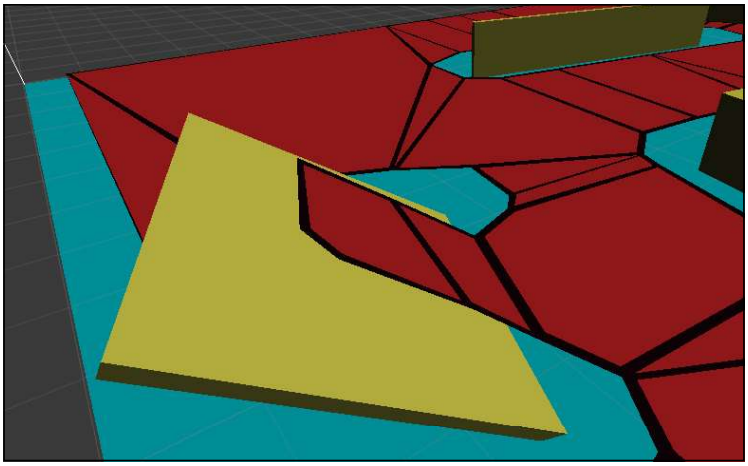
Note the difference between the two screenshots. In the former, walking through the two walls lower down in our picture is possible, but in the latter with a larger cell size, there is no path even though the character radius is the same. Problems like this become greater with larger cell sizes. The following is a navigation mesh with a cell size of **1**:

As you can see, the detail becomes jumbled and the mesh itself becomes unusable. With such differing results, the big question is how large should a cell size be for a level? The answer is that it depends on the required result. However, one important consideration is that as the processing time to generate one is done during development and not at runtime, even if it takes several minutes to generate a good mesh then it can be worth it to get a good result in the game.

Setting a small cell size on a large level can cause mesh processing to take a significant amount of time and consume a lot of memory. It is a good practice to save the scene before attempting to generate a complex navigation mesh.

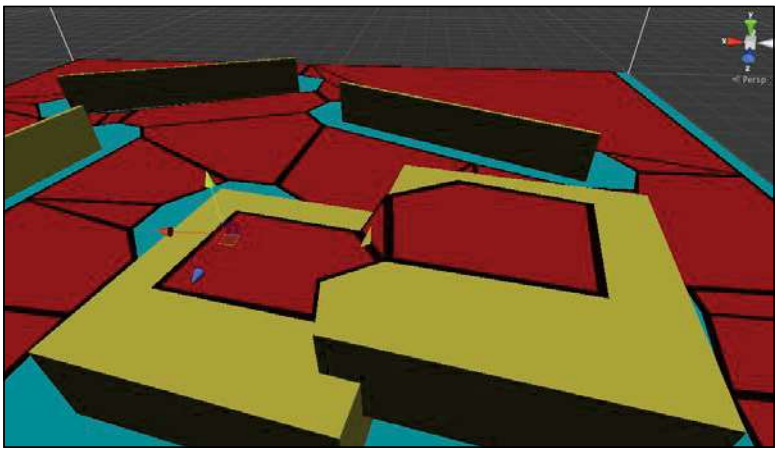
The **Size**, **Walkable Radius**, and **Cell Size** parameters are the most important parameters when generating the navigation mesh, but there are more that are used to customize the mesh further:

- **Max Slope**: This is the largest slope that a character can walk on. This is how much a piece of geometry that is tilted can still be walked on. If you take the wall and rotate it, you can see it is walkable:



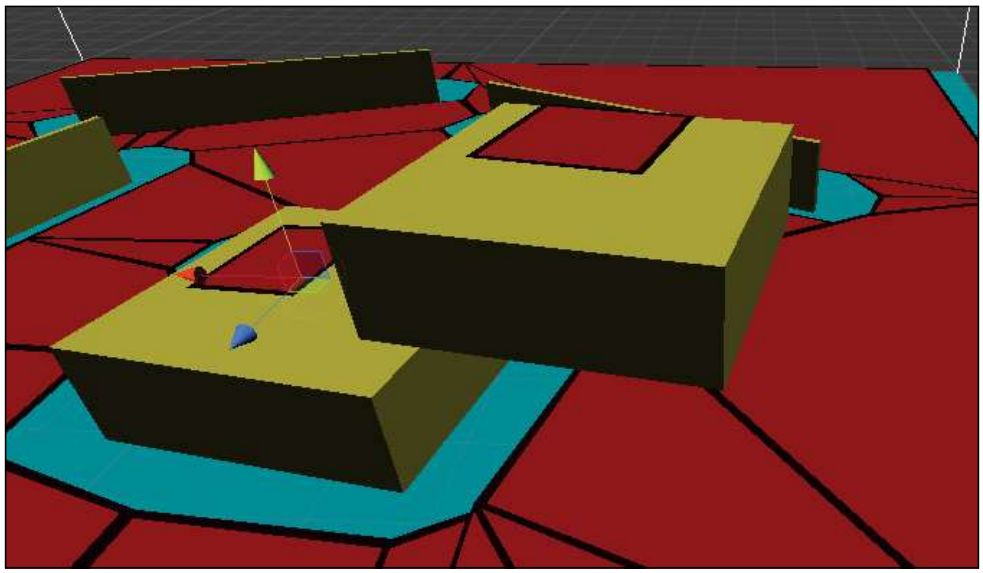
The preceding is a screenshot of a walkable object with a slope.

- **Step Height**: This is how high a character can step from one object to another. For example, if you have steps between two blocks, as shown in the following screenshot, this would define how far in height the blocks can be apart and whether the area is still considered walkable:



This is a screenshot of the navigation mesh with the step height set to connect adjacent blocks.

- **Walkable Height**: This is the vertical height that is needed for the character to walk. For example, in the previous screenshot, the second block is not walkable underneath because of the walkable height. If you raise it to at least one unit off the ground and set the walkable height to **1**, the area underneath would become walkable:



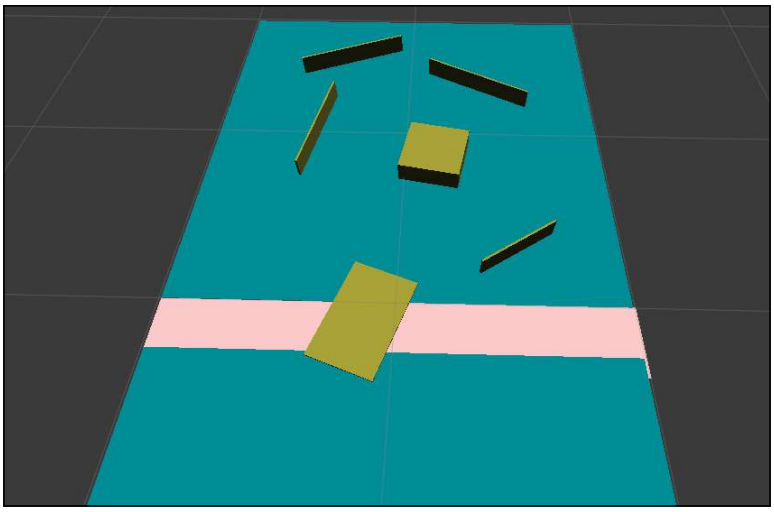
You can see a screenshot of the navigation mesh with the walkable height set to allow going under the higher block.

These are the most important parameters. There are some other parameters related to the visualization and to cull objects. We will look at culling more in the next section.

Culling areas

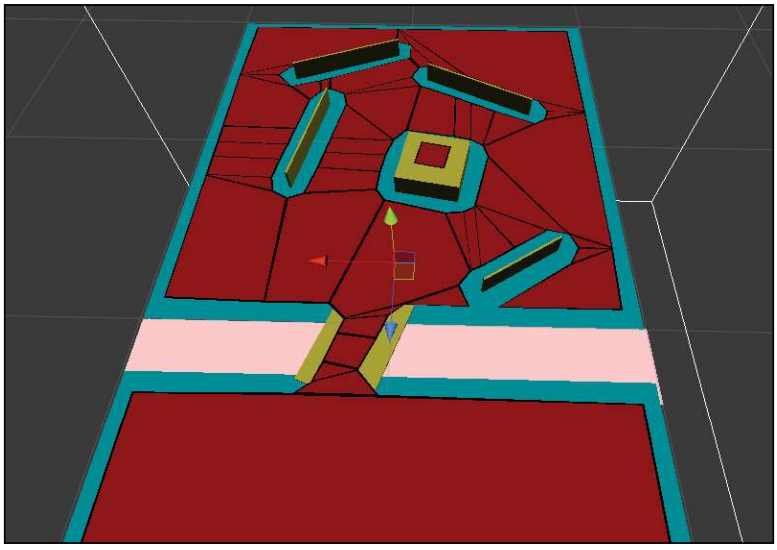
Being able to set up areas as walkable or not is an important part of creating a level. To demo this, let's divide the level into two parts and create a bridge between the two. Take our demo and duplicate the floor and pull it down. Then transform one of the walls to a bridge. Then, add two other pieces of geometry to mark areas that are dangerous to walk on, like lava.

Here is an example setup:



This is a basic scene with a bridge to cross.

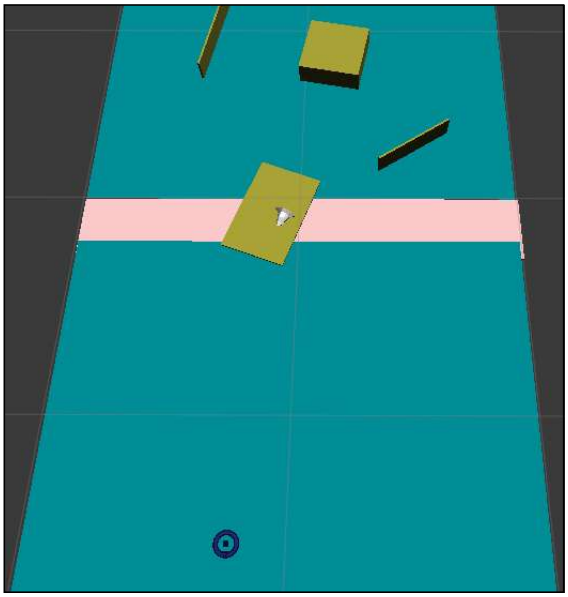
If you recreate the navigation mesh now, all the geometry will be covered and the bridge won't be recognized. To fix this, you can create a new tag called `Lava` and tag the geometry under the bridge with it. Then, in the navigation meshes' RAIN component, add `Lava` to the unwalkable tags. If you then regenerate the mesh, only the bridge is walkable. This is a screenshot of a navigation mesh with polygons that are under the bridge culled out:



To see this in action, create a new ship and add a target to the scene on different sizes of the bridge. Set the ship's behavior tree to have a **move** node with the target, as shown in the following screenshot:

The preceding screenshot shows a basic **move** node to a navigation point behavior tree.

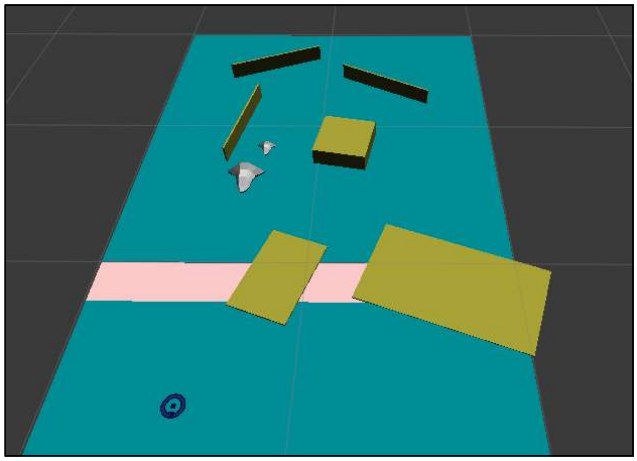
If you run the demo now, you will see the ship cross the bridge:



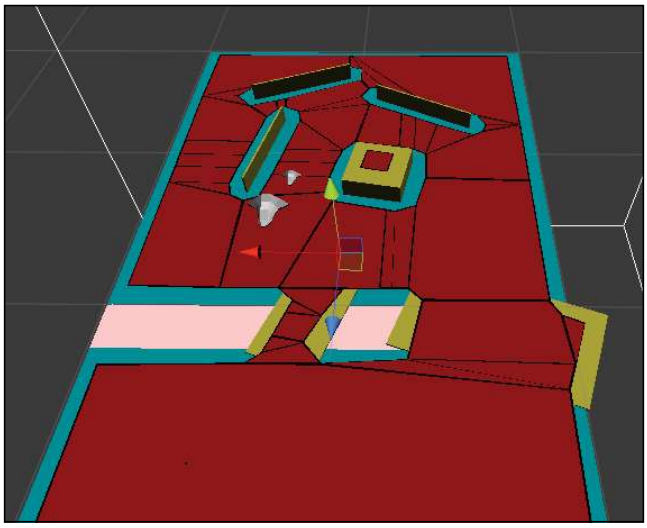
The preceding screenshot shows the ship crossing the bridge to go to its navigation target. Using layers and the walkable tag, you can customize navigation meshes.

Multiple navigation meshes

So far, we have only looked at setting up a single navigation mesh in a scene, but navigation meshes are designed to be per character and not just one for the entire scene. We need multiple navigation meshes, but there is no field to directly set which navigation mesh to use for a character. Instead, RAIN uses a field called **graph tags** to correlate meshes with characters. To see how this works, let's add a second bridge to our scene that is larger and a second ship with double the scale. Here is an example setup:

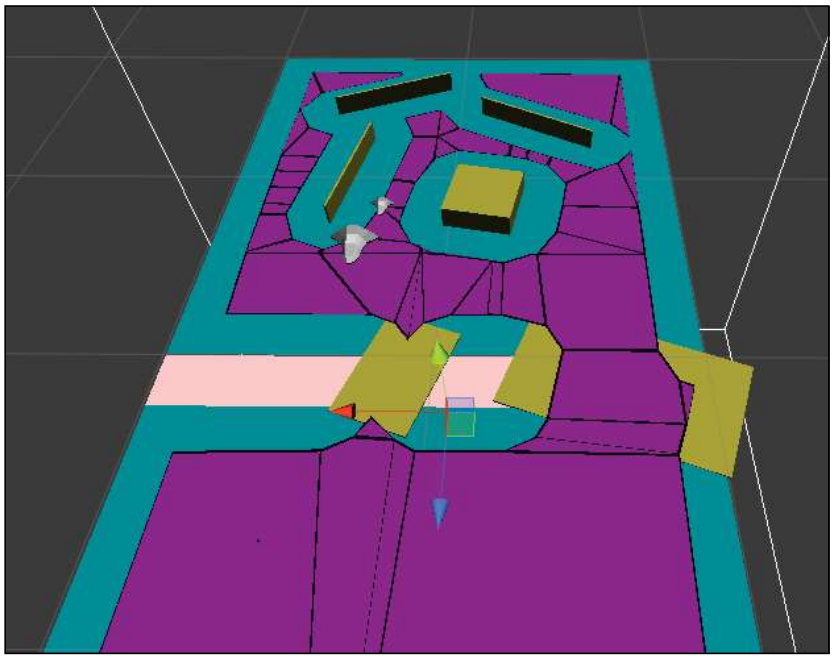


This is a demo scene setup with an additional larger ship and larger bridge. Regenerating the mesh gives us a path over both bridges:



This is a navigation mesh with a smaller walkable radius that can cross both bridges.

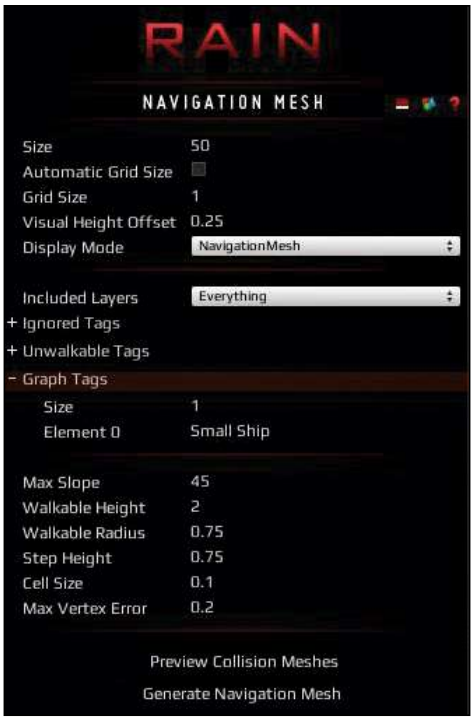
Any character using this mesh will be able to go across either bridge. For our larger character to not be able to cross the smaller bridge, we need to generate another mesh with a smaller **Walkable Radius**. Create a second navigation mesh in the scene (**RAIN | Create NavMesh**). Rename the first navigation mesh to `Navigation Mesh Small` and the new one to `Navigation Mesh Large`. In the large one, set the **Walkable Radius** parameter to **1.75**. Generate this mesh and see how it goes over the second bridge but not the first:



It is a navigation mesh with a larger walkable radius that can cross one bridge.

Then, to match the meshes with the characters for `Navigation Mesh Small`, make sure that you are in **Advanced Settings** and in **Graph Tags**, add an element called `Small Ship`.

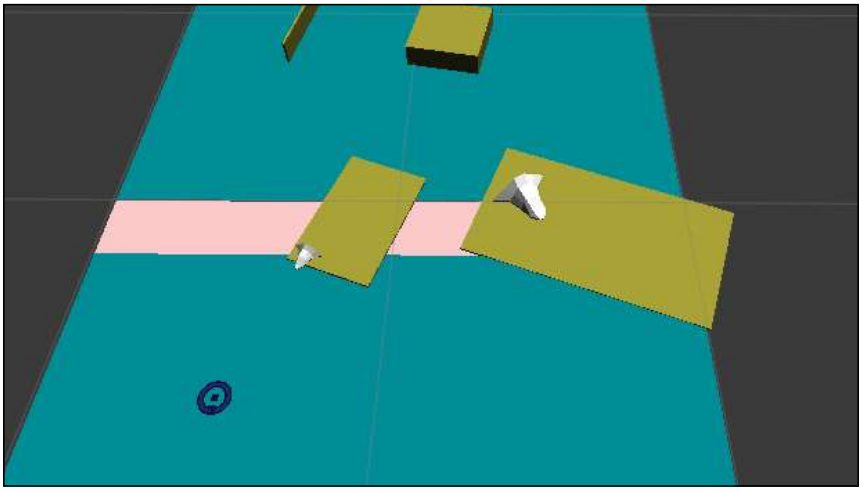
The following is a screenshot of a navigation mesh with a graph tag:



Do the same for `Navigation Mesh Large` with the **Graph Tags** field set to `Large Ship`. Then, for the smaller ship character AI, go to the **Navigator** tab and set its **Graph Tags** field to `Small Ship`:

This is a ship navigator with a graph tag setup.

Do the same with **Graph Tags** for larger ships. With the graph tags matched, if you run the demo now, the larger ship will not take the smaller bridge and go through the smaller one:



This is a demo with two ships. Only the larger ship can cross the larger bridge.

Summary

Navigation meshes are an important part of game AI. In this chapter, we looked at the different parameters to customize navigation meshes. We looked at things such as setting the character size and walkable slopes and discussed the importance of the cell size parameter. We then saw how to customize our mesh by tagging different areas as not walkable and how to set up multiple navigation meshes for different characters.

We now have all the essential skills we need to create AI in Unity. We've seen how to have a character move, navigate, and sense other characters in our game scenes as well as how to set up behavior trees to make decisions and integrate animation. We also looked at different AI use cases, such as crowds, driving, and had our characters attack and change behavior based on game events. This covers a lot about AI, but game AI is a huge and much studied topic and there is much more to learn. By doing some searching, you'll find that there are many online articles, textbooks, and conference talks that you can study to make even more advanced AI.