7
Adaptation

Having good AI for our characters is more than just giving them simple tasks to perform; we'd like to have our characters realistically react to the game environment. Game events such as seeing new objects appear or having a bomb go off in a scene should cause a reaction in the AI. Having the AI adapt to the environment is a huge topic, but we will focus on the basic ways to have AI adapt to the environment. In this chapter, we will look at taking AI skills we learned in previous chapters and combining them to create AI characters that adapt to the game environment in a realistic way, changing their tasks based on game events.

In this chapter, you will be:

- Creating AI characters that react and adapt to multiple game events
- Setting up more complex AI characters in RAIN
- Getting to know the importance of creating larger AI scenes with REACT AI

An overview

In previous chapters, we looked at how to do different specific AI tasks. We learned how to make characters patrol a path, have them wander an environment, change state with behavior trees, and sense objects in the game environment. These are all important, but it's more important to understand how we can combine these different elements to make AI that works well in a large game environment. We will need characters that can navigate an environment to perform tasks but then change based on game events that occur. To do this, the game needs to be designed at a high level, defining what the different AI character's main goals and actions are. These high-level goals are things such as wanting an enemy to patrol an area until it sees the player and then start to chase and attack him. From there, the different aspects of sensing need to be designed for the level, deciding what objects need to be tagged, so they can be used by the AI system. The characters then need sensors defined for the AI characters and high-level goals can be created using existing nodes and custom actions.

One of the ways we can define our character's adaptive behavior with RAIN is using RAIN's motor system. We have been using the motor system with the move node but not directly. The motor system controls moving the character, and it is available through the motion panel in RAIN, the icon with two feet. This is how the motion panel in RAIN looks:

RAIN supports three different kinds of motors:

- A basic motor, which we will use for most cases
- A character controller that uses the standard Unity character controller for movement
- A Mecanim controller (we will discuss Mecanim with RAIN in *Chapter 10, Animation and AI*)

The movement is target based: you give the motor a target position to go to and use the motor to get there.

Unity's character controller is very popular, but if you want to use it with RAIN, stick with RAIN's character controller. There are some known issues mixing Unity's basic character controller and RAIN 2.1.4. These should be fixed in a future version.

The fields for motors are pretty straightforward:

- **Speed / Rotation Speed**: This specifies how fast the character should move and rotate.
- **Close Enough Distance / Close Enough Angle**: This specifies how close the character needs to move to a target.
- **Face Before Move Angle**: This specifies how much of an angle the character needs to be facing its target before moving. This prevents weird movements with very close targets.
- **Step Up Height**: This specifies how much the character can step up; this is used to customize behavior for things such as steep terrain and staircases. We will discuss step up heights more in *Chapter 11*, *Advanced NavMesh Generation*.

We use the motor system from a **Custom Action** option in our demo, but you can use motors to move the character from any component.

Here's a little snippet that shows how to move from a standard Unity character script:

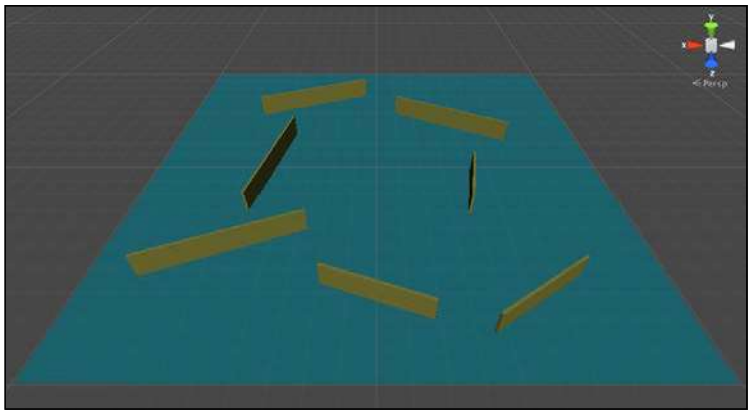
```
AIRigaiRig = GetComponentInChildren<AIRig>();
…
aiRig.AI.Motor.UpdateMotionTransforms();

aiRig.AI.Motor.MoveTarget.VectorTarget = targetPositon;
aiRig.AI.Motor.Move();
aiRig.AI.Motor.ApplyMotionTransforms();
```

First, we get `AIRig` attached to the character. Then, we call `UpdateMotionTransforms()` to make sure that the AI system has the latest transforms (position and rotation) from the character before updating. Next, we set `VectorTarget` to a `Vector3` variable as `targetPosition`, so the AI system knows where we want to go. Then, we call `Move()` to update the character's transforms in the AI system, and finally, we call `ApplyMotionTransforms()` to update our game to show the new transforms from the AI system. Using these methods, we can update game characters at any time.

With customized movements, we can have our characters adapt in any way we want. The best way to see how this works is to look at a demo. The demo that we will look at in this chapter is an extension of the ship demo from *Chapter 6*, *Sensors and Activities*. We will have a ship in a level searching for gold pieces, but we'll extend it to make the gold pieces appear more random and dynamic in the level over time. Then, we will have a bomb with a timer and when it goes off all of ships will be destroyed and stop updating their AI. This will illustrate how we can have AI characters react to game events.

RAIN's demo

The basic start of the demo will be similar to our others, a ground with several walls around for our ships to travel. This is how the basic starting point of our demo should look:



The basic starting point of our demo

One of the first things we will need is the ability to query a random location in the scene to spawn and find points to travel to. Create a class called `Ground` and add it to the ground plane. This class will be used to provide higher-level information about the level, the first of which is being able to find a random position in the level. Here is the `Ground` class with the random position chooser method:

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

public class Ground : MonoBehaviour {

    private static Vector3 min, max;

    private const float LevelHeight = 0.5f;

    public static Vector3 randomLevelPosition() {
        Vector3 position = new Vector3();
        position.x = Random.Range(min.x, max.x);
        position.y = LevelHeight;
        position.z = Random.Range(min.z, max.z);
        return position;
    }

    void Start () {
        const float innerEdge = 0.9f;
        min = renderer.bounds.min * innerEdge;
        max = renderer.bounds.max * innerEdge;
    }

    void Update () {

    }
}

```

In the preceding code, we are able to ask for a random position at anytime from anywhere in the game. In the `Start` method for the `Ground` class, we store the `max` and `min` positions for it, and as we don't want positions on the very edge of the level, it is scaled to 90 percent by multiplying by `0.9f`. The `min` and `max` positions are static, so we can add a static method, `randomLevelPosition()`, that returns a random 2D position on the level with a constant height. We'll be using this method in several other spots in the code.

We could do additional checks on this position finding to make sure that the spot never overlaps any of the walls in the scene, but to make the code simpler for this demo, we won't worry about this edge case. However, you would do this in a production game.

Reacting to game events

Next, we want to have some ships chase gold pieces, but we'll make it more dynamic than in the last demo. Create a **Sphere** object with a gold color and add a **RAIN** entity to it (by navigating to **RAIN | Create Entity**) and add a **Visual Aspect** called **Gold** to it so that AI characters can sense it. Turn this gold piece into a prefab. Instead of just placing it manually in the scene, we want them to be spawned randomly; add the code mentioned in the following screenshot to the `Ground` script:

```
7       public Transform gold;
8       private float goldTimer = 0.0f;
9       private const float goldCreateTime = 2.0f;
10
11      void Update () {
12
13          goldTimer += Time.deltaTime;
14          if(goldTimer>= goldCreateTime) {
15              Instantiate(gold, randomLevelPosition(), Quaternion.identity);
16              goldTimer = 0.0f;
17          }
18      }
```

In the Unity editor, drag the **Gold** prefab to the `Transform gold` in this script. This script randomly spawns a gold piece somewhere in the level every 2 seconds by tracking the time using `Time.deltaTime`. If you run the game now, you'll see a gold piece created randomly every 2 seconds. Next, we need ships to collect these.

Our AI ship characters will pick a random spot on the level and travel there and then after arriving, pick another random spot to go to; however, if they see a piece of gold along the way, they will stop and pick it up. To do this, create a ship object with a RAIN visual sensor with a horizontal angle of **120**, a vertical angle of **45**, and a range of **15**. The behavior tree for the ship will be straightforward. Set the root node to parallel and one **Detect** child set to look for **Gold** and store its form in the `gold` variable. Add another child to the root with a constraint to test if **gold == null**. If gold is not **null**, it should move to pick up the gold; if it is, pick a random spot on the level and move there. To pick a random spot in the level, create a new **Custom Action** option with a new script called `ChooseRandomSpot`. Set the following code for it:

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

[RAINAction]
public class ChooseRandomSpot : RAINAction
{
    public ChooseRandomSpot()
    {
        actionName = "ChooseRandomSpot";
    }

    public override void Start(AI ai)
    {
        Vector3 moveTarget = Ground.randomLevelPosition();
        ai.WorkingMemory.SetItem("moveTarget", moveTarget);
        base.Start(ai);
    }

    public override ActionResult Execute(AI ai)
    {
        return ActionResult.SUCCESS;
    }

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

The `Start` method uses our static `Ground` method to find a random position in the level and sets it to the `moveTarget` variable in the AI's memory. Next, add a **move** node to go to the `moveTarget` variable. If you need a review of how to set up these nodes, check *Chapter 6*, *Sensors and Activities*. The behavior tree for the ship should look like the following screenshot:

Change the ship to a prefab and add a few ships to the level. Now if you run the game, your ships will wander around, but if they see gold, they will race to pick it up and the first one there collects it.

Using RAIN's motor directly

However, if you run the game now, you'll see a problem. As expected, the ship will look for gold, and if it doesn't see any, it will pick a random position on the level and move toward it. If it sees gold along the way, it doesn't stop to pick it up; it keeps moving to its target location.

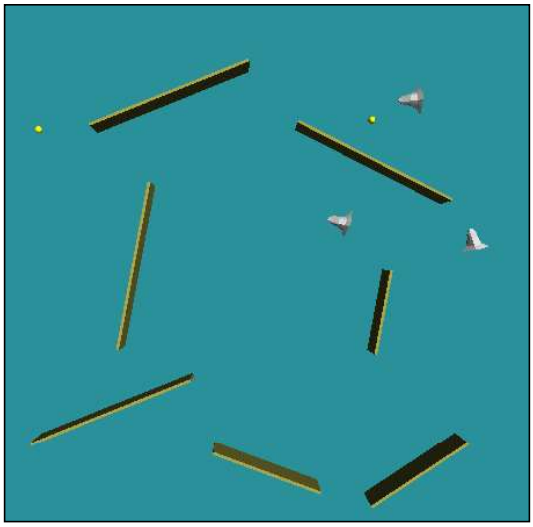
Our root node is a parallel type, so the character is always trying to detect gold, but it still ignores it while traveling. This is because our **move** node will keep running until it hits its destination, and even if it sees something, it is not interrupted until it gets there. To fix this, delete the **move** node underneath **ChooseRandomSpot Custom Action**. Then, change **ChooseRandomAction** to the code shown in the following screenshot:

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

[RAINAction]
public class ChooseRandomSpot : RAINAction
{
    public ChooseRandomSpot()
    {
        actionName = "ChooseRandomSpot";
    }

    public override void Start(AI ai)
    {
        Vector3 moveTarget = Ground.randomLevelPosition();
        ai.WorkingMemory.SetItem("moveTarget", moveTarget);
        base.Start(ai);
    }

    public override ActionResult Execute(AI ai)
    {
        GameObject gold = ai.WorkingMemory.GetItem<GameObject>("gold");
        if(gold != null) {
            return ActionResult.FAILURE;
        }

        Vector3 moveTarget = ai.WorkingMemory.GetItem<Vector3>("moveTarget");
        if(Vector3.Distance(moveTarget, ai.Body.transform.position) < 1.0f) {
            return ActionResult.SUCCESS;
        }

        ai.Motor.MoveTo(moveTarget);

        return ActionResult.RUNNING;
    }

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

This is a big change, so let's discuss what is going on. The `Start` method is the same as before: store a random position to move to in memory. However, our action method is different. The first thing it does is it queries the memory for gold. If we have gold, we don't need to keep moving to our target, so we return failure. Then, we get our `moveTarget` variable out of memory and check the position of the `Body` variable of our AI. If it is within one unit of the goal, we say that this is close enough and return success. Finally, if we don't have gold and aren't close to `moveTarget`, we call on the AI's motor system to move to the target and keep updating it by returning the running state.

With this update, we could have used a regular class variable to store `moveTarget`, but we keep it in memory to keep things consistent.

If you run the demo now, we will see the ships moving around as new gold appears in more expected ways, as shown in the following screenshot:



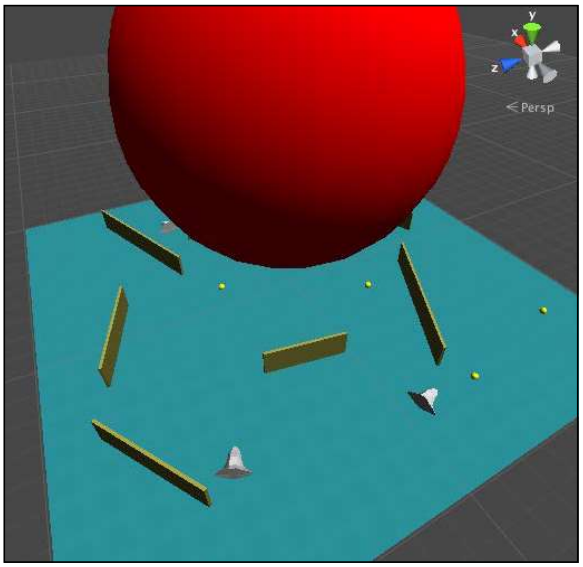
You can see the ships wandering and chasing gold in the preceding screenshot.

Adding large game events

As the last step of this demo, let's have a giant bomb go off in the scene and then have all of our AI stop to simulate having them all destroyed. To start, create a large red sphere to represent the bomb and turn it into a prefab. We will have the AI characters react to this bomb in the standard way by adding a RAIN Entity component to it and a visual aspect and have visual sensors on the ships detect it. But to show we can access the AI systems directly, let's have the bomb go off using the `Ground` class:

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

public class Ground : MonoBehaviour {

	...
    public Transform bomb;
    public float BombTime = 30.0f;

	...

    void Update () {

        if(BombTime < 0.0f) {
            return;
        }

        goldTimer += Time.deltaTime;
        if(goldTimer >= goldCreateTime) {
            Instantiate(gold, randomLevelPosition(), Quaternion.identity);
            goldTimer = 0.0f;
        }

        BombTime -= Time.deltaTime;
        if(BombTime <= 0.0f) {
            GameObject.Instantiate(bomb);

            AIRig[] AIs = GameObject.FindObjectsOfType(typeof(AIRig)) as AIRig[];
            for(int i = 0; i < AIs.Length; i++) {
                AIs[i].enabled = false;
            }
        }
    }
}

```

Here, we added a bomb transform to the script, so drag the bomb prefab in the Unity prefab over to it. There is also a field for a countdown that when it goes to 0, the bomb goes off and is instantiated into the scene. At this point, we grab all the AIs in the scene and send them a message, in this case, to disable it. We could have made this more complex than a simple disabling; this just shows us that we can have our game AI react to game events from anywhere. If you run the demo now, the ships stop when the bomb goes off, as shown in the following screenshot:



In the preceding screenshot, you can see the ships reacting to a bomb.

The React AI

We have been using RAIN for our adaption so far, but there is no reason you cannot create a demo like the one we just did with React. The basic behavior tree and node logic can stay the same. The main difference is that React doesn't use a built-in sensor system; instead, users define sensing based on what they think is the best. This can be done through Unity's built-in ray casting methods to query the scene. The following is a method adapted from React's sample that can be used with React to determine the visibility of a target. This code takes in a target and first does a simple test to see whether the target is within the field of view by finding the vector of the target from the player and comparing the angle of it and the forward direction of the AI character.

```
7     public bool IsTargetVisible (Transform target)
8     {
9         Vector3 targetDirection = target.position - transform.position;
10        Ray ray = new Ray (transform.position, targetDirection);
11        varinFOV = Vector3.Angle (transform.forward, targetDirection) < 45;
12        if (inFOV) {
13            RaycastHit hit;
14            if (Physics.Raycast (ray, out hit, 1000)) {
15                return hit.collider.transform == target;
16            }
17        }
18        return false;
19    }
```

This is a simple and quick test that does a basic check, in terms of collision detection, and this is called the broad phase. Then, the Unity physics system is used to ray cast from the character to the target; this is quite expensive but a more accurate test. Using this for sensing and React's built-in behavior tree demos, like the one in this chapter, can be created.

Summary

In this chapter, we looked at how we can make our AI adapt to events in the game. This was done using methods we learned in the previous chapters, and we also took a look at RAIN's motor system to allow our adaptions to be more customizable. Our demos in this chapter have been pretty straightforward, but there is no reason why this demo couldn't be extended to have more events to send and more reactions defined in the character behavior trees. However, our demos have been missing one important thing, which is yet to be discussed: the player. In the next chapter, we will discuss how AI characters attack by adding a player to our scene and having our characters react and attack. We will discuss how to create enemies for the player and have them attack the player.