8
Attacking

Fighting is an important part of a game's AI. For many games, fighting with the player is the main game mechanic and the most noticeable AI in the game. We will discuss the common methods for attack AI, how to make an enemy character chase and attack the player, and then have the enemy character take cover and hide from the player.

In this chapter, you will learn about:

- Designing attack AI in RAIN 2.1.4
- Creating basic chase attack AI
- Creating and covering attack AI
- Having AI attack in groups

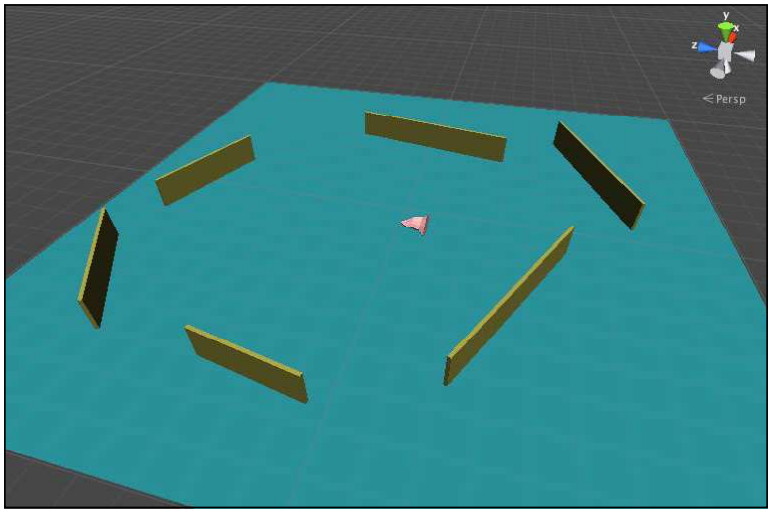
An overview of attack AI

Attack AI is a large and much studied subject. When you start dealing with things such as different attack moves based on different player actions or having enemies coordinate attacks, the AI can become quite complex. However, designing good AI that attacks is the same as designing for other AI scenarios we have looked at so far in this book. First, we need sensors for our AI characters to perceive game events and to create aspects in the game world, tagging what they can sense. Then, we define behavior trees for the characters, directing them to change actions based on sensor response or other game states, such as running out of ammo. Defining different behaviors is the main part of setting up attack AI.

We'll look at two foundational AI attack behaviors in our demos in this chapter. The first will use multiple sensors on the AI to determine when to chase and when to stop and attack. The second behavior we will look at is the duck and cover type, where the enemy attacker will retreat to a safe position after attacking, and this is based on set navigation points. These are both best illustrated through demos, so let's start one now.

The attack demo

Like our previous demos, we will start with a basic scene with a ground and walls. The demos here will involve an enemy ship attacking a player, so add a ship to the scene, name it `player`, and add simple controls to move the ship around. Also, tint the color of the material to make the player ship stand out from the enemy ship that we'll add in a moment. Of course, the player ship isn't an AI, so it doesn't need a RAIN AIRig, but it does need to have a RAIN Entity component. With the player selected, go to **RAIN | Create Entity**. Next, it needs a visual aspect for the AI enemies to see it; from the **Add Aspect** dropdown, select **Visual Aspect** and rename the aspect to `player`. This provides a base for our attack demo. This is how the RAIN attack demos will look with a player ship:



Next, we need an enemy for the attack. The enemies will also be ship models, and as we are focusing on just the AI, we won't worry about the actual game mechanics of attacking, such as having the ship fire projectiles at the player, then having the player respond to being hit, and so on. Usually, these kind of attack AI states involve playing different animations for the AI, and we will explore these more in *Chapter 10*, *Animation and AI*; for now, we just need a simple visualization to illustrate the attack. To visualize, we'll store a Boolean variable in RAIN's working memory flagging if the enemy is attacking and if so, start blinking.

To set this up, add a ship to the scene and add an AIRig to it by going to **RAIN | Create AI**. To add an attack flag, select the RAIN **Memory** tab on the ship AI (the light bulb icon) and from **Add Variable**, select **bool**. Rename the variable to `isAttacking` and leave it to the default value of false. The memory with the `isAttacking` variable set should look like the following screenshot:

To use this variable, create a new script called `Enemy.cs` and add it to the enemy ship. Change the code to the following:

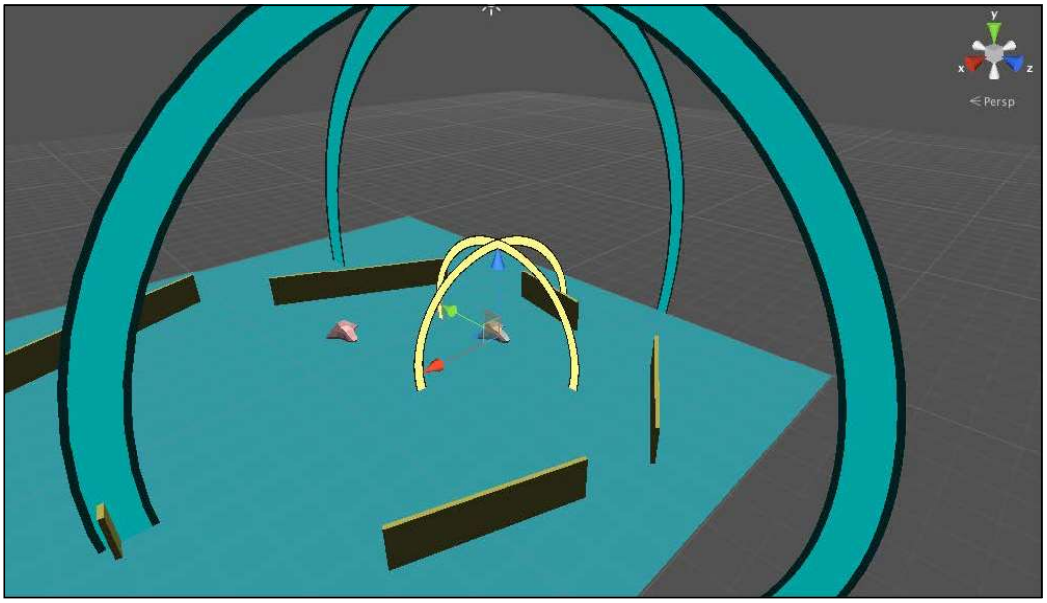
```
using UnityEngine;
using System.Collections;
using RAIN.Core;

public class Enemy : MonoBehaviour {

    float blinkTime = 0.0f;
    const float blinkLength = 0.1f;

    AIRig aiRig = null;

    void Start () {
        aiRig = GetComponentInChildren<AIRig>();
    }

    void Update () {

        bool isAttacking = aiRig.AI.WorkingMemory.GetItem<bool>("isAttacking");

        if(!isAttacking) {
            gameObject.renderer.material.color = Color.white;
            return;
        }

        blinkTime += Time.deltaTime;
        if(blinkTime > blinkLength) {
            blinkTime = -blinkLength;
        }

        gameObject.renderer.material.color = blinkTime < 0.0f ? Color.green : Color.white;
    }
}

```

Here, we store the AIRig for the entity by retrieving it at the start. Then, we get the `isAttacking` variable from the working memory, and if the enemy is attacking, the ship starts blinking green. If it's not attacking, the ship stays its default color, which is white. Create a new prefab in Unity named `Enemy` and drag the ship into it. Now we have enemies that can start attacking the player, and we can start setting up our AI.

The chase and attack demo

In the first demo, we will build an enemy ship that senses for the player, and if it sees the player, it starts moving toward it and then attacks it. A simple version of this would be to have the enemy wander with a visual sensor to detect the player, and if it sees the player, the enemy will move toward it and attack it. This would work but it really wouldn't be any different from the demo from *Chapter 7*, *Adaptation*, where the ship had to search for and collect gold. To make it a little different, we'll use a two-sensor approach. We will have one larger sensor on the enemy that detects the player, and if the enemy senses the player aspect, it will start chasing the player. Then, there is a second smaller sensor that attacks the player, that is, if it senses the player, then the enemy stops chasing and it instead attacks. This gives the effect of chasing the player but when the enemy gets closer, it stops and starts attacking, instead of just chasing and attacking at the same time.

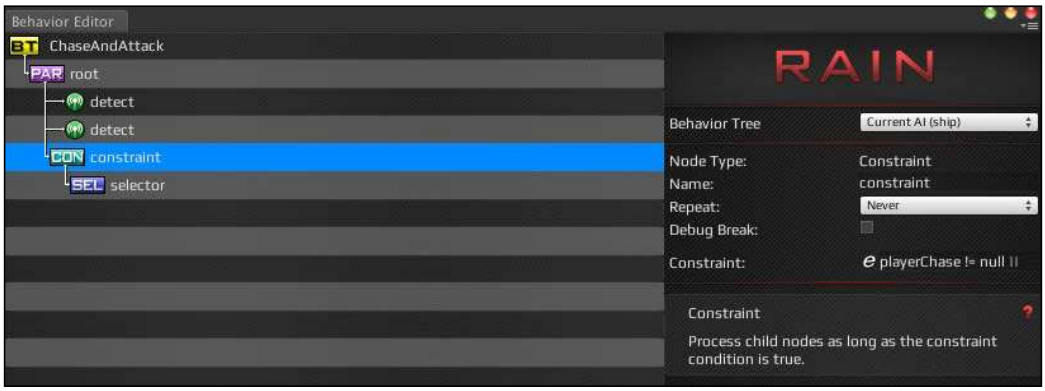
To begin setting these up, go to the **Perception** tab on the enemy AI rig (the little eye icon tab) and add a visual sensor called `ChaseSensor`. This should be pretty large and cover most of the scene. Then, add a second visual sensor and call it `AttackSensor`. Make this one about a third the size of `ChaseSensor`. The setup should look something like the following screenshot:



The preceding screenshot shows our enemy setup with two sensors: one will be used to chase and the smaller one will be used to attack.

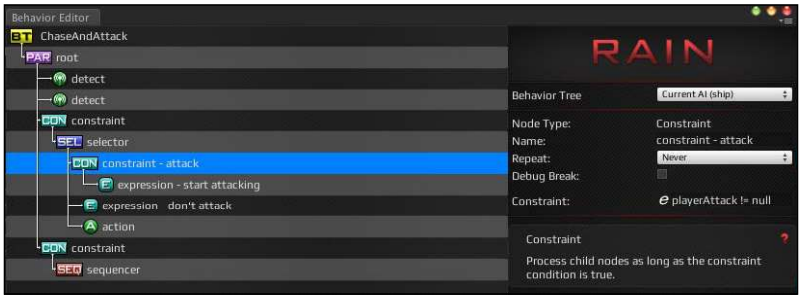
Using multiple sensors of the same type is a powerful tool to have AI characters react to things differently based on how far away they are.

Now we have our sensor, we can work on the behavior tree for the enemy. Select the **Mind** tab of the enemy AI rig and open the behavior editor. Create a new behavior tree called **ChaseAndAttack**. The enemy will detect and chase or attack the player at the same time, so right-click on the **root** node and change its type to **Parallel**. Then, add two detect nodes, one for the chase sensor and one for the attack sensor. For the chase detect node, set **Sensor** to **"ChaseSensor"**, **Aspect** to **"player"**, and the form variable to `playerChase` (remember to watch out for the quotes). For the attack sensor, set **Sensor** to **"AttackSensor"**, **Aspect** also to **"player"**, and the form variable to `playerAttack`. Then, add a constraint node, which will go off if either of the sensors has found something, so set its constraint to **playerChase != null || playerAttack != null**. Then, add a **selector** node under the **constraint** node that will handle the attack and chase logic. The multiple visual sensors behavior tree should look like the following screenshot:



Remember the **selector** node will continue to run its children until one succeeds, so first we will check for attack. If `playerAttack` has a value (that is, it is not null), we will set `isAttacking` to true, and if not, set it to false. Add a **constraint** node under the **selector** that checks for attacks and set its constraint to **playerAttack != null**. As the `playerAttack` variable is not null, add an **expression** node to start attacking with an expression value of `isAttacking`, which is equal to true.

Then, if `playerAttack` is null, we want the attack to stop, so add another expression with `isAttacking`, which is equal to false. The attack setup on our enemy behavior tree should look like the following screenshot:



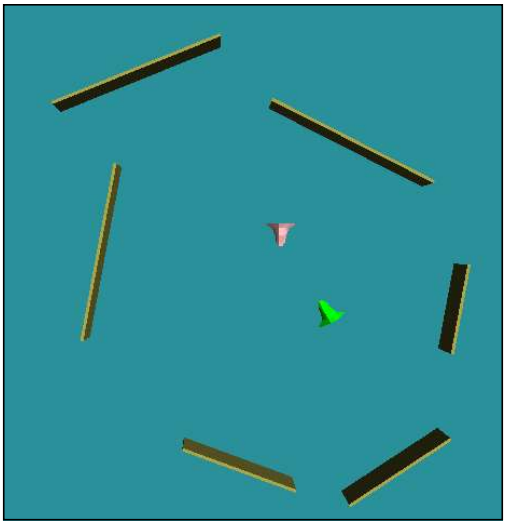
If you run the demo now, when the player gets sensed by `AttackSensor`, the enemy will start attacking and stop when the player is out of range.

Our attack behavior here is very simple; we are just flashing the ship. However, there's no reason why you couldn't add additional attack nodes and states to make the behavior more realistic.

Finally, we need to have the enemy chase the player if it is not attacking, so add a custom node called `Chase` to the bottom of the selector. Create a `Chase` script for it and set the `Chase` code to the following:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using RAIN.Core;
using RAIN.Action;

[RAINAction]
public class Chase : RAINAction
{
    public GameObject player;

    public override void Start(AI ai)
    {
        base.Start(ai);

        player = GameObject.Find("player");
    }

    public override ActionResult Execute(AI ai)
    {
        if(player == null) {
            return ActionResult.FAILURE;
        }

        ai.Motor.MoveTo(player.transform.position);

        return ActionResult.RUNNING;
    }

    public override void Stop(AI ai)
    {
        base.Stop(ai);
    }
}
```

This code first finds the `player` GameObject and then just moves to the player's position. This is unlike the code in the demos in *Chapter 7*, *Adaptation*, where we did a check to stop moving if the character gets very close to the moving target. When the character gets close to the target, it will stop moving and start attacking, so we don't need checks. If you run the demo now, the enemy will chase the player and start attacking:



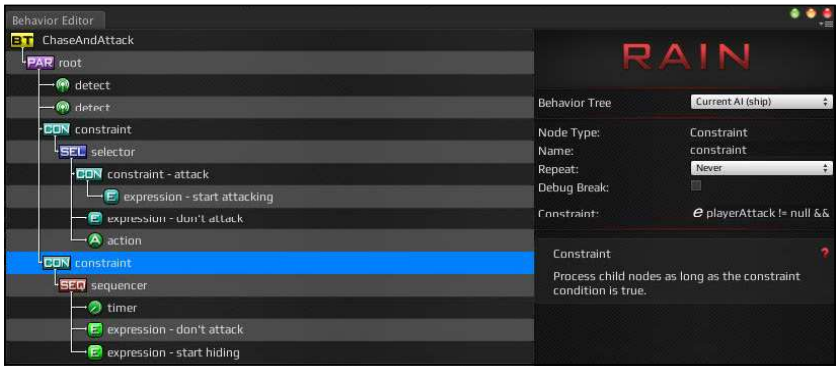
The preceding screenshot shows how an enemy attacking the player will look at the end.

Creating cover AI

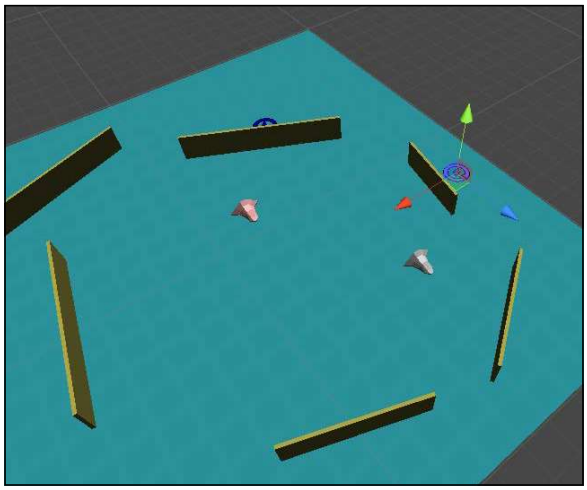
Our AI enemy will just keep attacking the player as long as it is close enough to the ship. However, this isn't very realistic; we'd like the enemy ship to attack for a little bit but then duck and head for cover. We could have this hiding behavior be based on a response to the player fighting back, but for this demo, we will make it a constant value of 5 seconds; after attacking the player for 5 seconds, it will hide.

To set this up, first we'll add an `isHidingbool` variable to our behavior tree that is set to true after 5 seconds of attacking. Create a new **constraint** node under the **root** parallel node with the **playerAttack != null && isHiding == false** expression. This node's children start when `playerAttack` is valid and we are not already hiding from the player. Add a **sequencer** node under this constraint so it will go through all of its children. The first child needs to be a new timer node with the **Seconds** value of **5** and **Returns** set to **Success**. Next, copy the **don't attack** node and add it below the timer so that the enemy won't attack as it's running to hide.

Then, add another **expression** node to set `isHiding` to true; its expression value should be **isHiding = true**. The behavior tree should be like the following screenshot:



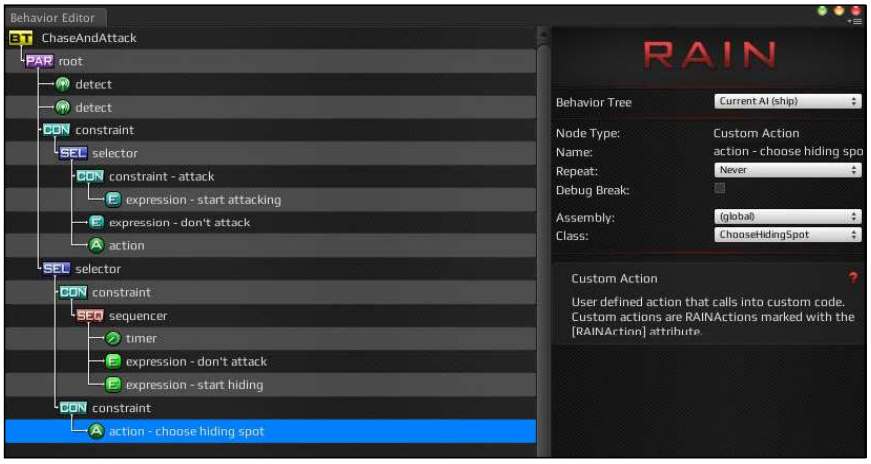
Finally, we need to have hiding spots to go to. These are often predefined; in shooting games, hiding spots are defined based on paths the player is expected to take. To do this, create a few navigation targets by going to **RAIN | Create Navigation Target** and add them to some good cover spots for the enemy. Here's how they can be arranged:



This is how we set up navigation targets for hiding spots.

Next, we need to have the AI choose a point to take cover. Lastly, we need to select and move to a hiding spot. To organize the tree better, add a **selector** node above the hiding **constraint** node. Then, add another **constraint** node below the **selector** node and create a custom action node with a new `ChooseHidingSpot` class.

The tree should look like the following screenshot:



When creating larger trees, giving the nodes descriptive names helps keep the tree organized and easy to understand.

The following is the code for our action to choose a hiding spot:

```
[RAINAction]
public class ChooseHidingSpot : RAINAction
{
    Vector3 hideTarget;

    public override void Start(AI ai)
    {
        NavigationTargetRig[] coverPoints = GameObject.FindObjectsOfType(typeof(NavigationTargetRig)) as NavigationTargetRig[];

        if(coverPoints.Length == 0)
        {
            return;
        }

        float length = float.MaxValue;
        Vector3 target = Vector3.zero;
        foreach(NavigationTargetRig obj in coverPoints)
        {
            if(Vector3.Distance(ai.Body.transform.position, obj.Target.PositionOffset) < length)
            {
                target = obj.Target.PositionOffset;
            }
        }

        hideTarget = target;

        base.Start(ai);
    }

    public override ActionResult Execute(AI ai)
    {
        if(Vector3.Distance(hideTarget, ai.Body.transform.position) < 1.0f) {
            return ActionResult.SUCCESS;
        }
        ai.Motor.MoveTo(hideTarget);

        return ActionResult.RUNNING;
    }

    public override void Stop(AI ai)
    {
        base.Stop(ai);
    }
}
```

Here, when we start the action, we find all the `NavigationTargetRig` objects and store them in the `coverPoints` array. Then, we go through each target and find the one closest to the enemy. Once we have the closest target, we store it in `hideTarget` and start moving to it.

As an addition to this, we can have the enemy start attacking again after hiding. Add the following line right before `ActionResult.SUCCESS` is returned:

```
ai.WorkingMemory.SetItem("isHiding", false);
```

This just updates the memory to set the hiding value to false and the attack will restart. This is a simple extension and the attack can be easily extended to better attack behaviors.

Group attacks

We spent *Chapter 4*, *Crowd Chaos*, and *Chapter 5*, *Crowd Control*, looking at group behaviors, and we won't go through a full demo of attacking in groups here, but we should discuss a few main points. With the demo in this chapter, we can add more ships and they will attack in a fairly believable manner. However, there are ways to make it better by considering other enemy positions.

When the enemy ships choose a cover position, a simple method for a group is to track each position if an enemy is already there. Then, when selecting a cover position, each enemy won't go to one that is occupied, making the enemies more diverse in their attacks.

Similarly, when attacking the player, instead of just going as close as possible, the attack pattern can be coordinated. Instead of just going directly to the player, a set of points can be defined radially around the player, so enemies surround and attack it. The key to these group behaviors is enemies taking into account the behavior of other enemies.

Summary

In this chapter, we looked at attack AI, focusing on how to have enemies chase and attack a player and then how to evade. These are basic attack behaviors and can be extended to more complex and game-specific behaviors, and we discussed how to do this when creating groups of enemies.

In the next chapter, we will look at another special AI case, which is driving and cars. However, instead of using a general-purpose AI system such as RAIN or React AI, we will use an AI plugin specifically designed for cars that takes into account physics to create realistic driving.