1
Behaviors – Intelligent Movement

In this chapter, we will develop AI algorithms for movement by covering the following recipes:

- Creating the behaviors' template
- Pursuing and evading
- Arriving and leaving
- Facing objects
- Wandering around
- Following a path
- Avoiding agents
- Avoiding walls
- Blending behaviors by weight
- Blending behaviors by priority
- Combining behaviors using a steering pipeline
- Shooting a projectile
- Predicting a projectile's landing spot
- Targeting a projectile
- Creating a jump system

Introduction

Unity has been one of the most popular game engines for quite a while now, and it's probably the de facto game development tool for indie developers, not only because of its business model, which has a low entry barrier, but also because of its robust project editor, year-by-year technological improvement, and most importantly, ease of use and an ever-growing community of developers around the globe.

Thanks to Unity's heavy lifting behind the scenes (rendering, physics, integration, and cross-platform deployment, just to name a few) it's possible for us to focus on creating the AI systems that will bring to life our games, creating great real-time experiences in the blink of an eye.

The goal of this book is to give you the tools to build great AI, for creating better enemies, polishing that final boss, or even building your own customized AI engine.

In this chapter, we will start by exploring some of the most interesting movement algorithms based on the steering behavior principles developed by Craig Reynolds, along with work from Ian Millington. These recipes are the stepping stones for most of the AI used in advanced games and other algorithms that rely on movement, such as the family of path-finding algorithms.

Creating the behavior template

Before creating our behaviors, we need to code the stepping stones that help us not only to create only intelligent movement, but also to build a modular system to change and add these behaviors. We will create custom data types and base classes for most of the algorithms covered in this chapter.

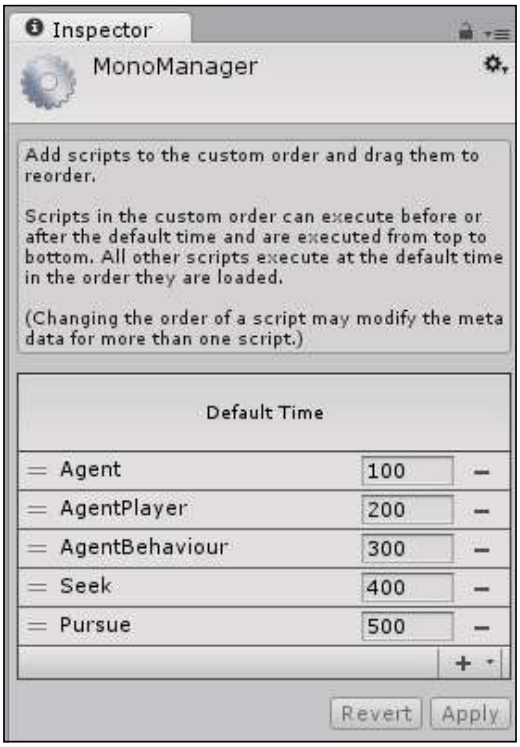
Getting ready

Our first step is to remember the update function order of execution:

- `Update`
- `LateUpdate`

Also, it's important to refresh so that we can select the scripts' order of execution. For our behaviors to work as intended, the rules for ordering are as follows:

- Agent scripts
- Behavior scripts
- Behaviors or scripts based on the previous ones



This is an example of how to arrange the order of execution for the movement scripts. We need to pursue derives from Seek, which derives from AgentBehaviour.

How to do it...

We need to create three classes: `Steering`, `AgentBehaviour`, and `Agent`:

1. `Steering` serves as a custom data type for storing the movement and rotation of the agent:

```
using UnityEngine;
using System.Collections;
public class Steering
{
    public float angular;
    public Vector3 linear;
    public Steering ()
    {
        angular = 0.0f;
        linear = new Vector3();
    }
}
```

2. Create the `AgentBehaviour` class, which is the template class for most of the behaviors covered in this chapter:

```
using UnityEngine;
using System.Collections;
public class AgentBehaviour : MonoBehaviour
{
    public GameObject target;
    protected Agent agent;
    public virtual void Awake ()
    {
        agent = gameObject.GetComponent<Agent>();
    }
    public virtual void Update ()
    {
            agent.SetSteering(GetSteering());
    }
    public virtual Steering GetSteering ()
    {
        return new Steering();
    }
}
```

3. Finally, `Agent` is the main component, and it makes use of behaviors in order to create intelligent movement. Create the file and its barebones:

```
using UnityEngine;
using System.Collections;
public class Agent : MonoBehaviour
{
    public float maxSpeed;
    public float maxAccel;
    public float orientation;
    public float rotation;
    public Vector3 velocity;
    protected Steering steering;
    void Start ()
    {
        velocity = Vector3.zero;
        steering = new Steering();
    }
    public void SetSteering (Steering steering)
    {
        this.steering = steering;
    }
}
```

4. Next, we code the `Update` function, which handles the movement according to the current value:

```
public virtual void Update ()
{
    Vector3 displacement = velocity * Time.deltaTime;
    orientation += rotation * Time.deltaTime;
    // we need to limit the orientation values
    // to be in the range (0 - 360)
    if (orientation < 0.0f)
        orientation += 360.0f;
    else if (orientation > 360.0f)
        orientation -= 360.0f;
    transform.Translate(displacement, Space.World);
    transform.rotation = new Quaternion();
    transform.Rotate(Vector3.up, orientation);
}
```

5. Finally, we implement the `LateUpdate` function, which takes care of updating the steering for the next frame according to the current frame's calculations:

```
public virtual void LateUpdate ()
{
    velocity += steering.linear * Time.deltaTime;
    rotation += steering.angular * Time.deltaTime;
    if (velocity.magnitude > maxSpeed)
    {
        velocity.Normalize();
        velocity = velocity * maxSpeed;
    }
    if (steering.angular == 0.0f)
    {
        rotation = 0.0f;
    }
    if (steering.linear.sqrMagnitude == 0.0f)
    {
        velocity = Vector3.zero;
    }
    steering = new Steering();
}
```

How it works...

The idea is to be able to delegate the movement's logic inside the `GetSteering()` function on the behaviors that we will later build, simplifying our agent's class to a main calculation based on those.

Besides, we are guaranteed to set the agent's steering value before it is used thanks to Unity script and function execution orders.

There's more...

This is a component-based approach, which means that we have to remember to always have an `Agent` script attached to `GameObject` for the behaviors to work as expected.

See also

For further information on Unity's game loop and the execution order of functions and scripts, please refer to the official documentation available online at:

- `http://docs.unity3d.com/Manual/ExecutionOrder.html`
- `http://docs.unity3d.com/Manual/class-ScriptExecution.html`

Pursuing and evading

Pursuing and evading are great behaviors to start with because they rely on the most basic behaviors and extend their functionality by predicting the target's next step.

Getting ready

We need a couple of basic behaviors called `Seek` and `Flee`; place them right after the `Agent` class in the scripts' execution order.

The following is the code for the `Seek` behaviour:

```
using UnityEngine;
using System.Collections;
public class Seek : AgentBehaviour
{
    public override Steering GetSteering()
    {
        Steering steering = new Steering();
        steering.linear = target.transform.position - transform.
position;
```

```
            steering.linear.Normalize();
            steering.linear = steering.linear * agent.maxAccel;
            return steering;
        }
    }
```

Also, we need to implement the `Flee` behavior:

```
using UnityEngine;
using System.Collections;
public class Flee : AgentBehaviour
{
    public override Steering GetSteering()
    {
        Steering steering = new Steering();
        steering.linear = transform.position - target.transform.
position;
        steering.linear.Normalize();
        steering.linear = steering.linear * agent.maxAccel;
        return steering;
    }
}
```

How to do it...

`Pursue` and `Evade` are essentially the same algorithm but differ in terms of the base class they derive from:

1. Create the `Pursue` class, derived from `Seek`, and add the attributes for the prediction:

```
using UnityEngine;
using System.Collections;

public class Pursue : Seek
{
    public float maxPrediction;
    private GameObject targetAux;
    private Agent targetAgent;
}
```

2. Implement the `Awake` function in order to set up everything according to the real target:

```
public override void Awake()
{
    base.Awake();
    targetAgent = target.GetComponent<Agent>();
    targetAux = target;
    target = new GameObject();
}
```

3. As well as implement the `OnDestroy` function, to properly handle the internal object:

```
void OnDestroy ()
{
    Destroy(targetAux);
}
```

4. Finally, implement the `GetSteering` function:

```
public override Steering GetSteering()
{
    Vector3 direction = targetAux.transform.position - transform.
position;
    float distance = direction.magnitude;
    float speed = agent.velocity.magnitude;
    float prediction;
    if (speed <= distance / maxPrediction)
        prediction = maxPrediction;
    else
        prediction = distance / speed;
    target.transform.position = targetAux.transform.position;
    target.transform.position += targetAgent.velocity *
prediction;
    return base.GetSteering();
}
```

5. To create the `Evade` behavior, the procedure is just the same, but it takes into account that `Flee` is the parent class:

```
public class Evade : Flee

{

    // everything stays the same

}
```

How it works...

These behaviors rely on `Seek` and `Flee` and take into consideration the target's velocity in order to predict where it will go next; they aim at that position using an internal extra object.

Arriving and leaving

Similar to `Seek` and `Flee`, the idea behind these algorithms is to apply the same principles and extend the functionality to a point where the agent stops automatically after a condition is met, either being close to its destination (arrive), or far enough from a dangerous point (leave).

Getting ready

We need to create one file for each of the algorithms, `Arrive` and `Leave`, respectively, and remember to set their custom execution order.

How to do it...

They use the same approach, but in terms of implementation, the name of the member variables change as well as some computations in the first half of the `GetSteering` function:

1. First, implement the `Arrive` behaviour with its member variables to define the radius for stopping (target) and slowing down:

```
using UnityEngine;
using System.Collections;

public class Arrive : AgentBehaviour
{
    public float targetRadius;
    public float slowRadius;
    public float timeToTarget = 0.1f;
}
```

2. Create the `GetSteering` function:

```
public override Steering GetSteering()
{
    // code in next steps
}
```

3. Define the first half of the `GetSteering` function, in which we compute the desired speed depending on the distance from the target according to the radii variables:

```
Steering steering = new Steering();
Vector3 direction = target.transform.position - transform.
position;
float distance = direction.magnitude;
float targetSpeed;
if (distance < targetRadius)
    return steering;
if (distance > slowRadius)
    targetSpeed = agent.maxSpeed;
else
    targetSpeed = agent.maxSpeed * distance / slowRadius;
```

4. Define the second half of the `GetSteering` function, in which we set the steering value and clamp it according to the maximum speed:

```
Vector3 desiredVelocity = direction;
desiredVelocity.Normalize();
desiredVelocity *= targetSpeed;
steering.linear = desiredVelocity - agent.velocity;
steering.linear /= timeToTarget;
if (steering.linear.magnitude > agent.maxAccel)
{
    steering.linear.Normalize();
    steering.linear *= agent.maxAccel;
}
return steering;
```

5. To implement `Leave`, the name of the member variables changes:

```
using UnityEngine;
using System.Collections;

public class Leave : AgentBehaviour
{
    public float escapeRadius;
    public float dangerRadius;
    public float timeToTarget = 0.1f;
}
```

6. Define the first half of the `GetSteering` function:

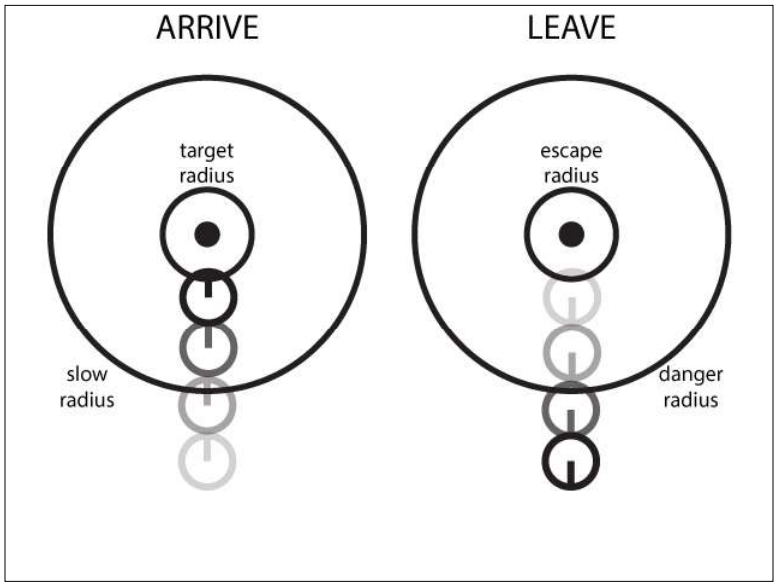
```
Steering steering = new Steering();
Vector3 direction = transform.position - target.transform.
position;
float distance = direction.magnitude;
```

```
if (distance > dangerRadius)
    return steering;
float reduce;
if (distance < escapeRadius)
    reduce = 0f;
else
    reduce = distance / dangerRadius * agent.maxSpeed;
float targetSpeed = agent.maxSpeed - reduce;
```

7. And finally, the second half of `GetSteering` stays just the same.

How it works...

After calculating the direction to go in, the next calculations are based on two radii distances in order to know when to go full throttle, slow down, and stop; that's why we have several `if` statements. In the `Arrive` behavior, when the agent is too far, we aim to full-throttle, progressively slow down when inside the proper radius, and finally to stop when close enough to the target. The converse train of thought applies to `Leave`.



A visual reference for the Arrive and Leave behaviors

Facing objects

Real-world aiming, just like in combat simulators, works a little differently from the widely-used *automatic* aiming in almost every game. Imagine that you need to implement an agent controlling a tank turret or a humanized sniper; that's when this recipe comes in handy.

Getting ready

We need to make some modifications to our `AgentBehaviour` class:

1. Add new member values to limit some of the existing ones:

```
public float maxSpeed;
public float maxAccel;
public float maxRotation;
public float maxAngularAccel;
```

2. Add a function called `MapToRange`. This function helps in finding the actual direction of rotation after two orientation values are subtracted:

```
public float MapToRange (float rotation) {
    rotation %= 360.0f;
    if (Mathf.Abs(rotation) > 180.0f) {
        if (rotation < 0.0f)
            rotation += 360.0f;
        else
            rotation -= 360.0f;
    }
    return rotation;
}
```

3. Also, we need to create a basic behavior called `Align` that is the stepping stone for the facing algorithm. It uses the same principle as `Arrive`, but only in terms of rotation:

```
using UnityEngine;
using System.Collections;

public class Align : AgentBehaviour
{
    public float targetRadius;
    public float slowRadius;
    public float timeToTarget = 0.1f;

    public override Steering GetSteering()
    {
```

```
        Steering steering = new Steering();
        float targetOrientation = target.GetComponent<Agent>().
orientation;
        float rotation = targetOrientation - agent.orientation;
        rotation = MapToRange(rotation);
        float rotationSize = Mathf.Abs(rotation);
        if (rotationSize < targetRadius)
            return steering;
        float targetRotation;
        if (rotationSize > slowRadius)
            targetRotation = agent.maxRotation;
        else
            targetRotation = agent.maxRotation * rotationSize /
slowRadius;
        targetRotation *= rotation / rotationSize;
        steering.angular = targetRotation - agent.rotation;
        steering.angular /= timeToTarget;
        float angularAccel = Mathf.Abs(steering.angular);
        if (angularAccel > agent.maxAngularAccel)
        {
            steering.angular /= angularAccel;
            steering.angular *= agent.maxAngularAccel;
        }
        return steering;
    }
}
```

How to do it...

We now proceed to implement our facing algorithm that derives from `Align`:

1. Create the `Face` class along with a private auxiliary target member variable:

```
using UnityEngine;
using System.Collections;

public class Face : Align
{
    protected GameObject targetAux;
}
```

2. Override the `Awake` function to set up everything and swap references:

```
public override void Awake()
{
    base.Awake();
```

```
        targetAux = target;
        target = new GameObject();
        target.AddComponent<Agent>();
    }
```

3. Also, implement the `OnDestroy` function to handle references and avoid memory issues:

```
void OnDestroy ()
{
    Destroy(target);
}
```

4. Finally, define the `GetSteering` function:

```
public override Steering GetSteering()
{
    Vector3 direction = targetAux.transform.position - transform.
position;
    if (direction.magnitude > 0.0f)
    {
        float targetOrientation = Mathf.Atan2(direction.x,
direction.z);
        targetOrientation *= Mathf.Rad2Deg;
        target.GetComponent<Agent>().orientation =
targetOrientation;
    }
    return base.GetSteering();
}
```

How it works...

The algorithm computes the internal target orientation according to the vector between the agent and the real target. Then, it just delegates the work to its parent class.

Wandering around

This technique works like a charm for random crowd simulations, animals, and almost any kind of NPC that requires random movement when idle.

Getting ready

We need to add another function to our `AgentBehaviour` class called `OriToVec` that converts an orientation value to a vector.

```
public Vector3 GetOriAsVec (float orientation) {
    Vector3 vector  = Vector3.zero;
    vector.x = Mathf.Sin(orientation * Mathf.Deg2Rad) * 1.0f;
    vector.z = Mathf.Cos(orientation * Mathf.Deg2Rad) * 1.0f;
    return vector.normalized;
}
```

How to do it...

We could see it as a big three-step process in which we manipulate the internal target position in a parameterized random way, face that position, and move accordingly:

1. Create the `Wander` class deriving from `Face`:

```
using UnityEngine;
using System.Collections;

public class Wander : Face
{
    public float offset;
    public float radius;
    public float rate;
}
```

2. Define the `Awake` function in order to set up the internal target:

```
public override void Awake()
{
    target = new GameObject();
    target.transform.position = transform.position;
    base.Awake();
}
```

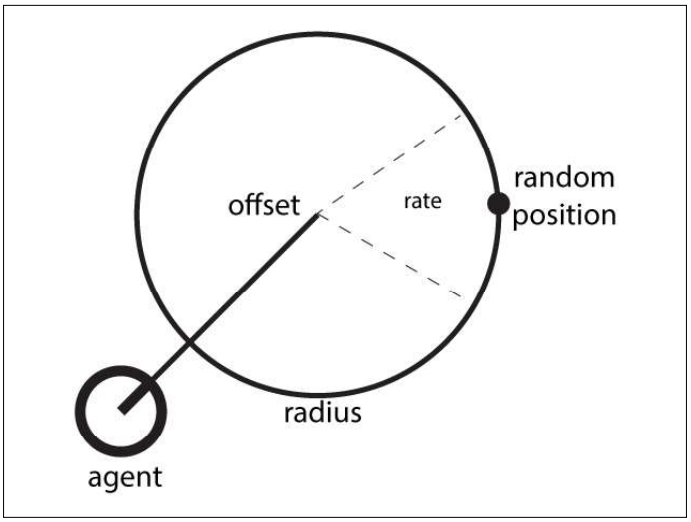
3. Define the `GetSteering` function:

```
public override Steering GetSteering()
{
    Steering steering = new Steering();
    float wanderOrientation = Random.Range(-1.0f, 1.0f) * rate;
    float targetOrientation = wanderOrientation + agent.
orientation;
    Vector3 orientationVec = OriToVec(agent.orientation);
```

```
    Vector3 targetPosition = (offset * orientationVec) +
transform.position;
    targetPosition = targetPosition + (OriToVec(targetOrientation)
* radius);
    targetAux.transform.position = targetPosition;
    steering = base.GetSteering();
    steering.linear = targetAux.transform.position - transform.
position;
    steering.linear.Normalize();
    steering.linear *= agent.maxAccel;
    return steering;
}
```

How it works...

The behavior takes into consideration two radii in order to get a random position to go to next, looks towards that random point, and converts the computed orientation into a direction vector in order to advance.



A visual description of the parameters for creating the Wander behavior

Following a path

There are times when we need scripted routes, and it's just inconceivable to do this entirely by code. Imagine you're working on a stealth game. Would you code a route for every single guard? This technique will help you build a flexible path system for those situations:

Getting ready

We need to define a custom data type called `PathSegment`:

```
using UnityEngine;
using System.Collections;

public class PathSegment
{
    public Vector3 a;
    public Vector3 b;

    public PathSegment () : this (Vector3.zero, Vector3.zero){}
    public PathSegment (Vector3 a, Vector3 b)
    {
        this.a = a;
        this.b = b;
    }
}
```

How to do it...

This is a long recipe that could be seen as a big two-step process. First, we build the `Path` class, which abstracts points in the path from their specific spatial representations, and then we build the `PathFollower` behavior, which makes use of that abstraction in order to get actual spatial points to follow:

1. Create the `Path` class, which consists of nodes and segments but only the nodes are public and assigned manually:

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

public class Path : MonoBehaviour
{
    public List<GameObject> nodes;
    List<PathSegment> segments;
}
```

2. Define the `Start` function to set the segments when the scene starts:

```
void Start()
{
    segments = GetSegments();
}
```

3. Define the `GetSegments` function to build the segments from the nodes:

```
public List<PathSegment> GetSegments ()
{
    List<PathSegment> segments = new List<PathSegment>();
    int i;
    for (i = 0; i < nodes.Count - 1; i++)
    {
        Vector3 src = nodes[i].transform.position;
        Vector3 dst = nodes[i+1].transform.position;
        PathSegment segment = new PathSegment(src, dst);
        segments.Add(segment);
    }
    return segments;
}
```

4. Define the first function for abstraction, called `GetParam`:

```
public float GetParam(Vector3 position, float lastParam)
{
    // body
}
```

5. We need to find out which segment the agent is closest to:

```
float param = 0f;
PathSegment currentSegment = null;
float tempParam = 0f;
foreach (PathSegment ps in segments)
{
    tempParam += Vector3.Distance(ps.a, ps.b);
    if (lastParam <= tempParam)
    {
        currentSegment = ps;
        break;
    }
}
if (currentSegment == null)
    return 0f;
```

6. Given the current position, we need to work out the direction to go to:

```
Vector3 currPos = position - currentSegment.a;
Vector3 segmentDirection = currentSegment.b - currentSegment.a;
segmentDirection.Normalize();
```

7. Find the point in the segment using vector projection:

```
Vector3 pointInSegment = Vector3.Project(currPos,
segmentDirection);
```

8. Finally, `GetParam` returns the next position to go to along the path:

```
param = tempParam - Vector3.Distance(currentSegment.a,
currentSegment.b);
param += pointInSegment.magnitude;
return param;
```

9. Define the `GetPosition` function:

```
public Vector3 GetPosition(float param)
{
    // body
}
```

10. Given the current location along the path, we find the corresponding segment:

```
Vector3 position = Vector3.zero;
PathSegment currentSegment = null;
float tempParam = 0f;
foreach (PathSegment ps in segments)
{
    tempParam += Vector3.Distance(ps.a, ps.b);
    if (param <= tempParam)
    {
        currentSegment = ps;
        break;
    }
}
if (currentSegment == null)
    return Vector3.zero;
```

11. Finally, `GetPosition` converts the parameter as a spatial point and returns it:

```
Vector3 segmentDirection = currentSegment.b - currentSegment.a;
segmentDirection.Normalize();
tempParam -= Vector3.Distance(currentSegment.a, currentSegment.b);
tempParam = param - tempParam;
position = currentSegment.a + segmentDirection * tempParam;
return position;
```

12. Create the `PathFollower` behavior, which derives from `Seek` (remember to set the order of execution):

```
using UnityEngine;
using System.Collections;

public class PathFollower : Seek
{
    public Path path;
    public float pathOffset = 0.0f;
    float currentParam;
}
```

13. Implement the `Awake` function to set the target:

```
public override void Awake()
{
    base.Awake();
    target = new GameObject();
    currentParam = 0f;
}
```

14. The final step is to define the `GetSteering` function, which relies on the abstraction created by the `Path` class to set the target position and apply `Seek`:

```
public override Steering GetSteering()
{
    currentParam = path.GetParam(transform.position,
currentParam);
    float targetParam = currentParam + pathOffset;
    target.transform.position = path.GetPosition(targetParam);
    return base.GetSteering();
}
```

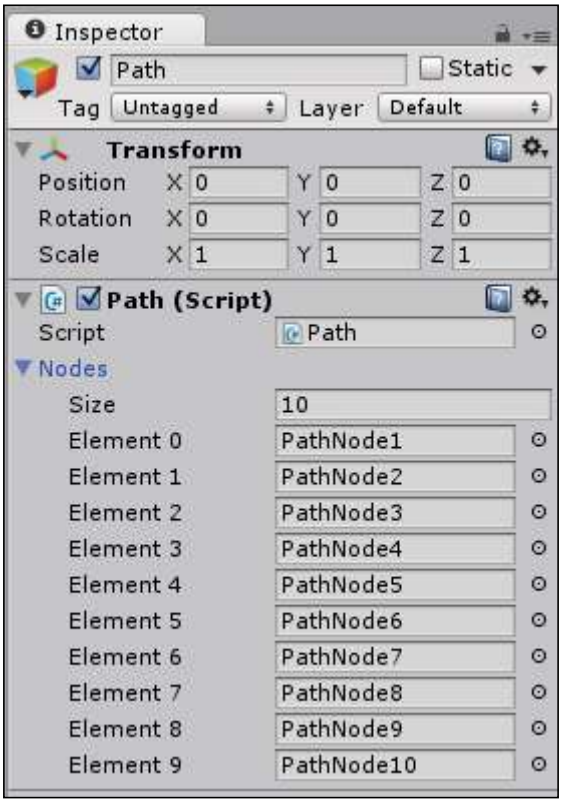
How it works...

We use the `Path` class in order to have a movement guideline. It is the cornerstone, because it relies on `GetParam` to map an offset point to follow in its internal guideline, and it also uses `GetPosition` to convert that referential point to a position in the three-dimensional space along the segments.

The path-following algorithm just makes use of the path's functions in order to get a new position, update the target, and apply the `Seek` behavior.

There's more...

It's important to take into account the order in which the nodes are linked in the Inspector for the path to work as expected. A practical way to achieve this is to manually name the nodes with a reference number.



An example of a path set up in the Inspector window

Also, we could define the `OnDrawGizmos` function in order to have a better visual reference of the path:

```
void OnDrawGizmos ()
{
    Vector3 direction;
    Color tmp = Gizmos.color;
    Gizmos.color = Color.magenta;//example color
    int i;
    for (i = 0; i < nodes.Count - 1; i++)
    {
```

```
            Vector3 src = nodes[i].transform.position;
            Vector3 dst = nodes[i+1].transform.position;
            direction = dst - src;
            Gizmos.DrawRay(src, direction);
        }
        Gizmos.color = tmp;
    }
```

Avoiding agents

In crowd-simulation games, it would be unnatural to see agents behaving entirely like particles in a physics-based system. The goal of this recipe is to create an agent capable of mimicking our peer-evasion movement.

Getting ready

We need to create a tag called **Agent** and assign it to those game objects that we would like to avoid, and we also need to have the **Agent** script component attached to them.

An example of how should look the Inspector of a dummy agent to avoid

How to do it...

This recipe will require the creation and handling of just one file:

1. Create the `AvoidAgent` behavior, which is composed of a collision avoidance radius and the list of agents to avoid:

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

public class AvoidAgent : AgentBehaviour
```

```
{
    public float collisionRadius = 0.4f;
    GameObject[] targets;
}
```

2. Implement the `Start` function in order to set the list of agents according to the tag we created earlier:

```
void Start ()
{
    targets = GameObject.FindGameObjectsWithTag("Agent");
}
```

3. Define the `GetSteering` function:

```
public override Steering GetSteering()
{
    // body
}
```

4. Add the following variables to compute distances and velocities from agents that are nearby:

```
Steering steering = new Steering();
float shortestTime = Mathf.Infinity;
GameObject firstTarget = null;
float firstMinSeparation = 0.0f;
float firstDistance = 0.0f;
Vector3 firstRelativePos = Vector3.zero;
Vector3 firstRelativeVel = Vector3.zero;
```

5. Find the closest agent that is prone to collision with the current one:

```
foreach (GameObject t in targets)
{
    Vector3 relativePos;
    Agent targetAgent = t.GetComponent<Agent>();
    relativePos = t.transform.position - transform.position;
    Vector3 relativeVel = targetAgent.velocity - agent.velocity;
    float relativeSpeed = relativeVel.magnitude;
    float timeToCollision = Vector3.Dot(relativePos, relativeVel);
    timeToCollision /= relativeSpeed * relativeSpeed * -1;
    float distance = relativePos.magnitude;
    float minSeparation = distance - relativeSpeed *
timeToCollision;
    if (minSeparation > 2 * collisionRadius)
        continue;
    if (timeToCollision > 0.0f && timeToCollision < shortestTime)
    {
```

```
                shortestTime = timeToCollision;
                firstTarget = t;
                firstMinSeparation = minSeparation;
                firstRelativePos = relativePos;
                firstRelativeVel = relativeVel;
            }
        }
```

6. If there is one, then get away:

```
if (firstTarget == null)
    return steering;
if (firstMinSeparation <= 0.0f || firstDistance < 2 *
collisionRadius)
    firstRelativePos = firstTarget.transform.position;
else
    firstRelativePos += firstRelativeVel * shortestTime;
firstRelativePos.Normalize();
steering.linear = -firstRelativePos * agent.maxAccel;
return steering;
```

How it works...

Given a list of agents, we take into consideration which one is closest, and if it is close enough, we make it so the agent tries to escape from the expected route of that first one according to its current velocity so that they don't collide.

There's more

This behavior works well when combined with other behaviors using blending techniques (some are included in this chapter); otherwise it's a starting point for your own collision avoidance algorithms.

Avoiding walls

This technique aims at imitating our capacity to evade walls by considering a safety margin, and creating repulsion from their surfaces when that gap is broken.

Getting ready

This technique uses the `RaycastHit` structure and the `Raycast` function from the physics engine, so it's recommended that you take a refresher on the docs in case you're a little rusty on the subject.

How to do it...

Thanks to our previous hard work, this recipe is a short one:

1. Create the `AvoidWall` behavior derived from `Seek`:

```
using UnityEngine;
using System.Collections;

public class AvoidWall : Seek
{
    // body
}
```

2. Include the member variables for defining the safety margin, and the length of the ray to cast:

```
public float avoidDistance;
public float lookAhead;
```

3. Define the `Awake` function to set up the target:

```
public override void Awake()
{
    base.Awake();
    target = new GameObject();
}
```

4. Define the `GetSteering` function for the following steps:

```
public override Steering GetSteering()
{
    // body
}
```

5. Declare and set the variable needed for ray casting:

```
Steering steering = new Steering();
Vector3 position = transform.position;
Vector3 rayVector = agent.velocity.normalized * lookAhead;
Vector3 direction = rayVector;
RaycastHit hit;
```

6. Cast the ray and make the proper calculations if a wall is hit:

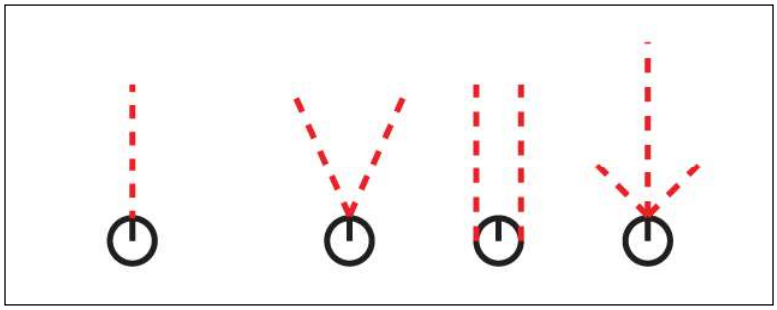
```
if (Physics.Raycast(position, direction, out hit, lookAhead))
{
    position = hit.point + hit.normal * avoidDistance;
    target.transform.position = position;
    steering = base.GetSteering();
}
return steering;
```

How it works...

We cast a ray in front of the agent; when the ray collides with a wall, the target object is placed in a new position taking into consideration its distance from the wall and the safety distance declared and delegating the steering calculations to the `Seek` behavior; this creates the illusion of the agent avoiding the wall.

There's more...

We could extend this behavior by adding more rays, like whiskers, in order to get better accuracy. Also, it is usually paired with other movement behaviors, such as `Pursue`, using blending.



The original ray cast and possible extensions for more precise wall avoidance

See also

For further information on the `RaycastHit` structure and the `Raycast` function, please refer to the official documentation available online at:

- `http://docs.unity3d.com/ScriptReference/RaycastHit.html`
- `http://docs.unity3d.com/ScriptReference/Physics.Raycast.html`

Blending behaviors by weight

Blending techniques allow you to add behaviors and mix them without creating new scripts every time you need a new type of hybrid agent.

This is one of the most powerful techniques in this chapter, and it's probably the most used behaviour-blending approach because of its power and the low cost of implementation.

Getting ready

We must add a new member variable to our `AgentBehaviour` class called `weight` and preferably assign a default value—in this case, `1.0f`. Besides this, we should refactor the `Update` function to incorporate `weight` as a parameter to the `Agent` class' `SetSteering` function. All in all, the new `AgentBehaviour` class should look something like this:

```
public class AgentBehaviour : MonoBehaviour
{
    public float weight = 1.0f;

    // ... the rest of the class

    public virtual void Update ()
    {
        agent.SetSteering(GetSteering(), weight);
    }
}
```

How to do it...

We just need to change the `SetSteering` agent function's signature and definition:

```
public void SetSteering (Steering steering, float weight)
{
    this.steering.linear += (weight * steering.linear);
    this.steering.angular += (weight * steering.angular);
}
```

How it works...

The weights are used to amplify the `steering` behavior result, and they're added to the main steering structure.

There's more...

The weights don't necessarily need to add up to `1.0f`. The `weight` parameter is a reference for defining the relevance that the `steering` behavior will have among the other ones.

See also

In this project, there is an example of avoiding walls, worked out using weighted blending.

Blending behaviors by priority

Sometimes, weighted blending is not enough because heavyweight behaviors dilute the contributions of the lightweights, but those behaviors need to play their part too. That's when priority-based blending comes into play, applying a cascading effect from high-priority to low-priority behaviors.

Getting ready

The approach is very similar to the one used in the previous recipe. We must add a new member variable to our `AgentBehaviour` class. We should also refactor the `Update` function to incorporate `priority` as a parameter to the `Agent` class' `SetSteering` function. The new `AgentBehaviour` class should look something like this:

```
public class AgentBehaviour : MonoBehaviour
{
    public int priority = 1;
    // ... everything else stays the same
    public virtual void Update ()
    {
        agent.SetSteering(GetSteering(), priority);
    }
}
```

How to do it...

Now, we need to make some changes to the `Agent` class:

1. Add a new namespace from the library:

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

2. Add the member variable for the minimum steering value to consider a group of behaviors:

   ```
   public float priorityThreshold = 0.2f;
   ```

3. Add the member variable for holding the group of behavior results:

   ```
   private Dictionary<int, List<Steering>> groups;
   ```

4. Initialize the variable in the `Start` function:

   ```
   groups = new Dictionary<int, List<Steering>>();
   ```

5. Modify the `LateUpdate` function so that the steering variable is set by calling `GetPrioritySteering`:

```
public virtual void LateUpdate ()
{
    //  funnelled steering through priorities
    steering = GetPrioritySteering();
    groups.Clear();
    // ... the rest of the computations stay the same
    steering = new Steering();
}
```

6. Modify the `SetSteering` function's signature and definition to store the steering values in their corresponding priority groups:

```
public void SetSteering (Steering steering, int priority)
{
    if (!groups.ContainsKey(priority))
    {
        groups.Add(priority, new List<Steering>());
    }
    groups[priority].Add(steering);
}
```

7. Finally, implement the `GetPrioritySteering` function to funnel the steering group:

```
private Steering GetPrioritySteering ()
{
    Steering steering = new Steering();
    float sqrThreshold = priorityThreshold * priorityThreshold;
    foreach (List<Steering> group in groups.Values)
    {
        steering = new Steering();
        foreach (Steering singleSteering in group)
        {
            steering.linear += singleSteering.linear;
            steering.angular += singleSteering.angular;
        }
        if (steering.linear.sqrMagnitude > sqrThreshold ||
                Mathf.Abs(steering.angular) > priorityThreshold)
        {
            return steering;
        }
}
```

How it works...

By creating priority groups, we blend behaviors that are common to one another, and the first group in which the steering value exceeds the threshold is selected. Otherwise, steering from the least-priority group is chosen.

There's more...

We could extend this approach by mixing it with weighted blending; in this way, we would have a more robust architecture by getting extra precision on the way the behaviors make an impact on the agent in every priority level:

```
foreach (Steering singleSteering in group)
{
    steering.linear += singleSteering.linear * weight;
    steering.angular += singleSteering.angular * weight;
}
```

See also

There is an example of avoiding walls using priority-based blending in this project.

Combining behaviors using a steering pipeline

This is a different approach to creating and blending behaviors that is based on goals. It tries to be a middle-ground between movement-blending and planning, without the implementation costs of the latter.

Getting ready

Using a steering pipeline slightly changes the train of thought used so far. We need to think in terms of goals, and constraints. That said, the heavy lifting rests on the base classes and the derived classes that will define the behaviors; we need to start by implementing them.

The following code is for the `Targeter` class. It can be seen as a goal-driven behavior:

```
using UnityEngine;
using System.Collections;

public class Targeter : MonoBehaviour
{
    public virtual Goal GetGoal()
```

```
        {
            return new Goal();
        }
    }
```

Now, we create the `Decomposer` class:

```
using UnityEngine;
using System.Collections;

public class Decomposer : MonoBehaviour
{
    public virtual Goal Decompose (Goal goal)
    {
        return goal;
    }
}
```

We also need a `Constraint` class:

```
using UnityEngine;
using System.Collections;

public class Constraint : MonoBehaviour
{
    public virtual bool WillViolate (Path path)
    {
        return true;
    }

    public virtual Goal Suggest (Path path) {
        return new Goal();
    }
}
```

And finally, an `Actuator` class:

```
using UnityEngine;
using System.Collections;

public class Actuator : MonoBehaviour
{
    public virtual Path GetPath (Goal goal)
    {
        return new Path();
```

```
        }

        public virtual Steering GetOutput (Path path, Goal goal)
        {
            return new Steering();
        }
    }
```

How to do it...

The `SteeringPipeline` class makes use of the previously implemented classes in order to work, maintaining the component-driven pipeline but with a different approach, as mentioned earlier:

1. Create the `SteeringPipeline` class deriving from the `Wander` behavior, including the array of components that it handles:

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

public class SteeringPipeline : Wander
{
    public int constraintSteps = 3;
    Targeter[] targeters;
    Decomposer[] decomposers;
    Constraint[] constraints;
    Actuator actuator;
}
```

2. Define the `Start` function to set the references to the attached components in the game object:

```
void Start ()
{
    targeters = GetComponents<Targeter>();
    decomposers = GetComponents<Decomposer>();
    constraints = GetComponents<Constraint>();
    actuator = GetComponent<Actuator>();
}
```

3. Define the `GetSteering` function to work out the goal and the steering value to reach it:

```
public override Steering GetSteering()
{
    Goal goal = new Goal();
```

```
        foreach (Targeter targeter in targeters)
            goal.UpdateChannels(targeter.GetGoal());
        foreach (Decomposer decomposer in decomposers)
            goal = decomposer.Decompose(goal);
        for (int i = 0; i < constraintSteps; i++)
        {
            Path path = actuator.GetPath(goal);
            foreach (Constraint constraint in constraints)
            {
                if (constraint.WillViolate(path))
                {
                    goal = constraint.Suggest(path);
                    break;
                }
                return actuator.GetOutput(path, goal);
            }
        }
        return base.GetSteering();
    }
```

How it works...

This code takes a composite goal generated by *targeters*, creates sub-goals using *decomposers*, and evaluates them to comply with defined *constraints* before "blending" them into a final goal in order to produce a steering result. If everything fails (the constraints are not satisfied), it uses the default `Wander` behavior.

There's more...

You should try to implement some of the behavior recipes in terms of targeters, decomposers, constraints, and an actuator. Take into account that there's room for one actuator only, and it's the one responsible for making the final decision. A good example is as follows:

- **Targeters**: seeking, arriving, facing, and matching velocity
- **Decomposers**: path-finding algorithms
- **Constraints**: avoiding walls/agents

See also

For more theoretical insights, refer to Ian Millington's book, *Artificial Intelligence for Games*.

Shooting a projectile

This is the stepping stone for scenarios where we want to have control over gravity-reliant objects, such as balls and grenades, so we can then predict the projectile's landing spot, or be able to effectively shoot a projectile at a given target.

Getting ready

This recipe differs slightly as it doesn't rely on the base `AgentBehaviour` class.

How to do it...

1. Create the `Projectile` class along with its member variables to handle the physics:

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour
{
    private bool set = false;
    private Vector3 firePos;
    private Vector3 direction;
    private float speed;
    private float timeElapsed;
}
```

2. Define the `Update` function:

```
void Update ()
{
    if (!set)
        return;
    timeElapsed += Time.deltaTime;
    transform.position = firePos + direction * speed *
timeElapsed;
    transform.position += Physics.gravity * (timeElapsed *
timeElapsed) / 2.0f;
    // extra validation for cleaning the scene
    if (transform.position.y < -1.0f)
        Destroy(this.gameObject);// or set = false; and hide it
}
```

3. Finally, implement the `Set` function in order to fire the game object (for example, calling it after it is instantiated in the scene):

```
public void Set (Vector3 firePos, Vector3 direction, float speed)
{
    this.firePos = firePos;
    this.direction = direction.normalized;
    this.speed = speed;
    transform.position = firePos;
    set = true;
}
```

How it works...

This behavior uses high-school physics in order to generate the parabolic movement.

There's more...

We could also take another approach: implementing public properties in the script or declaring member variables as public and, instead of calling the `Set` function, having the script disabled by default in the prefab and enabling it after all the properties have been set. That way, we could easily apply the object pool pattern.

See also

For further information on the object pool pattern, please refer to the following Wikipedia article and an official Unity Technologies video tutorial available online at the following addresses:

- `http://en.wikipedia.org/wiki/Object_pool_pattern`
- `http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/object-pooling`

Predicting a projectile's landing spot

After a projectile is shot, some agents need to make a run for it, if we're talking about a grenade, or look at it when we're developing a sports game. In either case, it's important to predict the projectile's landing spot in order to make decisions:

Getting ready

Before we get into predicting the landing position, it's important to know the time left before it hits the ground (or reaches a certain position). Thus, instead of creating new behaviors, we need to update the `Projectile` class.

How to do it...

1. First, we need to add the `GetLandingTime` function to compute the landing time:

```
public float GetLandingTime (float height = 0.0f)
{
    Vector3 position = transform.position;
    float time = 0.0f;
    float valueInt = (direction.y * direction.y) * (speed *
speed);
    valueInt = valueInt - (Physics.gravity.y * 2 * (position.y -
height));
    valueInt = Mathf.Sqrt(valueInt);
    float valueAdd = (-direction.y) * speed;
    float valueSub = (-direction.y) * speed;
    valueAdd = (valueAdd + valueInt) / Physics.gravity.y;
    valueSub = (valueSub - valueInt) / Physics.gravity.y;
    if (float.IsNaN(valueAdd) && !float.IsNaN(valueSub))
        return valueSub;
    else if (!float.IsNaN(valueAdd) && float.IsNaN(valueSub))
        return valueAdd;
    else if (float.IsNaN(valueAdd) && float.IsNaN(valueSub))
        return -1.0f;
    time = Mathf.Max(valueAdd, valueSub);
    return time;
}
```

2. Now, we add the `GetLandingPos` function to predict the landing spot:

```
public Vector3 GetLandingPos (float height = 0.0f)
{
    Vector3 landingPos = Vector3.zero;
    float time = GetLandingTime();
    if (time < 0.0f)
        return landingPos;
    landingPos.y = height;
    landingPos.x = firePos.x + direction.x * speed * time;
    landingPos.z = firePos.z + direction.z * speed * time;
    return landingPos;
}
```

How it works...

First, we solve the equation from the previous recipe for a fixed height and, given the projectile's current position and speed, we are able to get the time at which the projectile will reach the given height.

There's more...

Take into account the `NaN` validation. It's placed that way because there may be two, one, or no solution to the equation. Furthermore, when the landing time is less than zero, it means the projectile won't be able to reach the target height.

Targeting a projectile

Just like it's important to predict a projectile's landing point, it's also important to develop intelligent agents capable of aiming projectiles. It wouldn't be fun if our rugby-player agents aren't capable of passing the ball.

Getting ready

Just like the previous recipe, we only need to expand the `Projectile` class.

How to do it...

Thanks to our previous hard work, this recipe is a real piece of cake:

1. Create the `GetFireDirection` function:

```
public static Vector3 GetFireDirection (Vector3 startPos, Vector3
endPos, float speed)
{
    // body
}
```

2. Solve the corresponding quadratic equation:

```
Vector3 direction = Vector3.zero;
Vector3 delta = endPos - startPos;
float a = Vector3.Dot(Physics.gravity, Physics.gravity);
float b = -4 * (Vector3.Dot(Physics.gravity, delta) + speed *
speed);
float c = 4 * Vector3.Dot(delta, delta);
if (4 * a * c > b * b)
    return direction;
```

```
float time0 = Mathf.Sqrt((-b + Mathf.Sqrt(b * b - 4 * a * c)) /
(2*a));
float time1 = Mathf.Sqrt((-b - Mathf.Sqrt(b * b - 4 * a * c)) /
(2*a));
```

3. If shooting the projectile is feasible given the parameters, return a non-zero direction vector:

```
float time;
if (time0 < 0.0f)
{
    if (time1 < 0)
        return direction;
    time = time1;
}
else
{
    if (time1 < 0)
        time = time0;
    else
        time = Mathf.Min(time0, time1);
}
direction = 2 * delta - Physics.gravity * (time * time);
direction = direction / (2 * speed * time);
return direction;
```

How it works...

Given a fixed speed, we solve the corresponding quadratic equation in order to obtain the desired direction (when at least one time value is available), which doesn't need to be normalized because we already normalized the vector while setting up the projectile.

There's more...

Take into account that we are returning a *blank* direction when time is negative; it means that the speed is not sufficient. One way to overcome this is to define a function that tests different speeds and then shoots the projectile.

Another relevant improvement is to add an extra parameter of the type `bool` for those cases when we have two valid times (which means two possible arcs), and we need to shoot over an obstacle such as a wall:

```
if (isWall)
    time = Mathf.Max(time0, time1);
else
    time = Mathf.Min(time0, time1);
```

Creating a jump system

Imagine that we're developing a cool action game where the player is capable of escaping using cliffs and rooftops. In that case, the enemies need to be able to chase the player and be smart enough to discern whether to take the jump and gauge how to do it.

Getting ready

We need to create a basic matching-velocity algorithm and the notion of jump pads and landing pads in order to emulate a velocity math so that we can reach them.

Also, the agents must have the tag `Agent`, the main object must have a `Collider` component marked as trigger. Depending on your game, the agent or the pads will need the `Rigidbody` component attached.

The following is the code for the `VelocityMatch` behavior:

```
using UnityEngine;
using System.Collections;

public class VelocityMatch : AgentBehaviour {

    public float timeToTarget = 0.1f;

    public override Steering GetSteering()
    {
        Steering steering = new Steering();
        steering.linear = target.GetComponent<Agent>().velocity -
agent.velocity;
        steering.linear /= timeToTarget;
        if (steering.linear.magnitude > agent.maxAccel)
            steering.linear = steering.linear.normalized * agent.
maxAccel;

        steering.angular = 0.0f;
        return steering;
    }
}
```

Also, it's important to create a data type called `JumpPoint`:

```
using UnityEngine;

public class JumpPoint
{
```

```
        public Vector3 jumpLocation;
        public Vector3 landingLocation;
        //The change in position from jump to landing
        public Vector3 deltaPosition;

        public JumpPoint ()
            : this (Vector3.zero, Vector3.zero)
        {
        }

        public JumpPoint(Vector3 a, Vector3 b)
        {
            this.jumpLocation = a;
            this.landingLocation = b;
            this.deltaPosition = this.landingLocation - this.jumpLocation;
        }
    }
```

How to do it...

We will learn how to implement the `Jump` behavior:

1. Create the `Jump` class deriving from `VelocityMatch`, with its member variables:

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

public class Jump : VelocityMatch
{
    public JumpPoint jumpPoint;
    //Keeps track of whether the jump is achievable
    bool canAchieve = false;
    //Holds the maximum vertical jump velocity
    public float maxYVelocity;
    public Vector3 gravity = new Vector3(0, -9.8f, 0);
    private Projectile projectile;
    private List<AgentBehaviour> behaviours;

    // next steps
}
```

2. Implement the `Isolate` method. It disables all the agent behaviors, except for the `Jump` component:

```
public void Isolate(bool state)
{
    foreach (AgentBehaviour b in behaviours)
        b.enabled = !state;
    this.enabled = state;
}
```

3. Define the function for calling the jumping effect, using the projectile behavior we learned before:

```
public void DoJump()
{
    projectile.enabled = true;
    Vector3 direction;
    direction = Projectile.GetFireDirection(jumpPoint.
jumpLocation, jumpPoint.landingLocation, agent.maxSpeed);
    projectile.Set(jumpPoint.jumpLocation, direction, agent.
maxSpeed, false);
}
```

4. Implement the member function for setting up the behaviors' target for matching its velocity:

```
protected void CalculateTarget()
{
    target = new GameObject();
    target.AddComponent<Agent>();

    //Calculate the first jump time
    float sqrtTerm = Mathf.Sqrt(2f * gravity.y * jumpPoint.
deltaPosition.y + maxYVelocity * agent.maxSpeed);
    float time = (maxYVelocity - sqrtTerm) / gravity.y;

    //Check if we can use it, otherwise try the other time
    if (!CheckJumpTime(time))
    {
        time = (maxYVelocity + sqrtTerm) / gravity.y;
    }
}
```

5. Implement the function for computing the time:

```
//Private helper method for the CalculateTarget function
private bool CheckJumpTime(float time)
{
    //Calculate the planar speed
    float vx = jumpPoint.deltaPosition.x / time;
    float vz = jumpPoint.deltaPosition.z / time;
    float speedSq = vx * vx + vz * vz;

    //Check it to see if we have a valid solution
    if (speedSq < agent.maxSpeed * agent.maxSpeed)
    {
        target.GetComponent<Agent>().velocity = new Vector3(vx,
0f, vz);
        canAchieve = true;
        return true;
    }
    return false;
}
```

6. Override the `Awake` member function. The most important thing here is caching the references to other attached behaviors, so `Isolate` function makes sense:

```
public override void Awake()
{
    base.Awake();
    this.enabled = false;
    projectile = gameObject.AddComponent<Projectile>();
    behaviours = new List<AgentBehaviour>();
    AgentBehaviour[] abs;
    abs = gameObject.GetComponents<AgentBehaviour>();
    foreach (AgentBehaviour b in abs)
    {
        if (b == this)
            continue;
        behaviours.Add(b);
    }
}
```

7. Override the `GetSteering` member function:

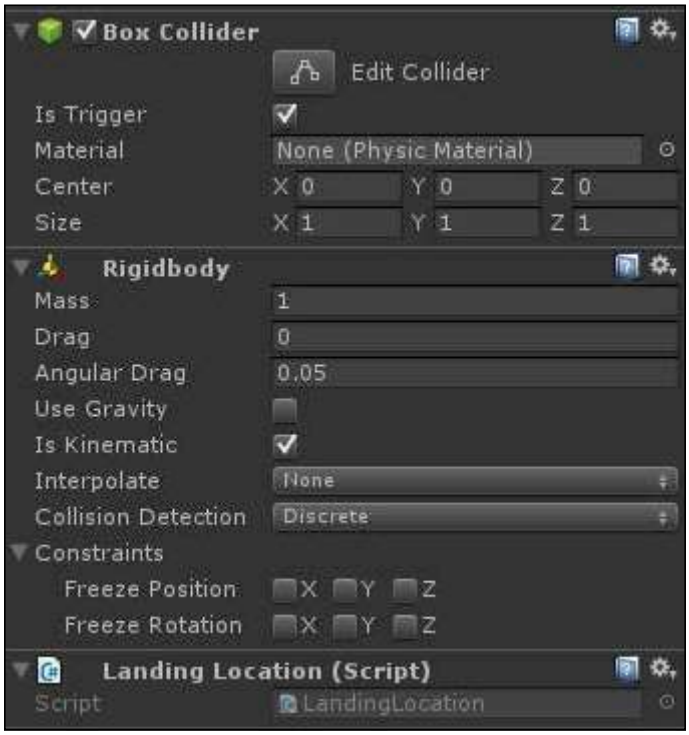
```
public override Steering GetSteering()
{
    Steering steering = new Steering();

    // Check if we have a trajectory, and create one if not.
    if (jumpPoint != null && target == null)
    {
        CalculateTarget();
    }
    //Check if the trajectory is zero. If not, we have no
acceleration.
    if (!canAchieve)
    {
        return steering;
    }

    //Check if we've hit the jump point
    if (Mathf.Approximately((transform.position - target.
transform.position).magnitude, 0f) &&
        Mathf.Approximately((agent.velocity - target.
GetComponent<Agent>().velocity).magnitude, 0f))
    {
        DoJump();
        return steering;
    }
    return base.GetSteering();
}
```

How it works...

The algorithm takes into account the agent's velocity and calculates whether it can reach the landing pad or not. The behavior's target is the one responsible for executing the jump, and if it judges that the agent can, it tries to match the targets' vertical velocity while seeking the landing pad's position.

There is more

We will need a jump pad and a landing pad in order to have a complete jumping system. Both the jump and landing pads need the `Collider` component marked as trigger. Also, as stated before, they will probably need to have a `Rigidbody` component, too, as seen in the image below.



The pads we will need a `MonoBehaviour` script attached as explained below.

The following code is to be attached to the jump pad:

```
using UnityEngine;

public class JumpLocation : MonoBehaviour
{
    public LandingLocation landingLocation;

    public void OnTriggerEnter(Collider other)
    {
        if (!other.gameObject.CompareTag("Agent"))
            return;
```

```
            Agent agent = other.GetComponent<Agent>();
            Jump jump = other.GetComponent<Jump>();
            if (agent == null || jump == null)
                return;
            Vector3 originPos = transform.position;
            Vector3 targetPos = landingLocation.transform.position;
            jump.Isolate(true);
            jump.jumpPoint = new JumpPoint(originPos, targetPos);
            jump.DoJump();
        }
    }
```

The following code is to be attached to the landing pad:

```
    using UnityEngine;

    public class LandingLocation : MonoBehaviour
    {
        public void OnTriggerEnter(Collider other)
        {
            if (!other.gameObject.CompareTag("Agent"))
                return;
            Agent agent = other.GetComponent<Agent>();
            Jump jump = other.GetComponent<Jump>();
            if (agent == null || jump == null)
                return;
            jump.Isolate(false);
            jump.jumpPoint = null;
        }
    }
```

See Also

The *Shooting a projectile* recipe