8

Player and Non-Player Character (NPC) gameObject Positions, Orientations, and Movement

In this chapter, we will cover:

- Player control of a 2D gameObject (and limiting movement within a rectangle)
- Player control of a 3D gameObject (and limiting movement within a rectangle)
- Choosing destinations: Find nearest (or a random) spawn point
- Choosing destinations: Respawn to most recently passed checkpoint
- NPC NavMeshAgent to seek or flee destination while avoiding obstacles
- NPC NavMeshAgent to follow waypoints in sequence
- NPC NavMeshAgent of group movement: flocking

Introduction

Many gameObjects in games **move**! Movement can be controlled by the player, by the (simulated) laws of physics in the environment, or by Non-Player Character (NPC) logic. For example objects that follow a path of waypoints, or seek (move towards) or flee (away) from the current position of a character. Unity provides several controllers, for first- and third- person characters, and for vehicles such as cars and airplanes. GameObject movement can also be controlled through the state machines of the Unity Mechanim animation system.

However, they may be times when you wish to *tweak* the Player character controllers from Unity, or you wish to write your own. You might wish to write *directional logic* – simple or sophisticated Artificial Intelligence (AI) to control the game's NPC and enemy characters. Such AI might involve your computer program making objects orient and move towards or away from characters or other game objects.

This chapter presents a range of such directional recipes, from which many games can benefit in terms of a richer and more exciting user experience.

Unity provides sophisticated classes and components including the Vector3 class and rigid body physics for modeling realistic movements, forces, and collisions in games. We make use of these game engine features to implement some sophisticated NPC and enemy character movements in the recipes in this chapter.

The big picture

For 3D games (and to some extent, 2D games as well), a fundamental class of object is the Vector3 class – objects that store and manipulate (x,y,z) values representing locations in 3D space. If we draw an imaginary *arrow* from the origin (0,0,0) to a point on space, then the direction and length of this *arrow* (vector) can represent a velocity or force (that is, a certain amount *magnitude* in a certain direction).

If we ignore all the character controller components, and colliders and physics system in Unity, we can write code that *teleports* objects directly to a particular (x,y,z) location in our scene. And sometimes that's just what we want to do, for example we may wish to *spawn* an object at a location. However, in most cases if we want objects to move in more physically realistic ways, then we either apply a force to the object, or change its *velocity* component, or if it has a Character Controller component then we send it a *Move()* message. With the introduction of Unity NavMeshAgents (and associated Navigation Meshes), we can now set a *destination* for an object with a NavMeshAgent, and the built-in pathfinding logic will do the work of moving our NPC object on a path towards the given (x,y,z) destination location.

As well as deciding **which** technique will be used to move an object, our game must also do the work of deciding how to choose the destination locations, or the direction and magnitude of changes to movement. This can involve logic to tell an NPC or enemy object the destination of the Player's character (to be moved towards, and then perhaps attacked when close enough). Or perhaps *shy* NPC objects will be given the direction to the Player's character, so that they can *flee* in the opposite direction, until they are a safe distance away.

Other core concepts in NPC object movement and creation (Instantiation) include:

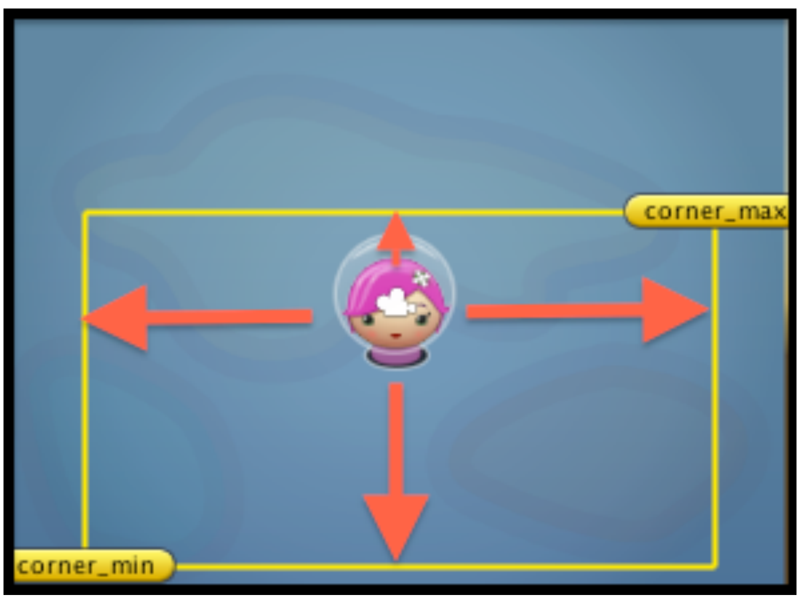
- Spawn points

        - Specific locations in the scene where objects are to be created, or moved to
- Waypoints
    - Sequence of locations to define a *path* for NPCs or perhaps the Player's character to follow
- Checkpoints
    - Locations (or colliders) which once passed through, change what happens in the game (for example, extra time, or if Player's character killed, they respawn to the last crossed checkpoint, and so on)

Player control of a 2D gameObject (and limiting movement within a rectangle)

While the rest of the recipes in this chapter are demonstrated in 3D projects, basic character movement in 2D, and also limiting movement to a bounding rectangle, are core skills for many 2D games, and so this first recipe illustrates how to achieve these features for a 2D game.

Since in *Chapter 3*, *Inventory GUI* we already have a basic 2D game, we'll adapt that game to restrict movement to our bounding rectangle.



Insert image 1362OT_08_11.png

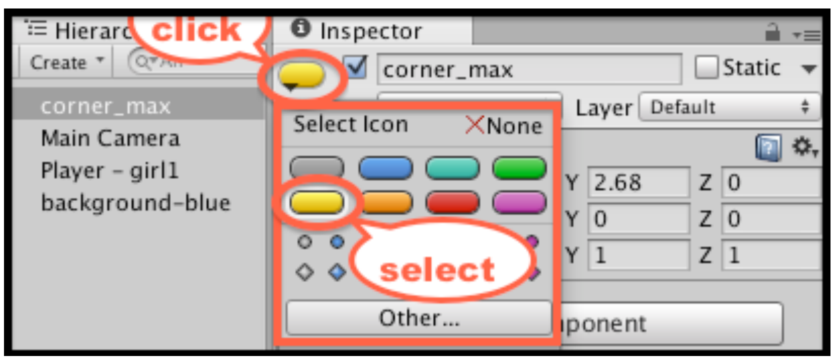
Getting ready

This recipe builds on the simple 2D game *Creating the Simple2DGame_SpaceGirl mini-game for this chapter* in *Chapter 3*, *Inventory GUI*. Start with a copy of that game, or use the provided completed recipe project as the basis for this recipe.

How to do it...

To create a 2D sprite controlled by the user with movement limited within a rectangle follow these steps:

1. Create a new empty gameObject named **corner_max**, and position it somewhere above and to the right of gameObject **player-spaceGirl1**. With this gameObject selected in the **Hierarchy** choose the yellow large oblong icon highlight in the **Inspector** panel.



Insert image 1362OT_08_18.png

2. Duplicate gameObect **corner_max**, naming the clone **corner_min**, and position this clone somewhere below and to the left of gameObject **player-spaceGirl1**. The co-ordinates of these 2 gameObjects will determine the maximum and minimum bounds of movement permitted for the player's character.
3. Modify the C# Script `PlayerMove` to declare some new variables at the beginning of the class:

```
public Transform corner_max;
public Transform corner_min;
private float x_min;
private float y_min;
private float x_max;
private float y_max;
```

4. Modify the C# Script `PlayerMove` so that method `Awake()` now gets a reference to the SpriteRenderer, and uses that object to help set up the maximum and minimum X and Y movement limits:

```
void Awake(){
    rigidBody2D = GetComponent<Rigidbody2D>();
    x_max = corner_max.position.x;
    x_min = corner_min.position.x;
    y_max = corner_max.position.y;
    y_min = corner_min.position.y;
}
```

5. Modify the C# Script `PlayerMove` to declare a new method `KeepWithinMinMaxRectangle()`:

```
    private void KeepWithinMinMaxRectangle(){
        float x = transform.position.x;
        float y = transform.position.y;
        float z = transform.position.z;
        float clampedX = Mathf.Clamp(x, x_min, x_max);
        float clampedY = Mathf.Clamp(y, y_min, y_max);
        transform.position = new Vector3(clampedX, clampedY,
z);
    }
```

6. Modify the C# Script `PlayerMove` to so that having done everything else in method `FixedUpdate()` a call is finally made to method `KeepWithinMinMaxRectangle()`:

```
    void FixedUpdate(){
        float xMove = Input.GetAxis("Horizontal");
        float yMove = Input.GetAxis("Vertical");

        float xSpeed = xMove * speed;
        float ySpeed = yMove * speed;

        Vector2 newVelocy = new Vector2(xSpeed, ySpeed);

        rigidBody2D.velocity = newVelocy;

        // restrict player movement
        KeepWithinMinMaxRectangle();
    }
```

7. With gameObject **player-spaceGirl1** in the **Hierarchy** drag gameObjects **corner_max**, and **corner_min** over the public variables `corner_max` and `corner_min` in the **Inspector**.

8. Before running the scene, in the Scene panel try repositioning gameObjects **corner_max**, and **corner_min**. When you run the scene, the positions of these 2 gameObjects (max and min X and Y) will be used as the limits of movement for the Player's **player-SpaceGirl1** character.
9. While this all works fine, let's make the rectangular bounds of movement visually explicit in the Scene panel, by having a yellow 'gizmo' rectangle drawn. Add the following method to C# script class `PlayerMove`:

```
void OnDrawGizmos(){
    Vector3 top_right = Vector3.zero;
    Vector3 bottom_right = Vector3.zero;
    Vector3 bottom_left = Vector3.zero;
    Vector3 top_left = Vector3.zero;

    if(corner_max && corner_min){
        top_right = corner_max.position;
        bottom_left = corner_min.position;

        bottom_right = top_right;
        bottom_right.y = bottom_left.y;

        top_left = top_right;
        top_left.x = bottom_left.x;
    }

    //Set the following gizmo colors to YELLOW
    Gizmos.color = Color.yellow;

    //Draw 4 lines making a rectangle
    Gizmos.DrawLine(top_right, bottom_right);
    Gizmos.DrawLine(bottom_right, bottom_left);
    Gizmos.DrawLine(bottom_left, top_left);
    Gizmos.DrawLine(top_left, top_right);
}
```

How it works...

You added empty gameObjects **corner_max** and **corner_min** to the scene. The X- and Y- co-ordinates of these gameObjects will be used to determine the bounds of movement we will permit for character **player-SpaceGirl1**. Since these are empty gameObjects, they will not be seen by the player when in play-mode. However we can see, and move them, in the Scene panel, and having added yellow oblong icons we can see their positions and names very easily.

Upon `Awake()` the `PlayerMoveWithLimits` object inside the **player-SpaceGirl1** gameObject records the maximum and minimum X- and Y- values of gameObjects **corner_max**, and **corner_min**. Each time the physics system is called, via the `FixedUpdate()` method, the velocity of the **player-SpaceGirl1** character is set according to the horizontal and vertical keyboard/joystick inputs. However, the final action of method `FixedUpdate()` is to call method `KeepWithinMinMaxRectangle()`, which uses the `Math.Clamp(…)` function to move the character back inside the X- and Y- limits, so that the player's character is not permitted to move outside the area defined by the gameObjects **corner_max**, and **corner_min**.

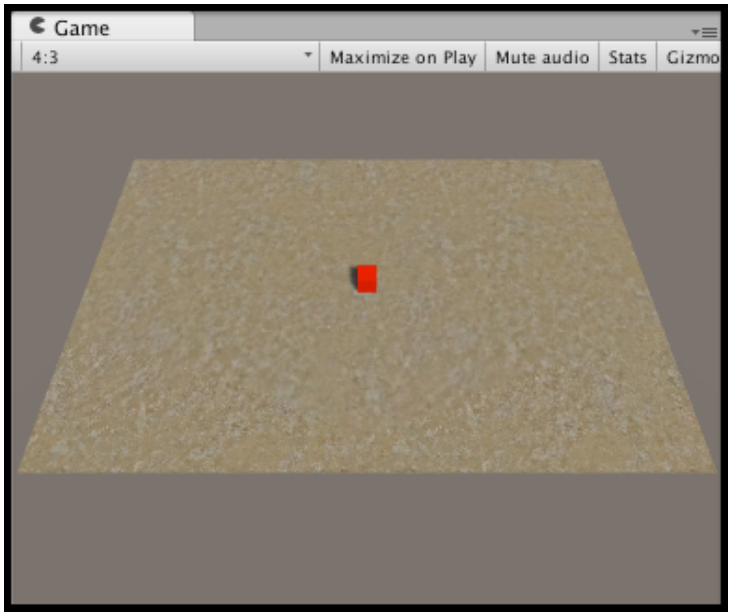
Method `OnDrawGizmos()` tests that the references to gameObjects **corner_max**, and **corner_min** are not null, and then sets the positions of 4 Vector3 objects, representing the 4 corners defined by the a rectangle with **corner_max**, and **corner_min** at opposite corners. It then sets the Gizmo color to yellow, and draws lines connecting the 4 corners in the **Scene** panel.

See also

Refer to the next recipe for more information about limiting player controlled character movements.

Player control of a 3D gameObject (and limiting movement within a rectangle)

Many of the 3D recipes in this chapter are built on this basic project, which constructs a scene with a textured terrain, a **Main Camera**, and a red cube that can be moved around by user with the 4 directional arrow keys. The bounds of movement of the cube are constrained using the same technique as in the previous 2D recipe.



Insert image 1362OT_08_01.png

How to do it...

To create a basic 3D cube controlled game, follow these steps:

1. Create a new, empty 3D project.
2. Once the project has been created, import the single Terrain Texture named `SandAlbedo` (it used to be named `GoodDirt` in Unity 4). Choose menu: **Assets | Import Package | Environments**, deselect everything, and then locate and tick the asset as follows: `Assets/Environment/TerrainAssets/SurfaceTextures/SandAlbedo.psd`.

> NOTE: You could have just added the Environment Asset Package when creating the project – but this would have imported 100s of files, and we only needed just this one … So starting a project in Unity, then selectively importing just what we need is the approach to take if you want to keep project Asset folders to small sizes.

3. Create a terrain, positioned at (-15, 0, -10) and sized 30 by 20.

> NOTE: Transform position for terrains relates to their corner not their center …

Since the Transform position of terrains relates to the corner of the object, we center such objects at (0,0,0) by setting the X-coordinate equal to (-1*width/2), and the Z-coordinate to (-1*length/2). In other words we slide the object by half its width and half its height, to ensure its center is just where we want it.

In this case width is 30 and length is 20, hence we get -15 for X (-1 * 30/2), and -10 for Z (-1 * 20/2).

4. Texture paint this terrain with your texture `SandAlbedo`.
5. Create a directional light (it should face downwards onto the terrain with default settings – but if it doesn't for some reason, then rotate it so the terrain is well lit).
6. Make the following changes to the Main Camera:
    - position = (0, 20, -15)
    - rotation = (60, 0, 0)
7. Change the **Aspect Ratio** of the **Game Panel** from **Free Aspect** to **4:3**. You should now see the whole of the **Terrain** in the **Game Panel**.
8. Create a new empty gameObject named **corner_max**, and position it at (14, 0, 9). With this gameObject selected in the **Hierarchy** choose the yellow large oblong icon highlight in the **Inspector** panel.
9. Duplicate gameObect **corner_max**, naming the clone **corner_min**, and position this clone at (-14, 0, -9). The co-ordinates of these 2 gameObjects will determine the maximum and minumum bounds of movement permitted for the player's character.
10. Create a new **Cube GameObject** named **Cube-player**, at position (0, 0.5, 0) and sized (1,1,1).
11. Add to gameObject **Cube-player** apply a component **Physics | RigidBody**, and uncheck the **RigidBody** property **Use Gravity**.
12. Create a red **Material** named **m_red**, and apply this **Material** to **Cube-player**.
13. Add the following C# script class `PlayerControl` to the **Cube-player**:

```
using UnityEngine;
using System.Collections;

public class PlayerControl : MonoBehaviour {
   public Transform corner_max;
   public Transform corner_min;

   public float speed = 40;
   private Rigidbody rigidBody;

   private float x_min;
```

```
    private float x_max;
    private float z_min;
    private float z_max;

    void Awake (){
        rigidBody = GetComponent<Rigidbody>();
        x_max = corner_max.position.x;
        x_min = corner_min.position.x;
        z_max = corner_max.position.z;
        z_min = corner_min.position.z;
    }

    void FixedUpdate() {
        KeyboardMovement();
        KeepWithinMinMaxRectangle();
    }

    private void KeyboardMovement (){
        float xMove = Input.GetAxis("Horizontal") * speed *
Time.deltaTime;
        float zMove = Input.GetAxis("Vertical") * speed *
Time.deltaTime;

        float xSpeed = xMove * speed;
        float zSpeed = zMove * speed;

        Vector3 newVelocy = new Vector3(xSpeed, 0, zSpeed);

        rigidBody.velocity = newVelocy;

        // restrict player movement
        KeepWithinMinMaxRectangle ();
    }

    private void KeepWithinMinMaxRectangle (){
        float x = transform.position.x;
        float y = transform.position.y;
        float z = transform.position.z;
        float clampedX = Mathf.Clamp(x, x_min, x_max);
        float clampedZ = Mathf.Clamp(z, z_min, z_max);
        transform.position = new Vector3(clampedX, y,
clampedZ);
    }

    void OnDrawGizmos (){
```

```
            Vector3 top_right = Vector3.zero;
            Vector3 bottom_right = Vector3.zero;
            Vector3 bottom_left = Vector3.zero;
            Vector3 top_left = Vector3.zero;

            if(corner_max && corner_min){
               top_right = corner_max.position;
               bottom_left = corner_min.position;

               bottom_right = top_right;
               bottom_right.z = bottom_left.z;

               top_left = bottom_left;
               top_left.z = top_right.z;
            }

            //Set the following gizmo colors to YELLOW
            Gizmos.color = Color.yellow;

            //Draw 4 lines making a rectangle
            Gizmos.DrawLine(top_right, bottom_right);
            Gizmos.DrawLine(bottom_right, bottom_left);
            Gizmos.DrawLine(bottom_left, top_left);
            Gizmos.DrawLine(top_left, top_right);
         }
      }
```

10. With gameObject **Cube-player** selected in the **Hierarchy** drag gameObjects **corner_max**, and **corner_min** over the public variables `corner_max` and `corner_min` in the **Inspector**.
11. When you run the scene, the positions of gameObjects **corner_max** and **corner_min** should define the bounds of movement for the Player's **Cube-player** character.

How it works...

The scene contains a terrain, positioned so its center is `(0,0,0)`. The red cube is controlled by the user's arrow keys, through the script `PlayerControl`.

Just as with the previous 2D recipe, a reference to the (3D) RigidBody component is stored when method `Awake()` executes, and the maximum and minimum X- and Z-values are retrieved from the 2 corner gameObjects and stored in variables `x_min, x_max, z_min`, and `z_max`. Note, for this basic 3D game we won't allow any Y-movement, although such movement (and bounding limits by adding a third 'max-height' corner gameObject) could be easily added by extending the code in this recipe.

Method `KeyboardMovement()` reads the horizontal and vertical input values (which the Unity default settings read from the 4 directional arrow keys). Based on these left-right and up-down values, the velocity of the cube is updated. The amount it will move depends on the speed variable.

Method `KeepWithinMinMaxRectangle()` uses the `Math.Clamp(…)` function to move the character back inside the X- and Z- limits, so that the player's character is not permitted to move outside the area defined by the gameObjects **corner_max**, and **corner_min**.

Method `OnDrawGizmos()` tests that the references to gameObjects **corner_max**, and **corner_min** are not null, and then sets the positions of 4 Vector3 objects, representing the 4 corners defined by the rectangle with **corner_max**, and **corner_min** at opposite corners. It then sets the Gizmo color to **yellow**, and draws lines connecting the 4 corners in the **Scene** panel.

Choosing destinations: Find random or nearest spawn point

Many games make use of spawn points and waypoints. This recipe demonstrates two very common examples of spawning – the choosing of (a) a random spawn point, or (b) the nearest one to an object of interest (such as the Player's character), and then the instantiation of an object at that chosen point.

Getting ready

This recipe builds upon the previous recipe. So make a copy of that project, open it and then follow the steps below.

How to do it...

To find a random spawn point follow these steps:

1. Create a Sphere sized (1,1,1) at position (2,2,2), and apply the **m_red** Material.
2. Create a new prefab named **Prefab-ball**, and drag your **Sphere** into it (and then delete the **Sphere** from the **Hierarchy**).
3. Create a new capsule object named **Capsule-spawnPoint** at (3, 0.5, 3), give it the tag **Respawn** (this is one of the default tags Unity provides).

> NOTE: For testing we'll leave these Respawn points visible. For final game we'd then uncheck the Mesh Rendered of each Respawn gameObject so they are not visible to the Player.

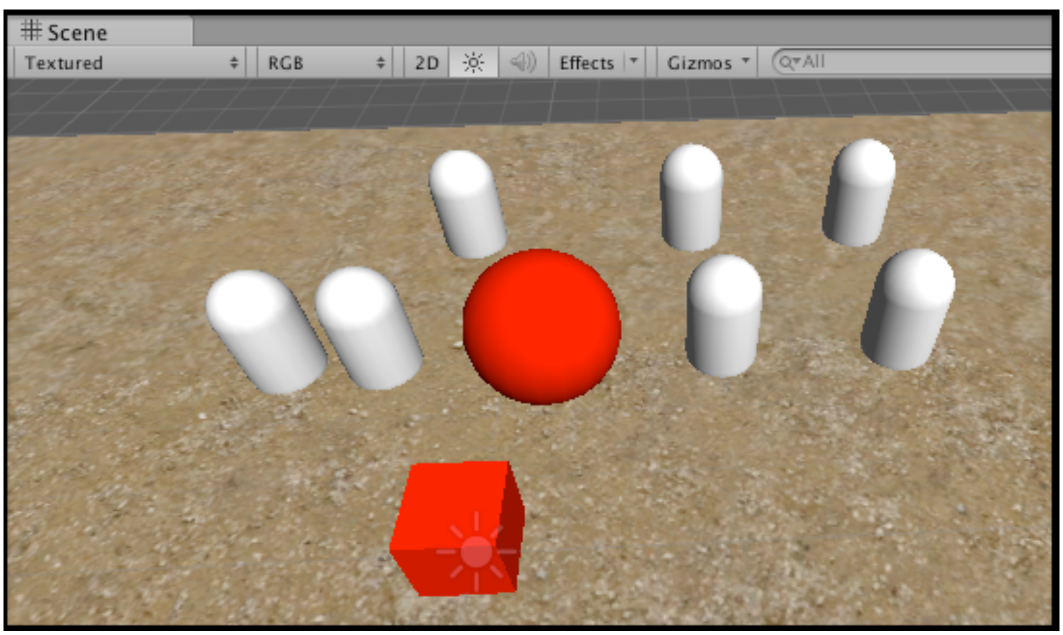
4. Make several copies of your **Capsule-spawnPoint** moving them to different locations on the terrain.
5. Add an instance of the following C# script class `SpawnBall` to gameObject **Cube-player**:

```
using UnityEngine;
using System.Collections;

public class SpawnBall : MonoBehaviour {
   public GameObject prefabBall;
   private SpawnPointManager spawnPointManager;
   private float destroyAfterDelay = 1;
   private float testFireKeyDelay = 0;

   void Start (){
      spawnPointManager = GetComponent<SpawnPointManager>
();
      StartCoroutine("CheckFireKeyAfterShortDelay");
   }

   IEnumerator CheckFireKeyAfterShortDelay () {
      while(true){
         yield return new WaitForSeconds(testFireKeyDelay);
         // having waited, now we check every frame
         testFireKeyDelay = 0;
         CheckFireKey();
      }
   }

   private void CheckFireKey() {
      if(Input.GetButton("Fire1")){
         CreateSphere();
         // wait half-second before alling next spawn
         testFireKeyDelay = 0.5f;
      }
   }

   private void CreateSphere(){
      GameObject spawnPoint =
spawnPointManager.RandomSpawnPoint ();
      GameObject newBall = (GameObject)Instantiate
(prefabBall, spawnPoint.transform.position,
Quaternion.identity);
      Destroy(newBall, destroyAfterDelay);
   }
}
```

6. Add an instance of the following C# script class `SpawnPointManager` to gameObject **Cube-player**:

```
using UnityEngine;
using System.Collections;

public class SpawnPointManager : MonoBehaviour {
   private GameObject[] spawnPoints;

   void Start() {
      spawnPoints =
GameObject.FindGameObjectsWithTag("Respawn");
   }

   public GameObject RandomSpawnPoint (){
      int r = Random.Range(0, spawnPoints.Length);
      return spawnPoints[r];
   }
}
```

7. Ensuring **Cube-player** is selected, in the **Inspector** for the `SpawnBall` scripted component drag **Prefab-ball** over public variable Projectile **Prefab Ball**.
8. Now run your game – when you click the mouse (fire) button, a sphere should be instantiated randomly to one of the capsule locations.



Insert image 1362OT_08_04.png

How it works...

The **Capsule-spawnPoint** objects represent candidate locations where we might wish to create an instance of our ball prefab. When our `SpawnPointManager` object inside gameObject **Cube-player** receives the `Start()` message, GameObject array **respawns** is set to the array returned from the call to `FindGameObjectsWithTag("Respawn")`. This creates an array of all objects in the scene with the tag **`Respawn`** – that is, all our **Capsule-spawnPoint** objects.

When our `SpawnBall` object gameObject **Cube-player** receives the `Start()` message, it sets variable `spawnPointManager` to be a reference to its sibling `SpawnPointManager` script component. Next we start co-routine method `CheckFireKeyAfterShortDelay()`.

Method `CheckFireKeyAfterShortDelay()` uses a typical Unity co-routine technique, going into an infinite loop, but using a delay controlled by the value of variable `testFireKeyDelay`. The delay is to make Unity wait before calling `CheckFireKey()` to test if the user wants a new sphere to be spawned.

> NOTE: Co-routines are an advanced technique, where execution inside the method can be paused, and resumed from the same point. The `Yield` command temporarily halts execution of code in the method allowing Unity to go off and execute code in other gameObjects and undertake physics and rendering work etc. They are perfect for situations where at regularl intervals we wish to check whether something has happended (such as testing for the Fire key or whether a response message has been received from an internet request and so on).
>
> Learn more about Unity co-routines at:
> `http://docs.unity3d.com/Manual/Coroutines.html`

The `SpawnBall` method `CheckFireKey()` tests whether at that instant the user is pressing the *Fire* button. If the *Fire* button is pressed, then method `CreateSphere()` is called. Also variable `testFireKeyDelay` is set to 0.5, this ensure that we won't test the for *Fire* button again until after waiting half a second.

The `SpawnBall` method `CreateSphere()`assigns variable `spawnPoint` to the `GameObject` returned by a call to the `RandomSpawnpoint(…)` method of our `spawnPointManager`. Then it creates a new instance of **prefab_Ball** (via the public variable) at the same position as the `spawnPoint` gameObject.

There's more...

Some details you don't want to miss:

Choosing nearest spawn point

Rather than just choosing a random spawn point, let's search through array `spawnpoints`, and choose the closest one to our player.

To find the nearest spawn point, we need to do the following:

1. Add the following method to C# script class `SpawnPointManager`:

```
public GameObject NearestSpawnpoint (Vector3 source){
   GameObject nearestSpawnPoint = spawnPoints[0];
   Vector3 spawnPointPos =
spawnPoints[0].transform.position;
   float shortestDistance = Vector3.Distance(source,
spawnPointPos);

   for (int i = 1; i < spawnPoints.Length; i++){
      spawnPointPos = spawnPoints[i].transform.position;
      float newDist = Vector3.Distance(source,
spawnPointPos);
      if (newDist < shortestDistance){
         shortestDistance = newDist;
         nearestSpawnPoint = spawnPoints[i];
      }
   }

   return nearestSpawnPoint;
}
```

2. We now need to change the first line in C# class `SpawnBall` so that variable `spawnPoint` is set by a call to our new method `NearestSpawnpoint(…)`:

```
private void CreateSphere(){
   GameObject spawnPoint =
spawnPointManager.NearestSpawnpoint(transform.position);

   GameObject newBall = (GameObject)Instantiate
(prefabBall, spawnPoint.transform.position,
Quaternion.identity);
   Destroy(newBall, lifeDuration);
}
```

In method `NearestSpawnpoint(…)`, we set `nearestSpawnpoint` to the first (array index 0) gameObject in the array, as our default. We then loop through the rest of the array (array index 1 up to `spawnPoints.Length`). For each gameObject in the array we test to see if its distance is less than the shortest distance so far, and if it is, then we update the shortest distance, and also set `nearestSpawnpoint` to the current element. When the array has been searched we return the gameObject that variable `nearestSpawnpoint` refers to.

Avoiding errors due to empty array

Let's make our code a little more robust, so that it can cope with the issue of an empty `spawnPoints` array – that is, when there are no objects tagged **Respawn** in the scene.

To cope with no objects tagged **Respawn** we need to do the following:

1. Improve our `Start()` method in C# script class `SpawnPointManager`: so that an ERROR is logged if the array of objects tagged **Respawn** is empty:

```
public GameObject NearestSpawnpoint (Vector3 source){
void Start() {
   spawnPoints =
GameObject.FindGameObjectsWithTag("Respawn");

   // logError if array empty
   if(spawnPoints.Length < 1) Debug.LogError
("SpawnPointManagaer - cannot find any objects tagged
'Respawn'!");
}
```

2. Improve methods `RandomSpawnPoint()` and `NearestSpawnpoint()` in C# script class `SpawnPointManager` so they still return a **GameObject** even if the array is empty:

```
public GameObject RandomSpawnPoint (){
   // return current gameObject if array empty
   if(spawnPoints.Length < 1) return null;

// the rest as before ...
```

3. Improve method `CreateSphere()` in C# class `SpawnBall` so that we only attempt to instantiate a new gameObject if methods `RandomSpawnPoint()` and `NearestSpawnpoint()` have returned a non-null object reference:

```
private void CreateSphere(){
   GameObject spawnPoint =
spawnPointManager.RandomSpawnPoint ();

   if(spawnPoint){
      GameObject newBall = (GameObject)Instantiate
(prefabBall, spawnPoint.transform.position,
Quaternion.identity);
      Destroy(newBall, destroyAfterDelay);
   }
}
```

See also

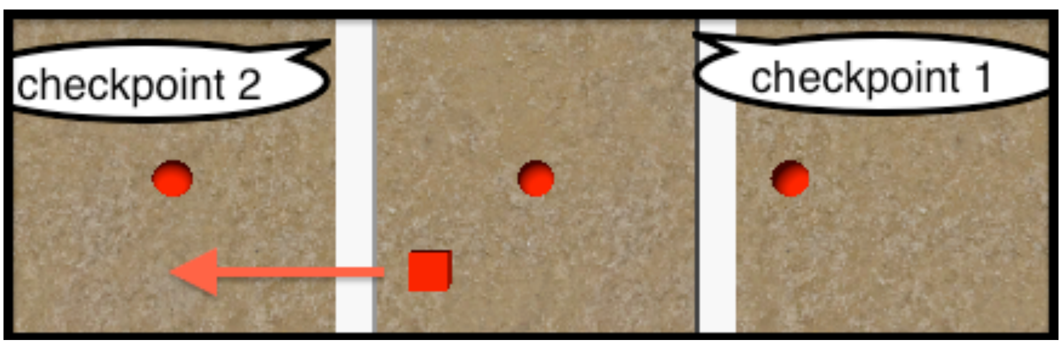
The same techniques and code can be used for selecting spawn points or waypoints.

Refer to the recipe *NPC NavMeshAgent control to follow waypoints in sequence* in this chapter for more information about waypoints.

Choosing destinations: Respawn to most recently passed checkpoint

A *checkpoint* usually represents a certain distance through the game (or perhaps a *track*) in which an agent (user or NPC) has succeeded reaching. Reaching (or passing) checkpoints often results in bonus awards, such as extra time or points or ammo, and so on. Also if a player has multiple lives, then often a player will be respawned only back as far as the most recently passed checkpoint, rather than right to the beginning of the level.

This recipe demonstrates a simple approach to checkpoints, whereby once the player's character has passed a checkpoint, if they die they are moved back only to the most recently passed checkpoint.



Insert image 1362OT_08_09.png

Getting ready

This recipe builds upon the player-controlled 3D cube Unity project you will have created at the beginning of this chapter. So make a copy of that project, open it and then follow the steps for this recipe.

How to do it...

To have respawn position upon losing a life to change depending on passed checkpoints follow these steps:

1. Move gameObject **Cube-player** to position (12, 0.5, 0).
2. Select **Cube-player** in the **Inspector** and add a **Character Controller** component, by clicking **Add Component | Physics | Character Controller** (this is to enable `OnTriggerEnter` collision messages to be received).
3. Create a cube named **Cube-checkpoint-1** at (5, 0, 0), scaled to (1, 1, 20).
4. With **Cube-checkpoint-1** selected check the **Is Trigger** property of its **Box Collider** Component in the **Inspector** panel.
5. Create a tag **CheckPoint**, and assign this tag to **Cube-checkpoint-1**.
6. Duplicate **Cube-checkpoint-1**, naming the clone **Cube-checkpoint-2** positioning it at (-5, 0, 0).
7. Create a sphere named **Sphere-Death** at (7, 0.5, 0). Assign the material **m_red** to this sphere to make it red.
8. With **Sphere-Death** selected check the **Is Trigger** property of its **Sphere Collider** Component in the **Inspector** panel.
9. Create a tag **Death**, and assign this tag to **Sphere-Death**.
10. Duplicate **Sphere-Death**, and positioning this clone at (0, 0.5, 0).
11. Duplicate **Sphere-Death** a second time, and positioning this second clone at (-10, 0.5, 0).
12. Add an instance of the following C# script class `CheckPoints` to gameObject **Cube-player**:

```
using UnityEngine;
using System.Collections;

public class CheckPoints : MonoBehaviour {
   private Vector3 respawnPosition;

   void Start (){
      respawnPosition = transform.position;
   }

   void OnTriggerEnter (Collider hit){
      if(hit.CompareTag("CheckPoint")){
         respawnPosition = transform.position;
      }

      if(hit.CompareTag("Death")){
         transform.position = respawnPosition;
      }
   }
}
```

13. Run the scene. If the cube runs into a red sphere **before** crossing a checkpoint, it will be respawned back to its starting position. Once the red cube has passed a checkpoint, if a red sphere is hit, then the cube will be moved back to the location of the most recent checkpoint that was passed through.

How it works...

C# script class `CheckPoints` has one variable `respawnPosition`, which is a Vector3 referring to the position the player's cube is to be moved to (respawned) if it collides with a **Death** tagged object. The default setting for this is the position of the player's cube when the scene begins – so in method `Start()` we set it to the player's position.

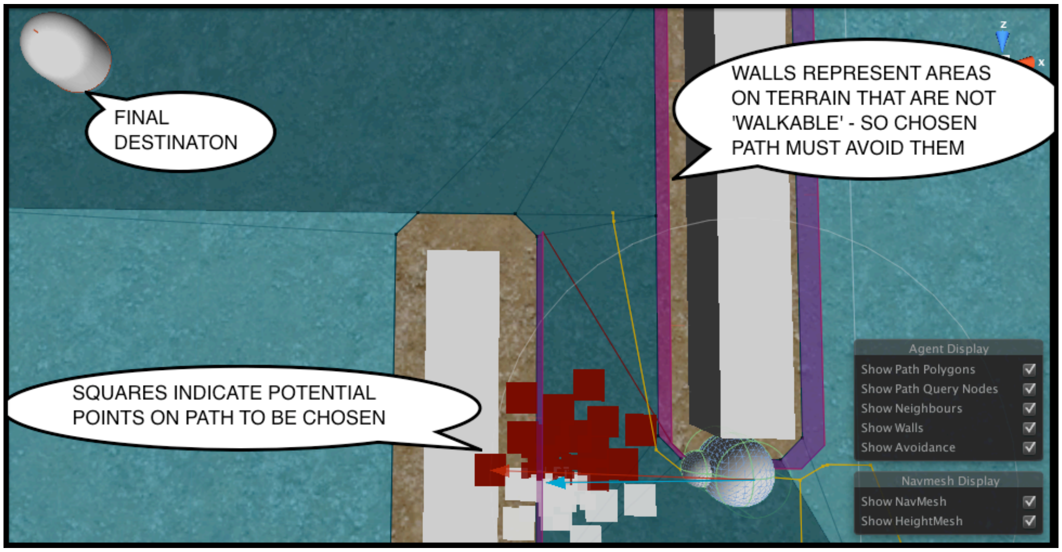
Each time an object tagged **CheckPoint** is collided with, the value of `respawnPosition` is updated to the current position of the player's red cube at that point in time (that is, where it is when it touches the stretched cube tagged **CheckPoint**). So that the next time an object tagged **Death** is hit, the cube will be respawned back to where it last touched an object tagged **CheckPoint**.

NPC NavMeshAgent control to seek or flee destination while avoiding obstacles

The introduction of Unity's NavMeshAgent has greatly simplified the coding for NPC and enemy agent behaviors. In this recipe we add some wall (scaled cubes) obstacles, and generate a NavMesh so Unity knows not to try to walk through the walls. We then add a NavMeshAgent component to our NPC gameObject, and tell it to head to a stated destination location; intelligently planning and following a path there while avoid the wall obstacles.

In the screenshot we can see in the Scene panel the squares representing potential points on the path, and lines showing current temporary direction and destination around the current obstacle.

When the Navigation panel is visible, then the Scene panel displays blue-shaded *walkable* areas, and un-shaded non-walkable areas at the edge of the terrain and around each of the 2 *wall* objects.



Insert image 1362OT_08_05.png

Getting ready

This recipe builds upon the player-controlled 3D cube Unity project you will have created at the beginning of this chapter. So make a copy of that project, open it and then follow the steps for this recipe.

How to do it...

To make an object seek or flee from a position follow these steps:

1. Delete the **Cube-player** gameobject, since we are going to be creating an NPC computer controlled agent.
2. Create a sphere named **Sphere-arrow**, positioned at (2, 0.5, 2) and with scale (1,1,1).
3. Create a second sphere, named **Sphere-small** with scale (0.5, 0.5, 0.5).
4. Child **Sphere-small** to **Sphere-arrow** and position it at (0, 0, 0.5).

> NOTE: Childing refer to making one gameObject in the Hierarcy a child of another gameObject. This is done by dragging the object to be childed over the object to be the parent, and once completed the parent-child relationship is indicated visually by all children being right-indented and positioned immediately below its parent in the Hierarchy panel. If a parent object is

transformed (moved / scaled / rotated) then all its children will also be transformed accordingy.

5. In the **Inspector** add a new NavMeshAgent to **Sphere-arrow**, choose **Add Component | Navigation | Nav Mesh Agent**.
6. Set the **Stopping Distance** property of **NavMeshAgent** component to 2.
7. Add the following C# script class `ArrowNPCMovement` to GameObject **Sphere-Arrow**:

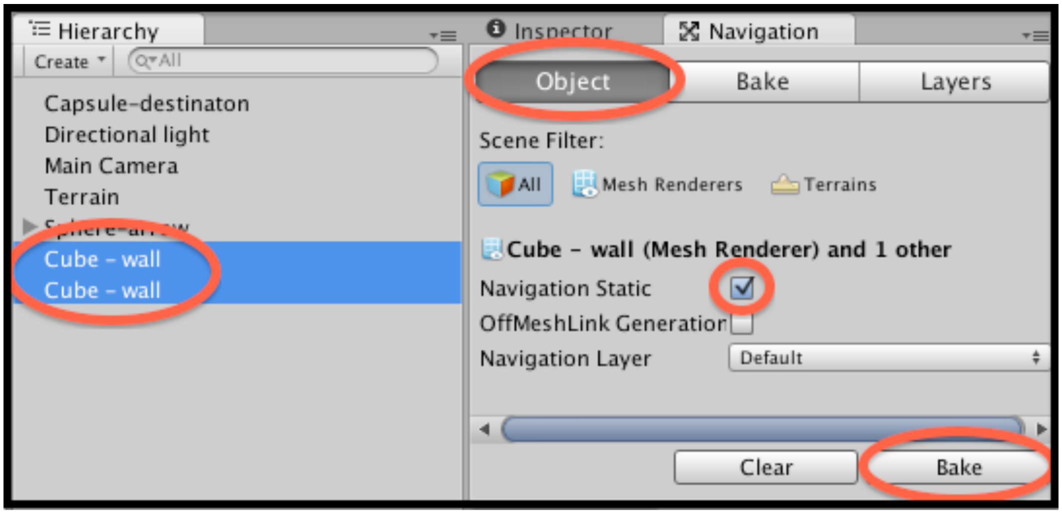
```
using UnityEngine;
using System.Collections;

public class ArrowNPCMovement : MonoBehaviour {
   public GameObject targetGO;
   private NavMeshAgent navMeshAgent;

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

   private void HeadForDestintation (){
      Vector3 destinaton = targetGO.transform.position;
      navMeshAgent.SetDestination (destinaton);
   }
}
```

8. Ensuring **Sphere-arrow** is selected, in the **Inspector** for the `ArrowNPCMovement` scripted component drag **Capsule-destination** over public variable Projectile **Target GO**.
9. Create a 3D Cube, named **Cube-wall** at (-6, 0, 0), scaled to (1, 2, 10).
10. Create another 3D Cube, named **Cube-wall** at (-2, 0, 6), scaled to (1, 2, 7).
11. Display the Navigation panel, by choosing **Window | Navigation**.

NOTE: A great place to *dock* the Navigation panel is next to the Inspector – since you are never using the Inspect and Navigation panels at the same time…

12. In the **Hierarchy** select both of the **Cube-wall** objects (we select the objects that are NOT to be part of the *walkable* parts of our scene), and then in the **Navigation** panel check the **Navigation Static** checkbox. Then click the **Bake** button at the bottom of the **Navigation** panel. When the **Navigation** panel is displayed you'll see a blue *tint* to the parts of the **Scene** that are *walkable*, and

candidate areas for a **NavMeshAgent** to consider as parts of a path to a destination.



Insert image 1362OT_08_06.png

13. Now run your game – you should see the **Sphere-arrow** gameObject automatically move towards the **Capsule-desintation** gameObject, following a path that avoids the 2 wall objects.

How it works...

The **NavMeshAgent** component that we added to gameObject **Sphere-arrow** does most of the work for us. **NavMeshAgents** need 2 things: a destination location to head towards, and a **NavMesh** of the terrain with walkable/non-walkable areas, so that it can plan a path avoiding obstacles. We created two obstacles (the **Cube-wall** objects), and these were selected when we created the **NavMesh** for this scene in the **Navigation** panel.

The location for our NPC object to travel towards is the position of gameObject **Capsule-destination** at (-12, 0, 8), but of course we could just move this object in the **Scene** at **Design-time**, and its new position will be the destination when we run the game.

C# script class `ArrowNPCMovement` has two variables, one is a reference to the destination game object, and the second is a reference to the NavMeshAgent component of the gameObject in which our instance of class `ArrowNPCMovement` is also a component. When the scene starts, via method `Start()`, the **NavMeshAgent** sibling component is found, and method `HeadForDestination()` is called, which sets the destination of the NavMeshAgent to the position of the destination gameObject.

Once the NavMeshAgent has a target to head towards, it will plan a path there and keep moving until it arrives (or gets within the **Stopping Distance** if that parameter has been set to a distance greater than zero).

> NOTE: Ensure the object with the NavMeshAgent component is selected in the Hierarchy at runtime to be able to see this navigation data in the Scene panel.

There's more...

Some details you don't want to miss:

Constantly update NavMeshAgent destination to Player's character current location

Rather than a destination that is fixed when the scene starts, let's allow the **Capsule-destination** object to be moved by the player while the scene is running, and every frame we'll get our NPC arrow to reset the NavMeshAgent's destination to wherever the **Capsule-destination** has been moved to.

To allow user movement of the destination object and frame-by-frame updating of NavMeshAgent destination, we need to do the following:

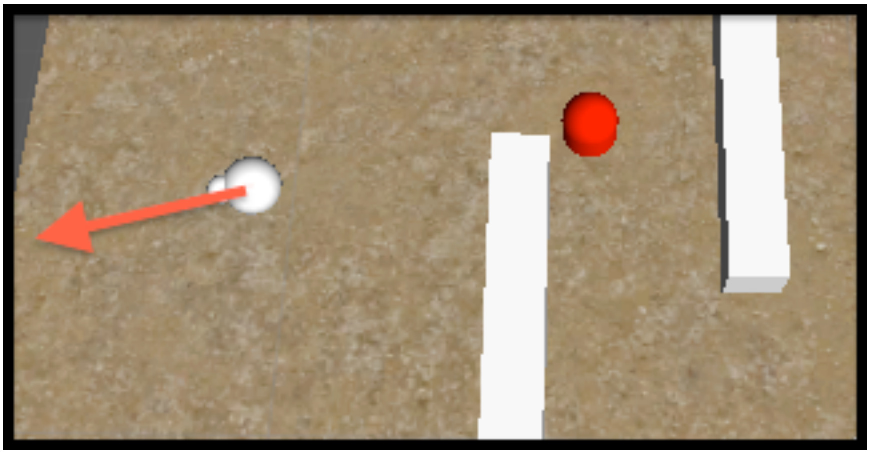
1. Add an instance of C# script class `PlayerControl` as a component of **Capsule-destination**.
2. Update C# script class `ArrowNPCMovement` so that we call method `HeadForDestintation()` every frame, that is, from `Update()`, rather than just once in `Start()`:

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

void Update (){
    HeadForDestintation();
}
```

Now when you run the game you can use the arrow keys to move the destination location, and the NavMeshAgent will update its paths each frame, based on the updated position of the **Capsule-destination** gameObject.

Constantly update NavMeshAgent destination to flee away from Player's character current location

Rather than seeking towards the player's current position, let's make our NPC agent always attempt to flee away from the player's location. For example, an enemy with very low health points might run away and so gain time to regain its health before fighting again.



Insert image 1362OT_08_10.png

To instruct our NavMeshAgent to flee away from the player's location we need to do the following:

1. Replace C# script class `ArrowNPCMovement` with the following:

```
using UnityEngine;
using System.Collections;

public class ArrowNPCMovement : MonoBehaviour {
   public GameObject targetGO;
   private NavMeshAgent navMeshAgent;
   private float runAwayMultiplier = 2;
   private float runAwayDistance;

   void Start(){
      navMeshAgent = GetComponent<NavMeshAgent>();
      runAwayDistance = navMeshAgent.stoppingDistance *
runAwayMultiplier;
   }

   void Update () {
      Vector3 enemyPosition = targetGO.transform.position;
      float distanceFromEnemy =
Vector3.Distance(transform.position, enemyPosition);
      if (distanceFromEnemy < runAwayDistance)
         FleeFromTarget (enemyPosition);
```

```
    }

    private void FleeFromTarget(Vector3 enemyPosition){
        Vector3 fleeToPosition =
Vector3.Normalize(transform.position - enemyPosition) *
runAwayDistance;
        HeadForDestintation(fleeToPosition);
    }

    private void HeadForDestintation (Vector3
destinationPosition){
        navMeshAgent.SetDestination (destinationPosition);
    }
}
```

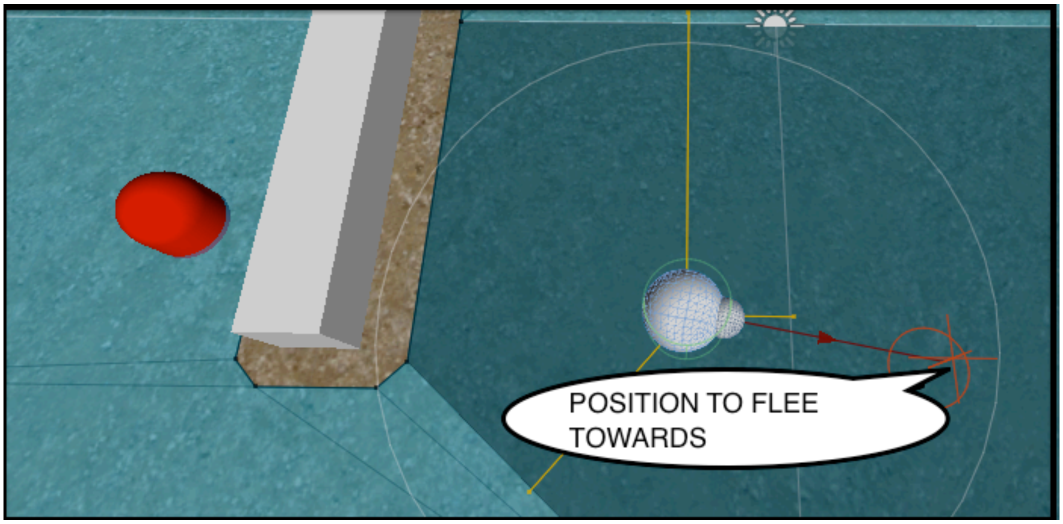
Method `Start()` caches a reference to the NavMeshAgent component, and also calculates the variable `runAwayDistance` to be twice the NavMeshAgent's stopping distance (although this can be changed by changing the value of variable `runAwayMultiplier` accordingly). When the distance to the enemy is less than the value of this variable then we'll instruct the computer-controlled object to flee in the opposite direction.

Method `Update()` calculates whether the distance to the enemy is within the `runAwayDistance`, and if so calls method `FleeFromTarget(…)` passing the location of the enemy as a parameter.

Method `FleeFromTarget(…)` calculates a point that is `runAwayDistance` Unity units away from the Player's cube, in a direction directly away from the computer-controlled object. This is achieved by subtracting the enemy position vector from the current transform's position. Finally method `HeadForDestintation(…)` is called, passing the flee-to position, which results in the NavMeshAgent being told to set location as its new destination.

> NOTE: Unit units are arbitrary, since they are just numbers in a computer. However, in most cases it simplifies things to think of distances in terms of meters (1 Unity unit = 1 meter), and mass in terms of kilograms (1 Unity unit = 1 kilogram). Of course if your game is based on a microscopic world, or a pan-galatic space travel etc. then you need to decide what each Unity unit corresponds to for your game context. See this link for more discussion of units in Unity:
>
> http://forum.unity3d.com/threads/best-units-of-measurement-in-unity.284133/#post-1875487

As the screenshot illustrates, the NavMeshAgent plans a path to the position to flee towards.



Insert image 1362OT_08_17.png

Create a mini point-and-click game

Another way to choose the destination for our **Sphere-arrow** gameObject to have its destination set is by the user clicking an object on screen, and then the **Sphere-arrow** gameObject moving to the location of the clicked object.

To allow the user to select destination objects with point-and-click we need to do the following:

1. Remove the `ArrowNPCMovement` component from gameObject **Sphere-arrow**.
2. Create some target objects, such as a black cube, a blue sphere, and a green cylinder. Note, each object to be a target needs to have a collider component in order to receive `OnMouseOver` event messages (when creating primitives objects from the Unity menu **Create | 3D Object** then colliders are automatically created).
3. Add an instance of the following C# script class `ClickMeToSetDestination` to each of the gameObjects you wish to be a clickable target:

```
using UnityEngine;
using System.Collections;

public class ClickMeToSetDestination : MonoBehaviour {
    private NavMeshAgent playerNavMeshAgent;
```

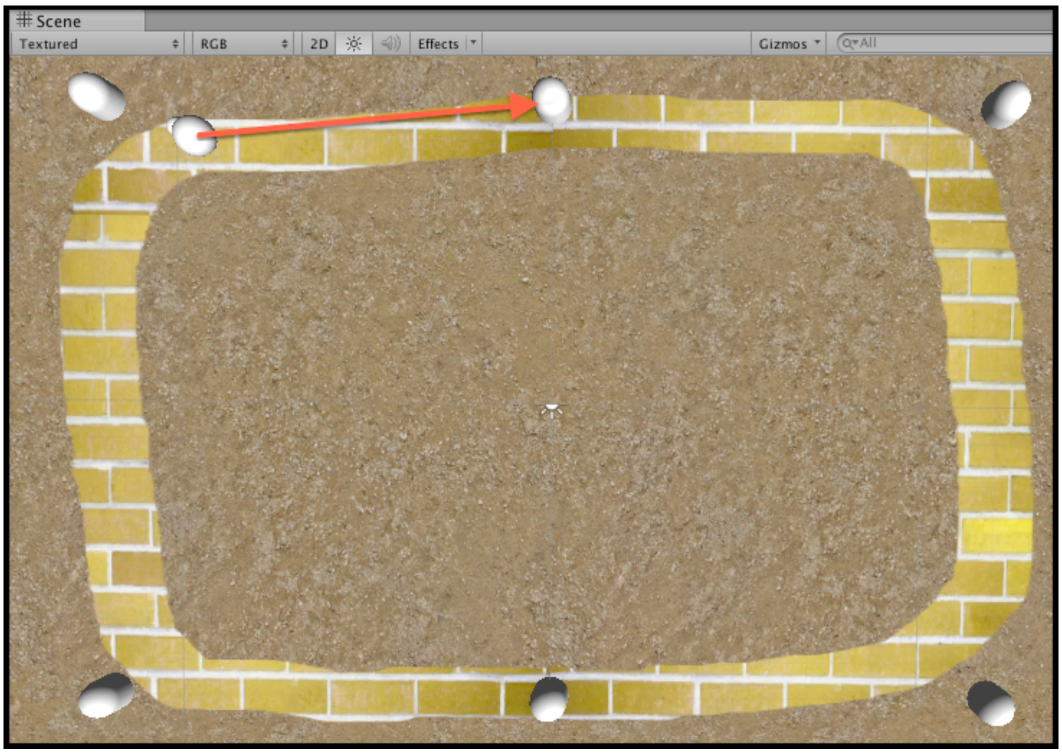
```
    private MeshRenderer meshRenderer;
    private bool mouseOver = false;

    private Color unselectedColor;

    void Start (){
        meshRenderer = GetComponent<MeshRenderer>();
        unselectedColor = meshRenderer.sharedMaterial.color;

        GameObject playerGO =
GameObject.FindGameObjectWithTag("Player");
        playerNavMeshAgent =
playerGO.GetComponent<NavMeshAgent>();
    }

    void Update (){
        if (Input.GetButtonDown("Fire1") && mouseOver)

    playerNavMeshAgent.SetDestination(transform.position);
    }

    void OnMouseOver (){
        mouseOver = true;
        meshRenderer.sharedMaterial.color = Color.yellow;
    }

    void OnMouseExit (){
        mouseOver = false;
        meshRenderer.sharedMaterial.color = unselectedColor;
    }
}
```

Now when you run the game when your mouse is over one of the 3 objects, that object will highlight with a yellow color. If you click the mouse button when the object is highlighted, the **Sphere-arrow** gameObject will make its way up to (but stopping just before) the clicked object.

NPC NavMeshAgent to follow waypoints in sequence

Waypoints are often used as a guide to make autonomously moving NPCs and enemies follow a path in a general way (but be able to respond with other directional behaviors such as flee or seek, if friends/predators/prey are sensed nearby). The waypoints are arranged in a sequence, so that when the character reaches, or gets close to, a waypoint, it will then select the next waypoint in the sequence as the target location to move towards. This recipe demonstrates an *arrow* object moving towards a waypoint, and then when it gets close enough, choosing the next waypoint in the sequence as the new target destination. When the last waypoint has been reached, it starts again heading towards the first waypoint.

Since Unity's NavMeshAgent has simplified coding NPC behavior, our work in this recipe becomes basically finding the position of the next waypoint and then telling the NavMeshAgent that this waypoint is its new destination.



Insert image 1362OT_08_08.png

Getting ready

This recipe builds upon the player-controlled 3D cube Unity project you will have created at the beginning of this chapter. So make a copy of that project, open it and then follow the steps for this recipe.

For this recipe, we have prepared the yellow brick texture image you need in a folder named `Textures` in folder `1362_08_06`.

How to do it...

To instruct an object to follow a sequence of waypoints follow these steps:

1. Delete the **Cube-player** gameobject, since we are going to be creating an NPC computer controlled agent.
2. Create a sphere named **Sphere-arrow**, positioned at (2, 0.5, 2) and with scale (1,1,1).
3. Create a second sphere, named **Sphere-small** with scale (0.5, 0.5, 0.5).
4. Child **Sphere-small** to **Sphere-arrow**, then position it at (0, 0, 0.5).
5. In the **Inspector** add a new NavMeshAgent to **Sphere-arrow**, choose **Add Component | Navigation | Nav Mesh Agent**.
6. Set the **Stopping Distance** property of the **NavMeshAgent** component to 2.
7. Display the **Navigation** panel, by choosing **Window | Navigation**.
8. Click the **Bake** button at the bottom of the **Navigation** panel. When the **Navigation** panel is displayed you'll see a blue *tint* to the parts of the **Scene** that are *walkable*, which should be all parts of the terrain except near the edges.
9. Add an instance of the following C# script class `ArrowNPCMovement` to gameObject **Sphere-arrow**:

```
using UnityEngine;
using System.Collections;

public class ArrowNPCMovement : MonoBehaviour {
   private GameObject targetGO = null;
   private WaypointManager waypointManager;
   private NavMeshAgent navMeshAgent;

   void Start (){
      navMeshAgent = GetComponent<NavMeshAgent>();
      waypointManager = GetComponent<WaypointManager>();
      HeadForNextWayPoint();
   }

   void Update (){
      float closeToDestinaton =
navMeshAgent.stoppingDistance * 2;
      if (navMeshAgent.remainingDistance <
closeToDestinaton){
         HeadForNextWayPoint ();
      }
   }

   private void HeadForNextWayPoint (){
```

```
        targetGO = waypointManager.NextWaypoint (targetGO);
        navMeshAgent.SetDestination
(targetGO.transform.position);
    }
}
```

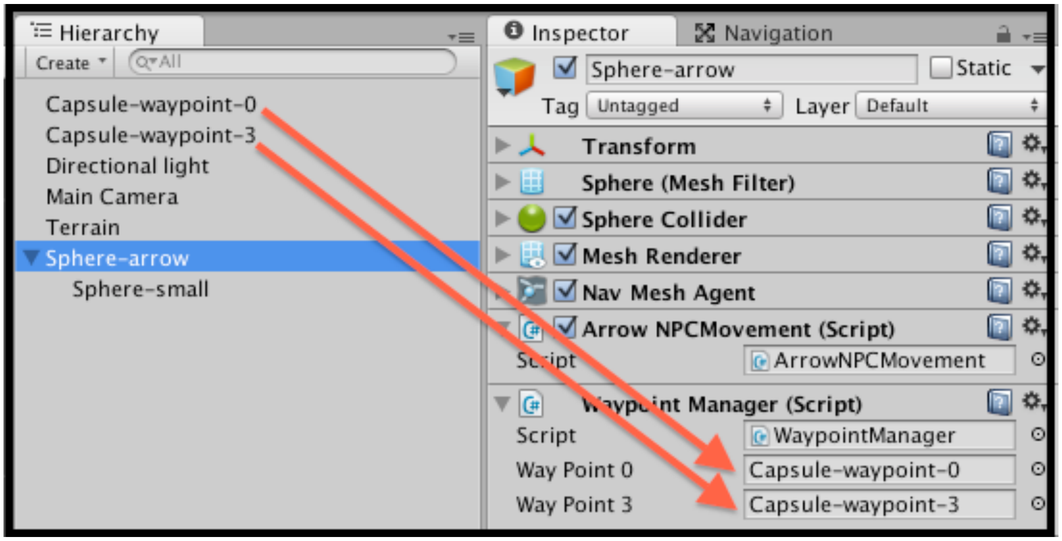
10. Create a new capsule object named **Capsule-waypoint-0** at (-12, 0, 8), give it the tag **waypoint**.
11. Copy **Capsule-waypoint -0** naming the copy **Capsule-waypoint -3** and position this copy at (8, 0, -8).

> NOTE: We are going to add some intermediate waypoints numbered 1 and 2 later on – that's why our second waypoint here is numbered 3, in case you were wondering …

12. Add the following C# script class `WaypointManager` to GameObject **Sphere-arrow**:

```
using UnityEngine;

public class WaypointManager : MonoBehaviour {
    public GameObject wayPoint0;
    public GameObject wayPoint3;

    public GameObject NextWaypoint(GameObject current){
        if(current == wayPoint0)
            return wayPoint3;
        else
            return wayPoint0;
    }
}
```

13. Ensuring **Sphere-arrow** is selected, in the **Inspector** for the `WaypointManager` scripted component drag **Capsule-waypoint-0** and **Capsule-waypoint -3** over public variable Projectile **Way Point 0** and **Way Point 3** respectively.



Insert image 1362OT_08_07.png

14. Display the Navigation panel, by choosing **Window | Navigation**.
15. Click the **Bake** button at the bottom of the **Navigation** panel. When the **Navigation** panel is displayed you'll see a blue *tint* to the parts of the **Scene** that are *walkable*, which should be all parts of the terrain except near the edges.
16. Now run your game – the arrow object should first move towards one of the waypoint capsules, then when it gets close to it, it should slow down, turn around and head towards the other waypoint capsule, and keep doing that continuously.

How it works...

The **NavMeshAgent** component that we added to gameObject **Sphere-arrow** does most of the work for us. **NavMeshAgents** need 2 things: a destination location to head towards, and a **NavMesh**, so that it can plan a path avoiding obstacles.

We created two possible waypoints to be the location for our NPC to move towards: **Capsule-waypoint-0** and **Capsule-waypoint -3**.

C# script class `WaypointManager` has one job – to return a reference to the *next* waypoint our NPC should head towards. There are two variables, `wayPoint0` and `wayPoint3`, references to the 2 waypoint gameObjects in our scene. Method `NextWaypoint(…)` takes a single parameter named `current`, which is a reference to the current waypoint the object was moving towards (or null), and this method's task is to return a reference to the **next** waypoint the NPC should travel towards. The logic for this method is simple, if `current` refers to `waypoint0`, then we'll return `waypoint3`, otherwise we'll return `waypoint0`. Note, if we pass this method `null`, then we'll get `waypoint0` back (so it is our default first waypoint).

C# script class `ArrowNPCMovement` has three variables; one is a reference to the destination game object named `targetGO`. The second is a reference to the `NavMeshAgent` component of the gameObject in which our instance of class `ArrowNPCMovement` is also a component. The third variable `waypointManager` is a reference to the sibling scripted component, an instance of our `WaypointManager` script class.

When the scene starts, via method `Start()`, the **NavMeshAgent** and `WaypointManager` sibling components are found, and method `HeadForDestination()` is called.

Method `HeadForDestination()` first sets variable `targetGO` to refer to the gameObject that is returned by a call to the `NextWaypoint(…)` of scripted component `waypointManager` (that is, `targetGO` is set to refer to either **Capsule-waypoint-0** and **Capsule-waypoint -3**). Next it instructs the `NavMeshAgent` make its destination the position of gameObject `targetGO`.

Each frame method `Update()` is called. A test is made to see if the distance from the NPC arrow object is close to the destination waypoint. If the distance is smaller than twice the *stopping distance* set in our `NavMeshAgent`, then a call is made to `waypointManager.NextWaypoint(…)`to update our target destination to be the next waypoint in the sequence.

There's more...

Some details you don't want to miss:

More efficient to avoid using navmeshes for waypoints

Navmeshes are far superior to way points since a location in a general area (not a specific point) can be used and the path finding algorithm will automatically find the shortest route. For a succinct recipe (such as the above) we can simplify the implementation of waypoints using navmeshes for calculating movements for us. However, for optimized real-world games the most common way to move from one waypoint to the next is via linear interpolation or implementing Craig Reynold's Seek algorithm (for details follow the link listed in the *Conclusion* section at the end of this chapter).

Working with arrays of waypoints

Having a separate C# script class `WaypointManager` to simply swap between **Capsule-waypoint-0** and **Capsule-waypoint -3** may have seemed heavy duty over-engineering, but this was actually a very good move. An object of script class `WaypointManager` has the job of returning the *next* waypoint. It is now very straightforward to add a more sophisticated approach of having an array of waypoints, without us having to change any code in of script class `ArrowNPCMovement`. We could choose a **random** waypoint to be the next destination (see recipe *Choosing destinations: Find nearest (or a random) spawnpoint*). Or we could have an array of waypoints, and choose the next one in the **sequence**.

To improve our game to work with an array of waypoints in the sequence to be followed we need to do the following:

1. Copy **Capsule-waypoint -0** naming the copy **Capsule-waypoint -1** and position this copy at (0, 0, 8).
2. Make 4 more copies (named **Capsule-waypoint-1,2,4,5**) positioning them as follows:
   - **Capsule-waypoint-1**: position = (-2, 0, 8).
   - **Capsule-waypoint-2**: position = (8, 0, 8).
   - **Capsule-waypoint-4**: position = (-2, 0, -8).
   - **Capsule-waypoint-5**: position = (-12, 0, -8).
3. Replace C# script class `WaypointManager` with the following code:

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

public class WaypointManager : MonoBehaviour {
   public GameObject[] waypoints;

   public GameObject NextWaypoint (GameObject current)
   {
      if( waypoints.Length < 1)
         Debug.LogError ("WaypointManager:: ERROR - no
waypoints have been added to array!");

      int currentIndex = Array.IndexOf(waypoints, current);
      int nextIndex = ((currentIndex + 1) %
waypoints.Length);
      return waypoints[nextIndex];
   }
}
```

4. Ensuring **Sphere-arrow** is selected, in the **Inspector** for the `WaypointManager` scripted component set the size of the Waypoints array to 6. Now drag in all 6 capsule waypoint objects **Capsule-waypoint -0/1/2/3/4/5**.
5. Run the game. Now the **Sphere-arrow** gameObject should first move towards waypoint 0 (top left, and then follow the sequence around the terrain).

6. Finally, you could make it look like the Sphere is following a *yellow brick road*. Import the provided yellow brick texture, add this to your terrain, and Terrain paint texture an oval-shaped path between the waypoints. You may also uncheck the Mesh Rendered component for each waypoint capsule, so the user does not see any of the way points, just the arrow object following the yellow brick path …

In method `NextWaypoint(…)` first we check in case the array is empty, in which case an error is logged. Next the array index for the current waypoint gameObject is found (if present in the array). Finally the array index for the next waypoint is calculated, using a modulus operator to support a cyclic sequence returning to the beginning of the array after the last element has been visited.

Increased flexibility with a WayPoint class

Rather than forcing a gameObject to follow a single rigid sequence of locations, we can make things more flexible by defining a WayPoint class, whereby each waypoint gameObject has an array of possible destinations, and each of those has its own array and so on. In this way a di-graph (directed graph) can be implemented, of which a linear sequence is just one possible instance.

To improve our game to work with di-graph of waypoints do the following:

1. Remove the scripted `WayPointManager` component from gameObject **Sphere-arrow**.
2. Replace C# script class `ArrowNPCMovement` with the following code:

```
using UnityEngine;
using System.Collections;

public class ArrowNPCMovement : MonoBehaviour {
   public Waypoint waypoint;
   private bool firstWayPoint = true;
   private NavMeshAgent navMeshAgent;

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

   void Update () {
      float closeToDestinaton =
navMeshAgent.stoppingDistance * 2;
      if (navMeshAgent.remainingDistance <
closeToDestinaton){
         HeadForNextWayPoint ();
      }
```

```
    }

    private void HeadForNextWayPoint (){
       if(firstWayPoint)
          firstWayPoint = false;
       else
          waypoint = waypoint.GetNextWaypoint();

       Vector3 target = waypoint.transform.position;
       navMeshAgent.SetDestination (target);
    }
}
```

3. Create a new C# script class `WayPoint` with the following code:

```
using UnityEngine;
using System.Collections;

public class Waypoint: MonoBehaviour {
   public Waypoint[] waypoints;

   public Waypoint GetNextWaypoint () {
      return waypoints[ Random.Range(0, waypoints.Length)
];
   }
}
```

4. Select all 6 gameObjects **Capsule-waypoint -0/1/2/3/4/5** and add to them a scripted instance of C# class `WayPoint`.
5. Select gameObject **Sphere-arrow** and add to it a scripted instance of C# class `WayPoint`.
6. Ensuring gameObject **Sphere-arrow** is selected, in the **Inspector** for the `ArrowNPCMovement` scripted component drag **Capsule-waypoint-0** into the **Waypoint** public variable slot.
7. Now we need to link **Capsule-waypoint-0** to **Capsule-waypoint-1**, **Capsule-waypoint-1** to **Capsule-waypoint -2**, and so on. Select **Capsule-waypoint-0**, set its Waypoints array size to 1, and drag in **Capsule-waypoint-1**. Next select **Capsule-waypoint-1**, set its Waypoints array size to 1, and drag in **Capsule-waypoint-2**. Do the following, until you finally link **Capsule-waypoint-5** back to **Capsule-waypoint-0**.

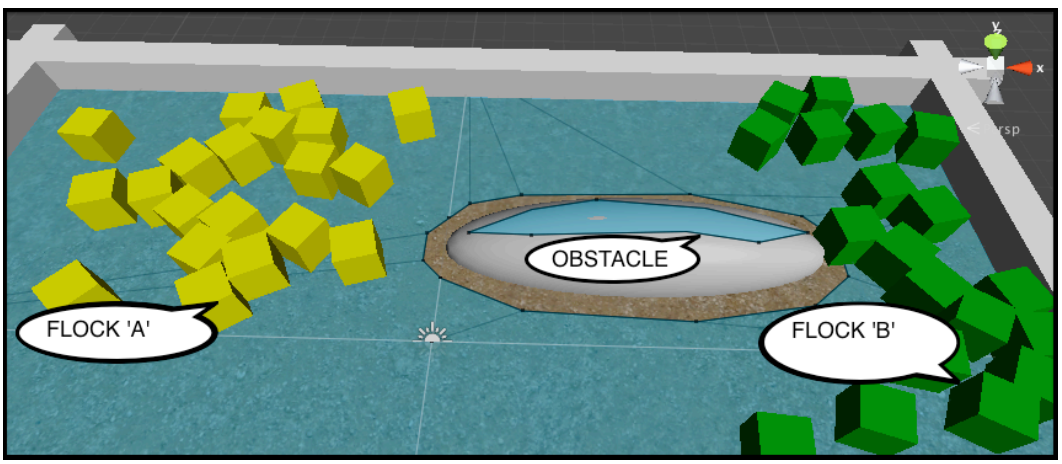
You now have a much more flexible game architecture, allowing gameObjects to randomly select one of several different paths at each waypoint reached. In this final recipe variation we have implemented a waypoint sequence, since each waypoint has an array of just one linked waypoint. However, if you change the array size to 2 or more, then you will then be creating a graph of linked waypoints, adding random variations in the sequence of waypoints a computer controlled character follows for any given run of your game.

Controlling object group movement through flocking

Realistic, natural looking flocking behavior (for example birds or antelopes or bats) can be created through creating collections of objects with the following 4 simple rules:

- Separation – avoid getting too close to neighbors
- Avoid Obstacle – turn away from an obstacle immediately ahead
- Alignment – move in the general direction the flock is heading
- Cohesion – move towards the location in the middle of the flock

Each member of the *flock* acts independently, but needs to know about the current heading and location of the members of its flock. This recipe shows you how to create a scene with 2 flocks of cubes, one flock of green cubes and one flock of yellow cubes. To keep things simple, we'll not worry about separation in our recipe.



Insert image 1362OT_08_15.png

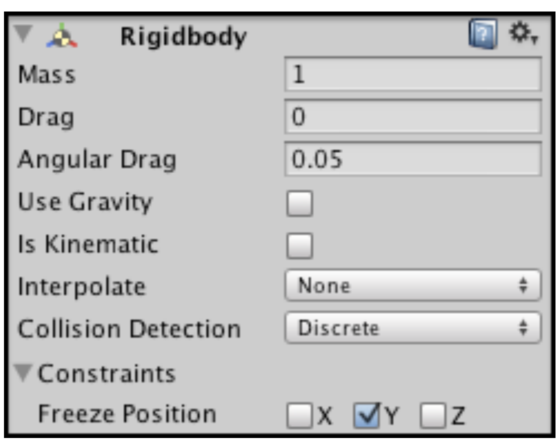
Getting ready

This recipe builds upon the player-controlled cube Unity project you will have created in the first recipe. So make a copy of that project, open it and then follow the steps for this recipe.

How to do it...

To make a group of objects flock together, please follow these steps:

1. Create a Material in the Project panel, named **m_green**, with Main Color tinted green.
2. Create a Material in the Project panel, named **m_yellow**, with Main Color tinted yellow.
3. Create a 3D Cube GameObject named **Cube-drone**, at (0,0,0). Drag Material **m_yellow** into this object.
4. Add a **Navigation | NavMeshAgent** component to **Cube-drone**. Set the **Stopping Distance** property of the **NavMeshAgent** component to 2.
5. Add a **Physics RigidBody** component to **Cube-drone** with the following properties:
    - **Mass** is 1
    - **Drag** is 0
    - **Angular Drag** is 0.05
    - **Use Gravity** and **Is Kinematic** are both un-checked
    - Under Constrains **Freeze Position** for the Y-axis is checked
6. You should see the following inspector values for your cube's rigid body component:



Insert image 1362OT_08_16.png

7. Create the following C# script class `Drone`, and add an instance as a component to gameObject **Cube-drone**:

```
using UnityEngine;
using System.Collections;

public class Drone : MonoBehaviour {
   private NavMeshAgent navMeshAgent;

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

   public void SetTargetPosition(Vector3
swarmCenterAverage, Vector3 swarmMovementAverage) {
      Vector3 destination = swarmCenterAverage +
swarmMovementAverage;
      navMeshAgent.SetDestination(destination);
   }
}
```

8. Create a new empty Prefab named **dronePrefabYellow**, and from the **Hierarchy** panel drag your **Cube-boid** GameObject into this prefab.
9. Now drag Material **m_green** into gameObject **Cube-boid**.
10. Create a new empty Prefab named **dronePrefabGreen**, and from the **Hierarchy** panel drag your **Cube-drone** GameObject into this prefab.
11. Delete gameObject **Cube-drone** from the **Scene** panel.
12. Add the following C# script `Swarm` class to the **Main Camera**:

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

public class Swarm : MonoBehaviour {
   public int droneCount = 20;
   public GameObject dronePrefab;

   private List<Drone> drones = new List<Drone>();

   void Awake() {
      for (int i = 0; i < droneCount; i++)
         AddDrone();
   }
```

```
    void FixedUpdate() {
        Vector3 swarmCenter = SwarmCenterAverage();
        Vector3 swarmMovement = SwarmMovementAverage();

        foreach(Drone drone in drones )
            drone.SetTargetPosition(swarmCenter,
swarmMovement);
    }

    private void AddDrone() {
        GameObject newDroneGO =
(GameObject)Instantiate(dronePrefab);
        Drone newDrone = newDroneGO.GetComponent<Drone>();
        drones.Add(newDrone);
    }

    private Vector3 SwarmCenterAverage() {
        // cohesion (swarm center point)
        Vector3 locationTotal = Vector3.zero;

        foreach(Drone drone in drones )
            locationTotal += drone.transform.position;

        return (locationTotal / drones.Count);
    }

    private Vector3 SwarmMovementAverage() {
        // alignment (swarm direction average)
        Vector3 velocityTotal = Vector3.zero;

        foreach(Drone drone in drones )
            velocityTotal += drone.rigidbody.velocity;

        return (velocityTotal / drones.Count);
    }
}
```

13. With **Main Camera** selected in the **Hierarchy**, then drag **prefab_boid_yellow** from the **Project** panel over the public variable of **Drone** *Prefab*.
14. With **Main Camera** selected in the **Hierarchy**, add a second instance of script class `Swarm` to this gameObject, and then drag **prefab_boid_green** from the **Project** panel over the public variable of **Drone** *Prefab*.
15. Create a new Cube named **wall-left**, with the following properties:
    - Position = (-15, 0.5, 0)

- Scale = (1, 1, 20)

16. Duplicate object **wall-left** naming the new object **wall-right**, and change the position of **wall-right** to (15, 0.5, 0).
17. Create a new Cube named **wall-top**, with the following properties:
    - Position = (0, 0.5, 10)
    - Scale = (31, 1, 1)
18. Duplicate object **wall-top** naming the new object **wall-bottom**, and change the position of **wall-bottom** to (0, 0.5, -10).
19. Create a new Sphere named **Sphere-obstacle**, with the following properties:
    - Position = (5, 0, 3)
    - Scale = (10, 3, 3)
20. In the **Hierarchy** select the **Sphere-obstacle** gameObject and then in the **Navigation** panel check the **Navigation Static** checkbox. Then click the **Bake** button at the bottom of the **Navigation** panel.
21. Finally, make the player's red cube larger, set its scale to (3,3,3).

How it works...

The `Swarm` class contains three variables:

- **Integer** `droneCount`, the number of swam members to create
- **GameObject** `dronePrefab` – reference to the prefab to be cloned to create swarm members
- **List** of `Drone` object references `drones`, a list of all the scripted `Drone` components inside all the swarm objects that have been created

Upon creation as the scene starts, the `Swarm` script class `Awake()` method loops to create `droneCount` swarm members by repeatedly calling method `AddDrone()`. This method instantiates a new `GameObject` from the prefab, and then sets variable `newDrone` to be a reference to the Drone scripted object inside the new swarm member. Each frame method `FixedUpdate()`loops through the list of `Drone` objects, calling their `SetTargetPosition(…)` method, passing in the swam center location and the average of all the swarm member velocities.

The rest of this Swarm class is made up of two methods, one (`SwarmCenterAverage`) returns a Vector3 object representing the average position of all the `Drone` objects, and the other (`SwarmMovementAverage`) returns a `Vector3` object representing the average velocity (movement force) of all the `Drone` objects.

- `SwarmMovementAverage()`:

    - What is the general direction the swarm is moving?
    - This is known as *alignment* – a swarm member attempting to move in the same direction as the swarm average
- `SwarmCenterAverage()`:
    - What is the center position of the swarm?
    - This is known as *cohesion* – a swarm member attempting to move towards the center of the swarm

The core work is undertaken by the `Drone` class. Each drone's method `Start(…)` finds and caches a reference to its NavMeshAgent component.

Each drone's `UpdateVelocity(…)` method takes as input two Vector3 arguments: `swarmCenterAverage` and `swarmMovementAverage`. This method then calculates the desired new velocity for this Drone (by simply adding the 2 vectors), and then uses the result (a Vector3 location) to update the NavMeshAgent's target location.

There's more...

Some details you don't want to miss:

Learn more about flocking Artificial Intelligence

Most of the flocking models in modern computing owe much to the work of Craig Reynolds in the 1980s. Learn more about Craig and his *boids* program at this URL:

http://en.wikipedia.org/wiki/Craig_Reynolds_(computer_graphics)

Conclusion

In this chapter, we have introduced recipes demonstrating a range of player and computer controlled characters, vehicles, and objects. Player character controllers are fundamental to the usability experience of every game, while NPC objects and characters add rich interactions to many games.

- Learn more about Unity NavMeshes from this Unity tutorial:
    - http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/navmeshes
- Learn more about Unity 2D character controllers:
    - http://unity3d.com/learn/tutorials/modules/beginner/2d/2d-controllers

- Lean lots about computer-controlled moving gameObjects from the classic paper entitled "Steering Behaviors For Autonomous Characters" by Craig W. Reynolds, presented at the GDC-99 (Game Developer's Conference):
    - `http://www.red3d.com/cwr/steer/gdc99/`
- Learn about the Unity 3D character component and control
    - `http://docs.unity3d.com/Manual/class-CharacterController.html`
    - `http://unity3d.com/learn/tutorials/projects/survival-shooter/player-character`

Every game needs textures – here are some sources of free textures suitable for many games:

- CG Textures:
  `http://www.cgtextures.com/`
- Naldz Graphics blog
  `http://naldzgraphics.net/textures/`