4
Crowd Chaos

Part of having a realistic game environment is having the nonplayer characters and NPCs act in a believable way. Crowd chaos is all about keeping NPCs busy to create crowded backgrounds for our games. Perhaps your game is set up in a mall, or a city, or any other place where lots of NPCs need to wander around and look like they are doing something. Crowds like these will be the subject of this chapter and the next.

In this chapter, you will learn about:

- Working with crowd chaos
- How to create crowd type characters in the React and RAIN AI packages
- Expanding our knowledge of behavior trees

An overview of crowd chaos

Crowd Chaos is all about giving separate interests to a large number of NPCs, so they look like they are living their own lives. In its lightest form, this can be something very simple, such as a whole bunch of NPCs picking random targets, walking to them, possibly sitting still for a moment, and then starting over. This stands out in real-time strategy games when buildings are constructed, and you see a construction worker walking to random points of the structure and waving their arms about.

Every game that needs crowd chaos will typically have a basic wandering base, and it can be extended as needed. Perhaps the crowd will form lines of more NPCs that are waiting at a spot. Perhaps the targets have changing values and AIs prefer higher values. They pick up a random block and put it somewhere else. The base wandering behavior needed for these and other crowd behaviors is what we will implement in both React and RAIN AI.

React AI

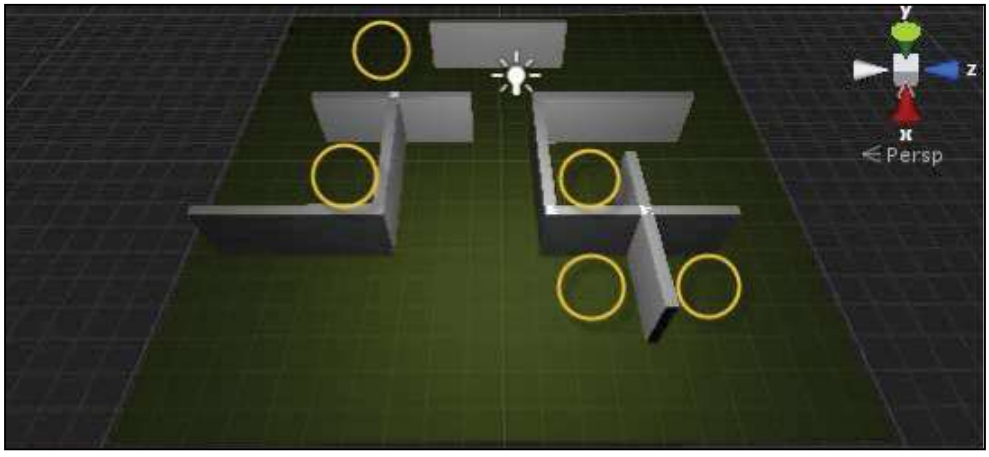
For this demo, we will duplicate the path-following behavior demo in React from *Chapter 1*, *Pathfinding*, and then update it to see some emergent behavior develop from it. We will need to complete the following:

- Create a world with some walls
- Create target markers in the scene
- Create a script with a custom editor to find the targets
- Create the behavior
- Create NPCs and assign the behavior

Setting up a scene with React

To start out with, we will need a basic environment for characters to walk in. Create a plane, call it **Floor**, and add some cubes, shaping them into walls. These will need to be static so that Unity's navigation mesh can find them. Then, we'll need to select the floor and add the navigation mesh. If you've forgotten how to do any of this, it is all covered in the React tutorial in *Chapter 1*, *Pathfinding*.

Next, we need some targets. We'll use a different approach for this from our previous demos and let GameObjects mark the targets. Create an empty GameObject and call it **Targets**. Underneath it, add more empty GameObjects. Give them all a tag, `NpcActivityTarget`, which you might need to create. Distribute these targets to different locations on the screen like this:



The preceding screenshot shows how our basic React scene setup with targets should look like.

Now, we need a script that can find these locations. It will be based on our earlier scripts and will contain three methods: one to find a target, one to move to a target, and another to hang around.

You can find the completed script at **Disk** | **Scripts** | **React AI** | **LookBusy.cs**. To get the tags to show up as a dropdown, we've also provided a custom editor, which is also available at **Disk** | **Scripts** | **React AI** | **TagOption.cs**. You will need to put this under `Assets/Editor` for it to work in Unity. **TagOptions** is a script that does nothing more than give a drop-down selector for the tag to be used. **LookBusy** uses the selected tag to find objects that are targets in the game.

Here are a couple of the methods inside the script. These are easy to reproduce or modify on your own:

```
GameObject[] targets = GameObject.FindGameObjectsWithTag(this.
SelectedTag);
// If there are not at least two targets to choose from return an
error
if(targets.Length < 2)
{
  Debug.LogWarning("LookBusy.cs:FindTarget() --> There are less than
2 targets with the tag, '" + this.SelectedTag + "'. This script wants
more positions.");
  yield return NodeResult.Failure;
  yield break;
}
// From the targets randomly select one and if it is closer than our
minimum distance return it, otherwise keep trying a constant number of
times before failing
int attempts = 0;

while(Vector3.Distance(this.Destination.transform.position, this.
transform.position) < this.MinimumDistance)
{
  this.Destination = targets[Random.Range(0, targets.Length)];
if(attempts >= 25)
{
    Debug.LogWarning("LookBusy.cs:FindTarget() --> Could not find a
target farther than the mininum distance. Either lower the mininum
distance or space the targets farther apart.");
    yield return NodeResult.Failure;
  }
}
yield return NodeResult.Success;
}
```

In this script, we first check whether we have at least two targets in the game tagged to select from, and if we don't have them, the script reports an error. You'll notice that the error doesn't break the game, it just gives a specific warning on the log of what you need and where the log was posted from. Next, the script selects a random position from the list of nodes, and if the position within a character's minimum distance (and not the same target the character is already on), the script returns the position. This random position is chosen from the list of nodes no more than a constant number of times, that is, 25 times. This random choosing method doesn't guarantee success, but it is a quick and easy way to choose a random target.

Besides picking a random place for a character to walk to, we also need a random amount of time for the NPC to stay at the location they go to. The `HangAround` method does this:

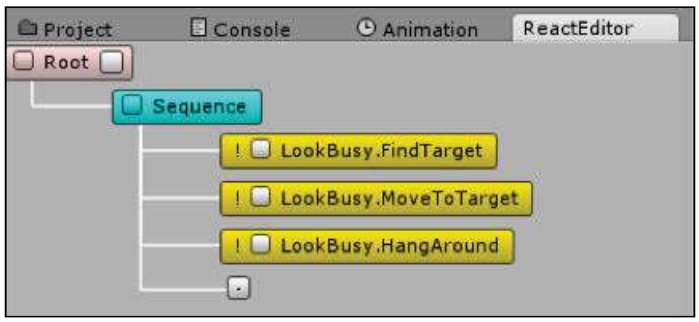
```
public Action HangAround()
{
  // Choose a random time to wait
  float randomtime = Random.Range(this.ShortWaitTime, this.
LongWaitTime);
  while(totalTime < randomtime)
  {
    totalTime += UnityEngine.Time.deltaTime;
    yield return NodeResult.Continue;
  }
  totalTime = 0;
  yield return NodeResult.Success;
}
```

The `HangAround` function just makes us wait a few seconds. First, it selects how long to wait, and then, once this amount of time passes, returns a success. Notice that we return `NodeResult.Continue`. This tells the script to wait until the next update and then try to get, then see if it is finished yet. (Yield is used so the game doesn't freeze up.)

The `MoveToTarget` function isn't given here as it is nearly identical to the function we used in *Chapter 1*, *Pathfinding*, except that now we are going after the target specified randomly from `FindTarget`.

Building behavior trees in React

Now that we have our behavior methods, we can build the behavior tree. Right-click on your project's `Assets` folder and navigate to **Create** | **Reactable**. Rename it to `LookBusyReactable`. Then, right-click on it and select **Edit Reactable**:



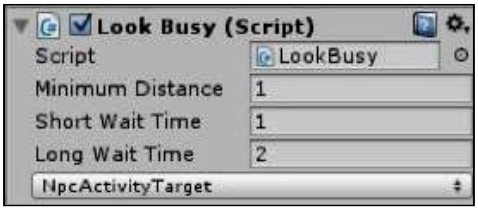
The preceding is a screenshot of the behavior tree editor completed. Right-click on **Root** and navigate to **Add** | **Branch** | **Sequence**. This is so it completes each step before moving on to the next. Right-click on **Sequence** and navigate to **Add** | **Leaf** | **Action**. Do this three times:

1. For the first one, click on the empty checkbox and navigate to **Scripts** | **LookBusy** | **FindTarget**. This is part of the `LookBusy.cs` script that we added earlier.
2. For the second one, do the same but instead navigate to **Scripts** | **LookBusy** | **MoveToTarget**.
3. For the third one, navigate to **Scripts** | **LookBusy** | **HangAround**.

The AI will find a random target from the list of GameObjects with the correct tag, **NpcActivityTarget**, as we set in the `LookBusy` script. Then, it moves to that target and hangs around for 2 to 4 seconds.

Setting up wandering characters with React

Finally, we will create the NPCs and assign a behavior. For this, you can use a character similar to the first, just a sphere stretch 2x tall, with a small cube on the front of it so that we can see the direction it is facing. Add the `LookBusy` script to the NPC:



This is how the `LookBusy` script options look like.

Minimum Distance is how far away you can be from the target and still be satisfied that you reached it. **Short Wait Time** and **Long Wait Time** are time ranges (in seconds) you hang around for, and **NpcActivityTarget** is the tag that the GameObjects have to identify as targets.

Next, add **Nav Mesh Agent** to the NPC so that it can navigate around the level. Finally, add the `Reactor` script and set its **Reactable** property to **LookBusyReactable**, which is the behavior tree we created earlier.

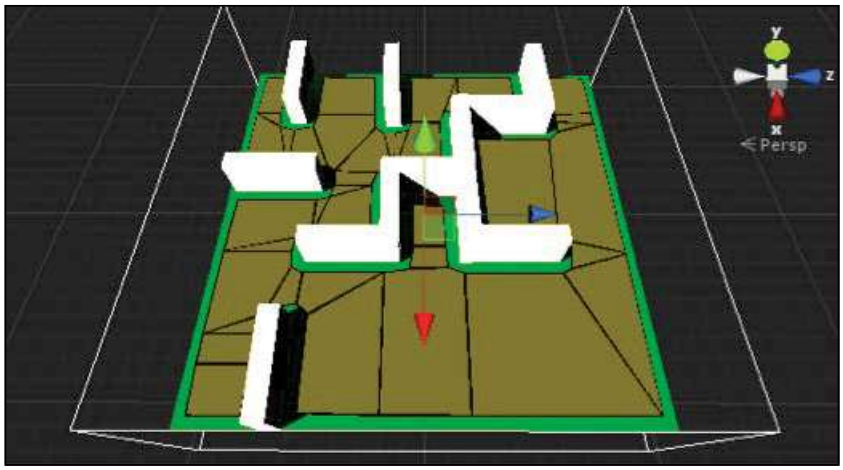
This completes all the steps needed to have NPC characters wander around in a game using React. You should now be able to create as many characters as you like and have them walk around a level.

RAIN AI

We have already looked at a basic wander behavior for RAIN in *Chapter 2*, *Patrolling*, when creating patrolling AI, but there, we manually created each possible location for the NPC to go to. In this demo, we will pick random points to wander to from anywhere in the navigation mesh. The NPCs won't have any interaction, though such features are not difficult to add. Here is a breakdown of the steps we will do in this section:

- Set up a world
- Build the behavior tree
- Add a script to pick new points
- Add the NPCs
- Learn about the RAIN AI world and behavior tree setup

First, we'll create a new world. Start with a large plane called `floor`, and add some cubes shaped into walls. You will need to add a navigation mesh and bake it into the scene. These are the same steps we have performed for RAIN demos in earlier chapters. The following is an example of how the scene could look:



Next comes the behavior tree. From the RAIN menu, select **Behavior Tree Editor**. Create a new tree called `RandomWalk`. The objective of this AI is broken into three steps, taken in this order: select a target, walk to it, and then wait a moment. This is a good case for the RAIN decision node sequenced. Under the root node, we will right-click and go to **Create** | **Decisions** | **Sequencer**.

As there is no behavior tree node built into RAIN that will choose a random location, we will use a custom action and write our own script. Right-click on the new SEQ node and navigate to **Create** | **Actions** | **Custom Action**. Set the **Repeat** property to **Until Success** because we want it to continue processing this node until it returns a success and then move on to the next node. Name the custom action node `Select Next Target`. You'll notice that you can put spaces in the name. This is useful as it shows up in the behavior tree, making it easier to follow:

We could add the script now, but we'll finish the rest of the nodes in the tree and then come back; for now, we will assume that the script will find a spot to walk to. The next action is to walk to it. This needs two things happening simultaneously, animation and actual walking, which means that we will use the **Parallel** decision node. Right-click on the sequence node and select **Create** | **Decisions** | **Parallel**. Name it **Walk to Target**.

Under **Walk to Target**, right-click and go to **Create** | **Actions** | **Animate**. Name it `Appear Walking`. Set the animation state to **walk**. Also under **Walk to Target**, right-click and navigate to **Create** | **Actions** | **Move**. Name it `Move`. Set the **Move Speed** property to `1`, so it moves 1 meter per second. The **Move Target** value should be set to **TargetPoint** without the quotes. **TargetPoint** doesn't exist yet; our script will create it.

The last step that the NPC must perform is generate a wait moment. To give the NPC more life, we will make sure that it uses an idle animation, which also means two things must happen simultaneously. Right-click on the root sequence node (**SEQ**) and go to **Create** | **Decisions** | **Parallel**. Name it `Look Busy`. Add an animation action under this and set **idle** as its **Animation State** and **Stand here** as its name.

Also, under the **Look Busy** node, we will right-click and go to **Create** | **Actions** | **Wait for Timer**. Name it `For a couple seconds`. Set the **Seconds** property to **2**.

The behavior tree is now complete. All we need to do now is fill in the script that gets our **TargetPoint**, so it knows where to move.

RAIN AI custom wander scripts

To start creating our needed wander scripts, first select the **Select Next Target** action in the behavior tree. Under the **Class** property, set it to **Create Custom Action**, which pops up a box to define the script. The following screenshot shows what a RAIN custom action creation dialog looks like:

Set the name to `SelectRandomTarget` and the script to **CSharp**. This will generate the default custom action script file already filled with a few common methods. In this demo, we only need to use the `Execute` function:

```
public override ActionResult Execute(AI ai)
{
  var loc = Vector3.zero;
  List<RAINNavigationGraph> found = new List<RAINNavigationGraph>();
  do
  {
    loc = ai.Kinematic.Position;
    loc.x += Random.Range(-8f, 8f);
    loc.z += Random.Range(-8f, 8f);
    found = NavigationManager.Instance.GraphsForPoints(
      ai.Kinematic.Position,
      loc,
      ai.Motor.StepUpHeight,
      NavigationManager.GraphType.Navmesh,
      ((BasicNavigator)ai.Navigator).GraphTags);
  }
  while ( Vector3.Distance(loc, ai.Kinematic.Position) < 2f
      || found.Count == 0);
  ai.WorkingMemory.SetItem<Vector3>("TargetPoint", loc);
  return ActionResult.SUCCESS;
}
```

In this code, we try to find a good value for the `loc` variable, a location variable that is set to a different random location, up to 18 meters away. The `found` variable identifies whether a path exists or not. These two things are determined in a loop, which ends as long as two conditions are met. First, the distance has to be greater than 2, as the movement should be detected by the player and second, we see if the points found is greater than zero. If it found none, then we would not be able to get to that location.

Once the `loc` variable has a location that works, the next thing it does is use RAIN's memory system and sets a memory entry, **TargetPoint**, to the new location. Remember that we have the **Move** action in our behavior set to find **TargetPoint**, so **Move** will go to our newly found location. Finally, we return a success.

This completes our behavior and script. The last thing we need to do is give that behavior to some NPCs and run the game.

Putting NPCs in the RAIN demo

Start by adding another simple NPC character to the game, like we did in the first chapter. We don't need to add any scripts directly to it. Instead, make sure that the NPC is selected in the hierarchy, and from the RAIN menu, select **Create AI**.

In the AI GameObject/component that was added, from the **Mind** tab, set the **Behavior Tree Asset** value to **RandomWalk**, which is found in **Assets**. Under the animation tab, click on the **Add Existing Animations** button.

Now, try the game. A single NPC should be walking around the screen, pausing, and then walking to another location at random. To create a larger crowd, just duplicate the NPC GameObject in the scene at several locations.

Summary

We were able to use scripting and behavior trees in both React AI and RAIN to effectively create a wandering AI. Each AI had strengths and weaknesses, though the weaknesses were more of a preference.

Behavior tree editors were used in both RAIN and React, and both work in a similar fashion. In RAIN, you can start editing a tree from the menu, or from the editor itself. (It had the option to select the behavior directly in the editor.) With React, you can do this from the **Project** tab, by right-clicking and choosing to edit it. React had premade scripts that can do nearly all the actions that were needed, except that instead of selecting randomly from a list of targets with the tag, it would select a target expecting only one object with that tag. With RAIN, we made a custom action node to choose a location to go to.

Both React and RAIN AI are general AI systems that are useful for many different types of game situations, so neither were designed specifically to handle crowds. In the next chapter, we will look at different tools with more focus on creating crowd AI.