2
Navigation

In this chapter, we will cover the following recipes:

- Representing the world with grids
- Representing the world with Dirichlet domains
- Representing the world with points of visibility
- Representing the world with a self-made navigation mesh
- Finding your way out of a maze with DFS
- Finding the shortest path in a grid with BFS
- Finding the shortest path with Dijkstra
- Finding the best-promising path with A*
- Improving A* for memory: IDA*
- Planning navigation in several frames: time-sliced search
- Smoothing a path

Introduction

In this chapter, we will learn path-finding algorithms for navigating complex scenarios. Game worlds are usually complex structures; whether a maze, an open world, or everything in between. That's why we need different techniques for approaching these kinds of problems.

We'll learn some ways of representing the world using different kinds of graph structures, and several algorithms for finding a path, each aimed at different situations.

It is worth mentioning that path-finding algorithms rely on techniques such as `Seek` and `Arrive`, learnt in the previous chapter, in order to navigate the map.

Representing the world with grids

A grid is the most used structure for representing worlds in games because it is easy to implement and visualize. However, we will lay the foundations for advanced graph representations while learning the basis of graph theory and properties.

Getting ready

First, we need to create an abstract class called `Graph`, declaring the virtual methods that every graph representation implements. It is done this way because, no matter how the vertices and edges are represented internally, the path-finding algorithms remain high-level, thus avoiding the implementation of the algorithms for each type of graph representation.

This class works as a parent class for the different representations to be learned in the chapter and it's a good starting point if you want to implement graph representations not covered in the book.

The following is the code for the `Graph` class:

1. Create the backbone with the member values:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public abstract class Graph : MonoBehaviour
{
    public GameObject vertexPrefab;
    protected List<Vertex> vertices;
    protected List<List<Vertex>> neighbours;
    protected List<List<float>> costs;
    // next steps
}
```

2. Define the Start function:

```
public virtual void Start()
{
    Load();
}
```

3. Define the Load function, mentioned previously:

```
public virtual void Load() { }
```

4. Implement the function for getting the graph's size:

```
public virtual int GetSize()
{
    if (ReferenceEquals(vertices, null))
        return 0;
    return vertices.Count;
}
```

5. Define the function for finding the nearest vertex given a position:

```
public virtual Vertex GetNearestVertex(Vector3 position)
{
    return null;
}
```

6. Implement the function for getting the vertex given its ID:

```
public virtual Vertex GetVertexObj(int id)
{
    if (ReferenceEquals(vertices, null) || vertices.Count == 0)
        return null;
    if (id < 0 || id >= vertices.Count)
        return null;
    return vertices[id];
}
```

7. Implement the function for retrieving a vertex' neighbours:

```
public virtual Vertex[] GetNeighbours(Vertex v)
{
    if (ReferenceEquals(neighbours, null) || neighbours.Count ==
0)
        return new Vertex[0];
    if (v.id < 0 || v.id >= neighbours.Count)
        return new Vertex[0];
    return neighbours[v.id].ToArray();
}
```

We also need a `Vertex` class, with the following code:

```
using UnityEngine;
using System.Collections.Generic;
[System.Serializable]
public class Vertex : MonoBehaviour
{
    public int id;
    public List<Edge> neighbours;
    [HideInInspector]
    public Vertex prev;
}
```

Following, we need to create a class for storing a vertex' neighbours with their costs. This class will be called `Edge`, and let's implement it:

1. Create the `Edge` class, deriving from `IComparable`:

```
using System;

[System.Serializable]
public class Edge : IComparable<Edge>
{
    public float cost;
    public Vertex vertex;
    // next steps
}
```

2. Implement its constructor:

```
public Edge(Vertex vertex = null, float cost = 1f)
{
    this.vertex = vertex;
    this.cost = cost;
}
```

3. Implement the comparison member function:

```
public int CompareTo(Edge other)
{
    float result = cost - other.cost;
    int idA = vertex.GetInstanceID();
    int idB = other.vertex.GetInstanceID();
    if (idA == idB)
        return 0;
    return (int)result;
}
```

4. Implement the function for comparing two edges:

```
public bool Equals(Edge other)
{
    return (other.vertex.id == this.vertex.id);
}
```

5. Override the function for comparing two objects:

```
public override bool Equals(object obj)
{
    Edge other = (Edge)obj;
    return (other.vertex.id == this.vertex.id);
}
```

6. Override the function for retrieving the hash code. This is necessary when overriding the previous member function:

```
public override int GetHashCode()
{
    return this.vertex.GetHashCode();
}
```

Besides creating the previous classes, it's important to define a couple of prefabs based on the cube primitive in order to visualize the ground (maybe a low-height cube) and walls or obstacles. The prefab for the ground is assigned to the `vertexPrefab` variable and the wall prefab is assigned to the `obstaclePrefab` variable that is declared in the next section.

Finally, create a directory called `Maps` to store the text files for defining the maps.

How to do it...

Now, it's time to go in-depth and be concrete about implementing our grid graph. First, we implement all the functions for handling the graph, leaving space for your own text files, and in a following section we'll learn how to read `.map` files, which is an open format used by a lot of games:

1. Create the `GraphGrid` class deriving from Graph

```
using UnityEngine;
using System;
using System.Collections.Generic;
using System.IO;

public class GraphGrid : Graph
{
    public GameObject obstaclePrefab;
    public string mapName = "arena.map";
    public bool get8Vicinity = false;
    public float cellSize = 1f;
    [Range(0, Mathf.Infinity)]
    public float defaultCost = 1f;
    [Range(0, Mathf.Infinity)]
    public float maximumCost = Mathf.Infinity;
    string mapsDir = "Maps";
    int numCols;
    int numRows;
    GameObject[] vertexObjs;
    // this is necessary for
    // the advanced section of reading
```

```
    // from an example test file
    bool[,] mapVertices;
    // next steps
}
```

2. Define the `GridToId` and `IdToGrid` functions for transforming a position in the grid into a vertex index, and vice versa, respectively

```
private int GridToId(int x, int y)
{
    return Math.Max(numRows, numCols) * y + x;
}

private Vector2 IdToGrid(int id)
{
    Vector2 location = Vector2.zero;
    location.y = Mathf.Floor(id / numCols);
    location.x = Mathf.Floor(id % numCols);
    return location;
}
```

3. Define the `LoadMap` function for reading the text file:

```
private void LoadMap(string filename)
{
    // TODO
    // implement your grid-based
    // file-reading procedure here
    // using
    // vertices[i, j] for logical representation and
    // vertexObjs[i, j] for assigning new prefab instances
}
```

4. Override the LoadGraph function:

```
public override void LoadGraph()
{
    LoadMap(mapName);
}
```

5. Override the `GetNearestVertex` function. This is the traditional way, without considering that the resulting vertex is an obstacle. In the next steps we will learn how to do it better:

```
public override Vertex GetNearestVertex(Vector3 position)
{
    position.x = Mathf.Floor(position.x / cellSize);
    position.y = Mathf.Floor(position.z / cellSize);
```

```
    int col = (int)position.x;
    int row = (int)position.z;
    int id = GridToId(col, row);
    return vertices[id];
   }
```

6. Override the `GetNearestVertex` function. It's is based on the Breadth-First Search algorithm that we will learn in depth later in the chapter:

```
public override Vertex GetNearestVertex(Vector3 position)
{
    int col = (int)(position.x / cellSize);
    int row = (int)(position.z / cellSize);
    Vector2 p = new Vector2(col, row);
    // next steps
}
```

7. Define the list of explored positions (vertices) and the queue of position to be explored:

```
List<Vector2> explored = new List<Vector2>();
Queue<Vector2> queue = new Queue<Vector2>();
queue.Enqueue(p);
```

8. Do it while the queue still have elements to explore. Otherwise, return null:

```
do
{
    p = queue.Dequeue();
    col = (int)p.x;
    row = (int)p.y;
    int id = GridToId(col, row);
    // next steps
} while (queue.Count != 0);
return null;
```

9. Retrieve it immediately if it's a valid vertex:

```
if (mapVertices[row, col])
    return vertices[id];
```

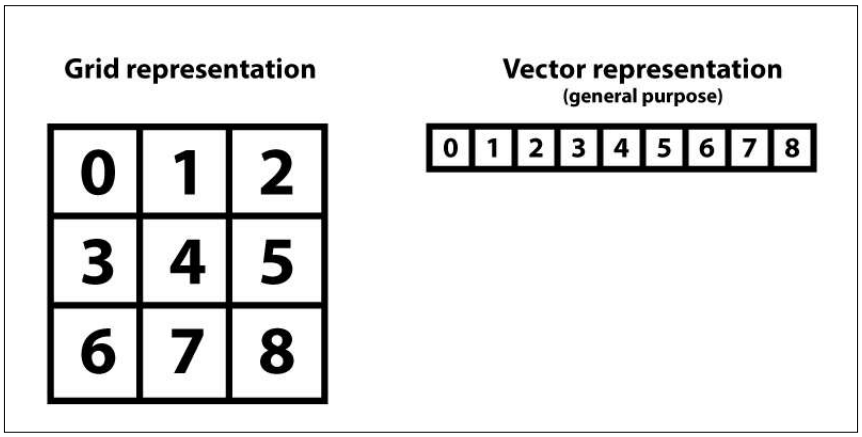
10. Add the position to the list of explored, if it's not already there:

```
if (!explored.Contains(p))
{
    explored.Add(p);
    int i, j;
    // next step
}
```

11. Add all its valid neighbors to the queue, provided they're valid:

```
for (i = row - 1; i <= row + 1; i++)
{
    for (j = col - 1; j <= col + 1; j++)
    {
        if (i < 0 || j < 0)
            continue;
        if (j >= numCols || i >= numRows)
            continue;
        if (i == row && j == col)
            continue;
        queue.Enqueue(new Vector2(j, i));
    }
}
```

How it works...



The algorithm makes use of its private functions in order to adapt itself to the general functions derived from the parent's class, and it relies on simple mathematical functions to convert from a two-dimensional vector position to a one-dimensional vector, or vertex index.

The `LoadMap` function is open to your own implementation, but in the next section we we'll learn a way to implement and read certain kinds of text files containing maps based on grids.

There's more...

We'll learn a way to implement the `LoadMap` function by using the `.map` file format as an example:

1. Define the function and create a `StreamReader` object for reading the file

```
private void LoadMap(string filename)
{
    string path = Application.dataPath + "/" + mapsDir + "/" +
filename;
    try
    {
        StreamReader strmRdr = new StreamReader(path);
        using (strmRdr)
        {
            // next steps in here
        }
    }
    catch (Exception e)
    {
        Debug.LogException(e);
    }
}
```

2. Declare and initialize the necessary variables

```
int j = 0;
int i = 0;
int id = 0;
string line;
Vector3 position = Vector3.zero;
Vector3 scale = Vector3.zero;
```

3. Read the header of the file containing its height and width

```
line = strmRdr.ReadLine();// non-important line
line = strmRdr.ReadLine();// height
numRows = int.Parse(line.Split(' ')[1]);
line = strmRdr.ReadLine();// width
numCols = int.Parse(line.Split(' ')[1]);
line = strmRdr.ReadLine();// "map" line in file
```

4. Initialize the member variables, allocating memory at the same time:

```
vertices = new List<Vertex>(numRows * numCols);
neighbours = new List<List<Vertex>>(numRows * numCols);
costs = new List<List<float>>(numRows * numCols);
vertexObjs = new GameObject[numRows * numCols];
   mapVertices = new bool[numRows, numCols];
```

5. Declare the for loop for iterating over the characters in the following lines

```
for (i = 0; i < numRows; i++)
{
    line = strmRdr.ReadLine();
    for (j = 0; j < numCols; j++)
    {
        // next steps in here
    }
}
```

6. Assign true or false to the logical representation depending on the character read

```
bool isGround = true;
if (line[j] != '.')
    isGround = false;
mapVertices[i, j] = isGround;
```

7. Instantiate the proper prefab

```
position.x = j * cellSize;
position.z = i * cellSize;
id = GridToId(j, i);
if (isGround)
    vertexObjs[id] = Instantiate(vertexPrefab, position,
Quaternion.identity) as GameObject;
else
    vertexObjs[id] = Instantiate(obstaclePrefab, position,
Quaternion.identity) as GameObject;
```

8. Assign the new game object as a child of the graph and clean-up its name

```
vertexObjs[id].name = vertexObjs[id].name.Replace("(Clone)",
id.ToString());
Vertex v = vertexObjs[id].AddComponent<Vertex>();
v.id = id;
vertices.Add(v);
neighbours.Add(new List<Vertex>());
costs.Add(new List<float>());
float y = vertexObjs[id].transform.localScale.y;
```

```
scale = new Vector3(cellSize, y, cellSize);
vertexObjs[id].transform.localScale = scale;
vertexObjs[id].transform.parent = gameObject.transform;
```

9. Create a pair of nested loops right after the previous loop, for setting up the neighbors for each vertex:

```
for (i = 0; i < numRows; i++)
{
    for (j = 0; j < numCols; j++)
    {
        SetNeighbours(j, i);
    }
}
```

10. Define the SetNeighbours function, called in the previous step:

```
protected void SetNeighbours(int x, int y, bool get8 = false)
{
    int col = x;
    int row = y;
    int i, j;
    int vertexId = GridToId(x, y);
    neighbours[vertexId] = new List<Vertex>();
    costs[vertexId] = new List<float>();
    Vector2[] pos = new Vector2[0];
    // next steps
}
```

11. Compute the proper values when we need vicinity of eight (top, bottom, right, left, and corners):

```
if (get8)
{
    pos = new Vector2[8];
    int c = 0;
    for (i = row - 1; i <= row + 1; i++)
    {
        for (j = col -1; j <= col; j++)
        {
            pos[c] = new Vector2(j, i);
            c++;
        }
    }
}
```

12. Set up everything for vicinity of four (no corners):

```
else
{
    pos = new Vector2[4];
    pos[0] = new Vector2(col, row - 1);
    pos[1] = new Vector2(col - 1, row);
    pos[2] = new Vector2(col + 1, row);
    pos[3] = new Vector2(col, row + 1);
}
```

13. Add the neighbors in the lists. It's the same procedure regarding the type of vicinity:

```
foreach (Vector2 p in pos)
{
    i = (int)p.y;
    j = (int)p.x;
    if (i < 0 || j < 0)
        continue;
    if (i >= numRows || j >= numCols)
        continue;
    if (i == row && j == col)
        continue;
    if (!mapVertices[i, j])
        continue;
    int id = GridToId(j, i);
    neighbours[vertexId].Add(vertices[id]);
    costs[vertexId].Add(defaultCost);
}
```

See also

For further information about the map's format used and getting free maps from several acclaimed titles, please refer to the *Moving AI Lab*'s website, led by Professor Sturtevant, available online at `http://movingai.com/benchmarks/`

Representing the world with Dirichlet domains

Also called a Voronoi polygon, a Dirichlet domain is a way of dividing space into regions consisting of a set of points closer to a given seed point than to any other. This graph representation helps in distributing the space using Unity's primitives or existing meshes, thus not really adhering to the definition, but using the concept as a means to an end. Dirichlet domains are usually mapped using cones for delimiting the area of a given vertex, but we're adapting that principle to our specific needs and tool.

Example of a Voronoi Diagram or Voronoi Polygon

Getting ready

Before building our new `Graph` class, it's important to create the `VertexReport` class, make some modifications to our `Graph` class, and add the `Vertex` tag in the project:

1. Prepend the `VertexReport` class to the `Graph` class specification, in the same file:

```
public class VertexReport
{
    public int vertex;
    public GameObject obj;
    public VertexReport(int vertexId, GameObject obj)
    {
```

```
            vertex = vertexId;
            this.obj = obj;
        }
    }
```

It's worth noting that the vertex objects in the scene must have a collider component attached to them, as well as the `Vertex` tag assigned. These objects can be either primitives or meshes, covering the maximum size of the area to be considered that vertex node.

How to do it...

This can be seen as a two-step recipe. First we define the vertex implementation and then the graph implementation, so everything works as intended:

1. First, create the `VertexDirichlet` class deriving from `Vertex`:

```
using UnityEngine;

public class VertexDirichlet : Vertex
{
    // next steps
}
```

2. Define the `OnTriggerEnter` function for registering the object in the current vertex:

```
public void OnTriggerEnter(Collider col)
{
    string objName = col.gameObject.name;
    if (objName.Equals("Agent") || objName.Equals("Player"))
    {
        VertexReport report = new VertexReport(id, col.
gameObject);
        SendMessageUpwards("AddLocation", report);
    }
}
```

3. Define the `OnTriggerExit` function for the inverse procedure

```
public void OnTriggerExit(Collider col)
{
    string objName = col.gameObject.name;
    if (objName.Equals("Agent") || objName.Equals("Player"))
    {
```

```
        VertexReport report = new VertexReport(id, col.
gameObject);
        SendMessageUpwards("RemoveLocation", report);
    }
}
```

4. Create the `GraphDirichlet` class deriving from `Graph`:

```
using UnityEngine;
using System.Collections.Generic;

public class GraphDirichlet : Graph
{
    Dictionary<int, List<int>> objToVertex;
}
```

5. Implement the `AddLocation` function we called before:

```
public void AddLocation(VertexReport report)
{
    int objId = report.obj.GetInstanceID();
    if (!objToVertex.ContainsKey(objId))
    {
        objToVertex.Add(objId, new List<int>());
    }
    objToVertex[objId].Add(report.vertex);
}
```

6. Define the `RemoveLocation` function as well:

```
public void RemoveLocation(VertexReport report)
{
    int objId = report.obj.GetInstanceID();
    objToVertex[objId].Remove(report.vertex);
}
```

7. Override the `Start` function to initialize the member variables:

```
public override void Start()
{
    base.Start();
    objToVertex = new Dictionary<int, List<int>>();
}
```

8. Implement the `Load` function for connecting everything:

```
public override void Load()
{
    Vertex[] verts = GameObject.FindObjectsOfType<Vertex>();
    vertices = new List<Vertex>(verts);
```

```
        for (int i = 0; i < vertices.Count; i++)
        {
            VertexVisibility vv = vertices[i] as VertexVisibility;
            vv.id = i;
            vv.FindNeighbours(vertices);
        }
    }
```

9. Override the `GetNearestVertex` function:

```
public override Vertex GetNearestVertex(Vector3 position)
{
    Vertex vertex = null;
    float dist = Mathf.Infinity;
    float distNear = dist;
    Vector3 posVertex = Vector3.zero;
    for (int i = 0; i < vertices.Count; i++)
    {
        posVertex = vertices[i].transform.position;
        dist = Vector3.Distance(position, posVertex);
        if (dist < distNear)
        {
            distNear = dist;
            vertex = vertices[i];
        }
    }
    return vertex;
}
```

10. Define the `GetNearestVertex` function, this time with a GameObject as input:

```
public Vertex GetNearestVertex(GameObject obj)
{
    int objId = obj.GetInstanceID();
    Vector3 objPos = obj.transform.position;
    if (!objToVertex.ContainsKey(objId))
        return null;
    List<int> vertIds = objToVertex[objId];
    Vertex vertex = null;
    float dist = Mathf.Infinity;
    for (int i = 0; i < vertIds.Count; i++)
    {
        int id = vertIds[i];
        Vertex v = vertices[id];
        Vector3 vPos = v.transform.position;
        float d = Vector3.Distance(objPos, vPos);
```

```
            if (d < dist)
            {
                vertex = v;
                dist = d;
            }
        }
        return vertex;
    }
```

11. Implement the `GetNeighbors` function:

```
public override Vertex[] GetNeighbours(Vertex v)
{
    List<Edge> edges = v.neighbours;
    Vertex[] ns = new Vertex[edges.Count];
    int i;
    for (i = 0; i < edges.Count; i++)
    {
        ns[i] = edges[i].vertex;
    }
    return ns;
}
```

12. Finally, define the `GetEdges` function:

```
public override Edge[] GetEdges(Vertex v)
{
    return vertices[v.id].neighbours.ToArray();
}
```

How it works...

When the agents or players enter into the area of a vertex, it sends a message to the graph parent class, and indexes that vertex into the proper dictionary of objects, making the appropriate quantization easier. The same inverse principle applies when the player leaves the area. When the player is mapped into more than one vertex, the function returns the index of the closest one.

Also, we're using a dictionary to facilitate the process of translating object instance IDs to the indices of our vertex array.

There's more...

Take into account that placing the vertices and making the connections between them (edges) must be done manually using the implemented method. You're encouraged to implement a way for getting a vertex's neighbors aimed at your own project if you need a more user-friendly (or automated) technique.

Finally, we'll explore is an automated way to get a vertex's neighbors in the next recipe, using ray casting that will probably serve as a starting point.

See also

- The *Representing the world with points of visibility* recipe

Representing the world with points of visibility

This is another widely-used technique for world representation based on points located throughout the valid area of navigation, whether manually placed or automated via scripting. We'll be using manually-placed points connected automatically via scripting.

Getting ready

Just like the previous representation, it's important to have several things in order before continuing:

- Having the `Edge` class prepended to the `Graph` class in the same file
- Defining the `GetEdges` function in the `Graph` class
- Having the `Vertex` class

The vertex objects in the scene must have a collider component attached to them, as well as the `Vertex` tag assigned. It's recommended for them to be unitary `Sphere` primitives.

How to do it...

We'll be creating the graph representation class as well as a custom `Vertex` class:

1. Create the `VertexVisibility` class deriving from `Vertex`:

```
using UnityEngine;
using System.Collections.Generic;

public class VertexVisibility : Vertex
{
    void Awake()
    {
        neighbours = new List<Edge>();
    }
}
```

2. Define the `FindNeighbours` function for automating the process of connecting vertices among them:

```
public void FindNeighbours(List<Vertex> vertices)
{
    Collider c = gameObject.GetComponent<Collider>();
    c.enabled = false;
    Vector3 direction = Vector3.zero;
    Vector3 origin = transform.position;
    Vector3 target = Vector3.zero;
    RaycastHit[] hits;
    Ray ray;
    float distance = 0f;
    // next step
}
```

3. Go over each object and cast a ray to validate whether it's completely visible and then add it to the list of neighbors:

```
for (int i = 0; i < vertices.Count; i++)
{
    if (vertices[i] == this)
        continue;
    target = vertices[i].transform.position;
    direction = target - origin;
    distance = direction.magnitude;
    ray = new Ray(origin, direction);
    hits = Physics.RaycastAll(ray, distance);
    if (hits.Length == 1)
    {
```

```
            if (hits[0].collider.gameObject.tag.Equals("Vertex"))
            {
                Edge e = new Edge();
                e.cost = distance;
                GameObject go = hits[0].collider.gameObject;
                Vertex v = go.GetComponent<Vertex>();
                if (v != vertices[i])
                    continue;
                e.vertex = v;
                neighbours.Add(e);
            }
        }
    }
    c.enabled = true;
```

4. Create the `GraphVisibility` class:

```
using UnityEngine;
using System.Collections.Generic;

public class GraphVisibility : Graph
{
    // next steps
}
```

5. Build the `Load` function for making the connections between vertices:

```
public override void Load()
{
    Vertex[] verts = GameObject.FindObjectsOfType<Vertex>();
    vertices = new List<Vertex>(verts);
    for (int i = 0; i < vertices.Count; i++)
    {
        VertexVisibility vv = vertices[i] as VertexVisibility;
        vv.id = i;
        vv.FindNeighbours(vertices);
    }
}
```

6. Define the `GetNearesteVertex` function:

```
public override Vertex GetNearestVertex(Vector3 position)
{
    Vertex vertex = null;
    float dist = Mathf.Infinity;
    float distNear = dist;
    Vector3 posVertex = Vector3.zero;
```

```
        for (int i = 0; i < vertices.Count; i++)
        {
            posVertex = vertices[i].transform.position;
            dist = Vector3.Distance(position, posVertex);
            if (dist < distNear)
            {
                distNear = dist;
                vertex = vertices[i];
            }
        }
        return vertex;
    }
```

7. Define the `GetNeighbours` function:

```
    public override Vertex[] GetNeighbours(Vertex v)
    {
        List<Edge> edges = v.neighbours;
        Vertex[] ns = new Vertex[edges.Count];
        int i;
        for (i = 0; i < edges.Count; i++)
        {
            ns[i] = edges[i].vertex;
        }
        return ns;
    }
```

8. Finally, override the `GetEdges` function:

```
    public override Edge[] GetEdges(Vertex v)
    {
        return vertices[v.id].neighbours.ToArray();
    }
```

How it works...

The parent class `GraphVisibility` indexes every vertex on the scene and makes use of the `FindNeighbours` function on each one. This is in order to build the graph and make the connections without total user supervision, beyond placing the visibility points where the user sees fit. Also, the distance between two points is used to assign the cost to that corresponding edge.

There's more...

It's important to make a point visible to one another for the graph to be connected. This approach is also suitable for building intelligent graphs considering stairs and cliffs, it just requires moving the `Load` function to an editor-friendly class in order to call it in edit mode, and then modify or delete the corresponding edges to make it work as intended.

Take a look at the previous recipe's *Getting ready* section so you can better understand the starting point in case you feel you're missing something.

For further information about custom editors, editor scripting, and how to execute code in edit mode, please refer to the Unity documentation, available online at:

- `http://docs.unity3d.com/ScriptReference/Editor.html`
- `http://docs.unity3d.com/ScriptReference/ExecuteInEditMode.html`
- `http://docs.unity3d.com/Manual/PlatformDependentCompilation.html`

See also

- *Representing the world with Dirichlet domains* recipe

Representing the world with a self-made navigation mesh

Sometimes, a custom navigation mesh is necessary for dealing with difficult situations such as different types of graphs, but placing the graph's vertices manually is troublesome because it requires a lot of time to cover large areas.

We will learn how to use a model's mesh in order to generate a navigation mesh based on its triangles' centroids as vertices, and then leverage the heavy lifting from the previous recipe we learned.

Getting ready

This recipe requires some knowledge of custom editor scripting and understanding and implementing the points of visibility in the graph representation. Also, it is worth mentioning that the script instantiates a `CustomNavMesh` game object automatically in the scene and requires a prefab assigned, just like any other graph representation.

Finally, it's important to create the following class, deriving from `GraphVisibility`:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class CustomNavMesh : GraphVisibility
{
    public override void Start()
    {
        instIdToId = new Dictionary<int, int>();
    }
}
```

How to do it...

We will create an editor window for easily handling the automation process without weighing down the graph's `Start` function, delaying the scene loading.

1. Create the `CustomNavMeshWindow` class and place it in a directory called `Editor`:

```
using UnityEngine;
using UnityEditor;
using System.Collections;
using System.Collections.Generic;

public class CustomNavMeshWindow : EditorWindow
{
    // next steps here
}
```

2. Add the attributes to the editor window:

```
static bool isEnabled = false;
static GameObject graphObj;
static CustomNavMesh graph;
static CustomNavMeshWindow window;
static GameObject graphVertex;
```

3. Implement the function for initializing and showing the window:

```
[MenuItem("UAIPC/Ch02/CustomNavMeshWindow")]
static void Init()
{
    window = EditorWindow.GetWindow<CustomNavMeshWindow>();
    window.title = "CustomNavMeshWindow";
    SceneView.onSceneGUIDelegate += OnScene;
```

```
        graphObj = GameObject.Find("CustomNavMesh");
        if (graphObj == null)
        {
            graphObj = new GameObject("CustomNavMesh");
            graphObj.AddComponent<CustomNavMesh>();
            graph = graphObj.GetComponent<CustomNavMesh>();
        }
        else
        {
            graph = graphObj.GetComponent<CustomNavMesh>();
            if (graph == null)
                graphObj.AddComponent<CustomNavMesh>();
            graph = graphObj.GetComponent<CustomNavMesh>();
        }
    }
```

4. Define the `OnDestroy` function:

```
    void OnDestroy()
    {
        SceneView.onSceneGUIDelegate -= OnScene;
    }
```

5. Implement the `OnGUI` function for drawing the window's interior:

```
    void OnGUI()
    {
        isEnabled = EditorGUILayout.Toggle("Enable Mesh Picking",
    isEnabled);
        if (GUILayout.Button("Build Edges"))
        {
            if (graph != null)
                graph.LoadGraph();
        }
    }
```

6. Implement the first half of the `OnScene` function for handling the left-click on the scene window:

```
    private static void OnScene(SceneView sceneView)
    {
        if (!isEnabled)
            return;
        if (Event.current.type == EventType.MouseDown)
        {
            graphVertex = graph.vertexPrefab;
            if (graphVertex == null)
```

```
            {
                Debug.LogError("No Vertex Prefab assigned");
                return;
            }
            Event e = Event.current;
            Ray ray = HandleUtility.GUIPointToWorldRay(e.
mousePosition);
            RaycastHit hit;
            GameObject newV;
            // next step
        }
    }
```

7. Implement the second half for implementing the behavior when clicking on the mesh:

```
if (Physics.Raycast(ray, out hit))
{
    GameObject obj = hit.collider.gameObject;
    Mesh mesh = obj.GetComponent<MeshFilter>().sharedMesh;
    Vector3 pos;
    int i;
    for (i = 0; i < mesh.triangles.Length; i += 3)
    {
        int i0 = mesh.triangles[i];
        int i1 = mesh.triangles[i + 1];
        int i2 = mesh.triangles[i + 2];
        pos = mesh.vertices[i0];
        pos += mesh.vertices[i1];
        pos += mesh.vertices[i2];
        pos /= 3;
        newV = (GameObject)Instantiate(graphVertex, pos,
Quaternion.identity);
        newV.transform.Translate(obj.transform.position);
        newV.transform.parent = graphObj.transform;
        graphObj.transform.parent = obj.transform;
    }
}
```

How it works...

We create a custom editor window and set up the delegate function `OnScene` for handling events on the scene window. Also, we create the graph nodes by traversing the mesh vertex arrays, computing each triangle's centroid. Finally, we make use of the graph's `LoadGraph` function in order to compute neighbors.

Finding your way out of a maze with DFS

The **Depth-First Search** (**DFS**) algorithm is a path-finding technique suitable for low-memory devices. Another common use is to build mazes with a few modifications to the list of nodes visited and discovered, however the main algorithm stays the same.

Getting ready

This is a high-level algorithm that relies on each graph's implementation of the general functions, so the algorithm is implemented in the `Graph` class.

It is important to

How to do it...

Even though this recipe is only defining a function, please take into consideration the comments in the code to understand the indentation and code flow for effectively:

1. Declare the `GetPathDFS` function:

```
public List<Vertex> GetPathDFS(GameObject srcObj, GameObject
dstObj)
{
    // next steps
}
```

2. Validate if input objects are null:

```
if (srcObj == null || dstObj == null)
    return new List<Vertex>();
```

3. Declare and initialize the variables we need for the algorithm:

```
Vertex src = GetNearestVertex(srcObj.transform.position);
Vertex dst = GetNearestVertex(dstObj.transform.position);
Vertex[] neighbours;
Vertex v;
int[] previous = new int[vertices.Count];
for (int i = 0; i < previous.Length; i++)
    previous[i] = -1;
previous[src.id] = src.id;
Stack<Vertex> s = new Stack<Vertex>();
s.Push(src);
```

4. Implement the DFS algorithm for finding a path:

```
while (s.Count != 0)
{
    v = s.Pop();
    if (ReferenceEquals(v, dst))
    {
        return BuildPath(src.id, v.id, ref previous);
    }

    neighbours = GetNeighbours(v);
    foreach (Vertex n in neighbours)
    {
        if (previous[n.id] != -1)
            continue;
        previous[n.id] = v.id;
        s.Push(n);
    }
}
```

How it works...

The algorithm is based on the iterative version of DFS. It is also based on the in-order traversing of a graph and the LIFO philosophy using a stack for visiting nodes and adding discovered ones.

There is more...

We called the function `BuildPath`, but we haven't implemented it yet. It is important to note that this function is called by almost every other path-finding algorithm in this chapter, that's why it's not part of the main recipe.

This is the code for the `BuildPath` method:

```
private List<Vertex> BuildPath(int srcId, int dstId, ref int[]
prevList)
{
    List<Vertex> path = new List<Vertex>();
    int prev = dstId;
    do
    {
        path.Add(vertices[prev]);
        prev = prevList[prev];
    } while (prev != srcId);
    return path;
}
```

Finding the shortest path in a grid with BFS

The **Breadth-First Search** (**BFS**) algorithm is another basic technique for graph traversal and it's aimed to get the shortest path in the fewest steps possible, with the trade-off being expensive in terms of memory; thus, aimed specially at games on high-end consoles and computers.

Getting ready

This is a high-level algorithm that relies on each graph's implementation of the general functions, so the algorithm is implemented in the `Graph` class.

How to do it...

Even though this recipe is only defining a function, please take into consideration the comments in the code to understand the indentation and code flow more effectively:

1. Declare the `GetPathBFS` function:

```
public List<Vertex> GetPathBFS(GameObject srcObj, GameObject
dstObj)
{
    if (srcObj == null || dstObj == null)
        return new List<Vertex>();
    // next steps
}
```

2. Declare and initialize the variables we need for the algorithm:

```
Vertex[] neighbours;
Queue<Vertex> q = new Queue<Vertex>();
Vertex src = GetNearestVertex(srcObj.transform.position);
Vertex dst = GetNearestVertex(dstObj.transform.position);
Vertex v;
int[] previous = new int[vertices.Count];
for (int i = 0; i < previous.Length; i++)
    previous[i] = -1;
previous[src.id] = src.id;
q.Enqueue(src);
```

3. Implement the BFS algorithm for finding a path:

```
while (q.Count != 0)
{
    v = q.Dequeue();
    if (ReferenceEquals(v, dst))
```

```
            {
                return BuildPath(src.id, v.id, ref previous);
            }

            neighbours = GetNeighbours(v);
            foreach (Vertex n in neighbours)
            {
                if (previous[n.id] != -1)
                    continue;
                previous[n.id] = v.id;
                q.Enqueue(n);
            }
        }
        return new List<Vertex>();
```

How it works...

The BFS algorithm is similar to the DFS algorithm because it's based on the same in-order traversing of a graph but, instead of a stack such as DFS, BFS uses a queue for visiting the discovered nodes.

There is more...

In case you haven't noticed, we didn't implement the method `BuildPath`. This is because we talked about it at the end of the Depth-First Search recipe.

See also

- *Finding your way out of a maze with DFS*, recipe.

Finding the shortest path with Dijkstra

The Dijkstra's algorithm was initially designed to solve the single-source shortest path problem for a graph. Thus, the algorithm finds the lowest-cost route to everywhere from a single point. We will learn how to make use of it with two different approaches.

Getting ready

The first thing to do is import the binary heap class from the **Game Programming Wiki** (**GPWiki**) into our project, given that neither the .Net framework nor Mono has a defined structure for handling binary heaps or priority queues.

For downloading the source file and more information regarding GP Wiki's binary heap, please refer to the documentation online available at `http://content.gpwiki.org/index.php/C_sharp:BinaryHeapOfT`.

How to do it...

We will learn how to implement the Dijkstra algorithm using the same number of parameters as the other algorithms, and then explain how to modify it to make maximum use of it according to its original purpose.

1. Define the `GetPathDijkstra` function with its internal variables:

```
public List<Vertex> GetPathDijkstra(GameObject srcObj, GameObject
dstObj)
{
    if (srcObj == null || dstObj == null)
        return new List<Vertex>();
    Vertex src = GetNearestVertex(srcObj.transform.position);
    Vertex dst = GetNearestVertex(dstObj.transform.position);
    GPWiki.BinaryHeap<Edge> frontier = new GPWiki.
BinaryHeap<Edge>();
    Edge[] edges;
    Edge node, child;
    int size = vertices.Count;
    float[] distValue = new float[size];
    int[] previous = new int[size];

    // next steps
}
```

2. Add the source node to the heap (working as a priority queue) and assign a distance value of infinity to all of them but the source node:

```
node = new Edge(src, 0);
frontier.Add(node);
distValue[src.id] = 0;
previous[src.id] = src.id;
for (int i = 0; i < size; i++)
{
    if (i == src.id)
        continue;
    distValue[i] = Mathf.Infinity;
    previous[i] = -1;
}
```

3. Define a loop to iterate while the queue is not empty:

```
while (frontier.Count != 0)
{
    node = frontier.Remove();
    int nodeId = node.vertex.id;
    // next steps
}
return new List<Vertex>();
```

4. Code the procedure when arriving at the destination:

```
if (ReferenceEquals(node.vertex, dst))
{
    return BuildPath(src.id, node.vertex.id, ref previous);
}
```

5. Otherwise, process the visited nodes and add its neighbors to the queue, and return the path (not empty if there is a path from source to destination vertex):

```
edges = GetEdges(node.vertex);
foreach (Edge e in edges)
{
    int eId = e.vertex.id;
    if (previous[eId] != -1)
        continue;
    float cost = distValue[nodeId] + e.cost;
    if (cost < distValue[e.vertex.id])
    {
        distValue[eId] = cost;
        previous[eId] = nodeId;
        frontier.Remove(e);
        child = new Edge(e.vertex, cost);
        frontier.Add(child);
    }
}
```

How it works...

The Dijkstra algorithm works in a similar way to BFS, but considers non-negative edge costs in order to build the best route from the source vertex to every other one. That's why we have an array for storing the previous vertex.

There's more...

We will learn how to modify the current Dijkstra algorithm in order to approach the problem using pre-processing techniques and optimizing the path-finding time. It can be seen as three big steps: modifying the main algorithm, creating the pre-processing function (handy in editor mode, for example), and, finally, defining the path-retrieval function.

1. Modify the main function's signature:

```
public int[] Dijkstra(GameObject srcObj)
```

2. Change the returning value:

```
return previous;
```

3. Remove the lines from step 4 in the *How to do it* section:
4. Also, delete the following line at the beginning:

```
Vertex dst = GetNearestVertex(dstObj.transform.position);
```

5. Create a new member value to the `Graph` class:

```
List<int[]> routes = new List<int[]>();
```

6. Define the pre-processing function, called `DijkstraProcessing`:

```
public void DijkstraProcessing()
{
    int size = GetSize();
    for (int i = 0; i < size; i++)
    {
        GameObject go = vertices[i].gameObject;
        routes.add(Dijkstra(go));
    }
}
```

7. Implement a new `GetPathDijkstra` function for path retrieval:

```
public List<Vertex> GetPathDijkstra(GameObject srcObj, GameObject
dstObj)
{
    List<Vertex> path = new List<Vertex>();
    Vertex src = GetNearestVertex(srcObj);
    Vertex dst = GetNearestVertex(dstObj);
    return BuildPath(src.id, dst.id, ref routes[dst.id]);
}
```

In case you haven't noticed, we didn't implement the method `BuildPath`. This is because we talked about it at the end of the Depth-First Search recipe.

See also

- *Finding your way out of a maze with DFS*, recipe.

Finding the best-promising path with A*

The A* algorithm is probably the most-used technique for path finding, given its implementation simplicity, and efficiency, and because it has room for optimization. It's no coincidence that there are several algorithms based on it. At the same time, A* shares some roots with the Dijkstra algorithm, so you'll find similarities in their implementations.

Getting ready

Just like Dijkstra's algorithm, this recipe uses the binary heap extracted from the GPWiki. Also, it is important to understand what delegates are and how they work for. Finally, we are entering into the world of informed search; that means that we need to understand what a heuristic is and what it is for.

In a nutshell, for the purpose of this recipe, a heuristic is a function for calculating the approximate cost between two vertices in order to compare them to other alternatives and take the minimum-cost choice.

We need to add small changes to the Graph class:

1. Define a member variable as delegate:

```
public delegate float Heuristic(Vertex a, Vertex b);
```

2. Implement Euclidean distance member function to use it as default heuristic:

```
public float EuclidDist(Vertex a, Vertex b)
{
    Vector3 posA = a.transform.position;
    Vector3 posB = b.transform.position;
    return  Vector3.Distance(posA, posB);
}
```

3. Implement Manhattan distance function to use as a different heuristic. It will help us in comparing results using different heuristics:

```
public float ManhattanDist(Vertex a, Vertex b)
{
    Vector3 posA = a.transform.position;
    Vector3 posB = b.transform.position;
    return Mathf.Abs(posA.x - posB.x) + Mathf.Abs(posA.y -
posB.y);
}
```

How to do it...

Even though this recipe covers defining a function, please take into consideration the comments in the code to understand the indentation and code flow more effectively:

1. Define the `GetPathAstar` function along with its member variables:

```
public List<Vertex> GetPathAstar(GameObject srcObj, GameObject
dstObj, Heuristic h = null)
{
    if (srcObj == null || dstObj == null)
        return new List<Vertex>();
    if (ReferenceEquals(h, null))
        h = EuclidDist;

    Vertex src = GetNearestVertex(srcObj.transform.position);
    Vertex dst = GetNearestVertex(dstObj.transform.position);
    GPWiki.BinaryHeap<Edge> frontier = new GPWiki.
BinaryHeap<Edge>();
    Edge[] edges;
    Edge node, child;
    int size = vertices.Count;
    float[] distValue = new float[size];
    int[] previous = new int[size];
    // next steps
}
```

2. Add the source node to the heap (working as a priority queue) and assign a distance value of infinity to all of them but the source node:

```
node = new Edge(src, 0);
frontier.Add(node);
distValue[src.id] = 0;
previous[src.id] = src.id;
for (int i = 0; i < size; i++)
{
    if (i == src.id)
        continue;
    distValue[i] = Mathf.Infinity;
    previous[i] = -1;
}
```

3. Declare the loop for traversing the graph:

```
while (frontier.Count != 0)
{
    // next steps
}
return new List<Vertex>();
```

4. Implement the conditions for returning a path when necessary:

```
node = frontier.Remove();
int nodeId = node.vertex.id;
if (ReferenceEquals(node.vertex, dst))
{
    return BuildPath(src.id, node.vertex.id, ref previous);
}
```

5. Get the vertex's neighbors (also called `successors` in some text books):

```
edges = GetEdges(node.vertex);
```

6. Traverse the neighbors for computing the `cost` function:

```
foreach (Edge e in edges)
{
    int eId = e.vertex.id;
    if (previous[eId] != -1)
        continue;
    float cost = distValue[nodeId] + e.cost;
    // key point
    cost += h(node.vertex, e.vertex);
    // next step
}
```

7. Expand the list of explored nodes (frontier) and updating costs, if necessary:

```
if (cost < distValue[e.vertex.id])
{
    distValue[eId] = cost;
    previous[eId] = nodeId;
    frontier.Remove(e);
    child = new Edge(e.vertex, cost);
    frontier.Add(child);
}
```

How it works...

A* works in a similar fashion to Dijkstra's algorithm. However, instead of choosing the real lowest-cost node from all the possible options, it chooses the most-promising one based on a given heuristic, and goes on from there. In our case, the default heuristic is based solely on the Euclidian distance between two vertices with the option of using Manhattan distance.

There's more...

You are welcome to play with different heuristic functions depending on the game and context, and the following is an example of how to do so:

1. Define a heuristic function in the `Graph` class:

```
public float Heuristic(Vertex a, Vertex b)
{
  float estimation = 0f;
  // your logic here
  return estimation;
}
```

The important thing here is that the heuristic we develop is both *admissible* and *consistent*. For more theoretical insights about these topics, please refer to *Artificial Intelligence: A Modern Approach* by Russel and Norvig.

In case you haven't noticed, we didn't implement the method `BuildPath`. This is because we talked about it at the end of the Depth-First Search recipe.

See also

- The *Finding the shortest path with Dijkstra* recipe
- The *Finding your way out of a maze with DFS* recipe

For further information about Delegates, please refer to the official documentation available online at:

- `https://unity3d.com/learn/tutorials/modules/intermediate/scripting/delegates`

Improving A* for memory: IDA*

IDA* is a variant of an algorithm called Iterative Deepening Depth-First Search. Its memory usage is lower than A* because it doesn't make use of data structures to store the looked-up and explored nodes.

Getting ready

For this recipe, it is important to have some understanding of how recursion works.

How to do it...

This is a long recipe that can be seen as an extensive two-step process: creating the main function, and creating an internal recursive one. Please take into consideration the comments in the code to understand the indentation and code flow more effectively:

1. Let's start by defining the main function called `GetPathIDAstar`:

```
public List<Vertex> GetPathIDAstar(GameObject srcObj, GameObject
dstObj, Heuristic h = null)
{
    if (srcObj == null || dstObj == null)
        return new List<Vertex>();
    if (ReferenceEquals(h, null))
        h = EuclidDist;
    // next steps;
}
```

2. Declare and compute the variables to use along with the algorithm:

```
List<Vertex> path = new List<Vertex>();
Vertex src = GetNearestVertex(srcObj.transform.position);
Vertex dst = GetNearestVertex(dstObj.transform.position);
Vertex goal = null;
bool[] visited = new bool[vertices.Count];
for (int i = 0; i < visited.Length; i++)
visited[i] = false;
visited[src.id] = true;
```

3. Implement the algorithm's loop:

```
float bound = h(src, dst);
while (bound < Mathf.Infinity)
{
    bound = RecursiveIDAstar(src, dst, bound, h, ref goal, ref
visited);
}
if (ReferenceEquals(goal, null))
    return path;
return BuildPath(goal);
```

4. Now it's time to build the recursive internal function:

```
private float RecursiveIDAstar(
        Vertex v,
        Vertex dst,
        float bound,
        Heuristic h,
        ref Vertex goal,
        ref bool[] visited)
{
    // next steps
}
```

5. Prepare everything to start the recursion:

```
// base case
if (ReferenceEquals(v, dst))
    return Mathf.Infinity;
Edge[] edges = GetEdges(v);
if (edges.Length == 0)
    return Mathf.Infinity;
```

6. Apply the recursion for each neighbor:

```
// recursive case
float fn = Mathf.Infinity;
foreach (Edge e in edges)
{
    int eId = e.vertex.id;
    if (visited[eId])
        continue;
    visited[eId] = true;
    e.vertex.prev = v;
    float f = h(v, dst);
    float b;
    if (f <= bound)
    {
        b = RecursiveIDAstar(e.vertex, dst, bound, h, ref goal,
ref visited);
        fn = Mathf.Min(f, b);
    }
    else
        fn = Mathf.Min(fn, f);
}
```

7. Return a value based on the recursion result:

```
return fn;
```

How it works...

As we can see, the algorithm is very similar to that of the recursive version of Depth-First Search, but uses the principle of making decisions on top of a heuristic from A*. The main function is responsible for starting the recursion and building the resulting path. The recursive function is the one responsible for traversing the graph, looking for the destination node.

There is more...

This time we will need to implement a different a `BuildPath` function, in case you have followed along with the previous path finding recipes. Otherwise, we will need to implement this method that we haven't defined yet:

```
private List<Vertex> BuildPath(Vertex v)
{
    List<Vertex> path = new List<Vertex>();
    while (!ReferenceEquals(v, null))
    {
        path.Add(v);
        v = v.prev;
    }
    return path;
}
```

Planning navigation in several frames: time-sliced search

When dealing with large graphs, computing paths can take a lot of time, even halting the game for a couple of seconds. This could ruins its overall experience, to say the least. Luckily enough there are methods to avoid this.

This recipe is built on top of the principle of using coroutines as a method to keep the game running smoothly while finding a path in the background; some knowledge about coroutines is required.

Getting ready

We'll learn how to implement path-finding techniques using coroutines by refactoring the A* algorithm learned previously, but we will handle its signature as a different function.

How to do it...

Even though this recipe is only defining a function, please take into consideration the comments in the code to understand the indentation and code flow more effectively:

1. Modify the `Graph` class and add a couple of member variables. One for storing the path and the other to know whether the coroutine has finished:

```
public List<Vertex> path;
public bool isFinished;
```

2. Declare the member function:

```
public IEnumerator GetPathInFrames(GameObject srcObj, GameObject
dstObj, Heuristic h = null)
{
    //next steps
}
```

3. Include the following member variables at the beginning:

```
isFinished = false;
path = new List<Vertex>();
if (srcObj == null || dstObj == null)
{
    path = new List<Vertex>();
    isFinished = true;
    yield break;
}
```

4. Modify the loop to traverse the graph:

```
while (frontier.Count != 0)
{
    // changes over A*
    yield return null;
    ////////////////////////////
    node = frontier.Remove();
```

5. Also, include the other path-retrieval validations:

```
if (ReferenceEquals(node.vertex, dst))
{
    // changes over A*
    path = BuildPath(src.id, node.vertex.id, ref previous);
    break;
    ////////////////////////////
}
```

6. Finally, reset the proper values and return control at the end of the function, after closing the main loop:

```
isFinished = true;
yield break;
```

How it works...

The yield return null statement inside the main loop works as a flag for delivering control to the higher-level functions, thus computing each new loop in each new frame using Unity's internal multi-tasking system.

See also

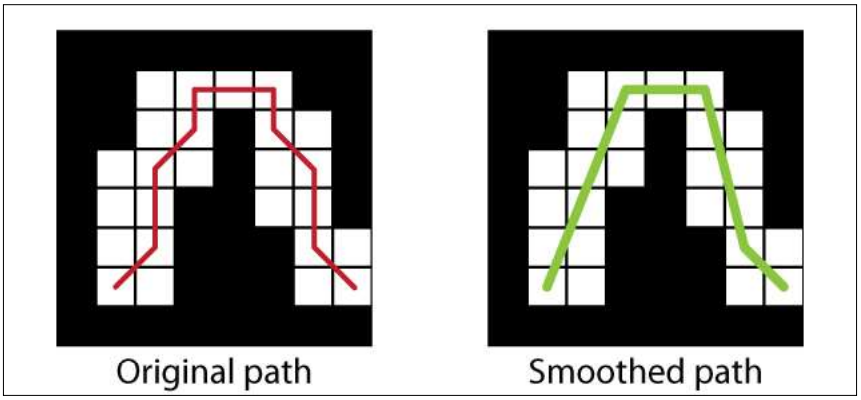
- The *Finding the best-promising path with A** recipe

For further information about Coroutines and more examples, please refer to the official documentation available online at:

- `http://docs.unity3d.com/Manual/Coroutines.html`
- `https://unity3d.com/learn/tutorials/modules/intermediate/scripting/coroutines`

Smoothing a path

When dealing with regular-size vertices on graph, such as grids, it's pretty common to see some kind of *robotic* movement from the agents in the game. Depending on the type of game we're developing, this could be avoided using path-smoothing techniques, such as the one we're about to learn.



Getting ready

Let's define a new tag in the Unity editor called `Wall` and assign it to every object in the scene that is intended to work as a wall or obstacle in the navigation.

How to do it...

This is an easy, yet powerful, function:

1. Define the `Smooth` function:

```
public List<Vertex> Smooth(List<Vertex> path)
{
    // next steps here
}
```

2. Check whether it is worth computing a new path:

```
List<Vertex> newPath = new List<Vertex>();
if (path.Count == 0)
    return newPath;
if (path.Count < 3)
    return path;
```

3. Implement the loops for traversing the list and building the new path:

```
newPath.Add(path[0]);
int i, j;
for (i = 0; i < path.Count - 1;)
{
    for (j = i + 1; j < path.Count; j++)
    {
        // next steps here
    }
    i = j - 1;
    newPath.Add(path[i]);
}
return newPath;
```

4. Declare and compute the variables to be used by the ray casting function:

```
Vector3 origin = path[i].transform.position;
Vector3 destination = path[j].transform.position;
Vector3 direction = destination - origin;
float distance = direction.magnitude;
bool isWall = false;
direction.Normalize();
```

5. Cast a ray from the current starting node to the next one:

```
Ray ray = new Ray(origin, direction);
RaycastHit[] hits;
hits = Physics.RaycastAll(ray, distance);
```

6. Check whether there is a wall and break the loop accordingly:

```
foreach (RaycastHit hit in hits)
{
    string tag = hit.collider.gameObject.tag;
    if (tag.Equals("Wall"))
    {
        isWall = true;
        break;
    }
}
if (isWall)
    break;
```

How it works...

We create a new path, taking the initial node as a starting point, and apply ray casting to the following node in the path, until we get a collision with a wall. When that happens, we take the previous node as the following node in the new path and the starting point for traversing the original one, until there are no nodes left to check. That way, we build a more intuitive path.