```
        curSpeed = speed * Time.deltaTime;

        //Rotate the vehicle to its target
          //directional vector
      var rot = Quaternion.LookRotation(dir);
      transform.rotation = Quaternion.Slerp
        (transform.rotation, rot, 5.0f *
        Time.deltaTime);

        //Move the vehicle towards
          transform.position += transform.forward *
            curSpeed;
    }
```

A* Pathfinding

Next up, we'll be implementing the A* algorithm in a Unity environment using C#. The A* Pathfinding algorithm is widely used in games and interactive applications even though there are other algorithms, such as Dijkstra's algorithm, because of its simplicity and effectiveness. We've briefly covered this algorithm previously in *Chapter 1*, *The Basics of AI in Games*, but let's review the algorithm again from an implementation perspective.

Revisiting the A* algorithm

Let's review the A* algorithm again before we proceed to implement it in the next section. First, we'll need to represent the map in a traversable data structure. While many structures are possible, for this example, we will use a 2D grid array. We'll implement the `GridManager` class later to handle this map information. Our `GridManager` class will keep a list of the `Node` objects that are basically titles in a 2D grid. So, we need to implement that `Node` class to handle things such as node type (whether it's a traversable node or an obstacle), cost to pass through and cost to reach the goal `Node`, and so on.

We'll have two variables to store the nodes that have been processed and the nodes that we have to process. We'll call them closed list and open list, respectively. We'll implement that list type in the `PriorityQueue` class. And then finally, the following A* algorithm will be implemented in the `AStar` class. Let's take a look at it:

1. We begin at the starting node and put it in the open list.
2. As long as the open list has some nodes in it, we'll perform the following processes:
    1. Pick the first node from the open list and keep it as the current node. (This is assuming that we've sorted the open list and the first node has the least cost value, which will be mentioned at the end of the code.)
    2. Get the neighboring nodes of this current node that are not obstacle types, such as a wall or canyon that can't be passed through.
    3. For each neighbor node, check if this neighbor node is already in the closed list. If not, we'll calculate the total cost (`F`) for this neighbor node using the following formula:

        ```
        F = G + H
        ```

    4. In the preceding formula, `G` is the total cost from the previous node to this node and `H` is the total cost from this node to the final target node.
    5. Store this cost data in the neighbor node object. Also, store the current node as the parent node as well. Later, we'll use this parent node data to trace back the actual path.
    6. Put this neighbor node in the open list. Sort the open list in ascending order, ordered by the total cost to reach the target node.
    7. If there's no more neighbor nodes to process, put the current node in the closed list and remove it from the open list.
    8. Go back to step 2.

Once you have completed this process your current node should be in the target goal node position, but only if there's an obstacle free path to reach the goal node from the start node. If it is not at the goal node, there's no available path to the target node from the current node position. If there's a valid path, all we have to do now is to trace back from current node's parent node until we reach the start node again. This will give us a path list of all the nodes that we chose during our pathfinding process, ordered from the target node to the start node. We then just reverse this path list since we want to know the path from the start node to the target goal node.

This is a general overview of the algorithm we're going to implement in Unity using C#. So let's get started.

Implementation

We'll implement the preliminary classes that were mentioned before, such as the `Node`, `GridManager`, and `PriorityQueue` classes. Then, we'll use them in our main `AStar` class.

Implementing the Node class

The `Node` class will handle each tile object in our 2D grid, representing the maps shown in the `Node.cs` file:

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

public class Node : IComparable {
  public float nodeTotalCost;
  public float estimatedCost;
  public bool bObstacle;
  public Node parent;
  public Vector3 position;

  public Node() {
    this.estimatedCost = 0.0f;
    this.nodeTotalCost = 1.0f;
    this.bObstacle = false;
    this.parent = null;
  }

  public Node(Vector3 pos) {
    this.estimatedCost = 0.0f;
    this.nodeTotalCost = 1.0f;
    this.bObstacle = false;
    this.parent = null;
    this.position = pos;
  }

  public void MarkAsObstacle() {
    this.bObstacle = true;
  }
```

The `Node` class has properties, such as the cost values (`G` and `H`), flags to mark whether it is an obstacle, its positions, and parent node. The `nodeTotalCost` is `G`, which is the movement cost value from starting node to this node so far and the `estimatedCost` is `H`, which is total estimated cost from this node to the target goal node. We also have two simple constructor methods and a wrapper method to set whether this node is an obstacle. Then, we implement the `CompareTo` method as shown in the following code:

```
    public int CompareTo(object obj) {
      Node node = (Node)obj;
      //Negative value means object comes before this in the sort
        //order.
      if (this.estimatedCost < node.estimatedCost)
        return -1;
      //Positive value means object comes after this in the sort
        //order.
      if (this.estimatedCost > node.estimatedCost) return 1;
      return 0;
    }
}
```

This method is important. Our `Node` class inherits from `IComparable` because we want to override this `CompareTo` method. If you can recall what we discussed in the previous algorithm section, you'll notice that we need to sort our list of node arrays based on the total estimated cost. The `ArrayList` type has a method called `Sort`. This method basically looks for this `CompareTo` method, implemented inside the object (in this case, our `Node` objects) from the list. So, we implement this method to sort the node objects based on our `estimatedCost` value.

The `IComparable.CompareTo` method, which is a .NET framework feature, can be found at `http://msdn.microsoft.com/en-us/library/system.icomparable.compareto.aspx`.

Establishing the priority queue

The `PriorityQueue` class is a short and simple class to make the handling of the nodes' `ArrayList` easier, as shown in the following `PriorityQueue.cs` class:

```
using UnityEngine;
using System.Collections;

public class PriorityQueue {
```

```
    private ArrayList nodes = new ArrayList();

    public int Length {
      get { return this.nodes.Count; }
    }

    public bool Contains(object node) {
      return this.nodes.Contains(node);
    }

    public Node First() {
      if (this.nodes.Count > 0) {
        return (Node)this.nodes[0];
      }
      return null;
    }

    public void Push(Node node) {
      this.nodes.Add(node);
      this.nodes.Sort();
    }

    public void Remove(Node node) {
      this.nodes.Remove(node);
      //Ensure the list is sorted
      this.nodes.Sort();
    }
  }
```

The preceding code listing should be easy to understand. One thing to notice is that after adding or removing node from the nodes' `ArrayList`, we call the `Sort` method. This will call the `Node` object's `CompareTo` method and will sort the nodes accordingly by the `estimatedCost` value.

Setting up our grid manager

The `GridManager` class handles all the properties of the grid, representing the map. We'll keep a singleton instance of the `GridManager` class as we need only one object to represent the map, as shown in the following `GridManager.cs` file:

```
using UnityEngine;
using System.Collections;

public class GridManager : MonoBehaviour {
```

```
private static GridManager s_Instance = null;

public static GridManager instance {
  get {
    if (s_Instance == null) {
      s_Instance = FindObjectOfType(typeof(GridManager))
          as GridManager;
      if (s_Instance == null)
        Debug.Log("Could not locate a GridManager " +
            "object. \n You have to have exactly " +
            "one GridManager in the scene.");
    }
    return s_Instance;
  }
}
```

We look for the `GridManager` object in our scene and if found, we keep it in our `s_Instance` static variable:

```
public int numOfRows;
public int numOfColumns;
public float gridCellSize;
public bool showGrid = true;
public bool showObstacleBlocks = true;

private Vector3 origin = new Vector3();
private GameObject[] obstacleList;
public Node[,] nodes { get; set; }
public Vector3 Origin {
  get { return origin; }
}
```

Next, we declare all the variables; we'll need to represent our map, such as number of rows and columns, the size of each grid tile, and some Boolean variables to visualize the grid and obstacles as well as to store all the nodes present in the grid, as shown in the following code:

```
void Awake() {
  obstacleList = GameObject.FindGameObjectsWithTag("Obstacle");
  CalculateObstacles();
}
// Find all the obstacles on the map
void CalculateObstacles() {
  nodes = new Node[numOfColumns, numOfRows];
  int index = 0;
```

```
    for (int i = 0; i < numOfColumns; i++) {
      for (int j = 0; j < numOfRows; j++) {
        Vector3 cellPos = GetGridCellCenter(index);
        Node node = new Node(cellPos);
        nodes[i, j] = node;
        index++;
      }
    }
    if (obstacleList != null && obstacleList.Length > 0) {
      //For each obstacle found on the map, record it in our list
      foreach (GameObject data in obstacleList) {
        int indexCell = GetGridIndex(data.transform.position);
        int col = GetColumn(indexCell);
        int row = GetRow(indexCell);
        nodes[row, col].MarkAsObstacle();
      }
    }
  }
```

We look for all the game objects with an `Obstacle` tag and put them in our `obstacleList` property. Then we set up our nodes' 2D array in the `CalculateObstacles` method. First, we just create the normal node objects with default properties. Just after that, we examine our `obstacleList`. Convert their position into row-column data and update the nodes at that index to be obstacles.

The `GridManager` class has a couple of helper methods to traverse the grid and get the grid cell data. The following are some of them with a brief description of what they do. The implementation is simple, so we won't go into the details.

The `GetGridCellCenter` method returns the position of the grid cell in world coordinates from the cell index, as shown in the following code:

```
  public Vector3 GetGridCellCenter(int index) {
    Vector3 cellPosition = GetGridCellPosition(index);
    cellPosition.x += (gridCellSize / 2.0f);
    cellPosition.z += (gridCellSize / 2.0f);
    return cellPosition;
  }

  public Vector3 GetGridCellPosition(int index) {
    int row = GetRow(index);
    int col = GetColumn(index);
```

```
    float xPosInGrid = col * gridCellSize;
    float zPosInGrid = row * gridCellSize;
    return Origin + new Vector3(xPosInGrid, 0.0f, zPosInGrid);
  }
```

The `GetGridIndex` method returns the grid cell index in the grid from the given position:

```
  public int GetGridIndex(Vector3 pos) {
    if (!IsInBounds(pos)) {
      return -1;
    }
    pos -= Origin;
    int col = (int)(pos.x / gridCellSize);
    int row = (int)(pos.z / gridCellSize);
    return (row * numOfColumns + col);
  }

  public bool IsInBounds(Vector3 pos) {
    float width = numOfColumns * gridCellSize;
    float height = numOfRows* gridCellSize;
    return (pos.x >= Origin.x &&  pos.x <= Origin.x + width &&
        pos.x <= Origin.z + height && pos.z >= Origin.z);
  }
```

The `GetRow` and `GetColumn` methods return the row and column data of the grid cell from the given index:

```
  public int GetRow(int index) {
    int row = index / numOfColumns;
    return row;
  }

  public int GetColumn(int index) {
    int col = index % numOfColumns;
    return col;
  }
```

Another important method is `GetNeighbours`, which is used by the `AStar` class to retrieve the neighboring nodes of a particular node:

```
  public void GetNeighbours(Node node, ArrayList neighbors) {
    Vector3 neighborPos = node.position;
    int neighborIndex = GetGridIndex(neighborPos);

    int row = GetRow(neighborIndex);
```

```
    int column = GetColumn(neighborIndex);

    //Bottom
    int leftNodeRow = row - 1;
    int leftNodeColumn = column;
    AssignNeighbour(leftNodeRow, leftNodeColumn, neighbors);

    //Top
    leftNodeRow = row + 1;
    leftNodeColumn = column;
    AssignNeighbour(leftNodeRow, leftNodeColumn, neighbors);

    //Right
    leftNodeRow = row;
    leftNodeColumn = column + 1;
    AssignNeighbour(leftNodeRow, leftNodeColumn, neighbors);

    //Left
    leftNodeRow = row;
    leftNodeColumn = column - 1;
    AssignNeighbour(leftNodeRow, leftNodeColumn, neighbors);
  }

  void AssignNeighbour(int row, int column, ArrayList neighbors) {
    if (row != -1 && column != -1 &&
        row < numOfRows && column < numOfColumns) {
      Node nodeToAdd = nodes[row, column];
      if (!nodeToAdd.bObstacle) {
        neighbors.Add(nodeToAdd);
      }
    }
  }
```

First, we retrieve the neighboring nodes of the current node in the left, right, top, and bottom, all four directions. Then, inside the `AssignNeighbour` method, we check the node to see whether it's an obstacle. If it's not, we push that neighbor node to the referenced array list, `neighbors`. The next method is a debug aid method to visualize the grid and obstacle blocks:

```
  void OnDrawGizmos() {
    if (showGrid) {
      DebugDrawGrid(transform.position, numOfRows, numOfColumns,
          gridCellSize, Color.blue);
    }
```

```
        Gizmos.DrawSphere(transform.position, 0.5f);
        if (showObstacleBlocks) {
          Vector3 cellSize = new Vector3(gridCellSize, 1.0f,
            gridCellSize);
          if (obstacleList != null && obstacleList.Length > 0) {
            foreach (GameObject data in obstacleList) {
              Gizmos.DrawCube(GetGridCellCenter(
                  GetGridIndex(data.transform.position)), cellSize);
            }
          }
        }
      }

      public void DebugDrawGrid(Vector3 origin, int numRows, int
        numCols,float cellSize, Color color) {
        float width = (numCols * cellSize);
        float height = (numRows * cellSize);

        // Draw the horizontal grid lines
        for (int i = 0; i < numRows + 1; i++) {
          Vector3 startPos = origin + i * cellSize * new Vector3(0.0f,
            0.0f, 1.0f);
          Vector3 endPos = startPos + width * new Vector3(1.0f, 0.0f,
            0.0f);
          Debug.DrawLine(startPos, endPos, color);
        }

        // Draw the vertical grid lines
        for (int i = 0; i < numCols + 1; i++) {
          Vector3 startPos = origin + i * cellSize * new Vector3(1.0f,
            0.0f, 0.0f);
          Vector3 endPos = startPos + height * new Vector3(0.0f, 0.0f,
            1.0f);
          Debug.DrawLine(startPos, endPos, color);
        }
      }
    }
```

Gizmos can be used to draw visual debugging and setup aids inside the editor scene view. The `OnDrawGizmos` method is called every frame by the engine. So, if the debug flags, `showGrid` and `showObstacleBlocks`, are checked, we just draw the grid with lines and obstacle cube objects with cubes. Let's not go through the `DebugDrawGrid` method, which is quite simple.

You can learn more about gizmos in the Unity reference documentation at `http://docs.unity3d.com/Documentation/ScriptReference/Gizmos.html`.

Diving into our A* implementation

The `AStar` class is the main class that will utilize the classes we have implemented so far. You can go back to the algorithm section if you want to review this. We start with our `openList` and `closedList` declarations, which are of the `PriorityQueue` type, as shown in the `AStar.cs` file:

```
using UnityEngine;
using System.Collections;

public class AStar {
  public static PriorityQueue closedList, openList;
```

Next, we implement a method called `HeuristicEstimateCost` to calculate the cost between the two nodes. The calculation is simple. We just find the direction vector between the two by subtracting one position vector from another. The magnitude of this resultant vector gives the direct distance from the current node to the goal node:

```
  private static float HeuristicEstimateCost(Node curNode,
      Node goalNode) {
    Vector3 vecCost = curNode.position - goalNode.position;
    return vecCost.magnitude;
  }
```

Next, we have our main `FindPath` method:

```
  public static ArrayList FindPath(Node start, Node goal) {
    openList = new PriorityQueue();
    openList.Push(start);
    start.nodeTotalCost = 0.0f;
    start.estimatedCost = HeuristicEstimateCost(start, goal);

    closedList = new PriorityQueue();
    Node node = null;
```

We initialize our open and closed lists. Starting with the start node, we put it in our open list. Then we start processing our open list:

```
    while (openList.Length != 0) {
      node = openList.First();
      //Check if the current node is the goal node
```

```
        if (node.position == goal.position) {
          return CalculatePath(node);
        }

        //Create an ArrayList to store the neighboring nodes
        ArrayList neighbours = new ArrayList();

        GridManager.instance.GetNeighbours(node, neighbours);

        for (int i = 0; i < neighbours.Count; i++) {
          Node neighbourNode = (Node)neighbours[i];

          if (!closedList.Contains(neighbourNode)) {
            float cost = HeuristicEstimateCost(node,
                neighbourNode);

            float totalCost = node.nodeTotalCost + cost;
            float neighbourNodeEstCost = HeuristicEstimateCost(
                neighbourNode, goal);

            neighbourNode.nodeTotalCost = totalCost;
            neighbourNode.parent = node;
            neighbourNode.estimatedCost = totalCost +
                neighbourNodeEstCost;

            if (!openList.Contains(neighbourNode)) {
              openList.Push(neighbourNode);
            }
          }
        }
        //Push the current node to the closed list
        closedList.Push(node);
        //and remove it from openList
        openList.Remove(node);
      }

      if (node.position != goal.position) {
        Debug.LogError("Goal Not Found");
        return null;
      }
      return CalculatePath(node);
    }
```

This code implementation resembles the algorithm that we have previously discussed, so you can refer back to it if you are not clear of certain things. Perform the following steps:

1. Get the first node of our `openList`. Remember our `openList` of nodes is always sorted every time a new node is added. So, the first node is always the node with the least estimated cost to the goal node.
2. Check whether the current node is already at the goal node. If so, exit the `while` loop and build the `path` array.
3. Create an array list to store the neighboring nodes of the current node being processed. Use the `GetNeighbours` method to retrieve the neighbors from the grid.
4. For every node in the `neighbors` array, we check whether it's already in `closedList`. If not, we calculate the cost values, update the node properties with the new cost values as well as the parent node data, and put it in `openList`.
5. Push the current node to `closedList` and remove it from `openList`. Go back to step 1.

If there are no more nodes in `openList`, our current node should be at the target node if there's a valid path available. Then, we just call the `CalculatePath` method with the current node parameter:

```
    private static ArrayList CalculatePath(Node node) {
      ArrayList list = new ArrayList();
      while (node != null) {
        list.Add(node);
        node = node.parent;
      }
      list.Reverse();
      return list;
    }
}
```

The `CalculatePath` method traces through each node's parent node object and builds an array list. It gives an array list with nodes from the target node to the start node. Since we want a path array from the start node to the target node, we just call the `Reverse` method.

So, this is our `AStar` class. We'll write a test script in the following code to test all this and then set up a scene to use them in.

Implementing a Test Code class

This class will use the `AStar` class to find the path from the start node to the goal node, as shown in the following `TestCode.cs` file:

```
using UnityEngine;
using System.Collections;

public class TestCode : MonoBehaviour {
  private Transform startPos, endPos;
  public Node startNode { get; set; }
  public Node goalNode { get; set; }

  public ArrayList pathArray;

  GameObject objStartCube, objEndCube;
  private float elapsedTime = 0.0f;
  //Interval time between pathfinding
  public float intervalTime = 1.0f;
```

First, we set up the variables that we'll need to reference. The `pathArray` is to store the nodes array returned from the `AStar FindPath` method:

```
  void Start () {
    objStartCube = GameObject.FindGameObjectWithTag("Start");
    objEndCube = GameObject.FindGameObjectWithTag("End");

    pathArray = new ArrayList();
    FindPath();
  }

  void Update () {
    elapsedTime += Time.deltaTime;
    if (elapsedTime >= intervalTime) {
      elapsedTime = 0.0f;
      FindPath();
    }
  }
```

In the `Start` method, we look for objects with the `Start` and `End` tags and initialize our `pathArray`. We'll be trying to find our new path at every interval that we set to our `intervalTime` property in case the positions of the start and end nodes have changed. Then, we call the `FindPath` method:

```
    void FindPath() {
      startPos = objStartCube.transform;
      endPos = objEndCube.transform;

      startNode = new Node(GridManager.instance.GetGridCellCenter(
          GridManager.instance.GetGridIndex(startPos.position)));

      goalNode = new Node(GridManager.instance.GetGridCellCenter(
          GridManager.instance.GetGridIndex(endPos.position)));

      pathArray = AStar.FindPath(startNode, goalNode);
    }
```

Since we implemented our pathfinding algorithm in the `AStar` class, finding a path has now become a lot simpler. First, we take the positions of our start and end game objects. Then, we create new `Node` objects using the helper methods of `GridManager` and `GetGridIndex` to calculate their respective row and column index positions inside the grid. Once we get this, we just call the `AStar.FindPath` method with the start node and goal node and store the returned array list in the local `pathArray` property. Next, we implement the `OnDrawGizmos` method to draw and visualize the path found:

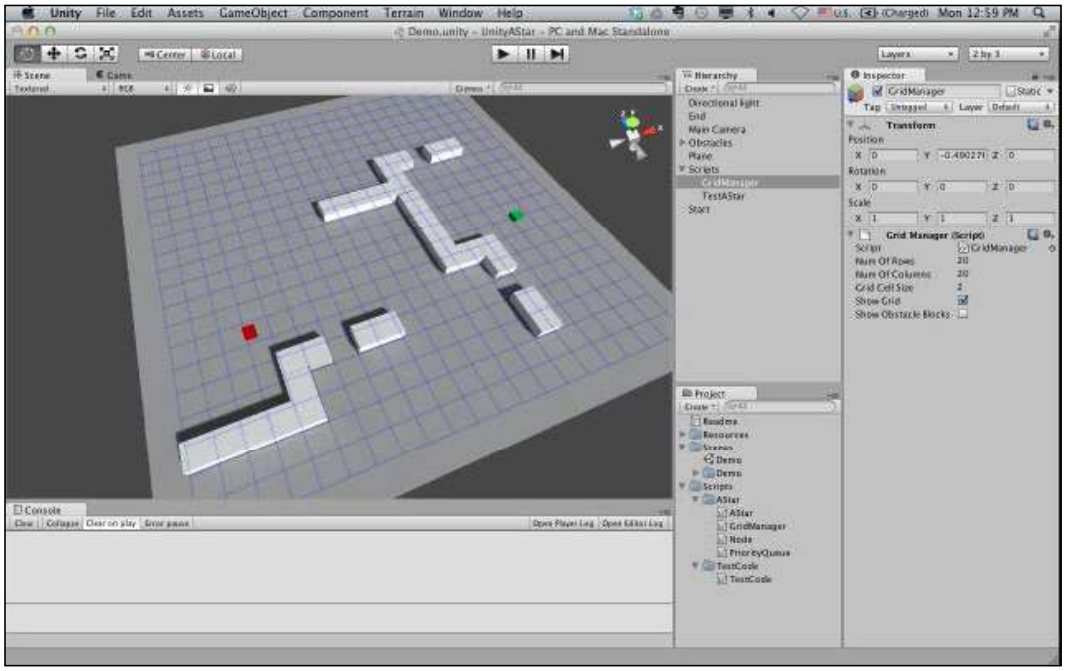
```
    void OnDrawGizmos() {
      if (pathArray == null)
        return;

      if (pathArray.Count > 0) {
        int index = 1;
        foreach (Node node in pathArray) {
          if (index < pathArray.Count) {
            Node nextNode = (Node)pathArray[index];
            Debug.DrawLine(node.position, nextNode.position,
              Color.green);
            index++;
          }
        }
      }
    }
  }
```

We look through our `pathArray` and use the `Debug.DrawLine` method to draw the lines connecting the nodes from the `pathArray`. With this, we'll be able to see a green line connecting the nodes from start to end, forming a path, when we run and test our program.

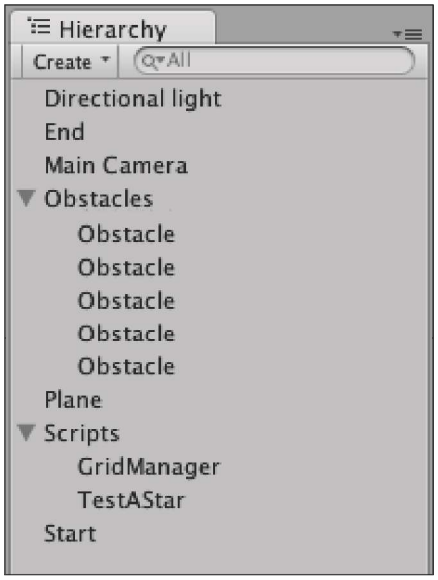
Setting up our sample scene

We are going to set up a scene that looks something similar to the following screenshot:



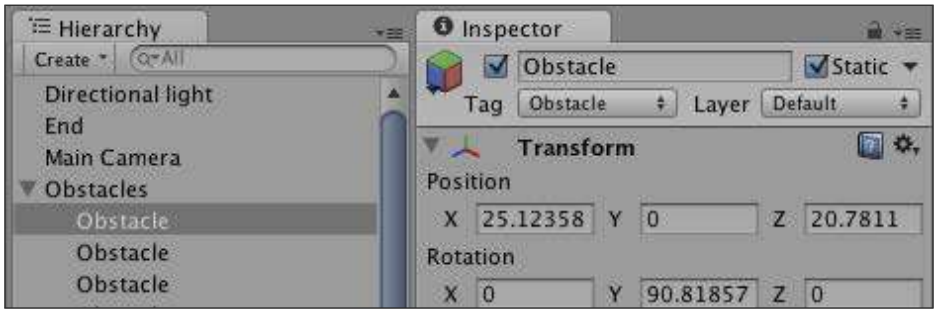
A sample test scene

We'll have a directional light, the start and end game objects, a few obstacle objects, a plane entity to be used as ground, and two empty game objects in which we put our `GridManager` and `TestAStar` scripts. This is our scene hierarchy:



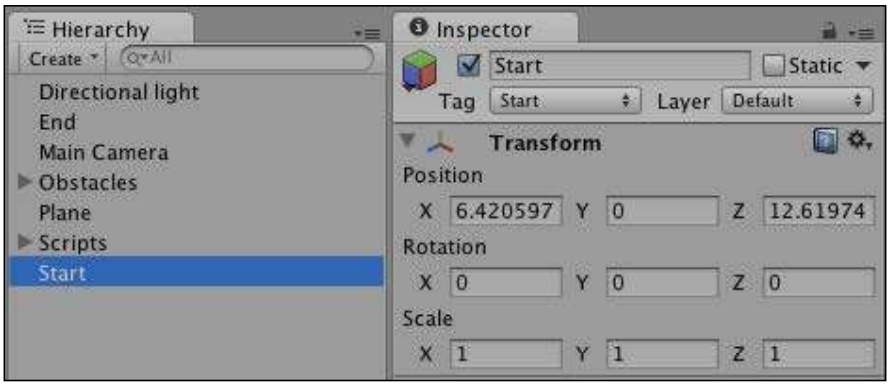
The scene Hierarchy

Create a bunch of cube entities and tag them as `Obstacle`. We'll be looking for objects with this tag when running our pathfinding algorithm.



The Obstacle node

Create a cube entity and tag it as `Start`.

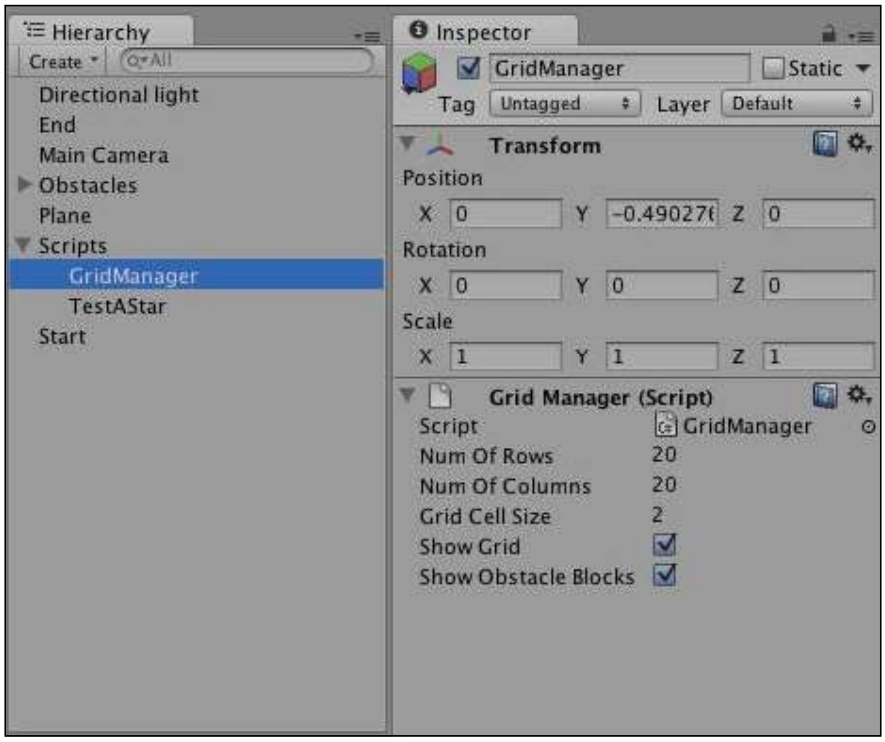


The Start node

Then, create another cube entity and tag it as `End`.

The End node

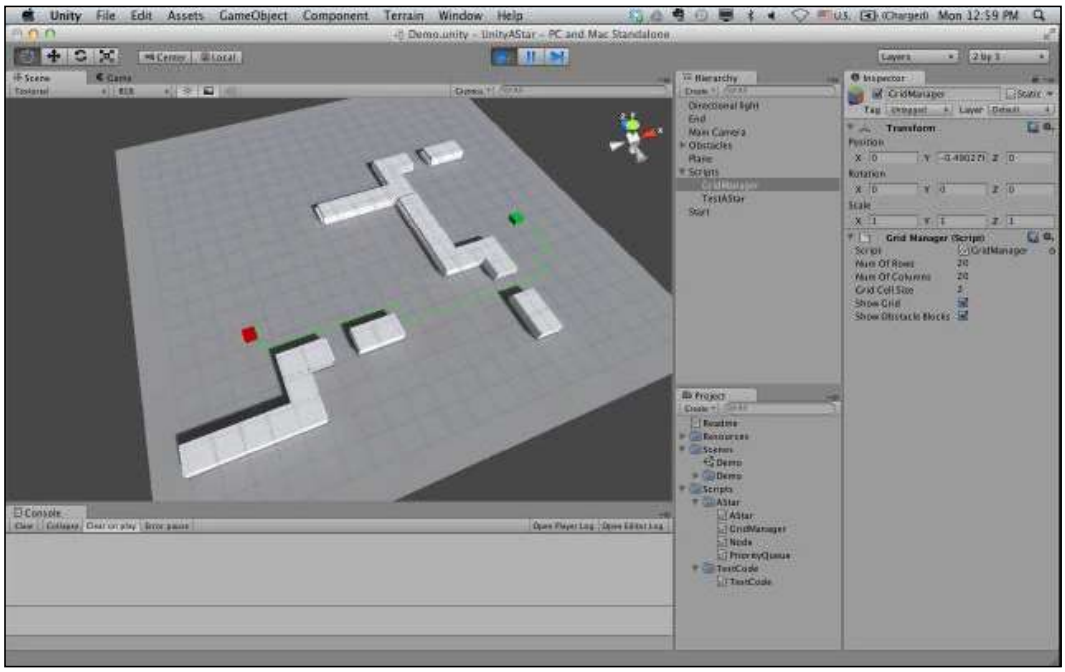
Now, create an empty game object and attach the `GridManager` script. Set the name as `GridManager` because we use this name to look for the `GridManager` object from our script. Here, we can set up the number of rows and columns for our grid as well as the size of each tile.



The GridManager script

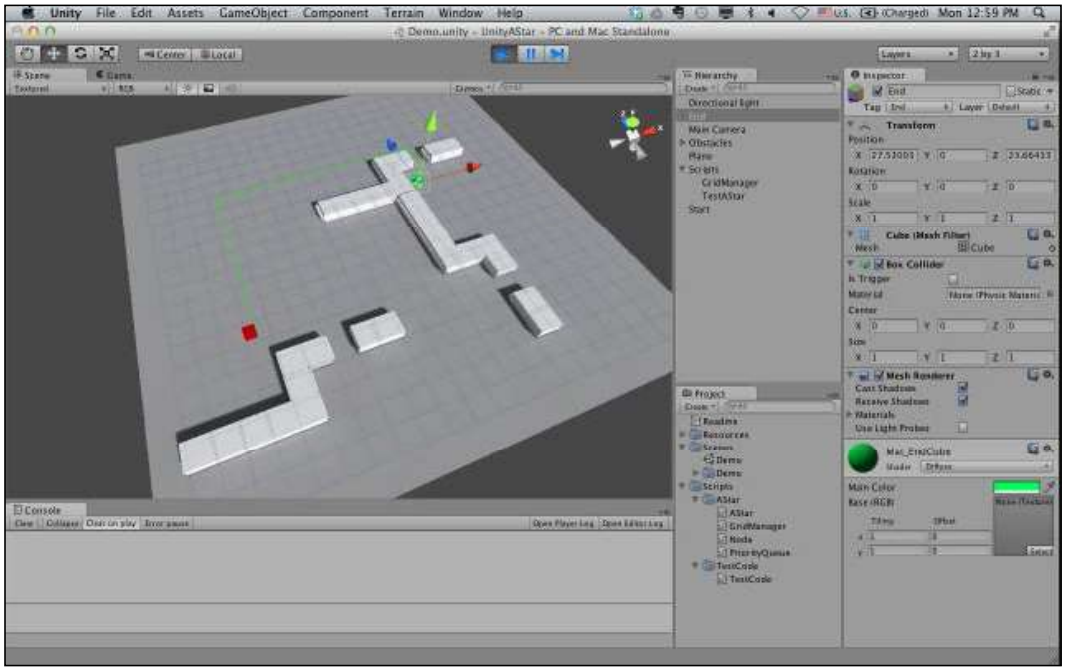
Testing all the components

Let's hit the play button and see our A* Pathfinding algorithm in action. By default, once you play the scene, Unity will switch to the **Game** view. Since our pathfinding visualization code is written for the debug drawn in the editor view, you'll need to switch back to the **Scene** view or enable **Gizmos** to see the path found.



Found path one

Now, try to move the start or end node around in the scene using the editor's movement gizmo (not in the **Game** view, but the **Scene** view).



Found path two

You should see the path updated accordingly if there's a valid path from the start node to the target goal node, dynamically in real time. You'll get an error message in the console window if there's no path available.