3
Random and Probability

In this chapter, we are going to look at how the concepts of probability can be applied to game AI. This chapter will be more about generic game AI development techniques in random and probability topics, and less about Unity3D in particular. Moreover, they can be applied to any game development middleware or technology framework. We'll be using mono C# in Unity3D for the demos mainly using the console to output data, and won't address much about the specific features of the Unity3D engine and the editor itself.

Game developers use probability or the confidence factor to add a little uncertainty to the behaviors of AI characters, as well as to the game world. This makes the artificial intelligence system a bit unpredictable of a certain outcome, and can provide the players with a more exciting and challenging experience.

Let us take an example of a typical soccer game. One of the rules of a soccer game is to award a direct free kick if one player is fouled while trying to possess the ball from the opposing team. Now, instead of giving a foul and a free kick all the time whenever that foul happens, the game developer can apply a probability so that only 98 percent of all the fouls will be rewarded with a direct free kick. As a result, most of the time, the player will get a direct free kick. But when that remaining two percent happens, like you have hit the other player and you know it's going to be a free kick but the referee passes it, it can provide a certain emotional feedback to the players from both the teams (assuming that you are playing against another human). The other player would feel angry and disappointed, while you'd feel lucky and satisfied. In the end, the referees are human, and like all other humans, they might not be 100 percent correct all the time.

So we use probability in a game AI to make the game and characters livelier and seem more realistic, by not making the same decision or taking the same action again and again. There are many topics to discuss and debate in the probability domain. So this small single chapter will only be able to address the basic concepts, and how we can implement some of them in Unity3D.

In this chapter, we will be going over random and probability. We will be creating a simple dice game. We will also give some application examples of probability and dynamic AI. Finally, we will finish the chapter with a simple slot machine, and then add on more probability features.

Random

Probability is basically a measure of how likely it is that a particular condition or a favorable outcome can be achieved among all the possible outcomes, if selected randomly. So speaking of probability, one can't neglect the importance of randomness. **Random number generation** (**RNG**) is very important when we need to produce unpredictable results. The simplest and probably the oldest technique would be throwing a dice to generate a random value between one and six. The random numbers are produced computationally by a **pseudorandom number generator** (**PRNG**), and they determine the same sequence of random numbers based on the initial seed value. So, if we theoretically know the seed value, we can regenerate the same sequence of random numbers again, and thus they are not considered as truly random. The seed value is usually generated from the state of the computer system, such as the elapsed time in milliseconds since the computer starts running. Some RNGs are more random than others. If we were creating an encryption program, we would want to look into a more random RNG. For the games we will be making, the RNG that comes with Unity will suffice. Now let's see how we can generate random numbers in Unity3D.

Random class

The Unity3D script has a `Random` class to generate random data. Two of the most widely used properties would be `seed` and `value`:

```
static var seed : int
```

You can set this `seed` property of the `Random` class to seed the random number generator. Usually, we will not want to seed the same value again and again, as this will result in the same predictable sequence of random numbers being generated. One of the reasons for keeping the same seed value is for testing purposes:

```
static var value : float
```

You can read the `Random.value` property to get a random number between `0.0` (inclusive) and `1.0` (inclusive). Both `0.0` and `1.0` may be returned by this property. Another class method that could be quite handy is the `Range` method.

```
static function Range (min : float, max : float) : float
```

The `Range` method can be used to generate a random number from a range. When given an `integer value`, it returns a random `integer` number between `min` (inclusive) and `max` (exclusive). This means that a zero may be returned, but never a one. If you pass in float values for the range, it'll return a random `float` number between `min` (inclusive) and `max` (inclusive). Take note of the exclusive and inclusive parts. Since the `integer` random value is exclusive of `max` in range, we'll need to pass in `n+1` as the `max` range, where `n` is our desired maximum random integer. However, for the `float` random value, the max value in range is inclusive.

Simple random dice game

Let's set up a very simple dice game in a new scene, where a random number is being generated between one and six, and checked against the input value. The player will win, if the input value matches the dice result generated randomly as shown in the following `DiceGame.cs` file:.

```
    using UnityEngine;
    using System.Collections;

public class DiceGame : MonoBehaviour {

    public string inputValue = "1";

    void OnGUI() {
      GUI.Label(new Rect (10, 10, 100, 20), "Input: ");
        inputValue = GUI.TextField(new Rect(120, 10, 50, 20),
          inputValue, 25);
        if (GUI.Button(new Rect(100,50,50,30),"Play")) {
          Debug.Log("Throwing dice...");
          Debug.Log("Finding random between 1 to 6...");
          int diceResult = Random.Range(1,7);
          Debug.Log("Result: " + diceResult);
        if (diceResult == int.Parse(inputValue)) {
          guiText.text = "DICE RESULT: " +
            diceResult.ToString() + "\r\nYOU WIN!";
          }
        else {
          guiText.text = "DICE RESULT: " +
            diceResult.ToString() + "\r\nYOU LOSE!";
          }
        }
    }
}
```

We implement this simple dice game in the `OnGUI()` method as we want to render some GUI controls such as a label, a text field to enter the input value, and a button to play. The `guiText` object will be used to display the result. Add a `guiText` to the scene, navigate to **Game Object | Create Other | GUI Text**, and add our script to the object. The output that you get if you run the game is shown in the following screenshot:

Simple dice game results

This is a purely random game, and there's no modified probability involved. Each side of the surface of the dice has an equal chance to be picked.

Definition of probability

There are many ways to define probability based on the situations and the domain context. The most commonly used notion of probability is to refer the possibility of an event to successfully occur. The probability of an event `A` to occur is usually written as `P(A)`. To calculate `P(A)` we need to know the number of ways or times it can occur `(n)`, and the total number of times all the other possible events can occur `(N)`.

So the probability of an event `A` can be calculated as

```
P(A) = n / N
```

`P(A)` is the probability of the event `A` to occur, and it's equal to the number of ways that `A` can occur `(n)` out of the number of all outcomes `(N)`. If `P(A)` is the probability of the event `A` to successfully occur, then the probability of the event `A` will not occur, or the probability of failure for event `A` is equal to:

```
Pf (A) = 1 - P.(A)
```

The range of probability is a decimal number from zero to one. Probability of zero means there's no chance for the desired event to occur, and one means that it's 100 percent certain for the event to occur. And `P(A) + Pf (A)` must equal to one. Since the probability values range from zero to one, we can get the percentage value by multiplying by 100.

Independent and related events

Another important concept in probability is whether the chance of a particular event to occur depends on any other event in some ways or not. For example, throwing a six-sided dice twice are two independent events. Each time you throw a dice, the probability of each side to turn up is one-sixth. On the other hand, drawing two cards from the same deck are two related events. If you drew a Jack in the first event, there's one less chance that you can get another Jack in the second event.

Conditional probability

When throwing two six-sided dices at the same time, what is the probability of getting a one on both the dices? Here there are two conditional events; to get one on the first dice, and also to get one on the second dice. They rely on each other to calculate the probability of getting one on both the dices. The probability to get one on the first dice is one-sixth, and also for the second dice is one-sixth. So the answer would be one-sixth times one-sixth which is 1/36, or a 2.8 percent chance.

Now let's consider another example, what's the probability that the sum of the numbers show up on two dices is equal to two? Since there's only one way to get this sum, which is one and one, the probability is still the same as getting the same number on both dices. In that case it would be still 1/36.

But how about getting the sum of the numbers that show up on the two dices to seven? As you can see, there are a total of six possible chances to get a total of seven, from the following table:

Dice 1	Dice 2
1	6
2	5
3	4
4	3
5	2
6	1

So the probability of getting a sum of seven from two dices is 6/36 or one-sixth, which is 16.7 percent. These are some examples of conditional probability, where two events rely on each other to achieve a desirable outcome.

A loaded dice

Now let's assume we haven't been that honest, and our dice is loaded so that the side of the number six has a double chance of landing facing upward. For a six-sided dice, the probability of each side facing upward is approximately one-sixth (`0.17`). Since we doubled the chance of getting six, we need to double the probability of getting six, let's say up to `0.34`. And the probability of the remaining five sides will be reduced to `0.132`.

The simplest way to implement this loaded dice algorithm is to generate a random value between `1` and `100`. Check if the random value is in a range of one to 35. If so `return 6`, otherwise get a random dice value between one and five, since these values have the same probability of `0.13`.

So here's our `throwLoadedDice()` method:

```
int throwDiceLoaded() {
  Debug.Log("Throwing dice...");
    int randomProbability = Random.Range(1,101);
    int diceResult = 0;
    if (randomProbability < 36) {
      diceResult = 6;
    }
    else {
      diceResult = Random.Range(1,5);
    }
  Debug.Log("Result: " + diceResult);
    return diceResult;
}
```

If we test our new loaded dice algorithm by throwing the dice multiple times, you'll notice that the value `6` yields more than usual. Here is our new `OnGUI()` function:

```
void OnGUI() {
  GUI.Label(new Rect (10, 10, 50, 20), "Input: ");
  inputValue = GUI.TextField(new Rect(60, 10, 50, 20),
    inputValue, 25);
    if (GUI.Button(new Rect(60,40,50,30),"Play")) {
      int totalSix = 0;
        for (int i=0;i<10;i++) {
          int diceResult = throwDiceLoaded();
          if (diceResult == 6) totalSix++;
          if (diceResult == int.Parse(inputValue)) {
            guiText.text = "DICE RESULT: " +
              diceResult.ToString()+"\r\nYOU WIN!";
          }
```

```
        else {
          guiText.text = "DICE RESULT: " +
            diceResult.ToString()+"\r\nYOU LOSE!";
              }
            }
          Debug.Log("Total of six: " + totalSix.ToString());
        }
      }
```

We throw the dice ten times in our `OnGUI()` method, and here I got `6` at least two to three times (which is approximately 33 percent of ten times). But, if you normally throw the dice without any loaded probability it's more possible that you won't get any `6` at all. Keep in mind that the value `6` is only favored for 35 percent, and thus there's still a chance that you will never get a `6` out of ten dice throws, though it's quite unlikely.

Character personalities

We can also use different probabilities to specify the in-game characters' specialties. Let's pretend we designed a game proposal for a population management game for the local government. We need to address and simulate issues like taxation versus global talent attraction, immigration versus social cohesion, and so on. We have three types of characters in our proposal, namely, workers, scientists, and professionals. Their efficiencies in performing the particular tasks are defined in the following table:

Characters	Construction	R&D	Corporate Jobs
Worker	95	2	3
Scientist	5	85	10
Professional	10	10	80

Let us take a look at how we can implement this mechanic. Let's say the player needs to build new houses to accommodate the increased population. A house construction would require 1,000 units of workload to finish. We use the value specified earlier as the workload that can be done per second per unit type for a particular task. So if you're building a house with one worker that will only take about 10 seconds to finish the construction (`1000/95`) whereas it'll take more than three minutes if you are trying to build with the scientists (`1000/5 = 200 seconds`). The same will be true for other tasks such as R&D and corporate jobs. These factors can be adjusted/enhanced later as the game progresses, making some of the early level tasks become simpler, and takes less time.

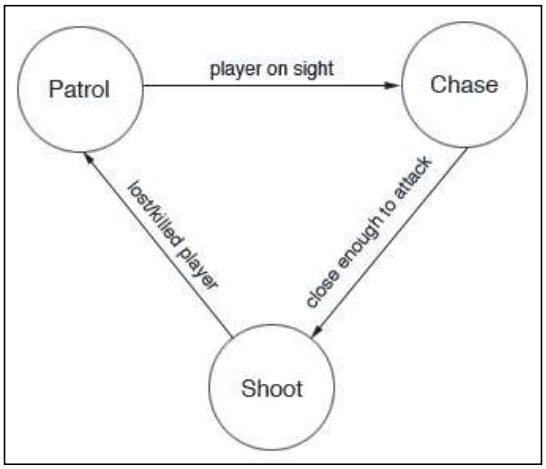
Then we introduce special items that could be discovered by the particular unit type. Now, we don't want to give these items every time a particular unit has done its tasks. Instead we want to reward the player as a surprise. So we associate the probability of finding such items according to the unit type, as described in the following table:

Special items	Worker	Scientist	Professional
Raw materials	0.3	0.1	0.0
New tech	0.0	0.3	0.0
Bonus	0.1	0.2	0.4

The preceding table means there's a 30 percent chance that a worker will find some raw materials, and a 10 percent chance to earn bonus income whenever they have built a factory or a house. This allows the players to anticipate the possible upcoming rewards, once they've done some tasks. This can make the game more fun because the players will not know the outcome of the event.

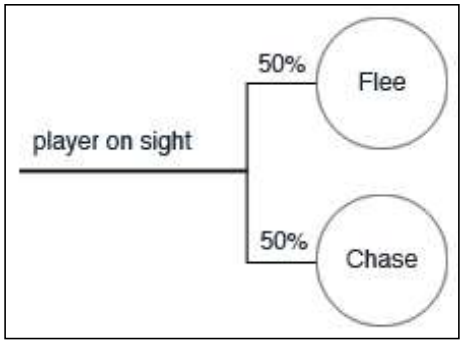
FSM with probability

We discussed Finite State Machines (FSM) in *Chapter 2*, *Finite State Machines*, using both simple switch statements as well as using the FSM framework. The decision to choose which state to execute was purely based on true or false value of a given condition. Remember the following FSM of our AI controlled tank entity?



Tank AI FSM

To make the AI more interesting, and a little bit unpredictable, we can give our tank entity some options with probabilities to choose from, instead of doing the same thing whenever a certain condition is met. For example, in our earlier FSM, our AI tank will chase the player tank once the player is in its line of sight. Instead we can give our AI another state, such as flee with some probability such as 50 percent as shown in the following figure:



FSM using probability

Now instead of chasing every time, the AI tank spots the player; there's a 50 percent chance that it'll flee, and maybe report to the headquarters or something. We can implement this mechanic the same way we did with our previous dice example. First we need to generate a random value between one and 100, and see if the value lies between one and 50 or 51 and 100. (Or we could randomly choose between zero and one.) Then choose a state accordingly. The other way to implement this is to fill an array with these options in proportion to their respective probabilities. Then pick a random state from this pool as if you were drawing a lottery winner. Let's see how to use this technique as shown in the following `FSM.cs` file:

```
using UnityEngine;
using System.Collections;

public class FSM : MonoBehaviour {
  public enum FSMState {
    Chase,
    Flee
  }

  public int chaseProbabiilty = 50;
  public int fleeProbabiilty = 50;

  //a poll to store the states according to their
    //probabilities
  public ArrayList statesPoll = new ArrayList();
```

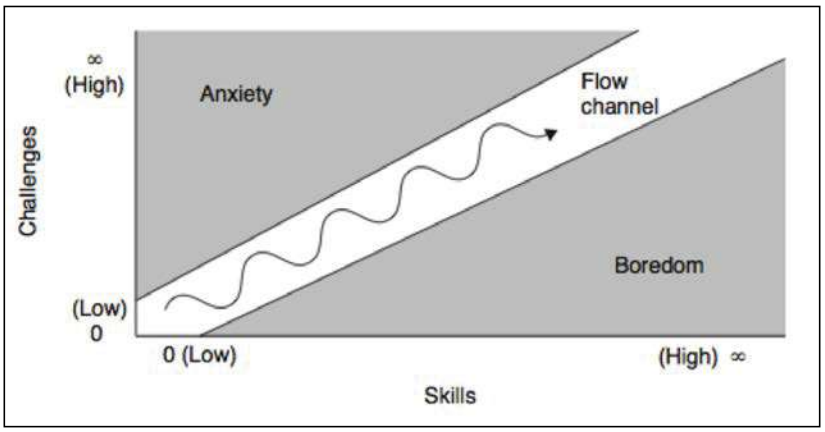
```
    void Start () {
      //fill the array
        for (int i = 0; i < chaseProbabiilty; i++) {
          statesPoll.Add(FSMState.Chase);
        }
        for (int i = 0; i < fleeProbabiilty; i++) {
          statesPoll.Add(FSMState.Flee);
        }
      }

    void OnGUI() {
        if (GUI.Button(new Rect(10,10,150,40),
          "Player on sight")) {
           int randomState = Random.Range(0, statesPoll.Count);
           Debug.Log(statesPoll[randomState].ToString());
        }
      }
}
```

In our `OnGUI()` method, when you click on the mouse button, we just choose one random item from our `statesPoll` array. Obviously, the one with more entries in the poll will have a higher chance to be selected. Try it out.

Dynamic AI

We can also use probability to specify the intelligence levels of AI characters, and the global game settings. This can in turn affect the overall difficulty level of the game, and keep it challenging and interesting enough to players. As described in the book, *The Art of Game Design*, *Jesse Schell*, *Morgan Kaufmann publications*, players will only continue to play our game if we keep them in their flow channel.



The Flow Channel

The players will feel anxious and get disappointed if we present tough challenges for them to solve before they have the necessary skills. On the other hand, once they've mastered the skills, and if we continue to keep the game at the same pace, then they will get bored. The grey area that can keep the players engaged for a long time is between these two extremes of hard and easy, which the original author referred to as the flow channel. To keep the players in the flow channel, the game designers need to feed the challenges and missions that match with the progressive skills that the players have acquired over time. However, it is not an easy task to find a value that works for all players, since the pace of learning and the expectations can be different individually.

One way to tackle this problem is to collect the player attempts and results during the game-play sessions, and to adjust the probability of the opponent's AI accordingly. Though this approach is supposed to help the games to be more engaging, there are many other players who don't like this approach, since this method takes away the pride and satisfaction of finishing a hard game. After all, beating a very hard boss AI character despite all the challenges can be much more rewarding and satisfying than winning the game because the AI is dumb. They would feel much worse if they find out that the AI becomes dumb because they don't have enough skills to match. So we must be careful about when we want to apply this technique in our games.

Demo slot machine

In this final demo, we'll design and implement a slot machine game with 10 symbols and three reels. Just to make it simple we'll just use the numbers from zero to nine as our symbols. Many slot machines would use fruit shapes and other simple shapes, such as bells, stars, and letters. Some other slot machines usually use a specific theme based on popular movies or TV programs as a franchise. Since there are 10 symbols and three reels, that's a total of 1,000 (10^3) possible combinations.

Random slot machine

This random slot machine demo is similar to our previous dice example. This time we are going to generate three random numbers for three reels. The only payout will be when you get three of the same symbols on the payline. To make it simpler, we'll only have one line to play against in this demo. And if the player wins, the game will return 500 times the bet amount.

We'll set up our scene with four GUI text objects to represent the three reels, and the result message.

Our GUI text objects

This is how our new script looks, as shown in the following `SlotMachine.cs` file::

```
using UnityEngine;
using System.Collections;

public class SlotMachine : MonoBehaviour {

  public float spinDuration = 2.0f;
  public int numberOfSym = 10;
  private GameObject betResult;

  private bool startSpin = false;
  private bool firstReelSpinned = false;
  private bool secondReelSpinned = false;
  private bool thirdReelSpinned = false;

  private string betAmount = "100";

  private int firstReelResult = 0;
  private int secondReelResult = 0;
  private int thirdReelResult = 0;

  private float elapsedTime = 0.0f;

    //Use this for initialization
  void Start () {
    betResult = gameObject;
    betResult.guiText.text = "";
  }
```

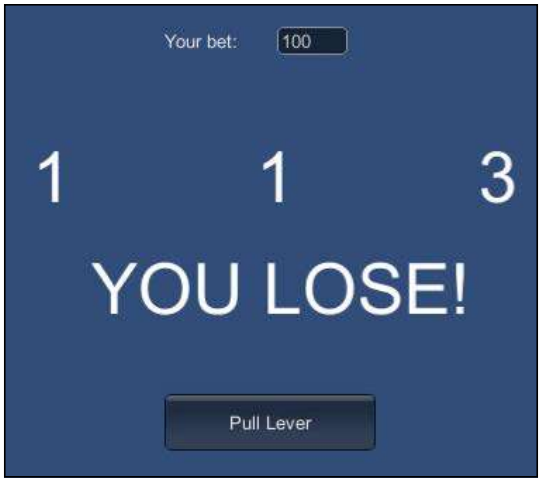
```
void OnGUI() {
  GUI.Label(new Rect(200, 40, 100, 20), "Your bet: ");
  betAmount = GUI.TextField(new Rect(280, 40, 50, 20),
    betAmount, 25);
    if (GUI.Button(new Rect(200, 300, 150, 40),
      "Pull Liver")) {
    Start();
      startSpin = true;
    }
  }

void checkBet() {
  if (firstReelResult == secondReelResult &&
    secondReelResult == thirdReelResult) {
      betResult.guiText.text = "YOU WIN!";
    }
  else {
    betResult.guiText.text = "YOU LOSE!";
  }
}

  //Update is called once per frame
  void FixedUpdate () {
    if (startSpin) {
      elapsedTime += Time.deltaTime;
      int randomSpinResult = Random.Range(0,
        numberOfSym);
    if (!firstReelSpinned) {
      GameObject.Find("firstReel").guiText.text =
        randomSpinResult.ToString();
    if (elapsedTime >= spinDuration) {
      firstReelResult = randomSpinResult;
      firstReelSpinned = true;
      elapsedTime = 0;
    }
  }
    else if (!secondReelSpinned) {
      GameObject.Find("secondReel").guiText.text =
        randomSpinResult.ToString();
    if (elapsedTime >= spinDuration) {
      secondReelResult = randomSpinResult;
      secondReelSpinned = true;
      elapsedTime = 0;
    }
```

```
        }
          else if (!thirdReelSpinned) {
            GameObject.Find("thirdReel").guiText.text =
              randomSpinResult.ToString();
          if (elapsedTime >= spinDuration) {
            thirdReelResult = randomSpinResult;
              startSpin = false;
              elapsedTime = 0;
              firstReelSpinned = false;
              secondReelSpinned = false;
            checkBet();
            }
          }
        }
      }
    }
```

Attach the script to our `betResult guiText` object, and then position the `guiText` element on the screen. We have a button called **Pull Lever** in the `OnGUI()` method that will set the `startSpin` flag to true when clicked. And in our `FixedUpdate()` method we generate a random value for each reel if the `startSpin` is true. Finally, once we've got the value for the third reel, then we reset the `startSpin` to false. While we are getting the random value for each reel, we also keep a track of how much time has elapsed, since the player pulled the lever. Usually in the real world slot machines, each reel would take three to five seconds before landing the result. Hence, we are also taking some time as specified in `spinDuration` before showing the final random value. If you play the scene and click on the **Pull Lever** button, you should see the final result as shown in the following screenshot:



Random slot game in action

Since your chance of winning is one out of 100, it becomes boring as you lose several times consecutively. And of course if you've ever played a slot machine, this is not how it works, or at least not anymore. Usually you can have several wins during your play. Even though these small wins don't recoup your principal bet, and in the long run most of the players go broke, the slot machines would render winning graphics and winning sounds, which researchers referred to as losses disguised as wins.

So instead of just one single way to win—winning the jackpot—we'd like to modify the rules a bit so that it pays out smaller returns during the play session.

Weighted probability

Real slot machines have something called a **Paytable and Reel Strips** (**PARS**) sheet, which is like the complete design document of the machine. The PARS sheet is used to specify what the payout percentage is, what the winning patterns, and what their prizes are, and so on. Obviously the number of the payout prizes and the frequencies of such wins need to be carefully selected, so that the house (slot machine) can collect the fraction of the bets over time, while making sure to return the rest to the players to make the machine attractive to play. This is known as payback percentage or **return to player** (**RTP**). For example, a slot machine with a 90 percent RTP means that over time the machine will return an average of 90 percent of all the bets to the players.

In this demo, we'll not be focusing on choosing the optimal value for the house to yield specific wins over time nor maintaining a particular payback percentage, but rather to demonstrate weighting probability to specific symbols, so that they show up more times than usual. So let's say we'd like to make the symbol zero to appear 20 percent more than by chance on the first and third reel, and return a small payout of half of the bet. In other words, a player will only lose half of their bet if they got zero symbols on the first and third reels, essentially disguising a loss as a small win. Currently, the zero symbol has a probability of 1/10 (`0.1`) or 10 percent probability to occur. Now we'll make it 30 percent for zero to land on the first and third reels as shown in the following `SlotMachineWeighted.cs` file:

```
using UnityEngine;
using System.Collections;

public class SlotMachineWeighted : MonoBehaviour {
  public float spinDuration = 2.0f;
  public int numberOfSym = 10;
  public GameObject betResult;

  private bool startSpin = false;
  private bool firstReelSpinned = false;
```

```
private bool secondReelSpinned = false;
private bool thirdReelSpinned = false;

private int betAmount = 100;

private int creditBalance = 1000;
private ArrayList weightedReelPoll = new ArrayList();
private int zeroProbability = 30;

private int firstReelResult = 0;
private int secondReelResult = 0;
private int thirdReelResult = 0;

private float elapsedTime = 0.0f;
```

New variable declarations are added, such as `zeroProbability` to specify the probability percentage of the zero symbol to land on the first and third reels. The `weightedReelPoll` array list will be used to fill with all the symbols (zero to nine) according to their distribution, so that we can later pick one randomly from the poll like we did in our earlier FSM example. And then we initialize the list in our `Start()` method as shown in the following code:

```
void Start () {
  betResult = gameObject;
  betResult.guiText.text = "";
    for (int i = 0; i < zeroProbability; i++) {
      weightedReelPoll.Add(0);
    }
  nt remainingValuesProb = (100 - zeroProbability)/9;
    for (int j = 1; j < 10; j++) {
      for (int k = 0; k < remainingValuesProb; k++) {
        weightedReelPoll.Add(j);
      }
    }
  }

void OnGUI() {
  GUI.Label(new Rect(150, 40, 100, 20), "Your bet: ");
  betAmount = int.Parse(GUI.TextField(new Rect(220, 40,
    50, 20), betAmount.ToString(), 25));
  GUI.Label(new Rect(300, 40, 100, 20), "Credits: " +
    creditBalance.ToString());
    if (GUI.Button(new Rect(200,300,150,40),"Pull Lever")) {
      betResult.guiText.text = "";
      startSpin = true;
    }
  }
```

And the following is our revised `checkBet()` method. Instead of just one jackpot win, we are now considering five conditions: jackpot, loss disguised as win, near miss, any two symbols matched on the first and third row, and of course the lose condition:

```
void checkBet() {
  if (firstReelResult == secondReelResult &&
    secondReelResult == thirdReelResult) {
    betResult.guiText.text = "JACKPOT!";
    creditBalance += betAmount * 50;
    }
  else if (firstReelResult ==0 && thirdReelResult ==0) {
    betResult.guiText.text = "YOU WIN" +
      (betAmount/2).ToString();
      creditBalance -= (betAmount/2);
    }
  else if (firstReelResult == secondReelResult) {
    betResult.guiText.text = "AWW... ALMOST JACKPOT!";
    }
  else if (firstReelResult == thirdReelResult) {
    betResult.guiText.text = "YOU WIN" +
      (betAmount*2).ToString();
      creditBalance -= (betAmount*2);
    }
  else {
    betResult.guiText.text = "YOU LOSE!";
       creditBalance -= betAmount;
    }
  }
```

In the `checkBet()` method, we designed our slot machine to return 50 times if they hit the jackpot, only to lose 50 percent of their bet, if the first and third reels are zero, and two times if the first and third reels are matched with any other symbol. And we generate values for the three reels in the `FixedUpdate()` method as shown in the following code:

```
void FixedUpdate () {
  if (!startSpin) {
    return;
  }
    elapsedTime += Time.deltaTime;
    int randomSpinResult = Random.Range(0,
      numberOfSym);
    if (!firstReelSpinned) {
       GameObject.Find("firstReel").guiText.text =
         randomSpinResult.ToString();
    if (elapsedTime >= spinDuration) {
      int weightedRandom = Random.Range(0,
        weightedReelPoll.Count);
        GameObject.Find("firstReel").guiText.text =
```

```
            weightedReelPoll[weightedRandom].ToString();
            firstReelResult =
              (int)weightedReelPoll[weightedRandom];
            firstReelSpinned = true;
            elapsedTime = 0;
          }
        }
      else if (!secondReelSpinned) {
        GameObject.Find("secondReel").guiText.text =
          randomSpinResult.ToString();
      if (elapsedTime >= spinDuration) {
        secondReelResult = randomSpinResult;
        secondReelSpinned = true;
        elapsedTime = 0;
      }
    }
```

For the first reel, during the spinning period, we really show the real random values. But once the time is up, we choose the value from our poll that is already populated with symbols according to the probability distributions. So our zero symbol would have 30 percent more chance of occurring than the rest, as shown in the following screenshot:

Loss disguised as a win

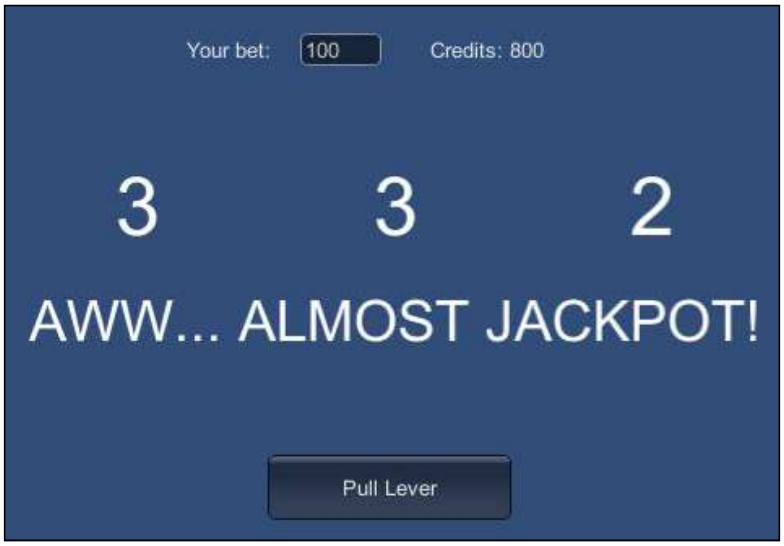
Actually the player is losing on his bets, if you get two zero symbols on the first and third reel. But we make it seem like a win. It's just a lame message here, but if we can combine it with nice graphics; maybe with fireworks, and nice winning sound effects, this can really work, and attract players to bet more, and pull that lever again and again.

Near miss

If the first and second reels return the same symbol, then we have to provide the near miss effect to the players by returning the random value to the third reel close to the second one. We can do this by checking the third random spin result first. If the random value is the same as the first and second results, then this is a jackpot, and we shouldn't alter the result. But if it's not, then we should modify the result so that it is close enough to the other two. Check the comments in the following code:

```
        else if (!thirdReelSpinned) {
          GameObject.Find("thirdReel").guiText.text =
            randomSpinResult.ToString();
        if (elapsedTime < spinDuration) {
          return;
        }
        if ((firstReelResult == secondReelResult)
          && randomSpinResult != firstReelResult) {
          randomSpinResult = firstReelResult - 1;
        if (randomSpinResult < firstReelResult)
          randomSpinResult = firstReelResult - 1;
        if (randomSpinResult > firstReelResult)
          randomSpinResult = firstReelResult + 1;
        if (randomSpinResult < 0) randomSpinResult = 9;
        if (randomSpinResult > 9) randomSpinResult = 0;
          GameObject.Find("thirdReel").guiText.text =
            randomSpinResult.ToString();
          thirdReelResult = randomSpinResult;
        }
        else {
          int weightedRandom = Random.Range(0,
            weightedReelPoll.Count);
          GameObject.Find("thirdReel").guiText.text =
            weightedReelPoll[weightedRandom].ToString();
            thirdReelResult =
              (int)weightedReelPoll[weightedRandom];
           }
           startSpin = false;
           elapsedTime = 0;
           firstReelSpinned = false;
           secondReelSpinned = false;
           checkBet();
        }
      }
  }
```

And if that "near miss" happens, you should see it as shown in the following screenshot:



A near miss

We can go even further by adjusting the probability in real-time based on the bet amount. But that'd be too creepy. Another thing we could add to our game is a check to make sure the player can't bet more money than they already have. Also, we could add a game over message that appears when the player has bet all their money.

Summary

In this chapter, we learned about the applications of probability in the game AI design. We experimented with some of the techniques by implementing them in Unity3D. As a bonus, we also learnt about the basics of how a slot machine works, and implemented a simple slot machine game using Unity3D. Probability in game AI is about making the game and characters seem more realistic by adding some uncertainty, so that the players cannot predict something for sure. One of the common usages and definitions of probability is to measure the possibility of a desired event to occur out of all the other possible events. A good reference to further study the advanced techniques on probability in game AI, such as decision making under uncertainty using Bayesian techniques, would be the *AI for Game Developers David M. Bourg, Glenn Seeman, O'Reilly*. In the next chapter, we will take a look at implementing sensors, and how they can be used to make our AI aware of its surroundings.