of which adjacent node is selected first. In Figure 5.6, nodes are selected in a counterclockwise order, beginning with the one immediately above the current node. If a clockwise order had been chosen, the path would be different. Of course, this algorithm could also be modified to perform an exhaustive search to find all paths to the destination; by finding the path with a minimum cost the best could be selected. However, it is an ineffective way of finding the shortest path.

The most popular algorithm used in games for searching graphs, is called A* (pronounced A-Star). What makes A* more efficient than BFS or DFS is that instead of picking the next adjacent node blindly, the algorithm looks for one that appears to be the most promising. From the starting node, the projected cost of all adjacent nodes is calculated, and the best node is chosen to be the next on the path. From this next node, the same calculations occur again, and the next best node is chosen. This algorithm ensures that all the best nodes are examined first. If one path of nodes does not work out, the algorithm can return to the next best in line and continue the search down a different path.

The algorithm determines the projected cost of taking paths based on the cost of getting to the next node and an estimate of getting from that node to the goal. The estimation is performed by a heuristic function. The term *heuristic* seems to be one of those funny words in AI that is difficult to define. Allen Newell first defined it in 1963 as a computation that performs the opposite function to that of an algorithm. A more useful definition of its meaning is given by Russell and Norvig in *Artificial Intelligence: A Modern Approach* published in 1995. They define a heuristic as any technique that can be used to improve the average performance of solving a problem that may not necessarily improve the worst performance. In the case of path finding, if the heuristic offers a perfect prediction—that is, if it can calculate the cost from the current node to the destination accurately—then the best path will be found. However, in reality, the heuristic is very rarely perfect and can offer only an approximation.

Unity Hands On

Pathfinding with A*

Step 1: Download *Chapter Five/Waypoints.zip* from the website. Open the project and the scene *patrolling*. Programming the A* algorithm is beyond the scope of this book and has therefore been provided with this project. If you are interested in the code, it can be found in the Project in the Plugins folder.

Step 2: Locate the robot model in the Robot Artwork > FBX folder in the Project. Drag the model into the Scene as shown in Figure 5.7. For the terrain and building models already in the scene, the robot will need to be scaled by 400 to match. If the textures are missing, find the material called *robot-robot1*, which is on the *roothandle* submesh of the robot; set it to *the Standard Shader*; and add the appropriate textures. The texture

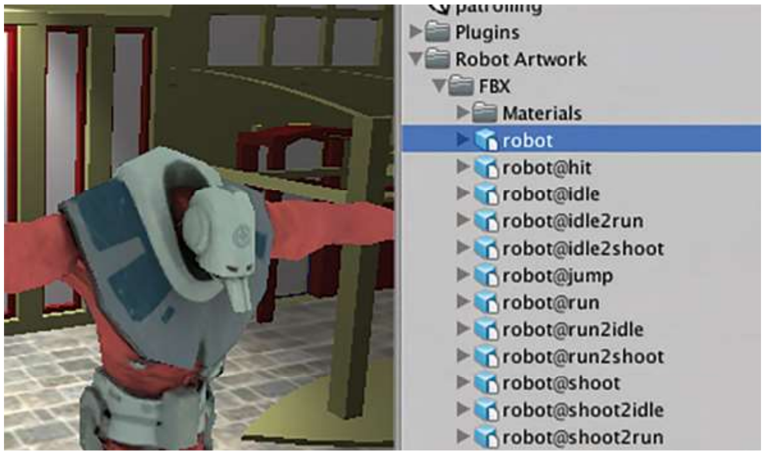


FIG 5.7 A robot model added to the scene and scaled by 400.

files will be in the Project. The model may also have a cube on its head. Make this invisible by finding the *headhandle* subobject of the robot in the Hierarchy and unticking its Mesh Renderer.

Step 3: Waypoints can be added to the scene in the same way as they were in Chapter 3. Any GameObject can act as a waypoint. Create nine spheres and arrange them in a circuit around the building. Add a ~~10th~~ 9th sphere out the front and an ~~11th~~ sphere in the driveway as shown in Figure 5.8. Name the spheres as Sphere1, Sphere2, and so on.

10th

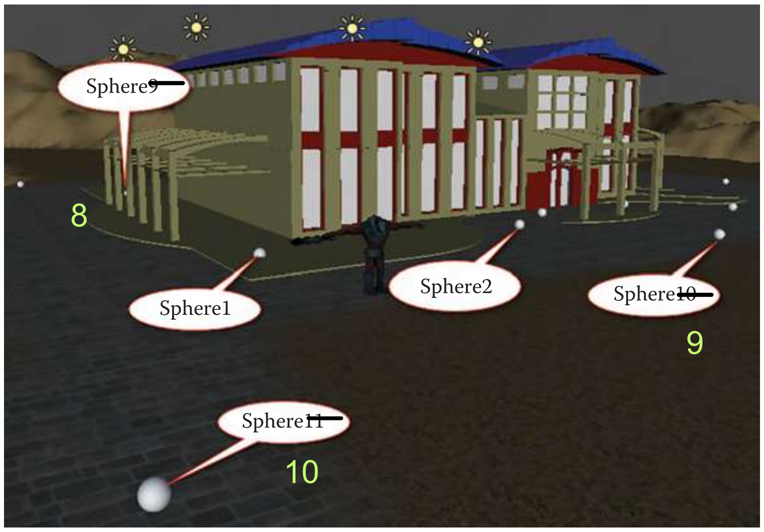
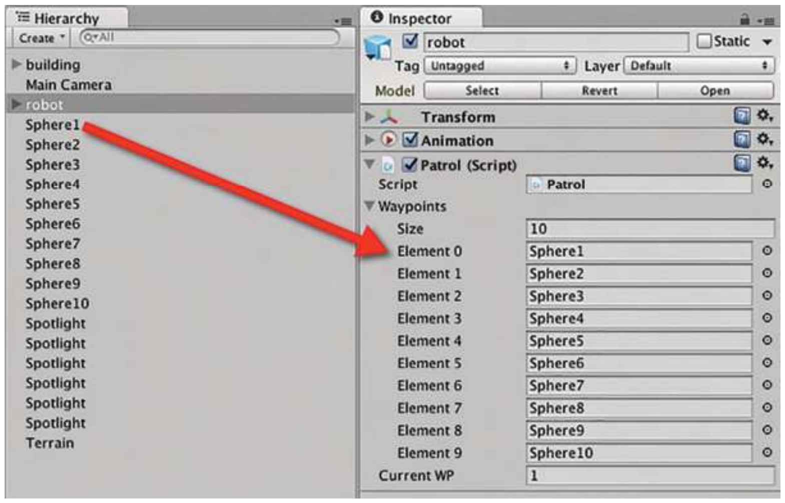


FIG 5.8 Waypoint layout.

Step 4: Create a new C# file called *Patrol* and open it in the script editor. Add the code shown in Listing 5.5.

Listing 5.5 Initializing waypoints in a graph object

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

public class Patrol : MonoBehaviour {

    public GameObject[] waypoints;
    Graph graph = new Graph();
    int currentWP = 0;
    GameObject currentNode;

    int speed = 8;
    int rotationSpeed = 5;
    float accuracy = 1.0f;

    // Use this for initialization
    void Start () {
        if(waypoints.Length > 0)
        {
            // add all the waypoints to the graph
            for(int i = 0; i < waypoints.Length; i++)
            {
                graph.AddNode(waypoints[i], true, true);
            }

            //create edges between the waypoints
            graph.AddEdge(waypoints[0], waypoints[1]);
            graph.AddEdge(waypoints[1], waypoints[2]);
            graph.AddEdge(waypoints[2], waypoints[3]);
            graph.AddEdge(waypoints[3], waypoints[4]);
            graph.AddEdge(waypoints[4], waypoints[5]);
            graph.AddEdge(waypoints[5], waypoints[6]);
            graph.AddEdge(waypoints[6], waypoints[7]);
            graph.AddEdge(waypoints[7], waypoints[8]);
            graph.AddEdge(waypoints[8], waypoints[0]);

        }
        currentNode = waypoints[0];
    }

    // Update is called once per frame
    void Update () {
        graph.debugDraw();
    }
}
```



building
Main Camera
robot
Sphere1
Sphere2
Sphere3
Sphere4
Sphere5
Sphere6
Sphere7
Sphere8
Sphere9
Sphere10
Spotlight
Spotlight
Spotlight
Spotlight
Spotlight
Spotlight
Terrain

robot ☐ Static
Tag Untagged Layer Default
Model

Patrol (Script)

Script	Patrol
Waypoints	
Size	10
Element 0	Sphere1
Element 1	Sphere2
Element 2	Sphere3
Element 3	Sphere4
Element 4	Sphere5
Element 5	Sphere6
Element 6	Sphere7
Element 7	Sphere8
Element 8	Sphere9
Element 9	Sphere10
Current WP	1

FIG 5.9 Adding waypoints to the *Patrol* script.

Step 5: Attach the *Patrol* script to the *robot* in the Hierarchy. Add the waypoints in order to the Waypoints array of the Patrol script as shown in Figure 5.9.
Step 6: Play. While playing, switch to the Scene. The code in the Update function will draw lines along the edges. The blue tip indicates the direction of the path. If you have all the waypoints collected correctly, there should be a circuit around the building.
Step 7: To get the robot guard to patrol around the building, modify the patrol code as shown in Listing 5.6.

Listing 5.6 Ordering a guard to traverse waypoints

```
public GameObject[] waypoints;
...
    void OnGUI ()
    {
        GUI.Box (new Rect (10,10,100,90), "Guard's
            Orders");
        if (GUI.Button (new Rect (20,65,80,20), "Patrol"))
        {
            graph.AStar(waypoints[0], waypoints[8]);

            this.GetComponent<Animation>().Play("run");
            this.GetComponent<Animation>()["run"].wrapMode
                = WrapMode.Loop;
        }
    }
```

```
...
    void Update () {
        graph.debugDraw();
        //if there is no path or at the end don't do
        //anything
        if(graph.getPathLength() == 0 || currentWP ==
            graph.getPathLength())
        {
            this.GetComponent<Animation>().Play("idle");
            return;
        }

        //the node we are closest to at this moment
        currentNode = graph.getPathPoint(currentWP);

        //if we are close enough to the current waypoint
        //move to next
        if(Vector3.Distance(
            graph.getPathPoint(currentWP).transform.
                position,transform.position) < accuracy)
        {
            currentWP++;
        }
        //if we are not at the end of the path
        if(currentWP < graph.getPathLength())
        {
            //keep on movin'
            Vector3 direction =
                graph.getPathPoint(currentWP).transform.
                   position - transform.position;
            transform.rotation = Quaternion.
                Slerp(transform.rotation, Quaternion.
                LookRotation(direction), rotationSpeed *
                Time.deltaTime);
            transform.Translate(0, 0, Time.deltaTime *
                speed);
        }
    }
```

Step 8: Play. When the Patrol button is pressed, the A* algorithm will calculate a path between the first and last waypoint, and the guard will start running around it.

Note
The character in this instance will move between the position of waypoints. If you have placed your spheres on the ground the character will sink into the ground as it is aiming its (0,0,0) position, which is in the center of the model, to the (0,0,0) of the sphere. To make the character appear to be moving on the terrain, move the spheres up to the right height. You can do this collectively by selecting all spheres in the Hierarchy by holding down shift while clicking on them and then dragging them up in the Scene.

In addition, if a character ever gets to a waypoint and starts circling it unexpectedly, it will be the accuracy setting. You may have it set to small and if the character can never get close enough to a waypoint, it will just keep trying. In this case, set the accuracy value to something higher.

Step 9: Using the A* algorithm to calculate a circuit is a little bit of overkill, as a circuit can be performed simply using the code from Chapter 3. So now we will put it through its paces by adding some more button commands to get the character to move about the building. Modify the patrol code to that in Listing 5.7.

Listing 5.7 Testing A* pathfinding by giving movement comments to a character

```
...
void OnGUI ()
{
    GUI.Box (new Rect (10,10,100,~~90~~ 150), "Guard's Orders");
    if (GUI.Button (new Rect (20,~~65~~ 90,80,20), "Front Door"))
    {
        graph.AStar(currentNode, waypoints[0]);
        currentWP = 0;

        this.GetComponent<Animation>().Play("run");
        this.GetComponent<Animation>()["run"].wrapMode =
            WrapMode.Loop;
    }
    if (GUI.Button (new Rect (20,~~90~~ 115,80,20), "Driveway"))
    {
        graph.AStar(currentNode, waypoints[9]);
        currentWP = 0;

        this.GetComponent<Animation>().Play("run");
        this.GetComponent<Animation>()["run"].wrapMode =
            WrapMode.Loop;
    }
    if (GUI.Button (new Rect (20,138~~115~~,80,20), "Front"))
    {
        graph.AStar(currentNode, waypoints[1]);
        currentWP = 0;

        this.GetComponent<Animation>().Play("run");
        this.GetComponent<Animation>()["run"].wrapMode =
            WrapMode.Loop;
    }
}

// Use this for initialization
void Start () {
    if(waypoints.Length > 0)
    {
        // add all the waypoints to the graph
        for(int i = 0; i < waypoints.Length; i++)
        {
            graph.AddNode(waypoints[i], true, true);
        }

        //create edges between the waypoints
        graph.AddEdge(waypoints[0], waypoints[1]);
        graph.AddEdge(waypoints[1], waypoints[2]);
        graph.AddEdge(waypoints[2], waypoints[3]);
        graph.AddEdge(waypoints[3], waypoints[4]);
        graph.AddEdge(waypoints[4], waypoints[5]);
        graph.AddEdge(waypoints[5], waypoints[6]);
        graph.AddEdge(waypoints[6], waypoints[7]);
        graph.AddEdge(waypoints[7], waypoints[8]);
        graph.AddEdge(waypoints[8], waypoints[0]);

        //and back the other way
        graph.AddEdge(waypoints[1], waypoints[0]);
        graph.AddEdge(waypoints[2], waypoints[1]);
        graph.AddEdge(waypoints[3], waypoints[2]);
        graph.AddEdge(waypoints[4], waypoints[3]);
        graph.AddEdge(waypoints[5], waypoints[4]);
        graph.AddEdge(waypoints[6], waypoints[5]);
        graph.AddEdge(waypoints[7], waypoints[6]);
        graph.AddEdge(waypoints[8], waypoints[7]);
        graph.AddEdge(waypoints[0], waypoints[8]);

        //create edges to extra to waypoints
        graph.AddEdge(waypoints[0], waypoints[8]);
        graph.AddEdge(waypoints[0], waypoints[9]);
        graph.AddEdge(waypoints[9], waypoints[9]);
        graph.AddEdge(waypoints~~[5]~~[4], waypoints[8]);
        //and back again
```

```
            graph.AddEdge(waypoints[8], waypoints[0]);
            graph.AddEdge(waypoints[9], waypoints[0]);
            graph.AddEdge(waypoints[9], waypoints[9]);
            graph.AddEdge(waypoints[8], waypoints~~[5]~~);
                                                    [4]
        }
        currentNode = waypoints[0];
    }
```

Step 10: The preceding code adds extra paths between the original circuit waypoints to point back the other way. This makes it possible to travel in any direction between points. Extra paths are also added between points in the driveway and out the front of the building. Play and switch to the Scene to see the red lines connecting the points (illustrated in Figure 5.10).

A new variable called currentNode has also been added to keep track of the waypoint the character last visited. This enables the algorithm to plot out paths based on the character's current position to the destination node.

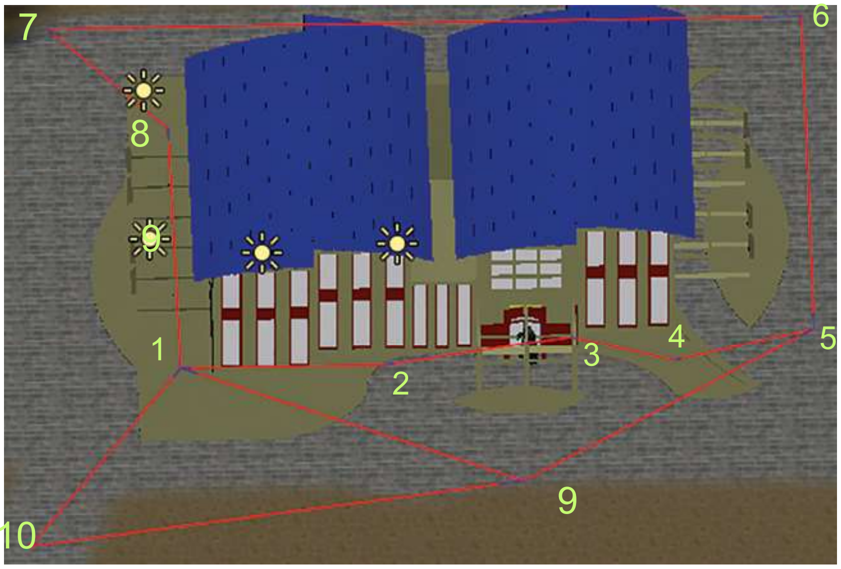


FIG 5.10 Debug lines showing paths in a waypoint graph.