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.

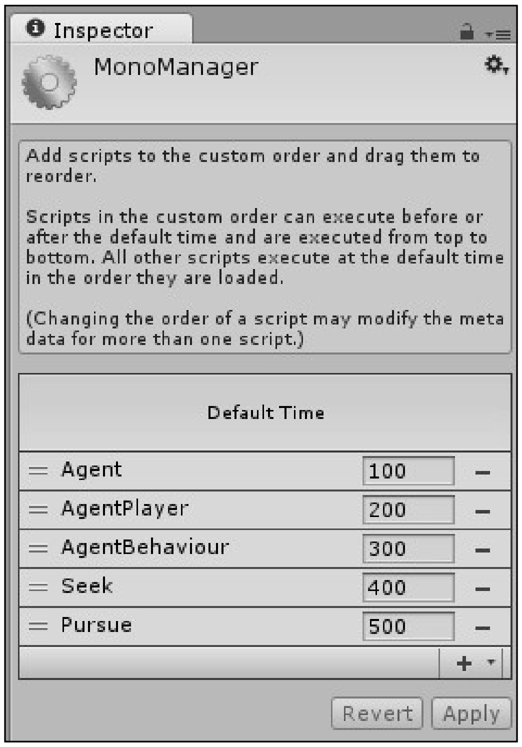
Getting ready

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



Add scripts to the custom order and drag them to reorder.

Scripts in the custom order can execute before or after the default time and are executed from top to bottom. All other scripts execute at the default time in the order they are loaded.

(Changing the order of a script may modify the meta data for more than one script.)

Default Time

Agent	100
AgentPlayer	200
AgentBehaviour	300
Seek	400
Pursue	500

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`:
Locate the script Steering.cs in Project panel> scripts>Ch01Behaviours

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: Project panel>Scripts>Ch01Behaviours>AgentBehaviour

```
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:

Project panel>Scripts>Ch01Behaviours>Agent

```
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();
}
```

Attach Agent.cs to the game objects Pursuit, Seek, Flee and Evade

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;
        }
    }
```

Attach Seek.cs to game object "Seek"

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;
    }
}
```

Attach Flee.cs to game object "Flee"

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();
}
```

Attach Pursue.cs to game object Pursue

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
}
```