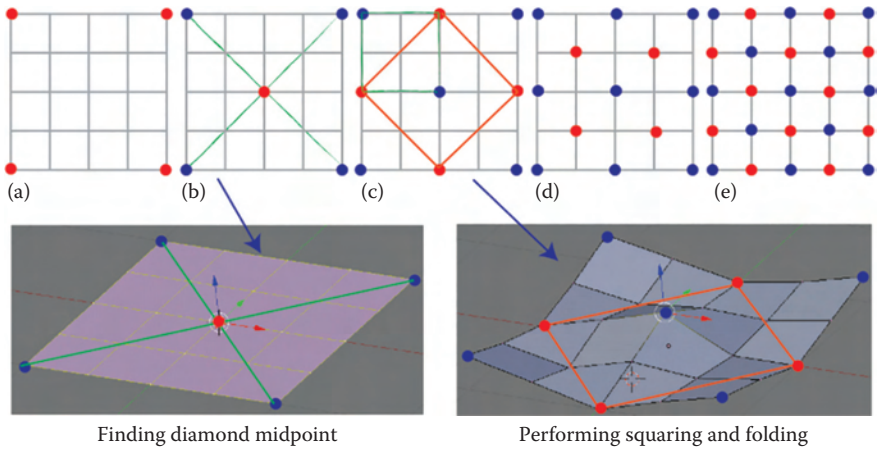

FIG 7.12 The Diamond-Square algorithm.

between the central point and the previous initial points. Now imagine fold lines between the red points and the height values assigned to the blue points as shown in green in Figure 7.12. Blue points determine height, and the green lines are where the mesh is allowed to fold.

The Diamond-Square method continues creating diamonds to find midpoints and squares for lowering and raising until all points in the grid have height values associated with them.

Perlin noise is another popular algorithmic way to generate landscapes. It is used in the next workshop to produce a natural-looking terrain from a plane object in Unity.

Unity Hands On

Generating a Terrain Procedurally

Step 1: Create a new Unity project and import the Character Controller package.

Step 2: Add a plane to the scene and a first person controller (FPC). Position the FPC just above the plane and add a directional light as shown in Figure 7.13. Scale the plane's *x* and *z* size to 10.

Note

If you want a smoother, more detailed terrain, you will need to use a plane with more vertices than the one supplied with Unity. To do this, use Blender or your favorite 3D modeling program to create a plane and apply subdivisions to give it the number of polygons you desire.

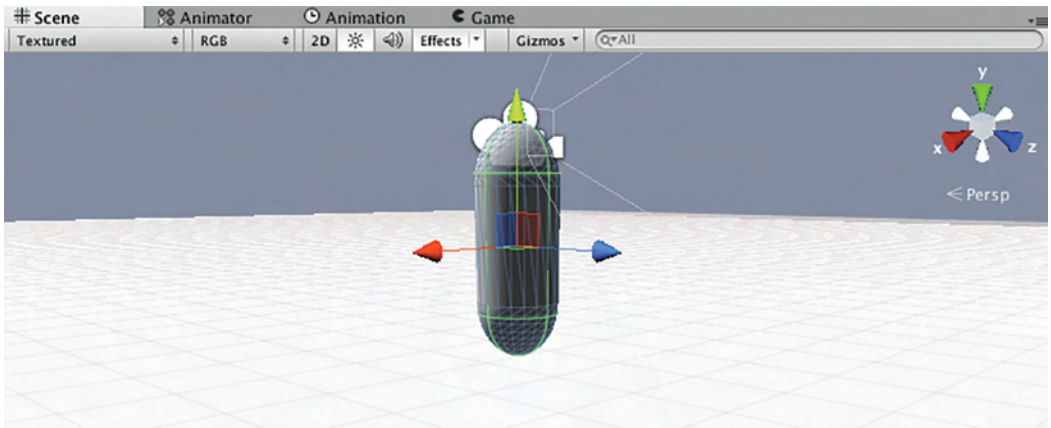

FIG 7.13 A plane and FPC for starting the project.

Step 3: Select the plane in the Hierarchy and remove its Mesh Collider component in the Inspector. We will add a new one shortly after its terrain heights have been modified.
Step 4: Create a C# file called *MakeTerrain.cs.* Add the code shown in Listing 7.1.

Listing 7.1 Script to create random height values on a mesh

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

public class MakeTerrain : MonoBehaviour {

     void Start()
     {
         Mesh mesh = this.
             GetComponent<MeshFilter>().mesh;
         Vector3[] vertices = mesh.vertices;
         for (int v = 0; v < vertices.Length; v++)
         {
             vertices[v].y = Random.Range(0f,10f);
         }
         mesh.vertices = vertices;
         mesh.RecalculateBounds();
         mesh.RecalculateNormals();
         this.gameObject.AddComponent<MeshCollider>();
     }
}
```

Step 5: Attach the script to the plane.
Step 6: Play. The terrain will have random heights across its surface. Note that the Mesh Collider is added after the heights have been set. This will ensure that the FPC does not fall through. Switch to the Scene while in Play mode for a better view of the created landscape.

After the mesh vertices are modified in the *y* direction for height, the bounding volume and normals are recreated. The bounding volume is used by the game engine to improve the processing of geometrical operations such as collision detection and overlap. The reason being that in the first instance, computing overlapping volumes is easier than lower level collision detection. Before exact collisions are calculated, determining if there might be a collision between two objects is more efficient with bounding volumes.

In addition, the plane's normals are also recalculated. Because the plane's mesh has changed shape, the normals must be adjusted for the new vertex values for the mesh to be shaded and shadowed correctly.

Using a random function to determine height is not a realistic way to produce rise and fall in a terrain, as the height can change erratically with each vertex. At small height values, it might look all right, but try changing the random range from 0 to 50 and the surface of the plane will become very jagged.
Step 7: To create smoother rises and falls in a surface, a mathematical curve such as sine or cosine can be used. Modify *MakeTerrain.cs* to reflect the changes shown in Listing 7.2.

Listing 7.2 Using a sine function to set the terrain height

```
...
    for (int v = 0; v < vertices.Length; v++)
    {
        vertices[v].y = Mathf. Sin(vertices[v].x * 10);
    }
...
```

Step 8: Play. The shape of the sine wave will be visually evident. The terrain will be smooth, but too uniform to replicate a natural landscape.

Another way to create a landscape procedurally is to use Perlin noise, a pseudo-random mathematical algorithm that provides smooth gradients between points. This is the method used for landscape generation in *Minecraft*. *Check out http://bit.ly/holisticminecraft for Unity tutorials showing you how.*
Step 9: Download *Chapter Seven/Perlin.cs* from the website. Add the file to your Project inside a new folder called *Plugins*.
Step 10: Modify *MakeTerrain.cs* to that shown in Listing 7.3.

Listing 7.3 Using Perlin noise to generate terrain heights

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

public class MakeTerrain : MonoBehaviour {

     void Start()
     {
         Perlin surface = new Perlin();
         Mesh mesh = this.
             GetComponent<MeshFilter>().mesh;
         Vector3[] vertices = mesh.vertices;
         for (int v = 0; v < vertices.Length; v++)
         {
             vertices[v].y = surface.Noise(
                 vertices[v].x * 2 + 0.1365143f,
                 vertices[v].z * 2 + 1.21688f) * 10;
         }
         mesh.vertices = vertices;
         mesh.RecalculateBounds();
         mesh.RecalculateNormals();
         this.gameObject.AddComponent<MeshCollider>();
     }
}
```

Note

Perlin Noise

Perlin noise is the same algorithm used in Photoshop for generating the *Render Clouds* filter. The image created by Perlin noise is grayscale where black indicates the lowest altitude and white the highest as shown in Figure 7.14a.

In fact, you can create a procedurally generated terrain in Unity (shown in Figure 7.14b) by (1) adding a terrain; (2) getting the flat terrain texture with Terrain > Export Heightmap; (3) opening this texture with Photoshop; (4) applying Photoshop's Render Clouds filter; (5) saving the texture in the same raw format; and (6) selecting Terrain > Import Heightmap in Unity to load the heights.

The difference in using a random number for the height and Perlin noise is illustrated in Figure 7.15.