9

Directional recipes and non-player-character control

In this chapter, we will cover:

- Individual movement: a player-controlled cube
- Individual movement: Look-at an object
- Individual movement: Seek-Flee-Follow target
- Group movement: Flock together
- Instantiating projectiles with forward velocity
- Spawnpoints and Waypoints: Find one randomly
- Spawnpoints and Waypoints: Identify nearest
- Spawnpoints and Waypoints: Follow waypoints in sequence
- Basic state-driven behavior (playing-win-lose)
- State behaviors with the state pattern

Introduction

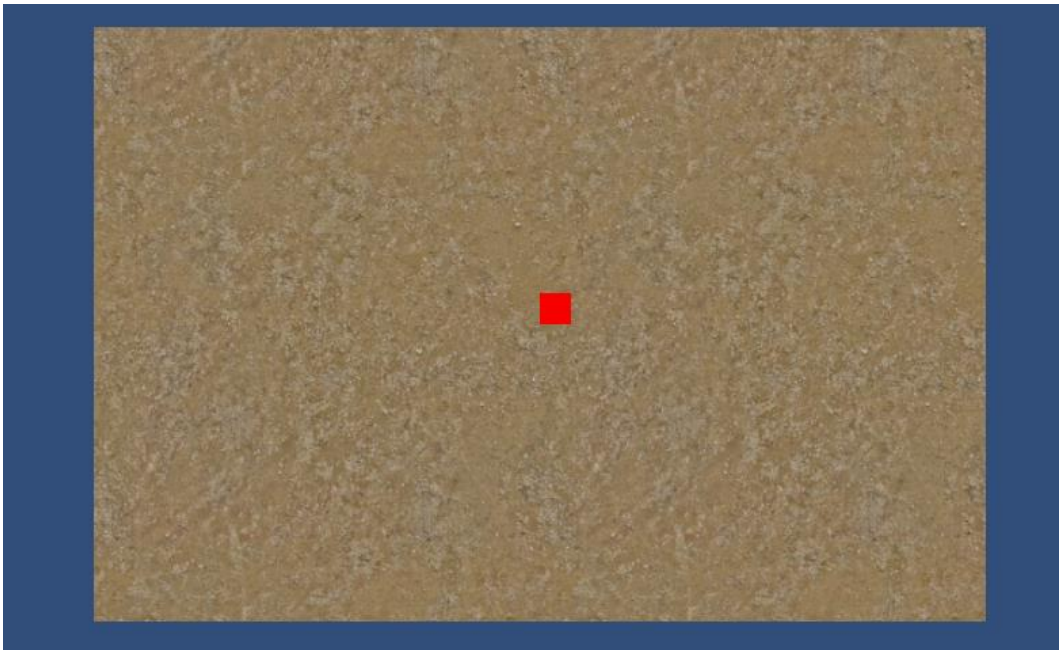
Many games involve moving computer controlled objects and characters. For some games animation components can be sufficient for all such movements. However, other games use 'directional logic' – simple or sophisticated artificial 'intelligence' involving the computer program making objects orient and move towards or away from the player or other game objects and characters. 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 (non-player character) object movements in the recipes in this chapter.

Individual movement: a player-controlled cube

Many of the the recipes in this chapter are built on this basic project, which constructs a scene with a textured terrain, a downward facing *Main Camera*, and a red cube that can be moved around by user with the 4 directional arrow keys.



0423_09_01.png

How to do it...

1. Start a new project, importing Terrain Textures > GoodDirt.psd from the *Terrain Assets* Unity package (uncheck all the other contents of this package, since we don't need them and they'll just bloat the size of the project unnecessarily)
2. 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).

3. Set the material for this terrain to be *GoodDirt*
4. 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)
5. Make the following changes to the Main Camera:
   - position = (0,20, 0)
   - rotation = (90, 0, 0)
6. 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*.
7. Create a new Cube GameObject named *Cube-player*, at position (0, 0.5, 0) and sized (1,1,1)
8. Create a new red material named *m_red*, and apply this material to *Cube-player*
9. Add the following C# script class to the *Cube-player*

```
// file: PlayerControl
using UnityEngine;
using System.Collections;

public class PlayerControl : MonoBehaviour {
   public float y;
   public const float MIN_X = -15;
   public const float MAX_X = 15;
   public const float MIN_Z = -10;
   public const float MAX_Z = 10;

   private float speed = 20;

   private void Awake(){
      y = transform.position.y;
   }

   private void Update () {
      KeyboardMovement();
      CheckBounds();
```

```
    }

    private void KeyboardMovement(){
        float dx = Input.GetAxis("Horizontal") * speed *
Time.deltaTime;
        float dz = Input.GetAxis("Vertical") * speed *
Time.deltaTime;
        transform.Translate( new Vector3(dx,y,dz) );
    }

    private void CheckBounds(){
        float x = transform.position.x;
        float z = transform.position.z;
        x = Mathf.Clamp(x, MIN_X, MAX_X);
        z = Mathf.Clamp(z, MIN_Z, MAX_Z);
        transform.position = new Vector3(x, y, z);
    }
}
```

10. Add the following script class to your project:

```
// file: UsefulFunctions.cs
using UnityEngine;

public class UsefulFunctions : MonoBehaviour{
    public static void DebugRay(Vector3 origin, Vector3 v,
Color c) {
        Debug.DrawRay(origin, v * v.magnitude, c);
    }

    public static Vector3 ClampMagnitude(Vector3 v, float
max) {
        if (v.magnitude > max)
            return v.normalized * max;
        else
            return v;
    }
}
```

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. Upon Awake() the starting Y-coordinate is stored, so that any changes to the position of the cube will always keep the object to the same Y-position. Each frame the Update() method first calls KeyboardMovement(), and then CheckBounds().

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 position of the player's cube is move (translated). The amount it is moved depends on the speed variable. Since our camera is looking DOWN, the Y-coordinate is fixed, and it is the X and Z values that are used to determine how to move the red cube.

Method CheckBounds() simply checks the X and Z positions against 4 constants for the maximum and minimum values, and if values are outside the range they are set to the maximum or minimum as appropriate.

> NOTE: Mathf.Clamp() a very handy function
>
> This function is very useful since it allows us to restrict a value between a minimum and maximum value. The first parameter is the value, the second is the minimum and the third is the maximum. If the value is smaller than the minimum, then the minimum is returned. If the value is larger than the maximum, the maximum is returned. Otherwise the value (which must then be somewhere within these limits) is returned.

Although not used in this project, the class UsefulFunctions() is used by several of the recipes in this chapter. It offers two useful functions to our project:

- method DebugRay() – described in the previous recipe
- method ClampVector() – which takes a Vector3 and a maximum size as input, and returns a vector with a maximum length (magnitude) of the input maximum size
    - i.e. if the length of the given vector is greater than the maximum, then a vector of maximum length in the same direction is returned

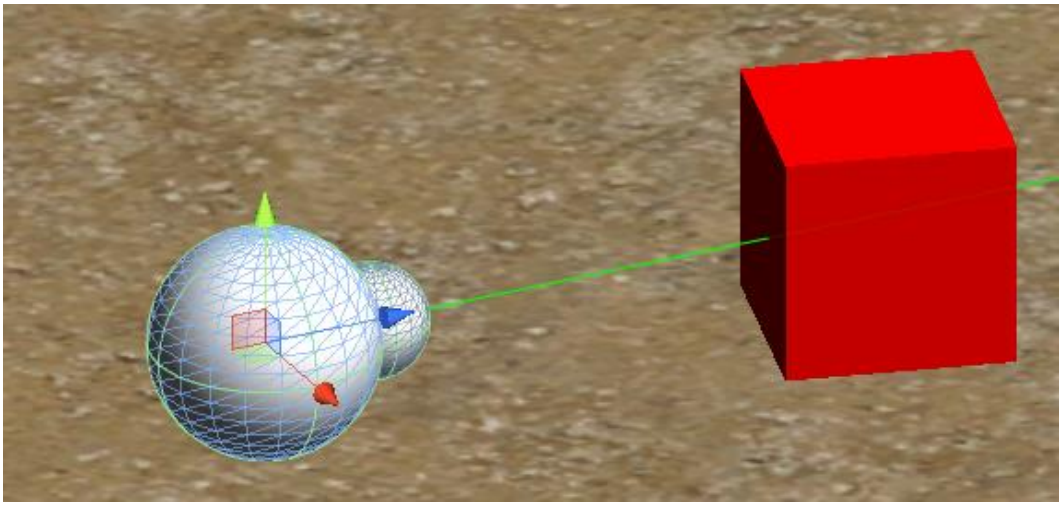
See also

- (many of the recipes in this chapter build upon this basic project)

Individual movement: Look-at an object

There are often situations where we want to make one object orient itself to look in the direction of some other object. Examples include enemies or gun turrets wanting to look at the player, or perhaps making the player start off looking at an object at a key point in

the game – such as after having been teleported or respawned to some location in the game.



0423_09_02.png

Getting ready

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

How to do it...

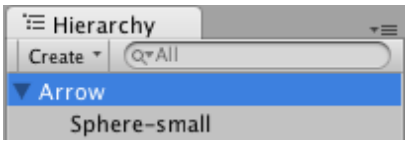
1. Create a sphere named *Arrow*, positioned at (2, 0.5, 2) and with scale (1,1,1)
2. Create a second sphere, named *Sphere-small*, positioned at (0, 0, 0.5) and with scale (0.5, 0.5, 0.5)

> NOTE: By default, when objects are first created they are 'looking' (facing) along the Z-axis. By positioning *Sphere-small* at (0, 0, 0.5) we have make this smaller sphere the 'point' of an arrow – i.e. when we look at the two spheres, they are facing in the direction of a line drawn from the center of the large sphere towards the center of the small sphere.

You can check this is correct in the Scene window – selecting *Arrow*, you should see the blue Z-axis gizmo arrow point from the center of the larger sphere towards the center of the smaller sphere.

3. Make *Sphere-small* a child-object of your GameObject *Arrow*

NOTE: To make an object a child of another object, in the Hierarchy window drag the child object into the parent object. The child object should then appear indented below the parent object. The parent object will also have a 'contents' triangle symbol next to it, to control whether its contents (child objects) are listed or not).



0423_09_03.png

4. Add the following C# script class to your GameObject *Arrow*:

```
// file: LookAt.cs
using UnityEngine;
using System.Collections;

public class LookAt : MonoBehaviour {
   public Transform playerTransform;

   private void Update () {
      transform.LookAt( playerTransform );
   }
}
```

5. Ensuring *Arrow* is selected, in the Inspector for the LookAt scripted component drag 3rd Person Controller over public variable *Player Transform*

How it works...

Since it is so useful, Unity has provided all Transform objects with the method LookAt(). This method will make the object whose transform has the method applied, orient itself at the object whose transform is provided as a parameter. Since our Arrow object's script has a reference to *Cube-Player* (via the public variable Player Transform), every

frame the *Arrow* object is made to re-orient itself to face towards the direction of the player's character.

There's more...

Some details you don't want to miss:

DrawRay() method to add visual information in scene panel

To illustrate that 'rays' (lines in 3D space) that can be drawn in the Scene window, and to aid debugging the Debug.DrawRay() statement can be used. Adding to the class as follows will result in a ray being drawn in green, from the position of *Arrow's* transform, 100 Unity units in the direction the Arrow is facing (its transform's forward Vector). Replace your Update() method with the following one, adds this new DebugRay() method to be called after making the *Arrow* object look at the position of the player's red cube:

```
   private void Update () {
      transform.LookAt( playerTransform );
      DebugRay();
   }

   private void DebugRay() {
      Debug.DrawRay(transform.position, transform.forward
* 100, Color.green);
```

See also

- Individual movement: Player controlled cube
- Individual movement: Seek-Flee-Follow target
- Group movement: Flock together

Individual movement: Seek-Flee-Follow target

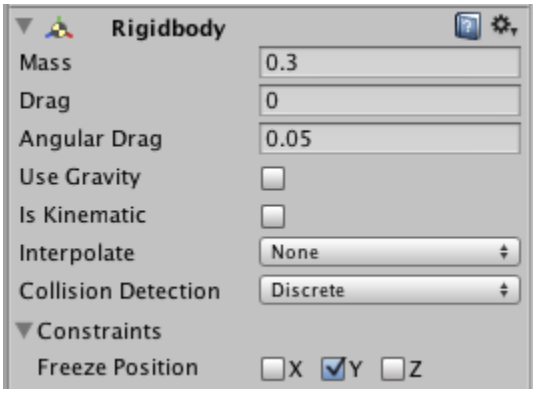
Computer controlled characters (NPC – non-player characters) often need to seek and move towards another object (such as the Player's character), or alternatively flee away from another character. The logic is the same for both seeking and fleeing, and is demonstrated in this recipe.

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...

1. Create a sphere named *Arrow*, positioned at (2, 0.5, 2) and with scale (1,1,1)
2. Create a second sphere, named *Sphere-small*, positioned at (0, 0, 0.5) and with scale (0.5, 0.5, 0.5)
3. Make *Sphere-small* a child-object of your GameObject *Arrow*
4. Add a Physics > RigidBody component to GameObject *Arrow*, changing the following properties from the defaults:
   - Mass = 0.3
   - Use Gravity = false
   - Check (tick) the Freeze Position for the Y axis only



0423_09_04.png

5. Add the following C# script class to your GameObject *Arrow*:

```
// file: SeekTarget.cs
using UnityEngine;
using System.Collections;

public class SeekTarget : MonoBehaviour {
   public GameObject playerGO;
   public const float MAX_SPEED = 500.0f;

   private float maxSpeedForFrame = 1;
```

```
   private void FixedUpdate () {
      maxSpeedForFrame = MAX_SPEED * Time.deltaTime;
      Vector3 source = transform.position;
      Vector3 target = playerGO.transform.position;

      Vector3 adjustVelocity= Seek(source, target);
      adjustVelocity  = UsefulFunctions.ClampMagnitude(
adjustVelocity, maxSpeedForFrame);
      rigidbody.AddForce(adjustVelocity);

      transform.LookAt( playerGO.transform );
      UsefulFunctions.DebugRay(transform.position,
adjustVelocity, Color.blue);
      UsefulFunctions.DebugRay(transform.position,
rigidbody.velocity, Color.yellow);
   }

   private Vector3 Seek(Vector3 source, Vector3 target){

      Vector3 directionToTarget = Vector3.Normalize(
target - source );
      Vector3 velocityToTarget = maxSpeedForFrame *
directionToTarget;
      return velocityToTarget - rigidbody.velocity;
   }
}
```

6. Ensuring *Arrow* is selected, in the Inspector for the *SeekTarget* scripted component drag *Cube-Player* over public variable Player Transform

How it works...

We have added a Unity physics *rigidbody* component to our *Arrow* object. This means that we can ask Unity to apply a 'force' to this object, and Unity will work out how to move the object, based on its mass and any previous velocity the object already had.

> NOTE: When working with physics and forces, we put our logic into method FixedUpdate(), which is called at a fixed frame rate, immediately before Unity runs its physics simulation for the frame. Had such logic been located in method Update(), none, or several Update() messages may have been received since the last physics simulation update, leading to uneven behavior.

So remember, once you have added a physics rigid body to an object you are influencing through code, that code must be placed into method FixedUpdate().

One danger when adding forces to already moving objects is that they start to move very fast, and this can lead to 'overshooting' the target, and sometimes even the (very unintelligent) behavior of a chasing object forever orbiting around the target, but never getting any closer. Therefore we define a constant MAX_SPEED, which will set a limit on the size of any force we will apply to *Arrow*. As usual, since the frame rate of games varies, we need to use the proportion of this speed maximum that is relative to the time since the last frame (`Time.deltaTime`).

Each time FixedUpdate() is called, first the maximum speed for this frame (*maxSpeedForFrame*) is calculated. Then method Seek() is called passing the current positions of the source (*Arrow*) and target (player cube) objects. This method returns a Vector3 object that represents the force we wish to apply to Arrow to make it move towards the player's cube. This force is limited to the size of *maxSpeedForFrame*, and then applied to the *Arrow* object as a force to the rigid body component. Finally, the *Arrow* object is made to face towards its target, and then two debug rays are drawn, a blue one to show the seeking force we have just applied, and a yellow one to show the current direction/speed velocity that the *Arrow* object now has.

The core of this target-seeking behavior is to make use of two key pieces of information, which we do in method Seek():

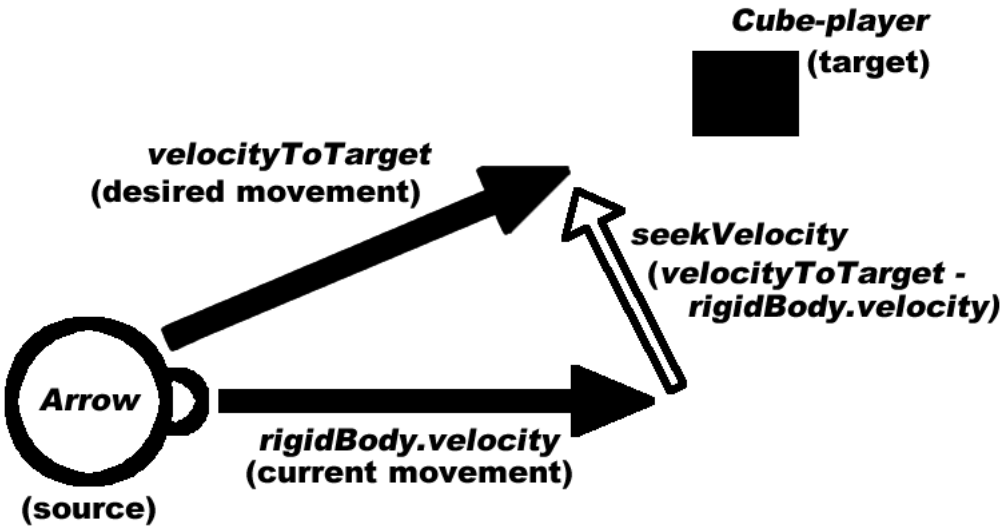
- the **direction** from the *Arrow* (source) object towards the player's cube (target) object
- and the **magnitude** (length) of the force vector we want to apply to our *Arrow* object

NOTE: A Vector3 object is the perfect data structure for forces, since they represent both a direction, and a magnitude (the length of the vector).

NOTE: A 'normalized' vector is simply a vector with a length (magnitude) of 1. This is very useful since we can then multiple a normalized vector by the speed (distance) we want an object to move. Sometimes we refer to a normalized vector as a 'unit vector', since it has a length of 1.

Method Seek() does the following:

- it calculates the **direction** from the *Arrow* (source) object towards the player's cube (target) object, by subtracting the source position from the target position; and then normalizing the result
- this direction is multiplied by the *maxSpeedForFrame* to give the velocity vector we'd like the *Arrow* object to move
- if the Arrow is already moving, then simply applying the force towards the target may 'nudge' the Arrow a little off its current direction towards the target, but what we really want is to calculate the force to apply to make Arrow move towards the target taking into account the Arrow's current movement – we do this by subtracting the Arrow's current movement (*rigidbody.velocity*) from the force in the direction of the target, the result is a force that will redirect the Arrow from its current direction towards the target



0423_09_05.png

There's more...

Some details you don't want to miss:

MoveTowards() simpler but less sophisticated

If the only movement behavior you wish to implement for an object is a simple 'seek' behavior, Unity does provide a very straightforward method for this purpose. The use of rigid bodies and forces is needed for the later recipes in this chapter, which is why this simple method approach was not taken for this recipe. However, you could replace the code above for class SeekTarget with the following, which uses the MoveTowards() Unity method for a simple seek behavior:

```
// file: MoveTowards.cs
using UnityEngine;
using System.Collections;

public class MoveTowards : MonoBehaviour {
   public Transform playerTransform;
   public float speed = 5.0F;

   private void Update () {
      transform.LookAt( playerTransform );
      float distance = speed * Time.deltaTime;
      Vector3 source = transform.position;
      Vector3 target = playerTransform.position;
      transform.position = Vector3.MoveTowards(source,
target, distance);
   }
}
```

Method MoveTowards() takes 3 arguments, being the source position, the target position, and how far to move towards the target.

Arrive – decelerate as get closer to target

A nicer version of the seek behavior involves the seeking object to slow down as it gets closer to the target object. This can be achieved by replacing method Seek() with a new method Arrive(). First replace the statement in FixedUpdate(), so that the value of *adjustVelocity* is returned from method Arrive()

```
Vector3 adjustVelocity = Arrive(source, target);
```

Next, define a constant DECELERATION_FACTOR that we need:

```
const float DECELERATION_FACTOR = 0.6f;
```

Finally, implement the Arrive() method with the following code:

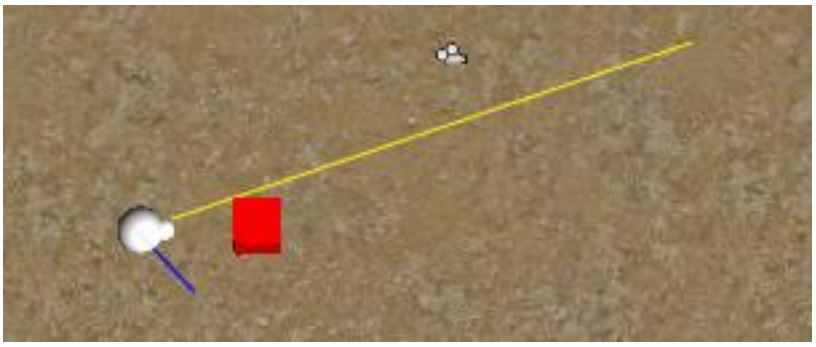
```
private Vector3 Arrive(Vector3 source, Vector3 target){
```

```
   float distanceToTarget = Vector3.Distance(source,
target);
   Vector3 directionToTarget = Vector3.Normalize( target
- source );
   float speed = distanceToTarget / DECELERATION_FACTOR;
   Vector3 velocityToTarget = speed * directionToTarget;

   return velocityToTarget - rigidbody.velocity;
}
```

The speed (magnitude) of the force to be applied to the Arrow object will vary, depending on the remaining distance to the target (player's cube). This rate of deceleration can be 'tweaked' by changing the DECELERATION_FACTOR constant. As the distance to the target is less (i.e. as the Arrow gets closer), then the size of the force to be applied to the Arrow is smaller, so it slows down as it gets closer to the target. This results in a more natural (and satisfying) movement in target seeking game objects.

The screenshot below shows the two debug rays of the forces on the Arrow object at a point in time:

- the longer ray shows the current movement velocity of the Arrow – upwards and to the right
- the shorter ray shows the force to be applied to Arrow to make it move towards the player's cube, downwards to the right – which should result in the Arrow colliding with the target player cube object very soon



0423_09_06.png

Flee target and follow at a distance

The opposite to arrive at is the 'flee' behavior – which involves moving in the opposite direction to where the target is. First change FixedUpdate(), so that the value of *adjustVelocity* is set differently according to the distance from target (its default is zero,

if within the FLEE_RADIUS it will be a force to move away from the target, if outside of the SEEK_RADIUS it will be a force to move towards the target):

```
Vector3 adjustVelocity = Vector3.zero;

float distanceToTarget = Vector3.Distance(target,
source);
if( distanceToTarget < FLEE_RADIUS )
   adjustVelocity += Flee(source, target);

if( distanceToTarget > SEEK_RADIUS )
   adjustVelocity += Arrive(source, target);
```

Next, define some constants we need:

```
const float DECELERATION_FACTOR = 0.6f;
public const float FLEE_RADIUS = 14f;
public const float SEEK_RADIUS = 6f;
```

Since the FLEE_RADIUS is larger than the SEEK_RADIUS, when the distance to the target is between 6 and 14, then BOTH a flee and an arrive force will be calculated – this will result in a deceleration in this overlap zone. The result is a realistic following behavior, whereby the Arrow follows at approximately 10 units, accelerating and decelerating smoothly based on the player's cube movements.

Add the Arrive() method listed above in the previous section, and also add a new method Flee() as follows:

```
private Vector3 Flee(Vector3 source, Vector3 target){
   float distanceToTarget = Vector3.Distance(target,
source);
   Vector3 directionAwayFromTarget =
Vector3.Normalize(source - target);
   float speed = (FLEE_RADIUS - distanceToTarget) /
DECELERATION_FACTOR;
   Vector3 velocityAwayFromTarget = speed *
directionAwayFromTarget;
   return velocityAwayFromTarget;
}
```

Method Flee() works in the opposite way to Arrive(), so the speed (magnitude) of the velocity force vector will be larger when the target is closer, so the Arrow will move quickly away from a nearby target.

A follow-at-a-distance behavior can be achieved by making an object flee a nearby target, but seek to arrive at a far away target.

See also

- Individual movement: Player controlled cube
- Individual movement: Look-at an object
- Group movement: Flock together
- Spawnpoints and Waypoints: Follow waypoints in sequence

Group movement: Flock together

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 generation 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 of flocking cubes. To keep things simple, we'll only implement basic versions of the last two rules above: alignment and cohesion.

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.

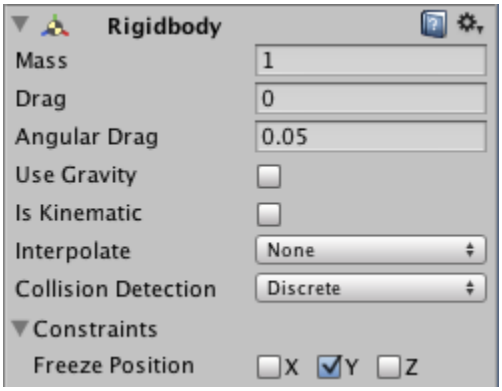
How to do it...

1. Create a new C# script class Swarm:

```
// file: Swarm
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Swarm : MonoBehaviour {
```

```
   private static List<GameObject> drones = new
List<GameObject>();

   public static void AddDrone(GameObject drone){
      drones.Add(drone);
   }

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

      foreach(GameObject drone in Swarm.drones ) {
         locationTotal += drone.transform.position;
      }

      return (locationTotal / Swarm.drones.Count);
   }

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

      foreach(GameObject drone in Swarm.drones ) {
         velocityTotal += drone.rigidbody.velocity;
      }

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

2. Create a new Cube GameObject named *Cube-boid*, at (0,0,0) with the following properties:
3. Add a Physics RigidBody component to *Cube-boid* 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



0423_09_07.png

4. Add the following C# script class to the *Cube-boid*

```
// file: Boid.cs
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Boid : MonoBehaviour {
   private void Awake(){
      Swarm.AddDrone( this.gameObject );
   }

   public float speed = 5f;
   public float maxSpeed = 10f;

   private void FixedUpdate() {
      rigidbody.velocity += (SwarmAdjustment() * speed);
      rigidbody.velocity = Limit(rigidbody.velocity,
maxSpeed);
   }

   private Vector3 SwarmAdjustment() {
      Vector3 moveTowardsSwarmCenter = VectorTowards(
Swarm.SwarmCenterAverage() );
      Vector3 adjustment = moveTowardsSwarmCenter + (2 *
Swarm.SwarmMovementAverage() );

      float maxDirectionChange = .05f;
      return Limit(adjustment, maxDirectionChange);
   }

   private Vector3 VectorTowards(Vector3 target) {
```

```
      Vector3 targetDirection = target -
transform.position;
      targetDirection.Normalize();
      targetDirection *= maxSpeed;
      return (targetDirection - rigidbody.velocity);
   }

   public Vector3 Limit(Vector3 v, float max) {
      if (v.magnitude > max)
         return v.normalized * max;
      else
         return v;
   }
}
```

5. Create a new empty Prefab named *prefab_Boid*, and from the Hierarchy panel drag your *Cube-boid* GameObject into this prefab
6. Delete *Cube-boid* from the Scene panel
7. Add the following C# script class to the *Main Camera*

```
// file: SwarmCreator.cs
using UnityEngine;
using System.Collections;

public class SwarmCreator : MonoBehaviour {
   // if your computer is slow, reduce this to less than
20!
   public int droneCount = 50;

   public GameObject prefab;

   protected virtual void Start () {
      for (int i = 0; i < droneCount; i++)
         Instantiate(prefab);
   }
}
```

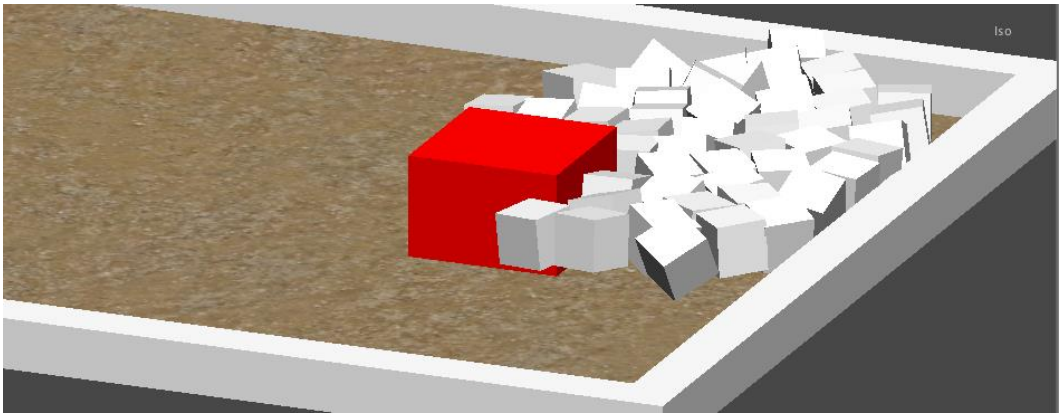
8. With *Main Camera* selected in the Hierarchy, drag *prefab_Boid* from the Project panel over the public variable of *prefab*
9. Create a new Cube named *wall-left*, with the following properties:
   - Position = (-15, 0.5, 0)
   - Scale = (1, 1, 20)
10. Duplicate object *wall-left* naming the new object *wall-right*, and change the position of *wall-right* to (15, 0.5, 0)

11. Create a new Cube named *wall-top*, with the following properties:
    - Position = (0, 0.5, 10)
    - Scale = (31, 1, 1)
12. Duplicate object *wall-top* naming the new object *wall-bottom*, and change the position of *wall-bottom* to (0, 0.5, -10)
13. Finally, make the player's red cube larger, set its scale to (3,3,3)

How it works...

The Swarm class contains a (static class) List variable *drones* of all the drone 'boids' that have been created. There is also a public static method Add(), which each Boid calls upon its created (via its Awake() method), so that *Swarm.drones* is a list of all the Boid objects that are currently in the Scene. 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 Boid objects, and the other (*SwarmMovementAverage*) returns a Vector3 object representing the average velocity (movement force) of all the Boid objects.

Class SwarmCreator has a single method *Start*(), which instantiates 50 Boid objects from the prefab linked in the Inspector. They are contained by the 4 walls that were created from scaled cubes (*wall-left, wall-right* etc.).



0423_09_08.png

The fundamental class of this recipe is the Boid class. Each Boid when created adds a reference to itself to the Swarm list *drones*. Two variables are defined, *speed* and *maxSpeed* – these are declared as public floats, so their values can be 'tweaked' by the game developer. Method *FixedUpdate*() adds to the current movement velocity of the

boid (*rigidbody.velocity*) the Vector3 value returned from method *SwarmAdjustment*(), multiplied by variable speed. Then the velocity of the Boid is limited to a maximum magnitude of *maxSpeed* (in case the addition of the new force

Each Boid object needs to know the following to decide what to do (it gets these values from calls to the Swarm class's static methods):

- SwarmMovementAverage
    - what is the general direction the flock is moving?
    - This is known as 'alignment' – a boid attempting to move in the same direction as the swarm average
- SwarmCenterAverage
    - what is the center position of the flock?
    - This is know as 'cohesion' – a boid attempting to move towards the center of the swarm

Method *SwarmAdjustment*() finds a vector pointing towards the centre of the swarm, by calling method *VectorTowards*() and passing the swarm center position. The total swarm adjustment vector is the sum of this vector towards the swarm center, added to two-times the average velocity of the swarm. This weighing of twice the average velocity compared to just one-times the swarm center position direction, means that each Boid will try to keep moving with the swarm, rather than just cluster at some center point. Without this weighting the swarm may simply stop moving after it forms a tight grouping. The size of this weighted velocity adjustment is limited to the value of variable *maxDirectionChange*, which prevents Boids from changing their direction too abruptly.

Method *VectorTowards*() creates a unit vector in the direction of the given target position, and then multiplies this by *maxSpeed*, to create a velocity vector of length *maxSpeed* in the direction of the given target position. It then returns this vector with the Boids current velocity subtracted from it – i.e. the force to be applied to make the already moving Boid move towards the target position at speed *maxSpeed*.

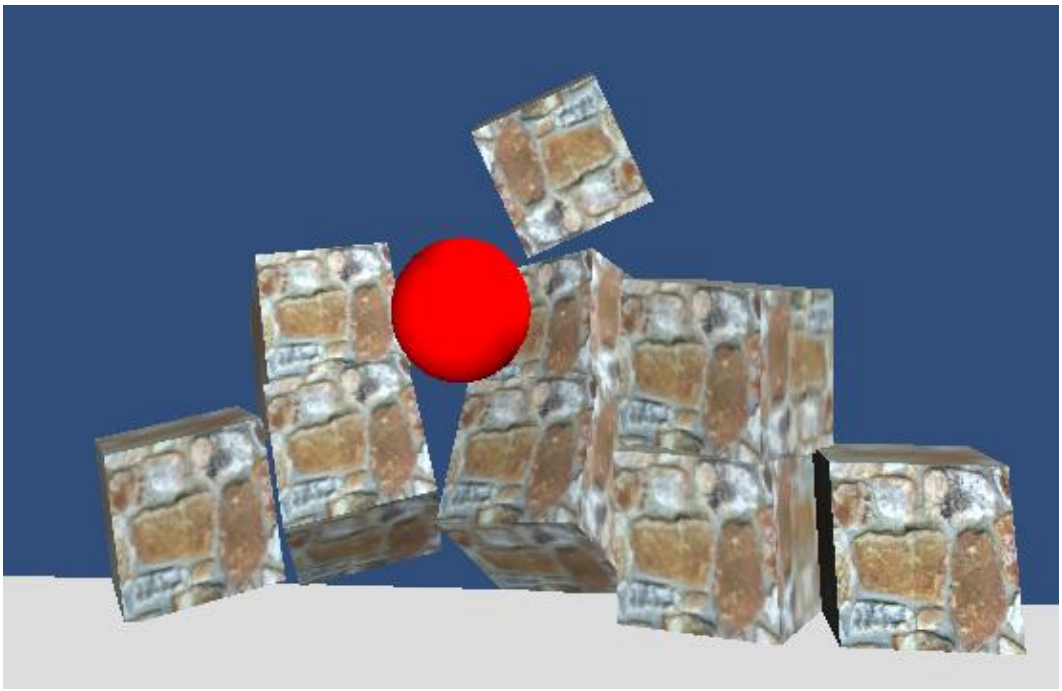
See also

- Individual movement: Player controlled cube
- Individual movement: Look-at an object
- Individual movement: Seek-Flee-Follow target

Instantiating projectiles with forward velocity

A very common game behavior is for the player (or a computer controlled non-player-character (NPC)) to fire a projectile in the direction they are facing. This recipe presents a way to achieve this behavior.

Although there are several steps in this recipe, there are just 3 script classes, and most of the work is the setting up of the cube wall and red sphere projectile.



0423_09_09.png

Getting ready

In folder 0423_09_05 you'll find a stone image texture file:

- stones.png

0423_09_10.png

How to do it...

1. Create a new scene, and add a directional light
2. Create a terrain, sized 10 x 10, positioned at (0,0,0)
3. Create a new cube named *BrickCube*, sized (1,1,1), apply the image texture *stones.png*, give it the tag 'brick', and add a Physics Rigid Body component to this GameObject
4. Create a new prefab named *Prefab-brick*, and drag *BrickCube* into it
5. Add the following C# script class to *Prefab-brick*:

```
// file: DestroyWhenFall.cs
using UnityEngine;
using System.Collections;

public class DestroyWhenFall : MonoBehaviour
{
   private const float MIN_Y = -1;

   void Update () {
      float y = transform.position.y;
      if( y < MIN_Y )
         Destroy( gameObject);
   }
}
```

6. Create a 'pyramid wall' of bricks, by placing 9 instances of *BrickPrefab* in the scene at the following positions:

- bottom row: (2, 0.5, 9) (3, 0.5, 9) (4, 0.5, 9) (5, 0.5, 9) (6, 0.5, 9)
- middle row: (3, 1.75, 9) (4, 1.75, 9) (5, 1.75, 9)
- top row: (4, 3, 9)

7. Create a new sphere named *SphereRed*, sized (1,1,1) with a red material, and add a Physics Rigid Body component to this GameObject
8. Create a new prefab named *Prefab-sphere*, and drag *SphereRed* into the prefab
9. Delete from the scene GameObject *SphereRed*
10. Add the *DestroyWhenFall* script class to *Prefab-sphere*
11. Remove the *Main Camera* GameObject (since there is already a *Main Camera* inside the controller you'll import in the next step)
12. Import the *Unity Character Controllers* package, and add an instance of the *First Person Controller* at (5, 1, 1)
13. Add the following script class to the *Main Camera* inside your *First Person Controller* (ensure *Main Camera* is selected in the Hierarchy):

```
// file: GameOverManager.cs
using UnityEngine;
using System.Collections;

public class GameOverManager : MonoBehaviour {
   private bool gameWon = false;

   void Update() {
      GameObject[] wallObjects =
GameObject.FindGameObjectsWithTag("brick");
      int numWallObjects = wallObjects.Length;

      if( numWallObjects < 1 )
         gameWon = true;
   }

   void OnGUI() {
      if( gameWon )
         GUILayout.Label("Well Done - you have destroyed
the whole wall!");
   }
}
```

14. Also add the following script class to the *Main Camera* inside your *First Person Controller* (ensure *Main Camera* is selected in the Hierarchy):

```
// file: FireProjectile.cs
using UnityEngine;
using System.Collections;
```

```
public class FireProjectile : MonoBehaviour {
   public Rigidbody projectilePrefab;
   private const float MIN_Y = -1;
   private float projectileSpeed = 15f;

   /** shortest time between firing */
   public const float FIRE_DELAY = 0.25f;
   private float nextFireTime = 0f;

   void Update() {
      if( Time.time > nextFireTime )
         CheckFireKey();
   }

   void CheckFireKey() {
      if( Input.GetButton("Fire1")) {
         CreateProjectile();

         nextFireTime = Time.time + FIRE_DELAY;
      }
   }

   void CreateProjectile() {
       Rigidbody projectile =
(Rigidbody)Instantiate(projectilePrefab,
transform.position, transform.rotation);

       // create and apply velocity
      Vector3 projectileVelocity = (projectileSpeed *
Vector3.forward);
       projectile.velocity =
transform.TransformDirection( projectileVelocity );
   }
}
```

15. Ensuring *Main Camera* inside your *First Person Controller* is selected, in the Inspector for the *FireProjectile* scripted component drag *SpherePrefab* over public variable *Projectile Prefab*

How it works...

Both prefabs (*SpherePrefab* and *BrickPrefab)* contain Rigid Body components, this allows the physics engine to control instances of such object. We can apply forces to objects with Rigid Bodies (e.g. we can 'throw' this object in a given direction), and the object will collide with and bounce off other objects. Both prefabs also have an instance of the script class *DestroyWhenFall* – this adds the simple behavior that when either the red

sphere or brick objects fall below the level of the terrain (Y = 0) the objects will be destroyed. This prevents the scene becoming full up with all the old sphere projectiles that have been fired. Also it allows us to detect when the game is completed – i.e. when all the brick objects have been pushed off the terrain. Testing this condition is the responsibility of the *GameOverManager* scripted component of the First Person Controller's Main Camera. Each frame it counts the number of objects tagged with 'brick', once that number is zero, the game completed message is displayed.

At the heart of this mini-game is the First Person Controller's *Main Camera* scripted component *FireProjectile*. Variable *nextFireTime* stores the next time a projectile may be fired, and each frame the current time is tested to see if that time has been reached, if it has, then method CheckFireKey() is called. This variable is initialized to zero, so the user doesn't have to wait at all to fire the first time.

Method *CheckFireKey*() tests whether at that instant the user is pressing the 'Fire' button. If the fire button is pressed, then method *CreateProjectile*() is called; and also the next time the can fire is set to FIRE_DELAY seconds in the future.

> NOTE: Unity input settings
>
> The default is the left mouse button but this can be changed in the project input settings: Edit | Project Settings | Input.

Method CreateProjectile() creates a new instance of the *SpherePrefab* (via the public variable *Projectile Prefab*) at the same position and rotation of the First Person Controller's Main Camera. Note that since Camera scripted component *FireProjectile* is attached to this camera, any references to *transform* will retrieve the transform component of that camera.

A velocity vector for the force (movement) to be applied to the sphere projectile is calculated by multiplying the a forward vector with the variable *projectileSpeed* – this variable can be 'tweaked' to achieve the speed of firing desired for a game. To ensure the sphere projectile is moved in the same direction the camera is facing, the TransformDirection() method of the camera's transform component is used to set the projectile's velocity property. The result is that a new sphere projectile instance is created at the camera location, and the force is applied to make it be fired in the direction the user has oriented the First Person Controller's camera.

There's more...

Some details you don't want to miss:

Efficiency issues with FindGameObjectsWithTag();

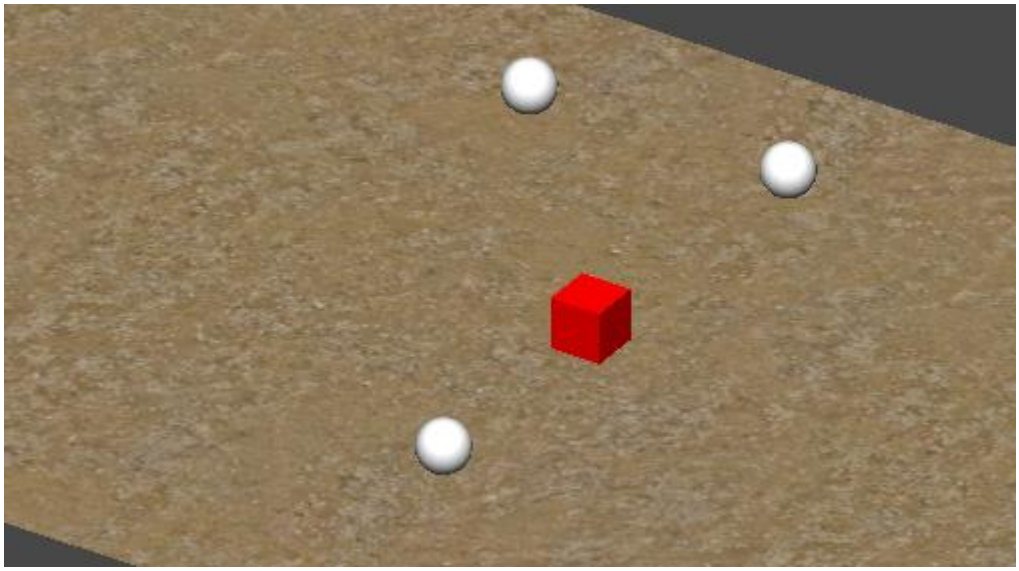
For small games, where efficiency isn't an issue, the use of FindGameObjectsWithTag() is fine. However, when tweaking a complex game to improve efficiency, the use of this method should be avoided, since it requires a search of all GameObjects in a scene and a test again their tag names. A more efficient approach would be to maintain a count of the number of instances that have been created, and then to ensure this count (list) is updated each time an object is destroyed. Thus the count (or size of the list) is all that would need to be tested against for our game over condition. Another alternative to FindGameObjectsWithTag() is to maintain a dynamic list al object references – see Chapter 10 for such optimisation recipes…

See also

- Individual movement: Seek-Flee-Follow target
- Cached component and object lookups (in Chapter 10)

Spawnpoints and Waypoints: Find one randomly

Many games make use of spawn points and waypoints. This recipe demonstrates a very common example of spawning – the choosing of a random spawn point, and the instantiation of an object at that point.



0423_09_11.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...

1. Add the following C# script class to the *Main Camera*:

```
// file: SpawnBall
using UnityEngine;
using System.Collections;

public class SpawnBall : MonoBehaviour {
   public GameObject prefabBall;
   public GameObject[] respawns;
   public const float FIRE_DELAY = 0.25f;
   private float nextFireTime = 0f;

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

   void Update() {
      if( Time.time > nextFireTime )
         CheckFireKey();
   }

   void CheckFireKey() {
      if( Input.GetButton("Fire1")) {
         CreateSphere();
         nextFireTime = Time.time + FIRE_DELAY;
      }
   }

   private void CreateSphere(){
      int r = Random.Range(0, respawns.Length);
      GameObject spawnPoint = respawns[r];
      if( spawnPoint )
         Instantiate( prefab_ball,
spawnPoint.transform.position,
spawnPoint.transform.rotation);
   }
```

```
}
```

2. Create a Sphere sized (1,1,1) at position (2,2,2)
3. Create a new prefab named *Prefab-ball*, and drag your Sphere into it (and then delete the Sphere from the Hierarchy)
4. Ensuring *Main Camera* is selected, in the Inspector for the *SpawnBall* scripted component drag *Prefab-ball* over public variable Projectile *Prefab Ball*
5. 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) and uncheck its Mesh Renderer (so it is invisible)
6. Copy your *Capsule-spawnPoint* and move the copy to (-3, 0.5, -3)
7. Now run your game – when you click the mouse (fire) button, a sphere should be instantiated randomly to one of the capsule locations.

How it works...

The two *Capsule-spawnPoint* objects represent candidate locations where we might wish to create an instance of our ball prefab. When the Start() message is received, 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" – i.e. the two *Capsule-spawnPoint* objects.

Variable *nextFireTime* stores the next time a projectile may be fired, and each frame the current time is tested to see if that time has been reached, if it has, then method CheckFireKey() is called. This variable is initialized to zero, so the user doesn't have to wait at all to fire the first time.

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; and also the next time the can fire is set to FIRE_DELAY seconds in the future.

Method *CreateSphere* () chooses a random index (*r*) in array *respawns*, and assigns variable *spawnPoint* to the GameObject at that location in the array. It then creates a new instance of *prefab_Ball* (via the public variable) at the same position and rotation of the *spawnPoint*.

See also

- Spawnpoints and Waypoints: Identify nearest
- Spawnpoints and Waypoints: Follow waypoints in sequence

Spawnpoints and Waypoints: Identify nearest

Rather than choosing a spawnpoint or waypoint randomly, in some cases we will wish to choose the closest such point to the current position of the player or camera. This recipe builds on the previous one to move the players red cube to the closest tagged "Respawn" object each time the fire key is clicked.

Getting ready

This recipe builds upon the previous one, so make a copy of that Unity project then follow the steps below.

How to do it...

1. Remove the SpawnBall scripted component from the Main Camera.
2. Enable the Mesh Rendered components of your two spawnpoint capsules (so you can see these white capsules on the terrain).
3. Add the following C# script class to Cube-player:

```
// file: NearestSpawnpoint
using UnityEngine;
using System.Collections;

public class NearestSpawnpoint : MonoBehaviour {
   public GameObject[] respawns;
   public const float FIRE_DELAY = 0.25f;
   private float nextFireTime = 0f;

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

   void Update() {
      if( Time.time > nextFireTime )
         CheckFireKey();
   }

   void CheckFireKey() {
      if( Input.GetButton("Fire1")) {
         MoveToNewPosition();
         nextFireTime = Time.time + FIRE_DELAY;
      }
   }
```

```
   private void MoveToNewPosition(){
      transform.position = NearestSpawnpointPosition();
   }

   private Vector3 NearestSpawnpointPosition(){
      if( respawns.Length < 1) return transform.position;
      Vector3 pos = respawns[0].transform.position;
      float shortestDistance = Vector3.Distance(
transform.position, pos);
      for(int i = 1; i < respawns.Length; i++){
         Vector3 newPos = respawns[i].transform.position;
         float newDist = Vector3.Distance(
transform.position, newPos);
         if( newDist < shortestDistance){
            pos = newPos;
            shortestDistance = newDist;
         }
      }

      return pos;
   }
}
```

4. Run your game. Move the red cube around with the arrow keys, and click the mouse button. The red cube should jump to the nearest capsule spawnpoint.

How it works...

Methods *Start*() and *CheckFireKey*() are the same, except that *CheckFireKey*() now calls method *MoveToNewPosition*().

Method *MoveToNewPosition*() has a single statement, setting the position of the (player cube's) transform to the Vector3 location returned from method *NearestSpawnpointPosition*().

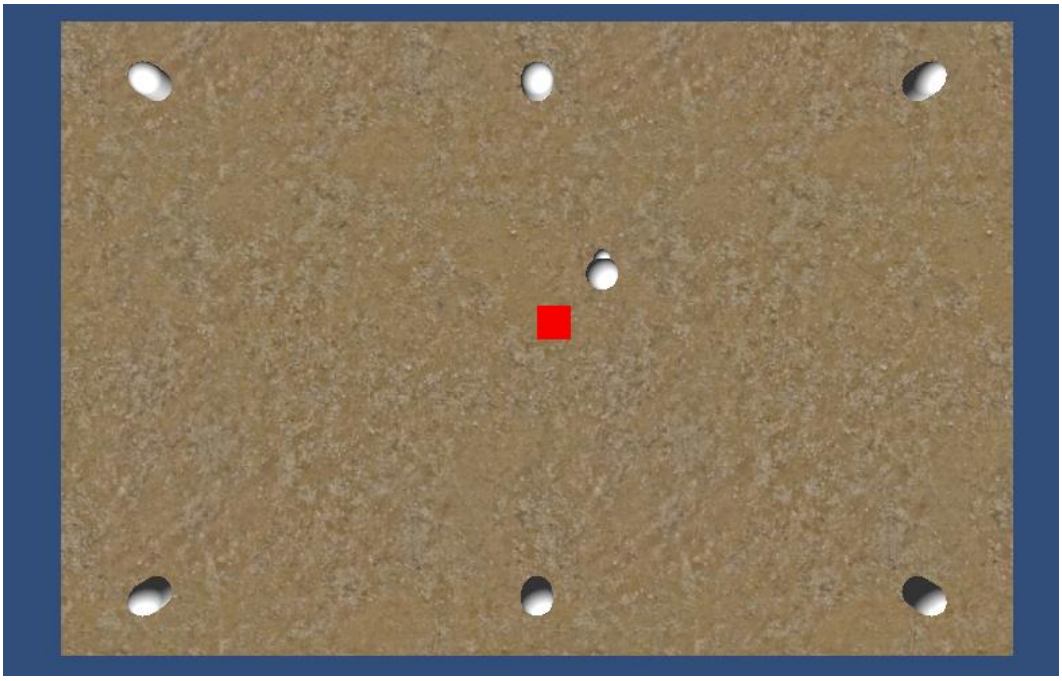
Method *NearestSpawnpointPosition*() has the job of finding and returning the position of the spawn point closest to the current location of the (player cube's) transform. First a check is made in case no respawn points can be found, in which case the current position of the player's cube is returned. Then the position of first element in array respawns is chosen as the initial closest spawnpoint and stored in variable *pos*, and the distance from the player cube to this object is stored in variable *shortestDistance*. A 'for' loop is used to loop through the remaining spawn points, and each time one is found to be closer than the distance in *shortestDistance*, then this closer position is stored into *pos* and this closer distance stored in *shortestDistance*. Once all spawn points have been tested, the method returns the Vector3 value in *pos*.

See also

- Spawnpoints and Waypoints: Find one randomly
- Spawnpoints and Waypoints: Follow waypoints in sequence

Spawnpoints and Waypoints: Follow waypoints in sequence

Waypoints are often used as a guide to make an autonomously moving NPC follow a path in a general way (but be able to respond with other directional behaviors such as flee or seek, if predators/prey are senses 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.



0423_09_12.png

Getting ready

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

How to do it...

1. Create a sphere named *Arrow*, positioned at (2, 0.5, 2) and with scale (1,1,1)
2. Create a second sphere, named *Sphere-small*, positioned at (0, 0, 0.5) and with scale (0.5, 0.5, 0.5)
5. Make *Sphere-small* a child-object of your GameObject *Arrow*
6. Create a new capsule object named *Capsule-waypoint-0* at (-12, 0, 8), give it the tag "*waypoint*"
7. Copy *Capsule-spawnPoint-0* naming the copy *Capsule-spawnPoint-1* and position this copy at (0, 0, 8)
8. Make 4 more copies (named *Capsule-spawnPoint-2..5*) positioning them as follows:
   - *Capsule-spawnPoint-2: position* = (12, 0, 8)
   - *Capsule-spawnPoint-3: position* = (12, 0, -8)
   - *Capsule-spawnPoint-4: position* = (0, 0, -8)
   - *Capsule-spawnPoint-5: position* = (-12, 0, -8)
9. Add the following C# script class to GameObject *Arrow*

```
// file: WaypointManager.cs
using UnityEngine;
using System.Collections;

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

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

      // default is first in the array
      int nextIndex = 0;

      // find array index of given waypoint
      int currentIndex = -1;
```

```
      for(int i = 0; i < waypoints.Length; i++){
         if( current == waypoints[i] )
            currentIndex = i;
      }

      int lastIndex = (waypoints.Length - 1);
      if(currentIndex > -1 && currentIndex < lastIndex)
         nextIndex = currentIndex + 1;

      return waypoints[nextIndex];
   }
}
```

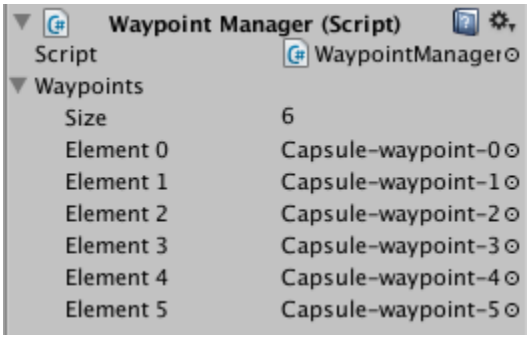
10. Add the following script class to GameObject *Arrow*

```
// file: MoveTowardsWaypoint.cs
using UnityEngine;
using System.Collections;

public class MoveTowardsWaypoint : MonoBehaviour {
   public const float ARRIVE_DISTANCE = 3f;
   public float speed = 5.0F;
   private GameObject targetGO;
   private WaypointManager waypointManager;

   private void Awake(){
      waypointManager = GetComponent<WaypointManager>();
      targetGO = waypointManager.NextWaypoint(null);
   }

   private void Update () {
      transform.LookAt( targetGO.transform );
      float distance = speed * Time.deltaTime;
      Vector3 source = transform.position;
      Vector3 target = targetGO.transform.position;
      transform.position = Vector3.MoveTowards(source,
target, distance);

      if( Vector3.Distance( source, target) <
ARRIVE_DISTANCE)
         targetGO = waypointManager.NextWaypoint(
targetGO );
   }
}
```

11. With *Arrow* selected in the Hierarchy, in the *Waypoint Manager* scripted component in the Inspector, do the following:

- set the size of the waypoints public array to 6

- drag the 6 capsule waypoint objects into the array in sequence



0423_09_13.png

How it works...

Class *WaypointManager* has an array waypoints, and a method *NextWaypoint*(). The array is public, and the level constructor assigns the waypoints in sequence into this array. Method *NextWaypoint*() takes as input a GameObject (or null), and returns a GameObject. The input is a waypoint, and the return value is the next waypoint GameObject in the array sequence, or the first in the array if the input object cannot be found.

Class *MoveTowardsWaypoint* has 4 variables:

- ARRIVE_DISTANCE: this is the distance from a waypoint, at which point the Arrow will then choose the next waypoint as its target
- speed: the distance to travel each second (so as usual it will be multiplied by Time.deltaTime for the distance per frame)
- targetGO: the GameObject of the current target waypoint object to move towards
- waypointManager: a reference to the WaypointManager object in the parent (Arrow) GameObject

When an *Awake*() message is received, the *waypointManager* object is assigned to the *WaypointManager* in the *Arrow* object; and targetGO is set to be the firsy waypoint in the sequence (by sending argument of 'null' the first waypoint in the array will always be returned).

Method *Update*() is responsible for making the *Arrow* object move towards the location of *targetGO*. It does this making use of the Vector3 method MoveTowards(). The final statement in the method then tests whether the current *targetGO* has been reached

(distance is less than ARRIVE_DISTANCE) – once it has been reached, then *targetGO* is reassigned to the next waypoint in the sequence.

See also

- Spawnpoints and Waypoints: Find one randomly
- Spawnpoints and Waypoints: Identify nearest

Basic state-driven behavior (playing-win-lose)

Games as a whole, and individual objects or characters, can often be thought of (or modeled as) passing through different 'states' or 'modes'. Modeling states and changes of state (due to 'events' or game conditions) is a very common way to manage the complexity of games, and game components. In this recipe we create a simple 3-state game (game playing / game won / game lost), using a single GameManager class.

How to do it...

1. Add the following C# script class to the Main Camera

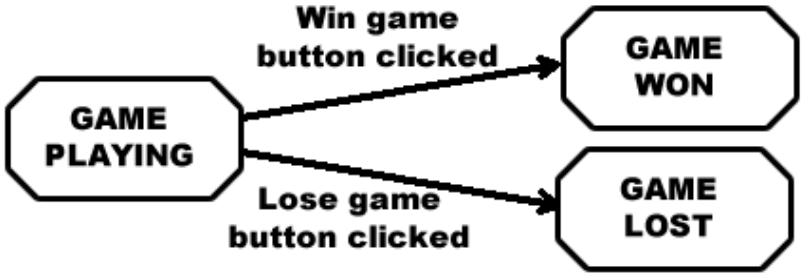
```
// file: GameManager
using UnityEngine;
using System.Collections;

public class GameManager : MonoBehaviour {
   private float gameStartTime;
   private float gamePlayingTime;

   private enum GameStateType {
      STATE_GAME_PLAYING,
      STATE_GAME_WON,
      STATE_GAME_LOST,
   }

   private GameStateType currentState;

   private void Start () {
      NewGameState( GameStateType.STATE_GAME_PLAYING );
   }

   private void NewGameState(GameStateType newState) {
      // (1) state EXIT actions
      switch( currentState ) {
      case GameStateType.STATE_GAME_PLAYING:
         gameStartTime = Time.time;
```

```
         break;
      }

      // (2) change current state
      currentState = newState;

      // (3) state ENTER actions
   }

   private void Update () {
      switch( currentState ) {
      case GameStateType.STATE_GAME_PLAYING:
         StateGamePlayingUpdate();
         break;
      case GameStateType.STATE_GAME_WON:
         StateGameWonUpdate();
         break;
      case GameStateType.STATE_GAME_LOST:
         StateGameLostUpdate();
         break;
      }
   }

   private void OnGUI () {
      switch( currentState ) {
      case GameStateType.STATE_GAME_PLAYING:
         StateGamePlayingGUI();
         break;
      case GameStateType.STATE_GAME_WON:
         StateGameWonGUI();
         break;
      case GameStateType.STATE_GAME_LOST:
         StateGameLostGUI();
         break;
      }
   }

   private void StateGamePlayingGUI() {
      GUILayout.Label("state: GAME PLAYING - time since
game started = "  + gamePlayingTime);
      bool winGameButtonClicked = GUILayout.Button("WIN
the game");
      bool loseGameButtonClicked = GUILayout.Button("LOSE
the game");

      if( winGameButtonClicked )
         NewGameState( GameStateType.STATE_GAME_WON );

      if( loseGameButtonClicked )
```

```
            NewGameState( GameStateType.STATE_GAME_LOST );
      }

      private void StateGameWonGUI() {
         GUILayout.Label("state: GAME WON - game duration =
" + gamePlayingTime);
      }

      private void StateGameLostGUI() {
         GUILayout.Label("state: GAME LOST - game duration =
" + gamePlayingTime);
      }

      private void StateGameWonUpdate() {print("update -
state: GAME WON"); }
      private void StateGameLostUpdate() {  print("update -
state: GAME LOST"); }

      private void StateGamePlayingUpdate() {
         gamePlayingTime = (Time.time - gameStartTime);
         print("update - state: GAME PLAYING :: time since
game started = " + gamePlayingTime);
      }

}
```

How it works...

As can be seen in the state chart figure, this recipe models a simple game which starts in the GAME PLAYING state, depending on the button clicked by the user, the game moves either into the GAME WON state or the GAME LOST state. The 3 possible states of the system are defined using the enumerated type *GameStateType*; and the current state of the system at any point in time is stored in variable *currentState*.



0423_09_14.png

Key aspects of state driven games include the following:

- When a state changes, they may be actions as a particular state is exited, then the system changes it data to represent the new state, then there may be actions as the new state is entered
    - in our example this is the responsibility of method NewGameState(). The code for this method is in three parts, implementing the logic for these three steps: (1) old state exit, (2) set new state, (3) new stated entered
- When the GameManager object receives messages (e.g. every frame for Update() and OnGUI()), its behavior must be appropriate for the current state. So we see in these methods 'switch' statements, that call state-specific methods. For example, if the current state is STATE_GAME_PLAYING, then when an Update() message is received, the method StateGamePlayingGUI() will be called, and when an OnGUI() message is received, the method StateGamePlayingGUI() will be called

However, the size of our methods, and the number of methods, in our GameManager class will grow significantly with as more states, and more complex game logic is needed for non-trivial games. The next recipe takes a more sophisticated approach to state-driven games, where each state has its own class.

See also

- State behaviors with the state pattern

State behaviors with the state pattern

The previous pattern illustrated the usefulness of modeling game 'states', but also how a game manager class can grown in size and become unmanageable. To manage the complexity of many states, and complex behaviors of states, the 'state pattern' has been proposed in the software development community. Design 'patterns' are general purpose software component architectures that have been tried and tested and found to be good solutions to commonly occurring software system features. The key features of the state pattern are that each state is modeled by its own class, and all states inherit (are sub-classed) from a single parent state class. The states need to know about each other, in order to tell the game manager to change the current state – this is a small

price to pay for the division of the complexity of overall game behaviors into separate state classes.

How to do it...

1. Create a new C# script class GameState:

```
// file: GameState
using UnityEngine;
using System.Collections;

public abstract class GameState : MonoBehaviour {
   protected GameManager gameManager;
   protected void Awake() {
      gameManager = GetComponent<GameManager>();
   }

   public abstract void OnStateEntered();
   public abstract void OnStateExit();
   public abstract void StateUpdate();
   public abstract void StateGUI();
}
```

2. Add the following script class to the Main Camera

```
// file: GameManager
using UnityEngine;
using System.Collections;

public class GameManager : MonoBehaviour {

   public StateGamePlaying stateGamePlaying;
   public StateGameWon stateGameWon;
   public StateGameLost stateGameLost;

   private GameState currentState;

   private void Awake () {
      stateGamePlaying =
GetComponent<StateGamePlaying>();
      stateGameWon = GetComponent<StateGameWon>();
      stateGameLost = GetComponent<StateGameLost>();
   }

   private void Start () {
      NewGameState( stateGamePlaying );
   }

   private void Update () {
```

```
      if (currentState != null)
         currentState.StateUpdate();
   }

   private void OnGUI () {
      if (currentState != null)
         currentState.StateGUI();
   }

   public void NewGameState(GameState newState)
   {
      if( null != currentState)
         currentState.OnStateExit();

      currentState = newState;
      currentState.OnStateEntered();
   }
}
```

3. Add the following script class to the Main Camera

```
// file: StateGamePlaying
using UnityEngine;
using System.Collections;

public class StateGamePlaying : GameState {
   public override void OnStateEntered(){}
   public override void OnStateExit(){}

   public override void StateGUI() {
      GUILayout.Label("state: GAME PLAYING");
      bool winGameButtonClicked = GUILayout.Button("WIN
the game");
      bool loseGameButtonClicked = GUILayout.Button("LOSE
the game");

      if( winGameButtonClicked )

   gameManager.NewGameState(gameManager.stateGameWon);

      if( loseGameButtonClicked )

   gameManager.NewGameState(gameManager.stateGameLost);
   }


   public override void StateUpdate() {
      print ("StateGamePlaying::StateUpdate() - would do
something here");
```

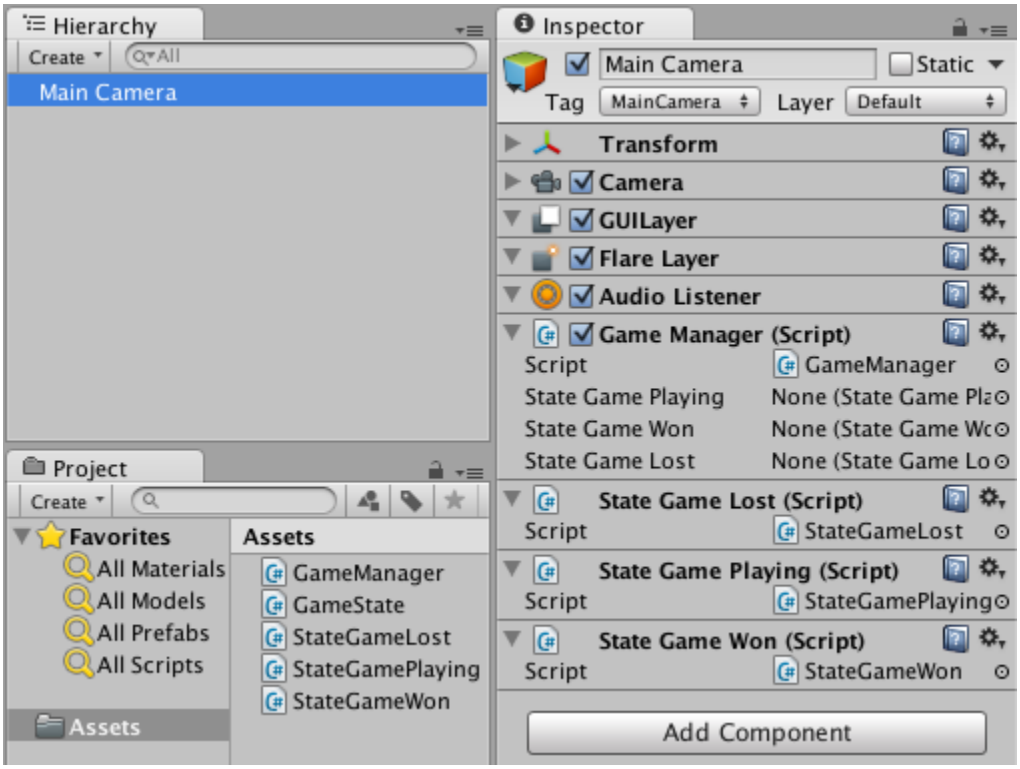
```
    }
}
```

4. Add the following script class to the Main Camera

```
// file: StateGameWon
using UnityEngine;
using System.Collections;

public class StateGameWon : GameState {
   public override void OnStateEntered(){}
   public override void OnStateExit(){}

   public override void StateGUI() {
      GUILayout.Label("state: GAME WON");
   }

   public override void StateUpdate() {
      print ("StateGameWon::StateUpdate() - would do
something here");
   }
}
```

5. Add the following script class to the Main Camera

```
// file: StateGameLost
using UnityEngine;
using System.Collections;

public class StateGameLost : GameState {
   public override void OnStateEntered(){}
   public override void OnStateExit(){}

   public override void StateGUI() {
      GUILayout.Label("state: GAME LOST");
   }

   public override void StateUpdate() {
      print ("StateGameLost::StateUpdate() - would do
something here");
   }
}
```



0423_09_15.png

How it works...

The *Scene* is very straightforward for this recipe, there is the single GameObject *Main Camera*, which has 4 scripted object components attached to it:

- GameManager
- StateGamePlaying
- StateGameLost
- StateGameWon

The GameManager class has 3 public variables, one for each of the states (StateGamePlaying, StateGameLost and StateGameWon). While these public variables are unassigned before the game runs, when the Awake() method is executed GetComponent() statements are used to assigned these variables to the 3 state object components in the Main Camera. See the information box to understand why this is necessary.

The final part of the Awake() method is to pass a reference to the GameManager object to any of the states that will need to communicate with the GameManager. For this recipe only the gamePlayingState needs to ask the game manager to change states, so the final statement of the Awake() method is `gamePlayingState.SetGameManager( this );`.

> NOTE: Cannot create MonoBehavior objects with ''new'
>
> Since we want our state objects to be able to respond to Unity messages such as Update() and OnGUI(), then the State class must inherit from MonoBehavior. However, due to the way Unity works, MonoBehavior objects **cannot** be created using the 'new' keyword.
>
> The straightforward solution to this is to attach an object of each state to the same GameObject as the GameManager. The GameManager is then able to use GetComponent() statements to access the objects each state.
>
> Another alternative would be to use prefabs and Instantiate().

The GameManager has one other variable *currentState*:

- this is a reference to the current state object at any time during the game running (initially it will be 'null'). Since it is of the class GameState (the parent of all state classes), it can refer to any of the different state objects.

After Awake(), the GameManager will receive a Start() message. This method initializes the *currentState* to be the *gamePlayingState* object.

Each frame the GameManager will receive Update() and OnGUI() messages, and upon receiving these messages the GameManager sends StateUpdate() and StateGUI() messages to the *currentState* object – so each frame the object for the current state of the game will execute those methods. For example, when the *currentState* is game playing, each frame the *gamePlayingObject* will display the win/lose buttons GUI, and will print() a console message.

When the user clicks a button, the *gamePlayingObject* will call the GameManager instance's NewState() method, passing it the object corresponding to the new state. So if the user clicks Win the Game then the NewState() method is passed `gameManager.stateGameWon`.

See also

- Basic state-driven behavior (playing-win-lose)