8
Miscellaneous

In this chapter, you will learn different techniques for:

- Handling random numbers better
- Building an air-hockey rival
- Devising a table-football competitor
- Creating a tennis rival
- Creating mazes procedurally
- Implementing a self-driving car
- Managing race difficulty using a rubber-banding system

Introduction

In this final chapter, we will introduce new techniques, and use algorithms that we have learned in the previous chapters in order to create new behaviors that don't quite fix in a definite category. This is a chapter to have fun and get another glimpse of how to mix different techniques in order to achieve different goals.

Handling random numbers better

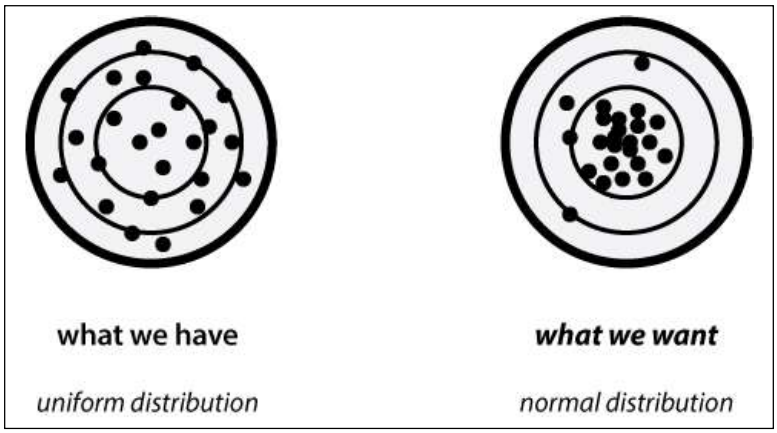
Sometimes, we need to create random behaviors that don't differ too much from a pivot point; this is the case of an aiming behavior. A normalized random behavior will shoot equally along the *x* and the *y* axes over a given distance from the aiming point. However, we would like most of the bullets to aim closer to the target because that's the expected behavior.

Most of the random functions out there return normalized values along the range given to them, and those are the expected results. Nonetheless, this is not completely useful for certain features in game development, as we just said. We will implement a random function to be used in our games with normal distribution instead of a normal distribution.

Getting ready

It is important to understand the differences between uniform and normal distribution. In the following figure, we can see a graphical representation of the behavior we're looking for by applying normal distribution with the example mentioned in the introductory text.

In the figure on the left-hand side, the uniform distribution spreads through the whole circle, and it is intended to be used in general random distributions. However, while developing other techniques, such as gun aiming, the desired random distribution will look more like the image on the right-hand side.



How to do it...

We will build a simple class as follows:

1. Create the `RandomGaussian` class:

```
using UnityEngine;

public class RandomGaussian
{
    // next steps
}
```

2. Define the `RangeAdditive` member function that initializes the necessary member variables:

```
public static float RangeAdditive(params Vector2[] values)
{
    float sum = 0f;
    int i;
    float min, max;
    // next steps
}
```

3. Check whether the number of parameters equals zero. If so, create three new values:

```
if (values.Length == 0)
{
    values = new Vector2[3];
    for (i = 0; i < values.Length; i++)
        values[i] = new Vector2(0f, 1f);
}
```

4. Sum all the values:

```
for (i = 0; i < values.Length; i++)
{
    min = values[i].x;
    max = values[i].y;
    sum += Random.Range(min, max);
}
```

5. Return the resulting random number:

```
return sum;
```

There's more...

We should always strive for efficiency. That's why there's another way of delivering a similar result. In this case, we could implement a new member function based on the solution offered by Rabin and others (refer to the proceeding *See also* section):

```
public static ulong seed = 61829450;
public static float Range()
{
    double sum = 0;
    for (int i = 0; i < 3; i++)
    {
        ulong holdseed = seed;
        seed ^= seed << 13;
```

```
            seed ^= seed >> 17;
            seed ^= seed << 5;
            long r = (long)(holdseed * seed);
            sum += r * (1.0 / 0x7FFFFFFFFFFFFFFF);
        }
        return (float)sum;
    }
```

See also

- For further information on the theory behind the Gaussian random generator and other advanced generators, please refer to the book *Game AI Pro* by Steve Rabin, article number 3

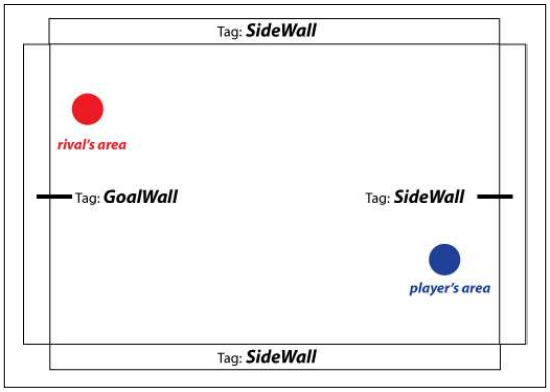
Building an air-hockey rival

Air hockey is probably one of the most popular games enjoyed by players of all ages during the golden age of arcades, and they are still found everywhere. With the advent of touchscreen mobile devices, developing an air-hockey game is a fun way to not only test physics engines, but also to develop intelligent rivals despite the apparently low complexity of the game.

Getting ready

This is a technique based on some of the algorithms that we learned in *Chapter 1*, *Movement*, such as `Seek`, `Arrive`, and `Leave`, and the ray casting knowledge that is employed in several other recipes, such as path smoothing.

It is necessary for the paddle game object to be used by the agent to have the `AgentBehaviour`, `Seek`, and `Leave` components attached, as it is used by the current algorithm. Also, it is important to tag the objects used as walls, that is, the ones containing the box colliders, as seen in the following figure:



Finally, it is important to create an `enum` type for handling the rival's state:

```
public enum AHRState
{
    ATTACK,
    DEFEND,
    IDLE
}
```

How to do it...

This is a long class, so it is important to carefully follow these steps:

1. Create the rival's class:

```
using UnityEngine;
using System.Collections;

public class AirHockeyRival : MonoBehaviour
{
    // next steps
}
```

2. Declare the public variables for setting it up and fine-tuning it:

```
public GameObject puck;
public GameObject paddle;
public string goalWallTag = "GoalWall";
public string sideWallTag = "SideWall";
[Range(1, 10)]
public int maxHits;
```

3. Declare the private variables:

```
float puckWidth;
Renderer puckMesh;
Rigidbody puckBody;
AgentBehaviour agent;
Seek seek;
Leave leave;
AHRState state;
bool hasAttacked;
```

4. Implement the `Awake` member function for setting up private classes, given the public ones:

```
public void Awake()
{
```

```
        puckMesh = puck.GetComponent<Renderer>();
        puckBody = puck.GetComponent<Rigidbody>();
        agent = paddle.GetComponent<AgentBehaviour>();
        seek = paddle.GetComponent<Seek>();
        leave = paddle.GetComponent<Leave>();
        puckWidth = puckMesh.bounds.extents.z;
        state = AHRState.IDLE;
        hasAttacked = false;
        if (seek.target == null)
            seek.target = new GameObject();
        if (leave.target == null)
            leave.target = new GameObject();
    }
```

5. Declare the `Update` member function. The following steps will define its body:

```
    public void Update()
    {
        // next steps
    }
```

6. Check the current state and call the proper functions:

```
    switch (state)
    {
        case AHRState.ATTACK:
            Attack();
            break;
        default:
        case AHRState.IDLE:
            agent.enabled = false;
            break;
        case AHRState.DEFEND:
            Defend();
            break;
    }
```

7. Call the function for resetting the active state for hitting the puck:

```
    AttackReset();
```

8. Implement the function for setting up the state from external objects:

```
    public void SetState(AHRState newState)
    {
        state = newState;
    }
```

9. Implement the function for retrieving the distance from paddle to puck:

```
private float DistanceToPuck()
{
    Vector3 puckPos = puck.transform.position;
    Vector3 paddlePos = paddle.transform.position;
    return Vector3.Distance(puckPos, paddlePos);
}
```

10. Declare the member function for attacking. The following steps will define its body:

```
private void Attack()
{
    if (hasAttacked)
        return;
    // next steps
}
```

11. Enable the agent component and calculate the distance to puck:

```
agent.enabled = true;
float dist = DistanceToPuck();
```

12. Check whether the puck is out of reach. If so, just follow it:

```
if (dist > leave.dangerRadius)
{
    Vector3 newPos = puck.transform.position;
    newPos.z = paddle.transform.position.z;
    seek.target.transform.position = newPos;
    seek.enabled = true;
    return;
}
```

13. Attack the puck if it is within reach:

```
hasAttacked = true;
seek.enabled = false;
Vector3 paddlePos = paddle.transform.position;
Vector3 puckPos = puck.transform.position;
Vector3 runPos = paddlePos - puckPos;
runPos = runPos.normalized * 0.1f;
runPos += paddle.transform.position;
leave.target.transform.position = runPos;
leave.enabled = true;
```

14. Implement the function for resetting the parameter for hitting the puck:

```
private void AttackReset()
{
    float dist = DistanceToPuck();
    if (hasAttacked && dist < leave.dangerRadius)
        return;
    hasAttacked = false;
    leave.enabled = false;
}
```

15. Define the function for defending the goal:

```
private void Defend()
{
    agent.enabled = true;
    seek.enabled = true;
    leave.enabled = false;
    Vector3 puckPos = puckBody.position;
    Vector3 puckVel = puckBody.velocity;
    Vector3 targetPos = Predict(puckPos, puckVel, 0);
    seek.target.transform.position = targetPos;
}
```

16. Implement the function for predicting the puck's position in the future:

```
private Vector3 Predict(Vector3 position, Vector3 velocity, int
numHit)
{
    if (numHit == maxHits)
        return position;
    // next steps
}
```

17. Cast a ray, given the position and the direction of the puck:

```
RaycastHit[] hits = Physics.RaycastAll(position, velocity.
normalized);
RaycastHit hit;
```

18. Check the hit results:

```
foreach (RaycastHit h in hits)
{
    string tag = h.collider.tag;
    // next steps
}
```

19. Check whether it collides with the goal wall. Base case:

```
if (tag.Equals(goalWallTag))
{
    position = h.point;
    position += (h.normal * puckWidth);
    return position;
}
```

20. Check whether it collides with a side wall. Recursive case:

```
if (tag.Equals(sideWallTag))
{
    hit = h;
    position = hit.point + (hit.normal * puckWidth);
    Vector3 u = hit.normal;
    u *= Vector3.Dot(velocity, hit.normal);
    Vector3 w = velocity - u;
    velocity = w - u;
    break;
}
// end of foreach
```

21. Enter the recursive case. This is done from the `foreach` loop:

```
return Predict(position, velocity, numHit + 1);
```

How it works...

The agent calculates the puck's next hits given its current velocity until the calculation results in the puck hitting the agent's wall. This calculation gives a point for the agent to move its paddle toward it. Furthermore, it changes to the attack mode when the puck is close to its paddle and is moving towards it. Otherwise, it changes to idle or defend depending on the new distance.

See also

- *Chapter 1*, *Movement* recipes *Pursuing and evading* and *Arriving and leaving* recipes

Devising a table-football competitor

Another common table game that has made its way into the digital realm is table football. In this recipe, we will create a competitor, imitating the way a human plays the game and using some techniques that emulate human senses and limitations.

Getting ready

In this recipe, we will use the knowledge gained from *Chapter 5*, *Agent Awareness*, and the emulation of vision.

First, it is important to have a couple of `enum` data structures, as shown in the following code:

```
public enum TFRAxisCompare
{
    X, Y, Z
}

public enum TFRState
{
    ATTACK, DEFEND, OPEN
}
```

How to do it...

This is a very extensive recipe. We'll build a couple of classes, one for the table-football bar and the other for the main AI agent that handles the bars, as follows:

1. Create a class for the bar that will be handled by the AI:

```
using UnityEngine;
using System.Collections;

public class TFRBar : MonoBehaviour
{
    [HideInInspector]
    public int barId;
    public float barSpeed;
    public float attackDegrees = 30f;
    public float defendDegrees = 0f;
    public float openDegrees = 90f;
    public GameObject ball;
    private Coroutine crTransition;
    private bool isLocked;
    // next steps
}
```

2. Implement the `Awake` function:

```
void Awake()
{
    crTransition = null;
    isLocked = false;
}
```

3. Define the function for setting the state of the bar:

```
public void SetState(TFRState state, float speed = 0f)
{
    // next steps
}
```

4. Check whether it is locked (after beginning a movement). This is optional:

```
// optional
if (isLocked)
    return;
isLocked = true;
```

5. Validate the speed:

```
if (speed == 0)
    speed = barSpeed;
float degrees = 0f;
```

6. Validate the state and make a decision out of it:

```
switch(state)
{
    case TFRState.ATTACK:
        degrees = attackDegrees;
        break;
    default:
    case TFRState.DEFEND:
        degrees = defendDegrees;
        break;
    case TFRState.OPEN:
        degrees = openDegrees;
        break;
}
```

7. Execute the transition:

```
if (crTransition != null)
    StopCoroutine(crTransition);
crTransition = StartCoroutine(Rotate(degrees, speed));
```

8. Define the function for rotating the bar:

```
public IEnumerator Rotate(float target, float speed)
{
    // next steps
}
```

9. Implement the internal body for the transition:

```
while (transform.rotation.x != target)
{
    Quaternion rot = transform.rotation;
    if (Mathf.Approximately(rot.x, target))
    {
        rot.x = target;
        transform.rotation = rot;
    }
    float vel = target - rot.x;
    rot.x += speed * Time.deltaTime * vel;
    yield return null;
}
```

10. Restore the bar to its default position:

```
isLocked = false;
transform.rotation = Quaternion.identity;
```

11. Implement the function for moving the bar from side to side:

```
public void Slide(float target, float speed)
{
    Vector3 targetPos = transform.position;
    targetPos.x = target;
    Vector3 trans = transform.position - targetPos;
    trans *= speed * Time.deltaTime;
    transform.Translate(trans, Space.World);
}
```

12. Create the class for the main AI:

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

public class TFRival : MonoBehaviour
{

    public string tagPiece = "TFPiece";
    public string tagWall = "TFWall";
    public int numBarsToHandle = 2;
    public float handleSpeed;
    public float attackDistance;
    public TFRAxisCompare depthAxis = TFRAxisCompare.Z;
    public TFRAxisCompare widthAxis = TFRAxisCompare.X;
    public GameObject ball;
```

```
    public GameObject[] bars;
    List<GameObject>[] pieceList;
    // next
}
```

13. Implement the `Awake` function for initializing the piece list:

```
void Awake()
{
    int numBars = bars.Length;
    pieceList = new List<GameObject>[numBars];
    for (int i = 0; i < numBars; i++)
    {
        pieceList[i] = new List<GameObject>();
    }
}
```

14. Start implementing the `Update` function:

```
void Update()
{
    int[] currBars = GetNearestBars();
    Vector3 ballPos = ball.transform.position;
    Vector3 barsPos;
    int i;
    // next steps
}
```

15. Define the status for each bar, depending on the ball's position:

```
for (i = 0; i < currBars.Length; i++)
{
    GameObject barObj = bars[currBars[i]];
    TFRBar bar = barObj.GetComponent<TFRBar>();
    barsPos = barObj.transform.position;
    float ballVisible = Vector3.Dot(barsPos, ballPos);
    float dist = Vector3.Distance(barsPos, ballPos);
    if (ballVisible > 0f && dist <= attackDistance)
        bar.SetState(TFRState.ATTACK, handleSpeed);
    else if (ballVisible > 0f)
        bar.SetState(TFRState.DEFEND);
    else
        bar.SetState(TFRState.OPEN);
}
```

16. Implement the `OnGUI` function. This will handle the prediction at 30 frames per second:

```
public void OnGUI()
{
    Predict();
}
```

17. Define the prediction function with its member values:

```
private void Predict()
{
    Rigidbody rb = ball.GetComponent<Rigidbody>();
    Vector3 position = rb.position;
    Vector3 velocity = rb.velocity.normalized;
    int[] barsToCheck = GetNearestBars();
    List<GameObject> barsChecked;
    GameObject piece;
    barsChecked = new List<GameObject>();
    int id = -1;
    // next steps
}
```

18. Define the main loop for checking the ball's trajectory:

```
do
{
    RaycastHit[] hits = Physics.RaycastAll(position, velocity.
normalized);
    RaycastHit wallHit = null;
    foreach (RaycastHit h in hits)
    {
        // next steps
    }

} while (barsChecked.Count == numBarsToHandle);
```

19. Get the object of the collision and check whether it is a bar and whether it has been checked already:

```
GameObject obj = h.collider.gameObject;
if (obj.CompareTag(tagWall))
    wallHit = h;
if (!IsBar(obj))
    continue;
if (barsChecked.Contains(obj))
    continue;
```

20. Check, if it is a bar, whether it is among those closest to the ball:

```
bool isToCheck = false;
for (int i = 0; i < barsToCheck.Length; i++)
{
    id = barsToCheck[i];
    GameObject barObj = bars[id];
    if (obj == barObj)
    {
        isToCheck = true;
        break;
    }
}
if (!isToCheck)
    continue;
```

21. Get the bar collision point and calculate the movement for blocking the ball with the closest piece:

```
Vector3 p = h.point;
piece = GetNearestPiece(h.point, id);
Vector3 piecePos = piece.transform.position;
float diff = Vector3.Distance(h.point, piecePos);
obj.GetComponent<TFRBar>().Slide(diff, handleSpeed);
barsChecked.Add(obj);
```

22. Otherwise, recalculate with the wall's hitting point:

```
C
```

23. Create the function for setting the pieces to the proper bar:

```
void SetPieces()
{
    // next steps
}
```

24. Create a dictionary for comparing the pieces' depth:

```
// Create a dictionary between z-index and bar
Dictionary<float, int> zBarDict;
zBarDict = new Dictionary<float, int>();
int i;
```

25. Set up the dictionary:

```
for (i = 0; i < bars.Length; i++)
{
    Vector3 p = bars[i].transform.position;
```

```
        float index = GetVectorAxis(p, this.depthAxis);
        zBarDict.Add(index, i);
    }
```

26. Start mapping the pieces to the bars:

```
    // Map the pieces to the bars
    GameObject[] objs = GameObject.FindGameObjectsWithTag(tagPiece);
    Dictionary<float, List<GameObject>> dict;
    dict = new Dictionary<float, List<GameObject>>();
```

27. Assign pieces to their proper dictionary entry:

```
    foreach (GameObject p in objs)
    {
        float zIndex = p.transform.position.z;
        if (!dict.ContainsKey(zIndex))
            dict.Add(zIndex, new List<GameObject>());
        dict[zIndex].Add(p);
    }
```

28. Define the function for getting a bar's index, given a position:

```
    int GetBarIndex(Vector3 position, TFRAxisCompare axis)
    {
        // next steps
    }
```

29. Validate it:

```
    int index = 0;
    if (bars.Length == 0)
        return index;
```

30. Declare the necessary member values:

```
    float pos = GetVectorAxis(position, axis);
    float min = Mathf.Infinity;
    float barPos;
    Vector3 p;
```

31. Traverse the list of bars:

```
    for (int i = 0; i < bars.Length; i++)
    {
        p = bars[i].transform.position;
        barPos = GetVectorAxis(p, axis);
        float diff = Mathf.Abs(pos - barPos);
        if (diff < min)
        {
```

```
            min = diff;
            index = i;
        }
    }
```

32. Retrieve the found index:

```
return index;
```

33. Implement the function for calculating the vector axis:

```
float GetVectorAxis(Vector3 v, TFRAxisCompare a)
{
    if (a == TFRAxisCompare.X)
        return v.x;
    if (a == TFRAxisCompare.Y)
        return v.y;
    return v.z;
}
```

34. Define the function for getting the nearest bars to the ball:

```
public int[] GetNearestBars()
{
    // next steps
}
```

35. Initialize all the necessary member variables:

```
int numBars = Mathf.Clamp(numBarsToHandle, 0, bars.Length);
Dictionary<float, int> distBar;
distBar = new Dictionary<float, int>(bars.Length);
List<float> distances = new List<float>(bars.Length);
int i;
Vector3 ballPos = ball.transform.position;
Vector3 barPos;
```

36. Traverse the bars:

```
for (i = 0; i < bars.Length; i++)
{
    barPos = bars[i].transform.position;
    float d = Vector3.Distance(ballPos, barPos);
    distBar.Add(d, i);
    distances.Add(d);
}
```

37. Sort the distances:

```
distances.Sort();
```

38. Get the distances and use the dictionary in an inverse way:

```
int[] barsNear = new int[numBars];
for (i = 0; i < numBars; i++)
{
    float d = distances[i];
    int id = distBar[d];
    barsNear[i] = id;
}
```

39. Retrieve the bar IDs:

```
return barsNear;
```

40. Implement the function for checking whether a given object is a bar:

```
private bool IsBar(GameObject gobj)
{
    foreach (GameObject b in bars)
    {
        if (b == gobj)
            return true;
    }
    return false;
}
```

41. Start implementing the function for retrieving the closest piece of a bar, given a position:

```
private GameObject GetNearestPiece(Vector3 position, int barId)
{
    // next steps
}
```

42. Define the necessary member variables:

```
float minDist = Mathf.Infinity;
float dist;
GameObject piece = null;
```

43. Traverse the list of pieces and calculate the closest one:

```
foreach (GameObject p in pieceList[barId])
{
    dist = Vector3.Distance(position, p.transform.position);
    if (dist < minDist)
    {
        minDist = dist;
        piece = p;
    }
}
```

44. Retrieve the piece:

```
return piece;
```

How it works...

The table-football competitor draws on the skills developed from the air-hockey rival. This means casting rays to get the trajectory of the ball and moving the nearest bar considering the pieces. It also moves the bar, depending on whether the rival is attacking or defending, so that it can block the ball or let it go further.

See also

- The *Seeing using a collider-based system* recipe in *Chapter 5*, *Agent Awareness*

Creating mazes procedurally

This is a completely new recipe oriented toward having fun while creating maps and levels procedurally. The main recipe works by creating a maze completely procedurally. Furthermore, we will explore a gray area, where both level design and procedurally generated content meet.

Getting ready

In this recipe, it is important to understand the concepts of Binary Space Partitioning and the Breadth-first Search algorithm learned in *Chapter 2*, *Navigation*.

How to do it...

We will implement two classes, one for the nodes to be partitioned and one for holding all the nodes and the maze representation, as follows:

1. Create the `BSPNode` class and its members:

```
using UnityEngine;

[System.Serializable]
public class BSPNode
{
    public Rect rect;
    public BSPNode nodeA;
    public BSPNode nodeB;
}
```

2. Implement the class constructor:

```
public BSPNode(Rect rect)
{
    this.rect = rect;
    nodeA = null;
    nodeB = null;
}
```

3. Define the function for splitting the node into two subregions:

```
public void Split(float stopArea)
{
    // next steps
}
```

4. Validate its base case:

```
if (rect.width * rect.height >= stopArea)
    return;
```

5. Initialize all the necessary function variables:

```
bool vertSplit = Random.Range(0, 1) == 1;
float x, y, w, h;
Rect rectA, rectB;
```

6. Compute the horizontal split:

```
if (!vertSplit)
{
    x = rect.x;
    y = rect.y;
    w = rect.width;
    h = rect.height / 2f;
    rectA = new Rect(x, y, w, h);
    y += h;
    rectB = new Rect(x, y, w, h);
}
```

7. Compute the vertical split:

```
else
{
    x = rect.x;
    y = rect.y;
    w = rect.width / 2f;
    h = rect.height;
```

```
        rectA = new Rect(x, y, w, h);
        x += w;
        rectB = new Rect(x, y, w, h);
    }
```

8. Create the class for handling the dungeon and declare all its member variables:

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

public class Dungeon : MonoBehaviour
{
    public Vector2 dungeonSize;
    public float roomAreaToStop;
    public float middleThreshold;
    public GameObject floorPrefab;

    private BSPNode root;
    private List<BSPNode> nodeList;
}
```

9. Implement the function for splitting:

```
public void Split()
{
    float x, y, w, h;
    x = dungeonSize.x / 2f * -1f;
    y = dungeonSize.y / 2f * -1f;
    w = dungeonSize.x;
    h = dungeonSize.y;
    Rect rootRect = new Rect(x, y, w, h);
    root = new BSPNode(rootRect);
}
```

10. Implement the function for drawing the maze using the nodes:

```
public void DrawNode(BSPNode n)
{
    GameObject go = Instantiate(floorPrefab) as GameObject;
    Vector3 position = new Vector3(n.rect.x, 0f, n.rect.y);
    Vector3 scale = new Vector3(n.rect.width, 1f, n.rect.height);
    go.transform.position = position;
    go.transform.localScale = scale;
}
```

How it works...

We divided the maze into two big data structures. The logical side that is handled via the BSP nodes and the visual and construction representation handled by the main `Maze` class. The idea behind this representation is to divide the space twice as many times as necessary until a condition is met. This is the Binary Space Partitioning.

We then created rooms for the leave nodes, and finally, we connected the regions on the tree from the bottom to the top (leaves to root).

There's more...

- There's another technique that is a little bit simpler, but it requires more input from the art or level-design team. It creates a level with BFS using random pieces in a list and connects them.
- The pieces can be rotated.
- It can be improved by using the random function learned previously and tuning the pieces' placement on the list.

See also

- The *Finding the shortest path in a grid with BFS* recipe in *Chapter 2*, *Navigation*

Implementing a self-driving car

What fun is a racing game without competitors? This is one of the most difficult subjects in artificial intelligence for games. It is usually tackled by creating *cheater* agents that disable certain limitations that are always imposed on the player, such as physics behaviors; this is because these limitations can create erratic or imprecise behaviors when evaluated by AI. In our case, we will approach the problem organically using techniques from a previous chapter.

Getting ready

In this chapter, we will explore how to create an autonomous car using advanced techniques from *Chapter 1*, *Movement*, such as following a path and avoiding walls. So, it is important to have grasped the knowledge behind them.

How to do it...

1. Create an empty **GameObject**.
2. Attach the **Agent** component.
3. Attach the **FollowPath** component.
4. Attach the **WallAvoid** component.
5. Create the track using the track pieces with the **PathNode** component.
6. Tag the track borders as walls.
7. Make sure the track is complete.

How it works...

By working with the system from the previous chapters, we can easily create a simple, yet flexible, system to create intelligent cars.

See also

- The *Following a path* and *Avoiding walls* recipes in *Chapter 1*, *Movement*

Managing race difficulty using a rubber-banding system

We usually want to create experiences that adapt to the player, and racing games are a good field for this, given that there is this gap of the cheater agent.

In this case, we will explore a middle ground for this using a framework that allows you to come up with your own heuristic for managing the speed of the vehicle given its status. It doesn't matter if it is an arcade racing game or simulation; the framework aims to work in a similar fashion for both the cases.

Getting ready

It is important to have grasped the basic skills in *Chapter 1*, *Movement*, in order to be able to develop a strategy to extend the framework for your own needs—that is, understanding the principles of how the agent class works and how the behaviors help the player move toward an object. In a nutshell, we are talking about vector operations.

How to do it...

We will implement three different classes for handling low-level and high-level AIs as follows:

1. Create the class for the basic rival agent:

```
using UnityEngine;

public class RacingRival : MonoBehaviour
{
    public float distanceThreshold;
    public float maxSpeed;
    public Vector3 randomPos;
    protected Vector3 targetPosition;
    protected float currentSpeed;
    protected RacingCenter ghost;
}
```

2. Implement the `Start` function:

```
void Start()
{
    ghost = FindObjectOfType<RacingCenter>();
}
```

3. Define the `Update` function for handling the target position to follow:

```
public virtual void Update()
{
    targetPosition = transform.position + randomPos;
    AdjustSpeed(targetPosition);
}
```

4. Define your function for adjusting the speed accordingly:

```
public virtual void AdjustSpeed(Vector3 targetPosition)
{
    // TODO
    // your own behaviour here
}
```

5. Create the class for handling the ghost rider or an invincible racer:

```
using UnityEngine;

public class RacingCenter : RacingRival
{
    public GameObject player;
}
```

6. Implement the initial function for finding its target:

```
void Start()
{
    player = GameObject.FindGameObjectWithTag("Player");
}
```

7. Override the `Update` function, so the invincible car can adapt to the player's behavior:

```
public override void Update()
{
    Vector3 playerPos = player.transform.position;
    float dist = Vector3.Distance(transform.position,
    playerPos);
    if (dist > distanceThreshold)
    {
        targetPosition = player.transform.position;
        base.Update();
    }
}
```

8. Implement its special behavior:

```
public override void AdjustSpeed(Vector3 targetPosition)
{

    // TODO
    // Use in case the base behaviour also applies
    base.AdjustSpeed(targetPosition);
}
```

9. Create the class for handling the high-level AI:

```
using UnityEngine;

public class Rubberband : MonoBehaviour
{
    RacingCenter ghost;
    RacingRival[] rivals;
}
```

10. Assign each racer its random position in the rubber system. We are using a circular rubber band in this case:

```
void Start()
{
    ghost = FindObjectOfType<RacingCenter>();
    rivals = FindObjectsOfType<RacingRival>();
```

```
            foreach (RacingRival r in rivals)
            {
                if (ReferenceEquals(r, ghost))
                    continue;
                r.randomPos = Random.insideUnitSphere;
                r.randomPos.y = ghost.transform.position.y;
            }
        }
```

How it works...

The high-level AI rubber system assigns the positions to be held by the racers. Each racer has its own behavior for adjusting speed, especially the invincible racer. This agent works as the center of the mass of the rubber band. If its dance from the player exceeds the threshold, it will adapt. Otherwise, it'll stay just the same, wobbling.