1 Navigation and Pathfinding

The Navigation system allows you to create characters that can intelligently move in the game world. The navigation system uses navigation meshes to reason about the environment. The navigation meshes are created automatically from your Scene geometry. Dynamic obstacles allow altering the navigation of the characters at runtime, and off-mesh links let you build specific actions such as opening doors, or jumping down from a ledge. This section describes Unity’s navigation and pathfinding in detail.

The Navigation System allows you to create characters which can navigate the game world. It gives your characters the ability to understand that they need to take stairs to reach second floor, or to jump to get over a ditch. The Unity NavMesh system consists of the following pieces:

- NavMesh (short for Navigation Mesh) is a data structure which describes the walkable surfaces of the game world and allows to find path from one walkable location to another in the game world. The data structure is built, or baked, automatically from your level geometry.

- NavMesh Agent component help you to create characters which avoid each other while moving towards their goal. Agents reason about the game world using the NavMesh and they know how to avoid each other as well as moving obstacles.

- Off-Mesh Link component allows you to incorporate navigation shortcuts which cannot be represented using a walkable surface. For example, jumping over a ditch or a fence, or opening a door before walking through it, can be all described as Off-mesh links.
• NavMesh Obstacle component allows you to describe moving obstacles the agents should avoid while navigating the world. A barrel or a crate controlled by the physics system is a good example of an obstacle. While the obstacle is moving the agents do their best to avoid it, but once the obstacle becomes stationary it will carve a hole in the navmesh so that the agents can change their paths to steer around it, or if the stationary obstacle is blocking the path way, the agents can find a different route.

2 Building a NavMesh

The process of creating a NavMesh from the level geometry is called NavMesh Baking. The process collects the Render Meshes and Terrains of all Game Objects which are marked as Navigation Static, and then processes them to create a navigation mesh that approximates the walkable surfaces of the level.

In Unity, NavMesh generation is handled from the Navigation window (menu: Window > Navigation).

Building a NavMesh for your scene can be done in 4 quick steps:

1. Select scene geometry that should affect the navigation - walkable surfaces and obstacles.

2. Click to include selected objects in the NavMesh baking process.

3. Adjust the bake settings to match your agent size.

4. Click bake to build the navmesh.

• Select scene geometry that should affect the navigation - walkable surfaces and obstacles.
- Check Navigation Static on to include selected objects in the NavMesh baking process.

- Adjust the bake settings to match your agent size. *Agent Radius defines how close the agent center can get to a wall or a ledge.*  
  *Agent Height defines how low the spaces are that the agent can reach.*  
  *Max Slope defines how steep the ramps are that the agent walk up.*  
  *Step Height defines how high obstructions are that the agent can step on.*

- Click bake to build the NavMesh.

The resulting NavMesh will be shown in the scene as a blue overlay on the underlying level geometry whenever the Navigation Window is open and visible.

As you may have noticed in the above pictures, the the walkable area in the generated NavMesh appears shrunk. The NavMesh represents the area where the center of the agent can move. Conceptually, it doesn’t matter whether you regard the agent as a point on a shrunken NavMesh or a circle on a full-size NavMesh since the two are equivalent. However, the point interpretation allows for better runtime efficiency and also allows the designer to see immediately whether an agent can squeeze through gaps without worrying about its radius.

Another thing to keep in mind is that the NavMesh is an approximation of the walkable surface. This can be seen for example in the stairs which are represented as a flat surface, while the source surface has steps. This is done in order to keep the NavMesh data size small. The side effect of the approximation is that sometimes you will need to have a little extra space in your level geometry to allows the agent to pass through a tight spot.

### 3 Creating a NavMesh Agent

Once you have a NavMesh baked for your level it is time to create a character which can navigate the scene. We’re going to build our prototype agent from a cylinder and set it in motion. This is done using a NavMesh Agent component and a simple script.

First let’s create the character:

Create a cylinder: GameObject > 3D Object > Cylinder. The default cylinder dimensions (height 2 and radius 0.5) are good for a humanoid shaped agent, so we will leave them as they are. Add a NavMesh Agent component: Component > Navigation > NavMesh Agent. Now you have simple NavMesh Agent set up ready to receive commands!

When you start to experiment with a NavMesh Agent, you most likely are going to adjust its dimensions for your character size and speed.

The NavMesh Agent component handles both the pathfinding and the movement control of a character. In your scripts, navigation can be as simple as setting the desired destination point - the NavMesh Agent can handle everything from there on. Next we need to build a simple script which allows you to send your character to the destination specified by another Game Object, and a Sphere which will be the destination to move to:

1. Create a new C# script and replace its contents with the above script.

2. Assign the MoveTo script to the character you’ve just created.

3. Create a sphere, this will be the destination the agent will move to.
4. Move the sphere away from the character to a location that is close to the NavMesh surface.

5. Select the character, locate the MoveTo script, and assign the Sphere to the Goal property.

6. Press Play; you should see the agent navigating to the location of the sphere.

4 NavMesh Obstacle

NavMesh Obstacle components can be used to describe obstacles the agents should avoid while navigating. For example, the agents should avoid physics controlled objects, such as crates and barrels while moving.

We’re going to add a crate to block the pathway at the top of the level.

- First create a cube to depict the crate: Game Object > 3D Object > Cube.
- Move the cube to the platform at the top, the default size of the cube is good for a crate so leave it as it is.
• Add a NavMesh Obstacle component to the cube. Choose Add Component from the inspector and choose Navigation > NavMesh Obstacle.

• Set the shape of the obstacle to box, changing the shape will automatically fit the center and size to the render mesh.

• Add a Rigid body to the obstacle. Choose Add Component from the inspector and choose Physics > Rigid Body.

• Finally turn on the Carve setting from the NavMesh Obstacle inspector so that the agent knows to find a path around the obstacle.

• Now we have a working crate that is physics controlled, and which the AI knows how to avoid while navigating.

5 NavMesh Areas and Cost

The Navigation Areas define how difficult it is to walk across a specific area, the lower cost areas will be preferred during path finding. In addition each NavMesh Agent has an Area Mask which can be used to specify on which areas the agent can move. Unity uses A* to calculate the shortest path on the NavMesh. A* works on a graph of connected nodes. The algorithm starts from the nearest node to the path start and visits the connect nodes until the destination is reached.

Since the Unity navigation representation is a mesh of polygons, the first thing the pathfinder needs to do is to place a point on each polygon, which is the location of the node. The shortest path is then calculated between these nodes.
The yellow dots and lines in the above picture shows how the nodes and links are placed on the NavMesh, and in which order they are traversed during the A*.

The cost to move between two nodes depends on the distance to travel and the cost associated with the area type of the polygon under the link, that is, distance * cost. In practice this means, that if the cost of an area is 2.0, the distance across such polygon will appear to be twice as long. The A* algorithm requires that all costs must be larger than 1.0.

The effect of the costs on the resulting path can be hard to tune, especially for longer paths. The best way to approach costs is to treat them as hints. For example, if you want the agents to not to use Off-Mesh Links too often, you could increase their cost. But it can be challenging to tune a behavior where the agents to prefer to walk on sidewalks.

Another thing you may notice on some levels is that the pathfinder does not always choose the very shortest path. The reason for this is the node placement. The effect can be noticeable in scenarios where big open areas are next to tiny obstacles, which results navigation mesh with very big and small polygons. In such cases the nodes on the big polygons may get placed anywhere in the big polygon and from the pathfinder’s point of view it looks like a detour.

The cost per area type can be set globally in the Areas tab, or you can override them per agent using a script. The area types are specified in the Navigation Window’s Areas tab. There are 29 custom types, and 3 built-in types: Walkable, Not Walkable, and Jump.

- Walkable is a generic area type which specifies that the area can be walked on.

- Not Walkable is a generic area type which prevents navigation. It is useful for cases where you want to mark certain object to be an obstacle, but without getting NavMesh on top of it.
• Jump is an area type that is assigned to all auto-generated Off-Mesh Links.

If several objects of different area types are overlapping, the resulting navmesh area type will generally be the one with the highest index. There is one exception however: Not Walkable always takes precedence. Which can be helpful if you need to block out an area.

Each agent has an Area Mask which describes which areas it can use when navigating. The area mask can be set in the agent properties, or the bitmask can be manipulated using a script at runtime.

The area mask is useful when you want only certain types characters to be able to walk through an area. For example, in a zombie evasion game, you could mark the area under each door with a Door area type, and uncheck the Door area from the zombie character's Area Mask.

6 Move to a clicked point

This script lets you choose the destination point on the NavMesh by clicking the mouse on the object's surface. The position of the click is determined by a raycast, rather like pointing a laser beam at the object to see where it hits (see the page Rays from the Camera for a full description of this technique). Since the GetComponent function is fairly slow to execute, the script stores its result in a variable during the Start function rather than call it repeatedly in Update.

```csharp
// MoveToClickPoint.cs
using UnityEngine;

public class MoveToClickPoint : MonoBehaviour
{
    NavMeshAgent agent;

    void Start()
    {
        agent = GetComponent<NavMeshAgent>();
    }

    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            RaycastHit hit;

            if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition), out hit, 100))
            {
                agent.destination = hit.point;
            }
        }
    }
}
```

7 Patrol between a set of points

Many games feature NPCs that patrol automatically around the playing area. The navigation system can be used to implement this behaviour but it is slightly more involved than standard pathfinding - merely using the shortest path between two points makes for a limited and predictable patrol route. You can get a more convincing patrol pattern by keeping a set of key points that are "useful" for the NPC to pass through and visiting them in some kind of sequence. For example, in a maze, you might place the key patrol points at junctions and corners to ensure the agent checks
every corridor. For an office building, the key points might be the individual offices and other rooms.

The ideal sequence of patrol points will depend on the way you want the NPCs to behave. For example, a robot would probably just visit the points in a methodical order while a human guard might try to catch the player out by using a more random pattern. The simple behaviour of the robot can be implemented using the code shown below.

The patrol points are supplied to the script using a public array of Transforms. This array can be assigned from the inspector using GameObjects to mark the points’ positions. The GotoNextPoint function sets the destination point for the agent (which also starts it moving) and then selects the new destination that will be used on the next call. As it stands, the code cycles through the points in the sequence they occur in the array but you can easily modify this, say by using Random.Range to choose an array index at random.

In the Update function, the script checks how close the agent is to the destination using the remainingDistance property. When this distance is very small, a call to GotoNextPoint is made to start the next leg of the patrol.

```csharp
public class Patrol : MonoBehaviour {
    public Transform[] points;
    private int destPoint = 0;
    private NavMeshAgent agent;

    void Start () {
        agent = GetComponent<NavMeshAgent> ();

        // Disabling auto-braking allows for continuous movement
        // between points (ie, the agent doesn’t slow down as it
        // approaches a destination point).
        agent.autoBraking = false;

        GotoNextPoint();
    }

    void GotoNextPoint() {
        // Returns if no points have been set up
        if (points.Length == 0)
            return;

        // Set the agent to go to the currently selected destination.
        agent.destination = points[destPoint].position;

        // Choose the next point in the array as the destination,
        // cycling to the start if necessary.
        destPoint = (destPoint + 1) % points.Length;
    }

    void Update () {
        // Choose the next destination point when the agent gets
        // close to the current one.
        if (agent.remainingDistance < 0.5f)
            GotoNextPoint();
    }
}
```