5
Flocks and Crowds

Flocks and crowds are other essential concepts we'll be exploring in this book. Luckily, flocks are very simple to implement, and they add a fairly extraordinary amount of realism to your simulation in just a few lines of code. Crowds can be a bit more complex, but we'll be exploring some of the powerful tools that come bundled with Unity to get the job done. In this chapter, we'll cover the following topics:

- Learning the history of flocks and herds
- Understanding the concepts behind flocks
- Flocking using the Unity concepts
- Flocking using the traditional algorithm
- Using realistic crowds

Learning the origins of flocks

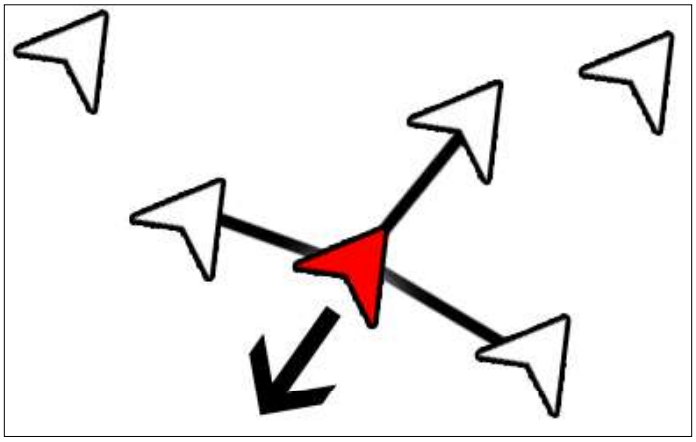
The flocking algorithm dates all the way back to the mid-80s. It was first developed by Craig Reynolds, who developed it for its use in films, the most famous adaptation of the technology being the swarm of bats in *Batman Returns* in 1992, for which he won an Oscar. Since then, the use of the flocking algorithm has expanded beyond the world of film into various fields from games to scientific research. Despite being highly efficient and accurate, the algorithm is also very simple to understand and implement.

Understanding the concepts behind flocks and crowds

As with previous concepts, it's easiest to understand flocks and herds by relating them to the real-life behaviors they model. As simple as it sounds, these concepts describe a group of objects, or boids, as they are called in artificial intelligence lingo, moving together as a group. The flocking algorithm gets its name from the behavior birds exhibit in nature, where a group of birds follow one another toward a common destination, keeping a mostly fixed distance from each other. The emphasis here is on the group. We've explored how singular agents can move and make decisions on their own, but flocks are a relatively computationally efficient way of simulating large groups of agents moving in unison while modeling unique movement in each boid that doesn't rely on randomness or predefined paths.

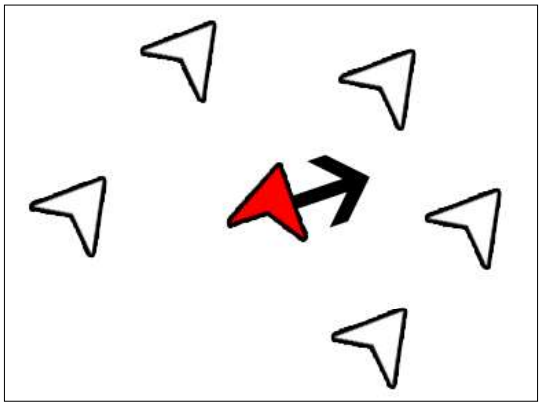
We'll implement two variations of flocking in this chapter. The first one will be based on a sample flocking behavior found in a demo project called Tropical Paradise Island. This demo came with Unity in Version 2.0, but has been removed since Unity 3.0. For our first example, we'll salvage this code and adapt it to our Unity 5 project. The second variation will be based on Craig Reynold's flocking algorithm. Along the way, you'll notice some differences and similarities, but there are three basic concepts that define how a flock works, and these concepts have been around since the algorithm's introduction in the 80s:

- **Separation**: This means to maintain a distance with other neighbors in the flock to avoid collision. The following diagram illustrates this concept:



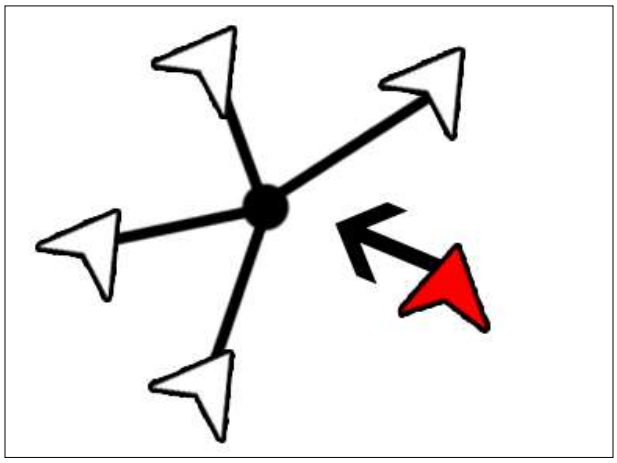
Here, the middle boid is shown moving in a direction away from the rest of the boids, without changing its heading

- **Alignment**: This means to move in the same direction as the flock, and with the same velocity. The following figure illustrates this concept:



Here, the boid in the middle is shown changing its heading toward the arrow to match the heading of the boids around it

- **Cohesion**: This means to maintain a minimum distance with the flock's center. The following figure illustrates this concept:

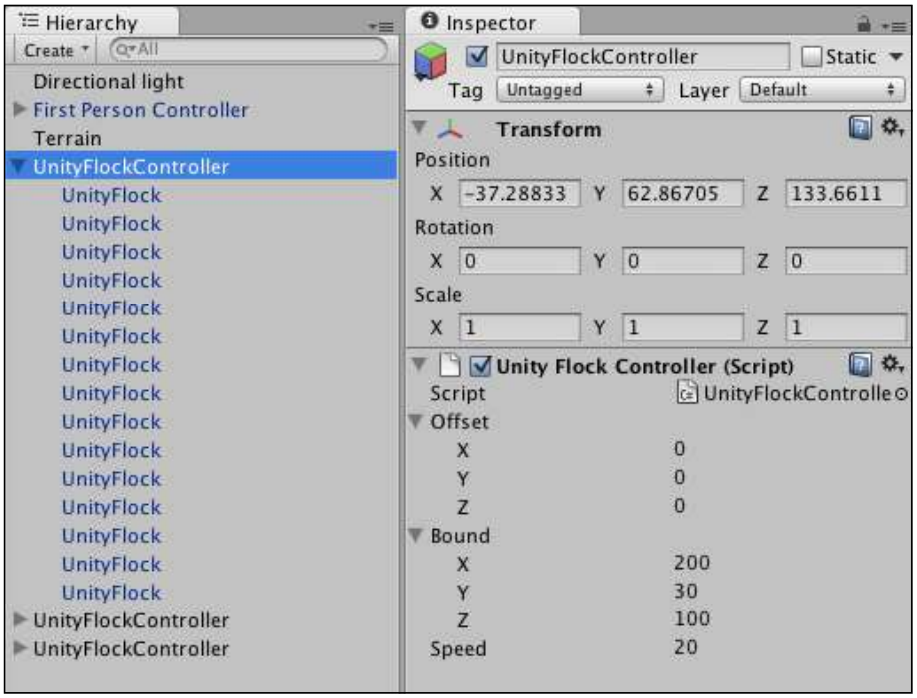


Here, the boid to the right of the rest moves in the direction of the arrow to be within the minimum distance to its nearest group of boids

Flocking using Unity's samples

In this section, we'll create our own scene with flocks of objects and implement the flocking behavior in C#. There are two main components in this example: the individual boid behavior and a main controller to maintain and lead the crowd.

Our scene hierarchy is shown in the following screenshot:



The scene hierarchy

As you can see, we have several boid entities, `UnityFlock`, under a controller named `UnityFlockController`. The `UnityFlock` entities are individual boid objects and they'll reference to their parent `UnityFlockController` entity to use it as a leader. The `UnityFlockController` entity will update the next destination point randomly once it reaches the current destination point.

The `UnityFlock` prefab is a prefab with just a cube mesh and a `UnityFlock` script. We can use any other mesh representation for this prefab to represent something more interesting, like birds.

Mimicking individual behavior

Boid is a term, coined by Craig Reynold, that refers to a bird-like object. We'll use this term to describe each individual object in our flock. Now let's implement our boid behavior. You can find the following script in `UnityFlock.cs`, and this is the behavior that controls each boid in our flock.

The code in the `UnityFlock.cs` file is as follows:

```
using UnityEngine;
using System.Collections;

public class UnityFlock : MonoBehaviour {
  public float minSpeed = 20.0f;
  public float turnSpeed = 20.0f;
  public float randomFreq = 20.0f;
  public float randomForce = 20.0f;

  //alignment variables
  public float toOriginForce = 50.0f;
  public float toOriginRange = 100.0f;

  public float gravity = 2.0f;

  //seperation variables
  public float avoidanceRadius = 50.0f;
  public float avoidanceForce = 20.0f;

  //cohesion variables
  public float followVelocity = 4.0f;
  public float followRadius = 40.0f;

  //these variables control the movement of the boid
  private Transform origin;
  private Vector3 velocity;
  private Vector3 normalizedVelocity;
  private Vector3 randomPush;
  private Vector3 originPush;
  private Transform[] objects;
  private UnityFlock[] otherFlocks;
  private Transform transformComponent;
```

We declare the input values for our algorithm that can be set up and customized from the editor. First, we define the minimum movement speed, `minSpeed`, and rotation speed, `turnSpeed`, for our boid. The `randomFreq` value is used to determine how many times we want to update the `randomPush` value based on the `randomForce` value. This force creates a randomly increased and decreased velocity and makes the flock movement look more realistic.

The `toOriginRange` value specifies how spread out we want our flock to be. We also use `toOriginForce` to keep the boids in range and maintain a distance with the flock's origin. Basically, these are the properties to deal with the alignment rule of our flocking algorithm. The `avoidanceRadius` and `avoidanceForce` properties are used to maintain a minimum distance between individual boids. These are the properties that apply the separation rule to our flock.

The `followRadius` and `followVelocity` values are used to keep a minimum distance with the leader or the origin of the flock. They are used to comply with the cohesion rule of the flocking algorithm.

The `origin` object will be the parent object to control the whole group of flocking objects. Our boid needs to know about the other boids in the flock. So, we use the `objects` and `otherFlocks` properties to store the neighboring boids' information.

The following is the initialization method for our boid:

```
void Start () {
  randomFreq = 1.0f / randomFreq;

  //Assign the parent as origin
  origin = transform.parent;

  //Flock transform
  transformComponent = transform;

  //Temporary components
  Component[] tempFlocks= null;

  //Get all the unity flock components from the parent
  //transform in the group
  if (transform.parent) {
```

```
        tempFlocks = transform.parent.GetComponentsInChildren
            <UnityFlock>();
      }

      //Assign and store all the flock objects in this group
      objects = new Transform[tempFlocks.Length];
      otherFlocks = new UnityFlock[tempFlocks.Length];

      for (int i = 0;i<tempFlocks.Length;i++) {
        objects[i] = tempFlocks[i].transform;
        otherFlocks[i] = (UnityFlock)tempFlocks[i];
      }

      //Null Parent as the flock leader will be
      //UnityFlockController object
      transform.parent = null;

      //Calculate random push depends on the random frequency
  //provided
      StartCoroutine(UpdateRandom());
    }
```

We set the parent of the object of our boid as `origin`; it means that this will be the controller object to follow generally. Then, we grab all the other boids in the group and store them in our own variables for later references.

The `StartCoroutine` method starts the `UpdateRandom()` method as a co-routine:

```
  IEnumerator UpdateRandom() {
    while (true) {
      randomPush = Random.insideUnitSphere * randomForce;
      yield return new WaitForSeconds(randomFreq +
          Random.Range(-randomFreq / 2.0f, randomFreq / 2.0f));
    }
  }
```

The `UpdateRandom()` method updates the `randomPush` value throughout the game with an interval based on `randomFreq`. The `Random.insideUnitSphere` part returns a `Vector3` object with random *x*, *y*, and *z* values within a sphere with a radius of the `randomForce` value. Then, we wait for a certain random amount of time before resuming the `while(true)` loop to update the `randomPush` value again.

Now, here's our boid behavior's `Update()` method that helps our boid entity comply with the three rules of the flocking algorithm:

```
void Update () {
  //Internal variables
  float speed = velocity.magnitude;
  Vector3 avgVelocity = Vector3.zero;
  Vector3 avgPosition = Vector3.zero;
  float count = 0;
  float f = 0.0f;
  float d = 0.0f;
  Vector3 myPosition = transformComponent.position;
  Vector3 forceV;
  Vector3 toAvg;
  Vector3 wantedVel;

  for (int i = 0;i<objects.Length;i++){
    Transform transform= objects[i];
    if (transform != transformComponent) {
      Vector3 otherPosition = transform.position;

      // Average position to calculate cohesion
      avgPosition += otherPosition;
      count++;

      //Directional vector from other flock to this flock
      forceV = myPosition - otherPosition;

      //Magnitude of that directional vector(Length)
      d= forceV.magnitude;

      //Add push value if the magnitude, the length of the
      //vector, is less than followRadius to the leader
      if (d < followRadius) {
        //calculate the velocity, the speed of the object, based
         //on the avoidance distance between flocks if the
        //current magnitude is less than the specified
        //avoidance radius
        if (d < avoidanceRadius) {
          f = 1.0f - (d / avoidanceRadius);
```

```
            if (d > 0) avgVelocity +=
                (forceV / d) * f * avoidanceForce;
          }

          //just keep the current distance with the leader
          f = d / followRadius;
          UnityFlock tempOtherFlock = otherFlocks[i];
          //we normalize the tempOtherFlock velocity vector to get
          //the direction of movement, then we set a new velocity
          avgVelocity += tempOtherFlock.normalizedVelocity * f *
              followVelocity;
        }
      }
    }
```

The preceding code implements the separation rule. First, we check the distance between the current boid and the other boids and update the velocity accordingly, as explained in the comments.

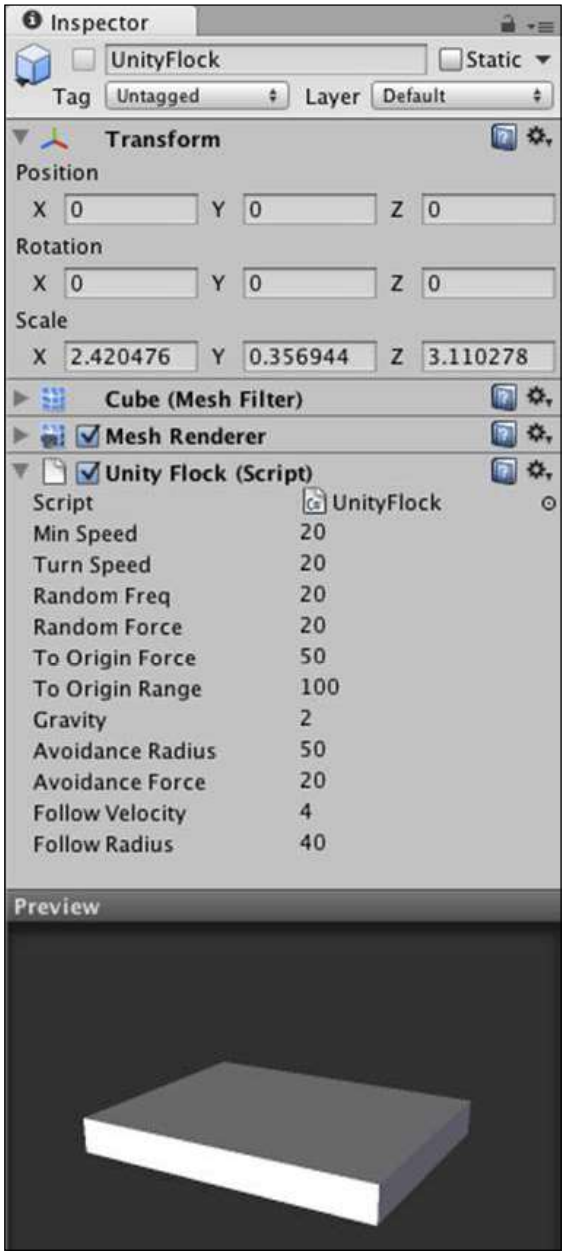
Next, we calculate the average velocity of the flock by dividing the current velocity with the number of boids in the flock:

```
if (count > 0) {
  //Calculate the average flock velocity(Alignment)
  avgVelocity /= count;

  //Calculate Center value of the flock(Cohesion)
  toAvg = (avgPosition / count) - myPosition;
}
else {
  toAvg = Vector3.zero;
}

//Directional Vector to the leader
forceV = origin.position -  myPosition;
d = forceV.magnitude;
f = d / toOriginRange;

//Calculate the velocity of the flock to the leader
if (d > 0) //if this void is not at the center of the flock
```

```
            originPush = (forceV / d) * f * toOriginForce;

        if (speed < minSpeed && speed > 0) {
          velocity = (velocity / speed) * minSpeed;
        }

        wantedVel = velocity;

        //Calculate final velocity
        wantedVel -= wantedVel *  Time.deltaTime;
        wantedVel += randomPush * Time.deltaTime;
        wantedVel += originPush * Time.deltaTime;
        wantedVel += avgVelocity * Time.deltaTime;
        wantedVel += toAvg.normalized * gravity * Time.deltaTime;

        //Final Velocity to rotate the flock into
        velocity = Vector3.RotateTowards(velocity, wantedVel,
            turnSpeed * Time.deltaTime, 100.00f);

        transformComponent.rotation =
    Quaternion.LookRotation(velocity);

        //Move the flock based on the calculated velocity
        transformComponent.Translate(velocity * Time.deltaTime,
            Space.World);

        //normalise the velocity
        normalizedVelocity = velocity.normalized;
      }
    }
```

Finally, we add up all the factors such as `randomPush`, `originPush`, and `avgVelocity` to calculate our final target velocity, `wantedVel`. We also update our current `velocity` to `wantedVel` with linear interpolation using the `Vector3.RotateTowards` method. Then, we move our boid based on the new velocity using the `Translate()` method.

Next, we create a cube mesh and add this `UnityFlock` script to it, and make it a prefab, as shown in the following screenshot:



The Unity flock prefab

Creating the controller

Now it is time to create the controller class. This class updates its own position so that the other individual boid objects know where to go. This object is referenced in the `origin` variable in the preceding `UnityFlock` script.

The code in the `UnityFlockController.cs` file is as follows:

```
using UnityEngine;
using System.Collections;

public class UnityFlockController : MonoBehaviour {
  public Vector3 offset;
  public Vector3 bound;
  public float speed = 100.0f;

  private Vector3 initialPosition;
  private Vector3 nextMovementPoint;

  // Use this for initialization
  void Start () {
    initialPosition = transform.position;
    CalculateNextMovementPoint();
  }

  // Update is called once per frame
  void Update () {
    transform.Translate(Vector3.forward * speed * Time.deltaTime);
    transform.rotation = Quaternion.Slerp(transform.rotation,
        Quaternion.LookRotation(nextMovementPoint -
        transform.position), 1.0f * Time.deltaTime);

    if (Vector3.Distance(nextMovementPoint,
        transform.position) <= 10.0f)
        CalculateNextMovementPoint();
  }
```

In our `Update()` method, we check whether our controller object is near the target destination point. If it is, we update our `nextMovementPoint` variable again with the `CalculateNextMovementPoint()` method we just discussed:

```
  void CalculateNextMovementPoint () {
    float posX = Random.Range(initialPosition.x - bound.x,
        initialPosition.x + bound.x);
    float posY = Random.Range(initialPosition.y - bound.y,
```

```
            initialPosition.y + bound.y);
        float posZ = Random.Range(initialPosition.z - bound.z,
            initialPosition.z + bound.z);

        nextMovementPoint = initialPosition + new Vector3(posX,
            posY, posZ);
    }
}
```

The `CalculateNextMovementPoint()` method finds the next random destination position in a range between the current position and the boundary vectors.

Putting it all together, as shown in the previous scene hierarchy screenshot, you should have flocks flying around somewhat realistically:

Flocking using the Unity seagull sample

Using an alternative implementation

Here's a simpler implementation of the flocking algorithm. In this example, we'll create a cube object and place a rigid body on our boids. With Unity's rigid body physics, we can simplify the translation and steering behavior of our boid. To prevent our boids from overlapping each other, we'll add a sphere collider physics component.

We'll have two components in this implementation as well: individual boid behavior and controller behavior. The controller will be the object that the rest of the boids try to follow.

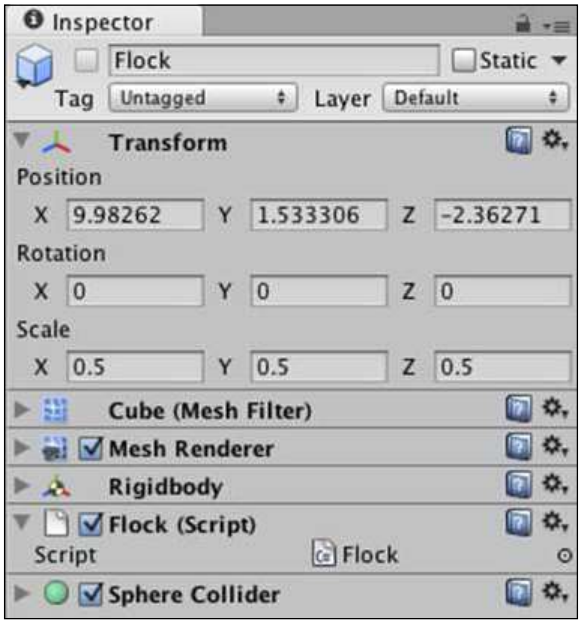
The code in the `Flock.cs` file is as follows:

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

public class Flock : MonoBehaviour {
  internal FlockController controller;

  void Update () {
    if (controller) {
      Vector3 relativePos = steer() * Time.deltaTime;

      if (relativePos != Vector3.zero)
          rigidbody.velocity = relativePos;

      // enforce minimum and maximum speeds for the boids
      float speed = rigidbody.velocity.magnitude;
      if (speed > controller.maxVelocity) {
        rigidbody.velocity = rigidbody.velocity.normalized *
          controller.maxVelocity;
      }
      else if (speed < controller.minVelocity) {
        rigidbody.velocity = rigidbody.velocity.normalized *
            controller.minVelocity;
      }
    }
  }
```

The `FlockController` will be created in a moment. In our `Update()` method, we calculate the velocity for our boid using the following `steer()` method and apply it to its rigid body velocity. Next, we check the current speed of our rigid body component to verify whether it's in the range of our controller's maximum and minimum velocity limits. If not, we cap the velocity at the preset range:

```
    private Vector3 steer () {
      Vector3 center = controller.flockCenter -
          transform.localPosition;  // cohesion

      Vector3 velocity = controller.flockVelocity -
          rigidbody.velocity;  // alignment

      Vector3 follow = controller.target.localPosition -
          transform.localPosition;  // follow leader

      Vector3 separation = Vector3.zero;

      foreach (Flock flock in controller.flockList) {
        if (flock != this) {
          Vector3 relativePos = transform.localPosition -
              flock.transform.localPosition;

          separation += relativePos / (relativePos.sqrMagnitude);
        }
      }

      // randomize
      Vector3 randomize = new Vector3( (Random.value * 2) - 1,
          (Random.value * 2) - 1, (Random.value * 2) - 1);

      randomize.Normalize();

      return (controller.centerWeight * center +
          controller.velocityWeight * velocity +
          controller.separationWeight * separation +
          controller.followWeight * follow +
          controller.randomizeWeight * randomize);
    }
  }
```

The `steer()` method implements separation, cohesion, and alignment, and follows the leader rules of the flocking algorithm. Then, we sum up all the factors together with a random weight value. With this `Flock` script together with rigid body and sphere collider components, we create a `Flock` prefab, as shown in the following screenshot:



The Flock

Implementing the FlockController

The `FlockController` is a simple behavior to generate the boids at runtime and update the center as well as the average velocity of the flock.

The code in the `FlockController.cs` file is as follows:

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

public class FlockController : MonoBehaviour {
  public float minVelocity = 1;  //Min Velocity
  public float maxVelocity = 8;  //Max Flock speed
  public int flockSize = 20;  //Number of flocks in the group

  //How far the boids should stick to the center (the more
```

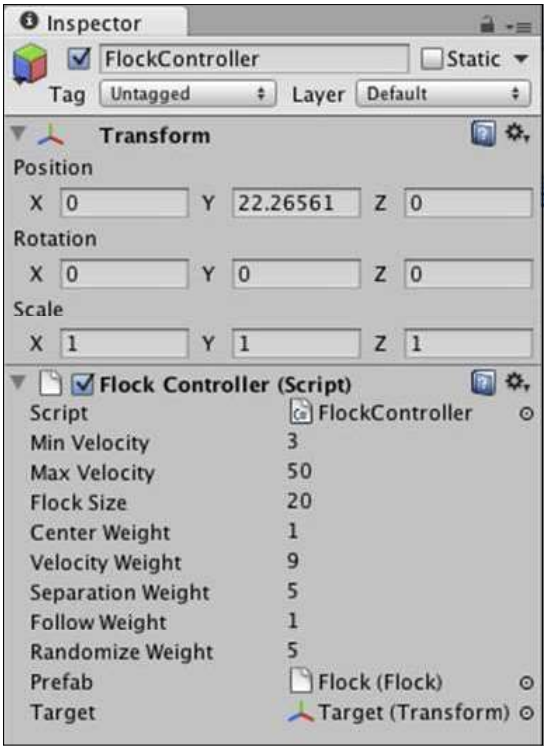
```
//weight stick closer to the center)
public float centerWeight = 1;

public float velocityWeight = 1;  //Alignment behavior

//How far each boid should be separated within the flock
public float separationWeight = 1;

//How close each boid should follow to the leader (the more
//weight make the closer follow)
public float followWeight = 1;

//Additional Random Noise
public float randomizeWeight = 1;

public Flock prefab;
public Transform target;

//Center position of the flock in the group
internal Vector3 flockCenter;
internal Vector3 flockVelocity;  //Average Velocity

public ArrayList flockList = new ArrayList();

void Start () {
  for (int i = 0; i < flockSize; i++) {
    Flock flock = Instantiate(prefab, transform.position,
        transform.rotation) as Flock;
    flock.transform.parent = transform;
    flock.controller = this;
    flockList.Add(flock);
  }
}
```

We declare all the properties to implement the flocking algorithm and then start with the generation of the boid objects based on the flock size input. We set up the controller class and parent transform object as we did last time. Then, we add the created boid object in our `ArrayList` function. The `target` variable accepts an entity to be used as a moving leader. We'll create a sphere entity as a moving target leader for our flock:

```
void Update () {
  //Calculate the Center and Velocity of the whole flock group
  Vector3 center = Vector3.zero;
```

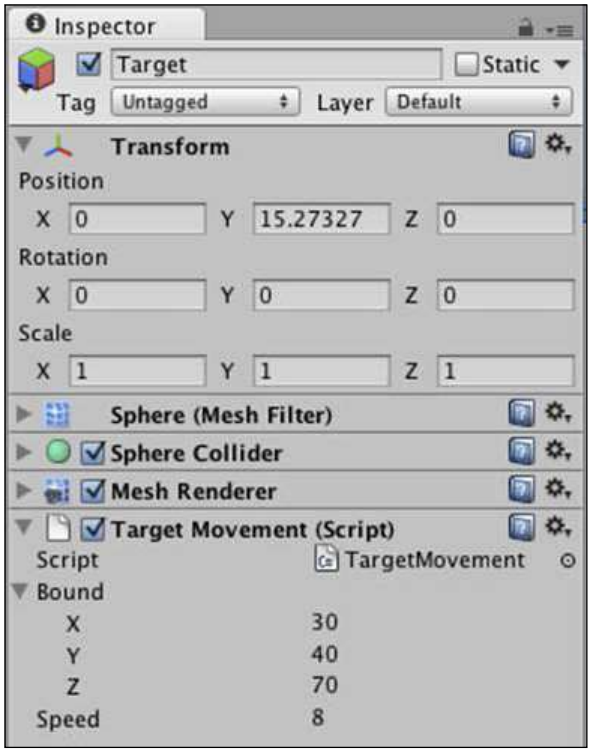
```
        Vector3 velocity = Vector3.zero;

        foreach (Flock flock in flockList) {
          center += flock.transform.localPosition;
          velocity += flock.rigidbody.velocity;
        }

        flockCenter = center / flockSize;
        flockVelocity = velocity / flockSize;
      }
    }
```

In our `Update()` method, we keep updating the average center and velocity of the flock. These are the values referenced from our boid object and they are used to adjust the cohesion and alignment properties with the controller.



The Flock controller

The following is our `Target` entity with the `TargetMovement` script, which we will create in a moment. The movement script is the same as what we saw in our previous Unity sample controller's movement script:



The Target entity with the TargetMovement script

Here is how our `TargetMovement` script works. We pick a random point nearby for the target to move to. When we get close to that point, we pick a new point. The boids will then follow the target.

The code in the `TargetMovement.cs` file is as follows:

```
using UnityEngine;
using System.Collections;

public class TargetMovement : MonoBehaviour {
  //Move target around circle with tangential speed
  public Vector3 bound;
  public float speed = 100.0f;

  private Vector3 initialPosition;
  private Vector3 nextMovementPoint;

  void Start () {
    initialPosition = transform.position;
    CalculateNextMovementPoint();
  }
  void CalculateNextMovementPoint () {
    float posX = Random.Range(initialPosition.x = bound.x,
        initialPosition.x+bound.x);
    float posY = Random.Range(initialPosition.y = bound.y,
        initialPosition.y+bound.y);
    float posZ = Random.Range(initialPosition.z = bound.z,
        initialPosition.z+bound.z);

    nextMovementPoint = initialPosition+
        new Vector3(posX, posY, posZ);
  }
  void Update () {
    transform.Translate(Vector3.forward * speed * Time.deltaTime);
    transform.rotation = Quaternion.Slerp(transform.rotation,
        Quaternion.LookRotation(nextMovementPoint -
        transform.position), 1.0f * Time.deltaTime);

    if (Vector3.Distance(nextMovementPoint, transform.position)
        <= 10.0f) CalculateNextMovementPoint();
  }
}
```

After we put everything together, we should have nice flocking boids flying around in our scene, chasing the target:

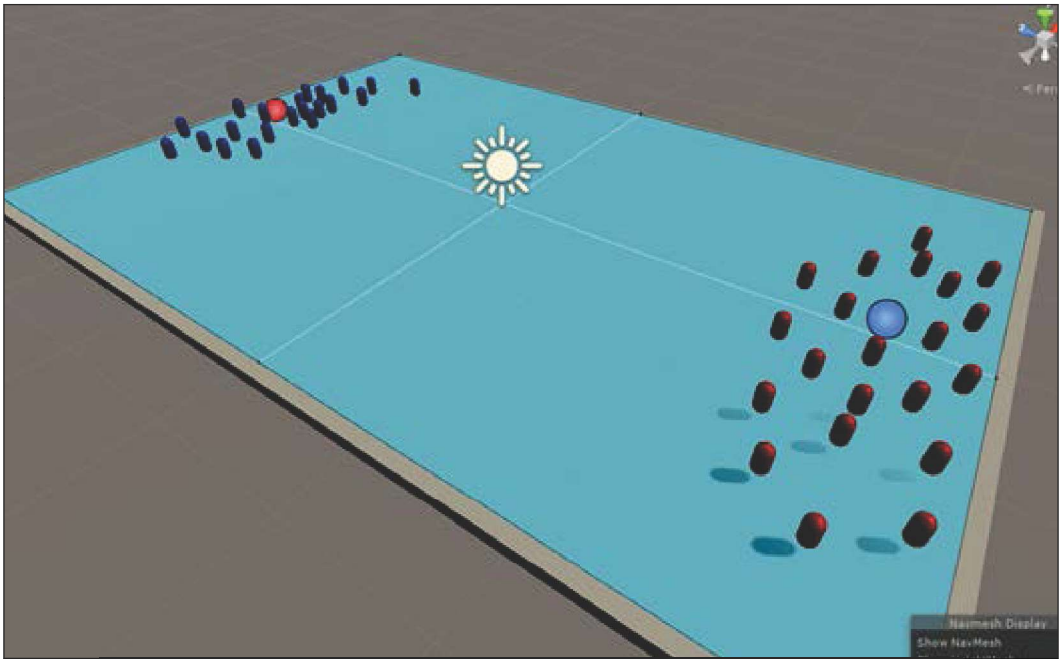
Flocking with Craig Reynold's algorithm

Using crowds

Crowd simulations are far less cut and dry. There really isn't any one way to implement them in a general sense. While not a strict restriction, the term generally refers to simulating crowds of humanoid agents navigating an area while avoiding each other and the environment. Like flocks, the use of crowd simulations has been widely used in films. For example, the epic armies battling one another in *Lord of the Rings* were completely procedurally generated using the crowd simulation software Massive, which was created for using it in the film. While the use of crowd algorithms is not as widespread in video games as in films, certain genres rely on the concept more than others. Real-time strategy games often involve armies of characters, moving in unison across the screen.

Implementing a simple crowd simulation

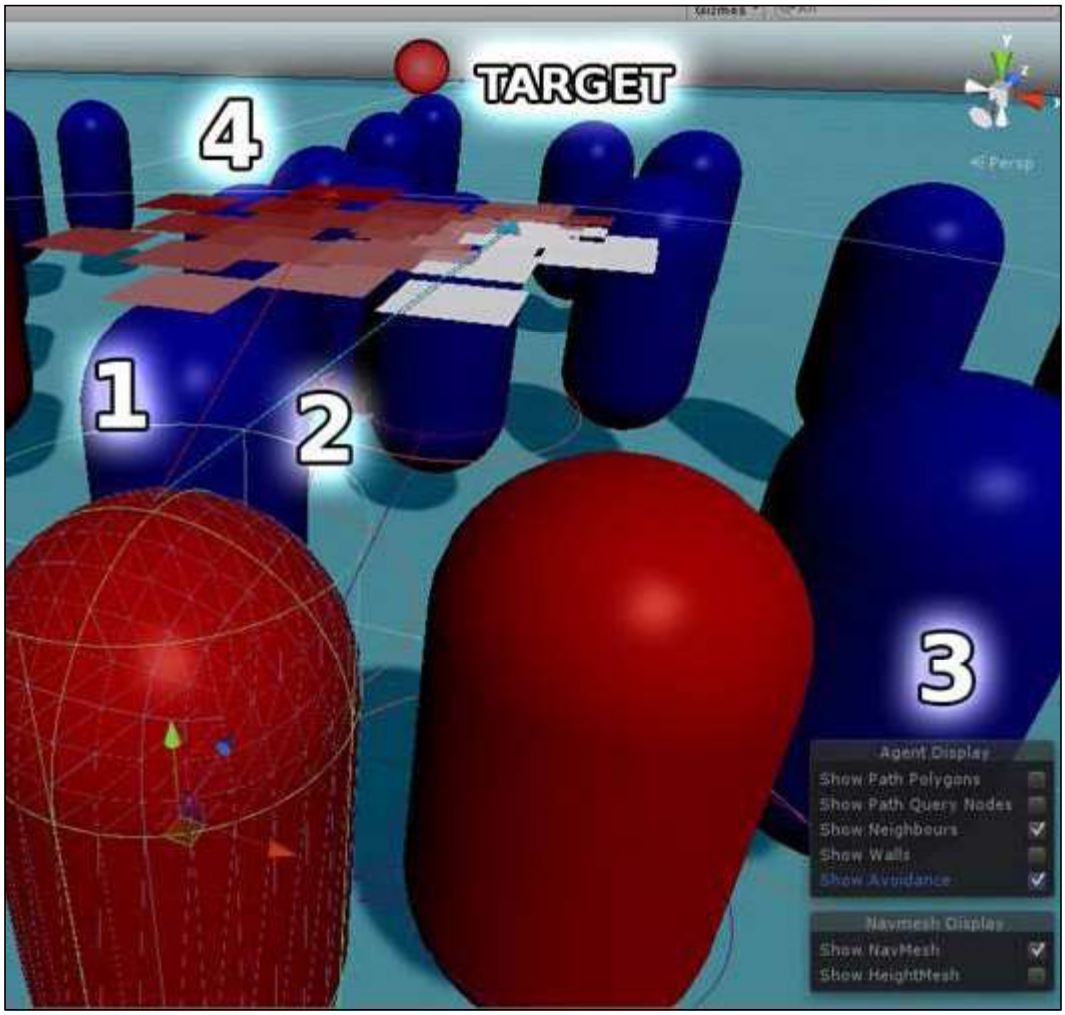
Our implementation will be quick, simple, and effective, and it will focus on using Unity's NavMesh feature. Thankfully, NavMesh will handle much of the heavy lifting for us. Our scene has a simple walking surface with a NavMesh baked onto it, a couple of targets, and two teams of capsules, as shown in the following screenshot:



The classic scenario: red versus blue

In the previous screenshot, we can see that our red and blue targets are opposite to their teams—red and blue, respectively. The setup is straightforward. Each capsule has a `CrowdAgent.cs` component attached to it, and when you hit play, each agent will head towards their target while avoiding each other and the oncoming capsules from the opposite team. Once they reach their destination, they will gather around the target.

While the game is running, you can even select a single capsule or a group of them in the editor to see their behavior visualized. As long as you have the navigation window active, you'll be able to see some debugging information about your NavMesh and the agents on it, as you can see in the following screenshot:



It's worth checking this out in the editor to really get an idea of how this looks in motion, but we've labeled a few key elements in the preceding screenshot:

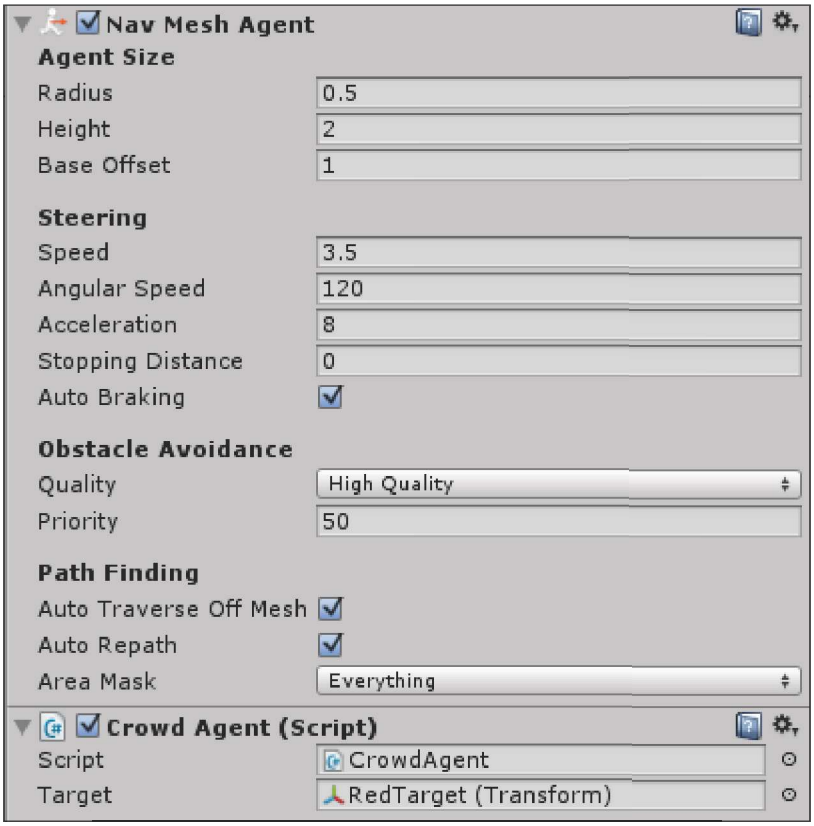
- **1**: This is the destination arrow that points toward the `NavMeshAgent` destination, which for this little guy is `RedTarget`. All this arrow cares about is where the destination is, regardless of the direction the agent is facing or moving toward.
- **2**: This arrow is the heading arrow. It shows the actual direction the agent is moving in. The direction of the agent takes into account several factors, including the position of its neighbors, space on the NavMesh, and the destination.
- **3**: This debug menu allows you to show a few different things. In our case, we enabled **Show Avoidance** and **Show Neighbours**.
- **4**: Speaking of avoidance, this cluster of squares, ranging from dark to light and floating over the agents, represents the areas to avoid between our agent and the destination. The darker squares indicate areas that are densely populated by other agents or blocked by the environment, while the lighter-white squares indicate areas that are safe to walk through. Of course, this is a dynamic display, so watch it change as you play in the editor.

Using the CrowdAgent component

The `CrowdAgent` component is incredibly simple, but gets the job done. As mentioned earlier, Unity does most of the heavy lifting for us. The following code gives our `CrowdAgent` a destination:

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(NavMeshAgent))]
public class CrowdAgent : MonoBehaviour {

    public Transform target;

    private NavMeshAgent agent;

   void Start () {
        agent = GetComponent<NavMeshAgent>();
        agent.speed = Random.Range(4.0f, 5.0f);
        agent.SetDestination(target.position);
   }
}
```

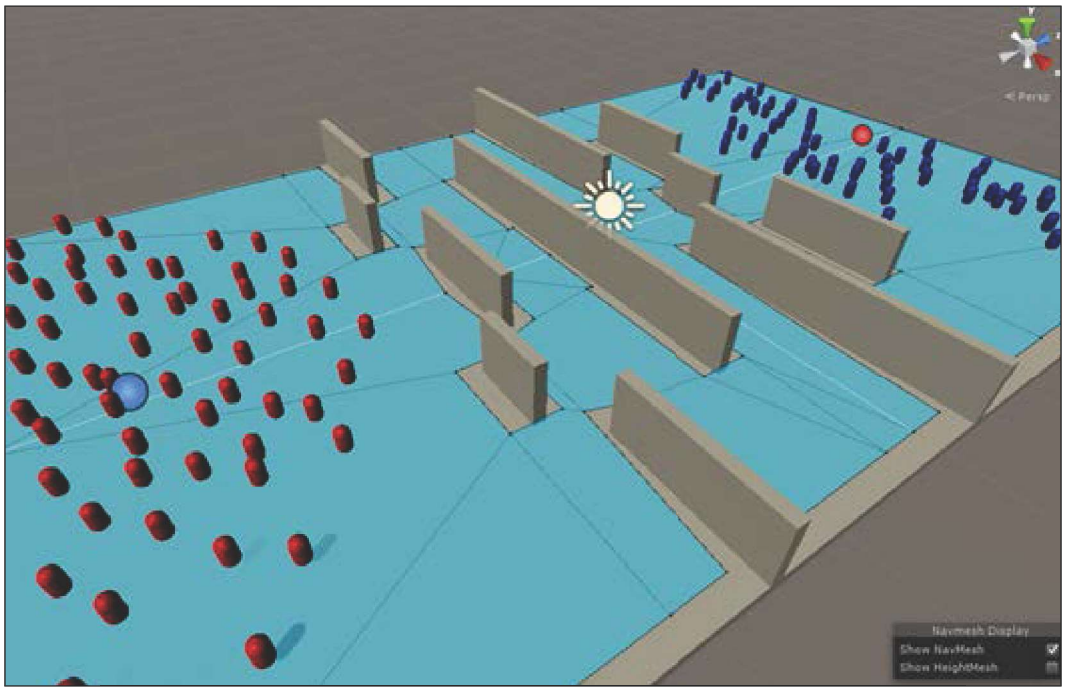
The script requires a component of type `NavMeshAgent`, which it assigns to the `agent` variable on `Start()`. We then set its speed randomly between two values for some added effect. Lastly, we set its destination to be the position of the target marker. The target marker is assigned via the inspector, as you can see in the following screenshot:



The preceding screenshot illustrates a red capsule as it has **RedTarget (Transform)** set as its **Target**.

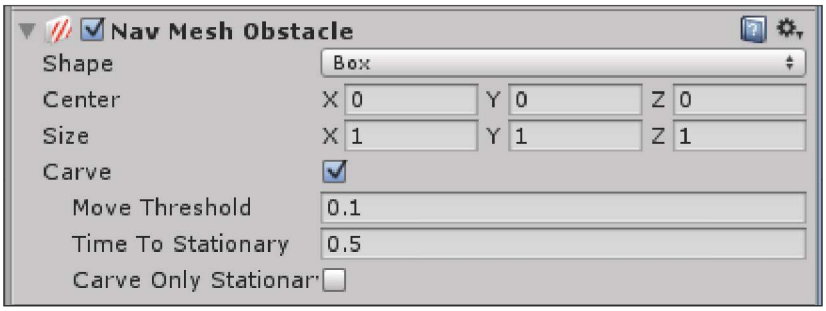
Adding some fun obstacles

Without having to do anything else in our code, we can make a few changes to our scene layout and enable a few components provided by Unity to dramatically alter the behavior of our agents. In our `CrowdsObstacles` scene, we've added a few walls to the environment, creating a maze-like layout for our red and blue teams of capsules to traverse, as you can see in the following screenshot:



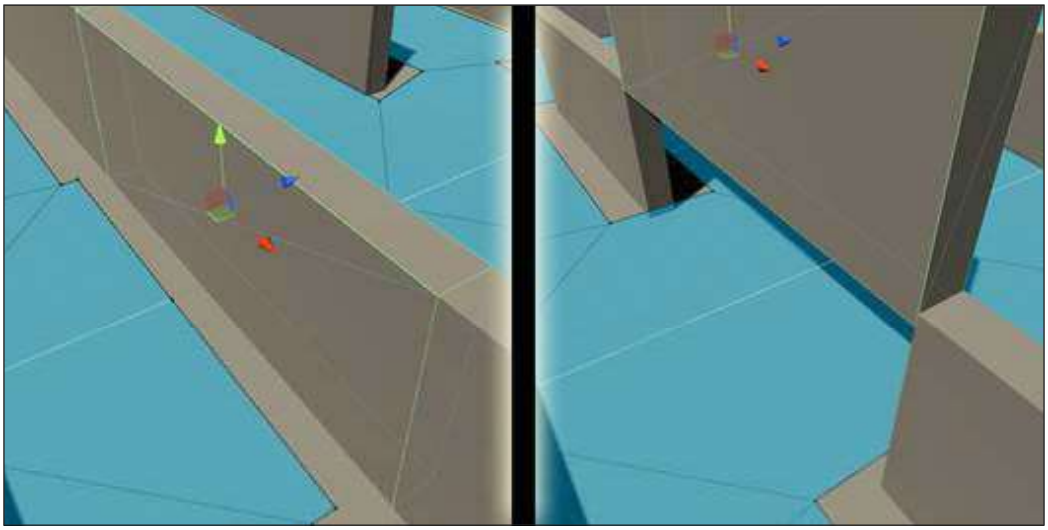
Let the game begin!

The fun part about this example is that because of the randomized speed of each agent, the results will be totally different each time. As the agents move through the environment, they'll be blocked by teammates or opposing agents and will be forced to re-route and find the quickest route to their target. Of course, this concept is not new to us, as we saw `NavMeshAgent` avoiding obstacles in *Chapter 4*, *Finding Your Way*, except that we have many, many more agents in this scenario. To add a bit more fun to the example, we've also added a simple up-down animation to one of the walls and a `NavMeshObstacle` component, which looks something like this:



Nav Mesh Obstacle looks a bit different in Unity 5

Note that our obstacle does not need to be set to **Static** when we are using this component. Our obstacle is mostly box-like, so we leave the default **Shape** setting as **Box** (**Capsule** is another choice). The **Size** and **Center** options let us move the outline of our shape around and resize it, but the default settings fit our shape perfectly, which is what we want, so let's leave that alone. The next option **Carve** is important. It essentially does exactly what it says; it carves a space out of the NavMesh, as shown in the following screenshot:

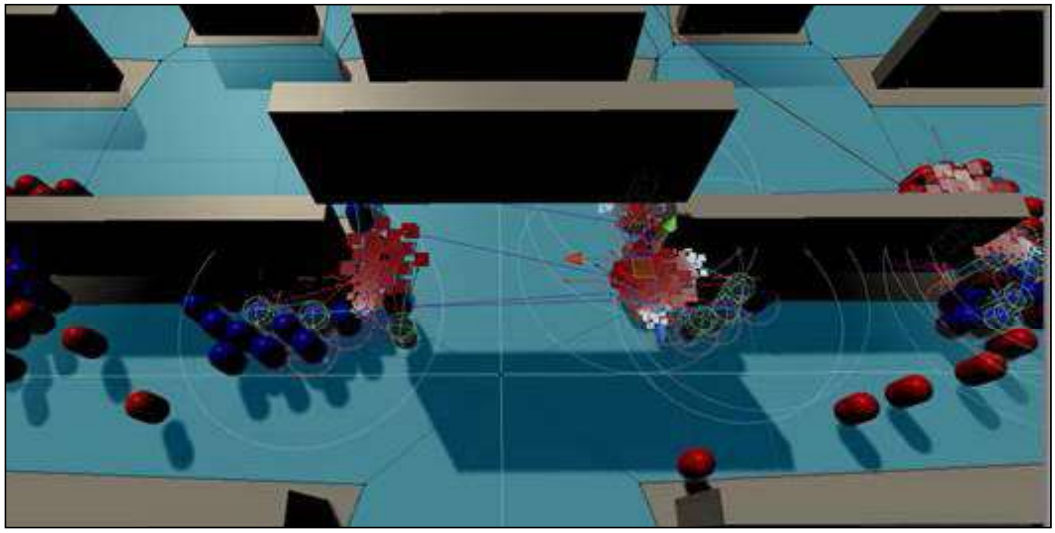


The sane obstacle at two different points of its up-down animation

The left screenshot shows the space carved out when the obstacle is on the surface, while the NavMesh is connected in the right screenshot when the obstacle is raised off the surface. We can leave **Time to Stationary** and **Move Threshold** as they are, but we do want to make sure that **Carve Only Stationary** is turned off. This is because our obstacle is moving, and if we didn't tick this box, it would not carve out the space from the NavMesh, and our agents would be trying to move through the obstacle whether it was up or down, which is not the behavior we are after in this case.

As the obstacle moves up and down and the mesh is carved out and reconnected, you'll notice the agents changing their heading. With the navigation debug options enabled, we can also see a very interesting visualization of everything going on with our agents at any given moment. It may seem a bit cruel to mess with our poor agents like this, but we're doing it for science!

The following screenshot gives us a glimpse into the chaos and disorder we're subjecting our poor agents to:



I'm secretly rooting for the blue team

Summary

In this chapter, we learned how to implement flocking behavior in two ways. First, we examined, dissected, and learned how to implement a flocking algorithm based on Unity's Tropical Island Demo project. Next, we implemented it using rigid body to control the boid's movement and sphere collider to avoid collision with other boids. We applied our flocking behavior to the flying objects, but you can apply the techniques in these examples to implement other character behaviors such as fish shoaling, insects swarming, or land animals herding. You'll only have to implement different leader movement behaviors such as limiting movement along the y axis for characters that can't move up and down. For a 2D game, we would just freeze the y position. For 2D movement along uneven terrain, we would have to modify our script to not put any forces in the y direction.

We also took a look at crowd simulation and even implemented our own version of it using Unity's NavMesh system, which we first learned about in *Chapter 4*, *Finding Your Way*. We learned how to visualize our agents' behavior and decision-making process.

In the next chapter, *Behavior Trees*, we'll look at the behavior tree pattern and learn to implement our own version of it from scratch.