3
Decision Making

In this chapter, we will cover the following recipes:

- Choosing through a decision tree
- Working a finite-state machine
- Improving FSMs: hierarchical finite-state machines
- Combining FSMs and decision trees
- Implementing behavior trees
- Working with fuzzy logic
- Representing states with numerical values: Markov system
- Making decisions with goal-oriented behaviors

Introduction

Making decisions or changing the game flow according to the game's state could get really messy if we rely only on simple control structures. That's why we will learn different decision-making techniques that are flexible enough to adapt to different types of games, and robust enough to let us build modular decision-making systems.

The techniques covered in the chapter are mostly related to trees, automata, and matrices. Also, some topics require a good understanding of how recursion, inheritance, and polymorphism work, so it is important that we review those topics if that is the case.

Choosing through a decision tree

One of the simplest mechanisms for tackling decision-making problems is decision trees, because they are fast and easy to grasp and implement. As a consequence, it's one of the most used techniques today; it is extensively used in other character-controlled scopes such as animations.

Getting ready

This recipe requires a good understanding of recursion and inheritance as we will constantly be implementing and calling virtual functions throughout the sections.

How to do it...

This recipe requires a lot of attention due to the number of files that we will need to handle. Overall, we will create a parent class `DecisionTreeNode`, from which we will derive the other ones. Finally, we will learn how to implement a couple of standard decision nodes:

1. First, create the parent class, `DecisionTreeNode`:

```
using UnityEngine;
using System.Collections;
public class DecisionTreeNode : MonoBehaviour
{
    public virtual DecisionTreeNode MakeDecision()
    {
        return null;
    }
}
```

2. Create the pseudo-abstract class, `Decision`, deriving from the parent class, `DecisionTreeNode`:

```
using UnityEngine;
using System.Collections;
public class Decision : DecisionTreeNode
{
    public Action nodeTrue;
    public Action nodeFalse;

    public virtual Action GetBranch()
    {
        return null;
    }
}
```

3. Define the pseudo-abstract class, `Action`:

```
using UnityEngine;
using System.Collections;
public class Action : DecisionTreeNode
{
    public bool activated = false;

    public override DecisionTreeNode MakeDecision()
    {
        return this;
    }
}
```

4. Implement the virtual function, `LateUpdate`:

```
public virtual void LateUpdate()
{
    if (!activated)
        return;
    // Implement your behaviors here
}
```

5. Create the final class, `DecisionTree`:

```
using UnityEngine;
using System.Collections;
public class DecisionTree : DecisionTreeNode
{
    public DecisionTreeNode root;
    private Action actionNew;
    private Action actionOld;
}
```

6. Override the function, `MakeDecision`:

```
public override DecisionTreeNode MakeDecision()
{
    return root.MakeDecision();
}
```

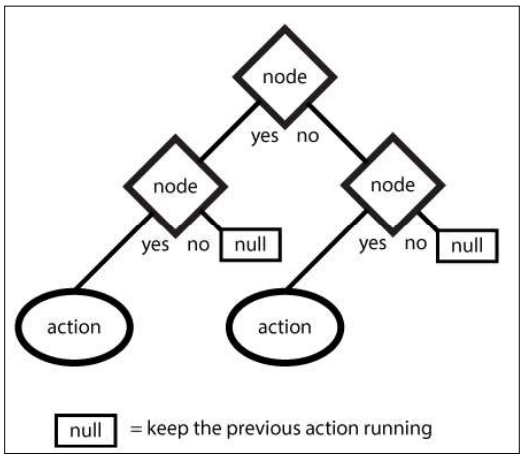
7. Finally, implement the `Update` function:

```
void Update()
{
    actionNew.activated = false;
    actionOld = actionNew;
    actionNew = root.MakeDecision() as Action;
    if (actionNew == null)
```

```
            actionNew = actionOld;
        actionNew.activated = true;
    }
```

How it works...

Decision nodes choose which path to take, calling the `MakeDecision` function recursively. It is worth mentioning that branches must be decisions and leaves must be actions. Also, we should be careful not to create cycles within the tree.



There's more...

We can create custom decisions and actions starting from the pseudo-abstract classes we already created. For example, a decision on whether to attack or run away from the player.

The custom Boolean decision:

```
using UnityEngine;
using System.Collections;

public class DecisionBool : Decision
{
    public bool valueDecision;
    public bool valueTest;

    public override Action GetBranch()
    {
        if (valueTest == valueDecision)
```

```
            return nodeTrue;
        return nodeFalse;
    }
}
```

Working a finite-state machine

Another interesting yet easy-to-implement technique is **finite-state machines** (**FSM**). They move us to change the train of thought from what it was in the previous recipe. FSMs are great when our train of thought is more event-oriented, and we think in terms of holding behavior until a condition is met changing to another.

Getting ready

This is a technique mostly based on automata behavior, and will lay the grounds for the next recipe, which is an improved version of the current one.

How to do it...

This recipe breaks down into implementing three classes from the ground up, and everything will make sense by the final step:

1. Implement the `Condition` class:

```
public class Condition
{
    public virtual bool Test()
    {
        return false;
    }
}
```

2. Define the `Transition` class:

```
public class Transition
{
    public Condition condition;
    public State target;
}
```

3. Define the `State` class:

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

public class State : MonoBehaviour
{
    public List<Transition> transitions;
}
```

4. Implement the Awake function:

```
public virtual void Awake()
{
    transitions = new List<Transition>();
    // TO-DO
    // setup your transitions here
```

5. Define the initialization function:

```
public virtual void OnEnable()
{
    // TO-DO
    // develop state's initialization here
}
```

6. Define the finalization function:

```
public virtual void OnDisable()
{
    // TO-DO
    // develop state's finalization here
}
```

7. Define the function for developing the proper behavior for the state:

```
public virtual void Update()
{
    // TO-DO
    // develop behaviour here
}
```

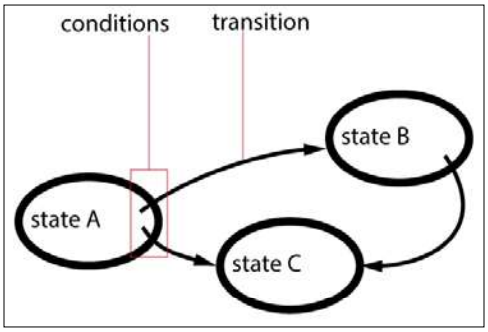
8. Implement the function for deciding if and which state to enable next:

```
public void LateUpdate()
{
    foreach (Transition t in transitions)
    {
        if (t.condition.Test())
        {
```

```
                t.target.enabled = true;
                this.enabled = false;
                return;
            }
        }
    }
```

How it works...

Each state is a `MonoBehaviour` script that is enabled or disabled according to the transitions it comes from; we take advantage of `LateUpdate` in order not to change the usual train of thought when developing behaviors, and we use it to check whether it is time to transition to a different state. It is important to disable every state in the game object apart from the initial one.



There's more...

In order to illustrate how to develop child classes deriving from `Condition`, let's take a look at a couple of examples: one that is aimed at validating a value in a range and the other one at being a logic comparer between two conditions:

The code for `ConditionFloat` is as follows:

```
using UnityEngine;
using System.Collections;

public class ConditionFloat : Condition
{
    public float valueMin;
    public float valueMax;
    public float valueTest;

    public override bool Test()
```

```
        {
            if (valueMax >= valueTest && valueTest >= valueMin)
                return true;
            return false;
        }
    }
```

The following is an example of code for `ConditionAnd`:

```
using UnityEngine;
using System.Collections;

public class ConditionAnd : Condition
{
    public Condition conditionA;
    public Condition conditionB;

    public override bool Test()
    {
        if (conditionA.Test() && conditionB.Test())
            return true;
        return false;
    }
}
```

Improving FSMs: hierarchical finite-state machines

Finite-state machines can be improved in terms of having different layers or hierarchies. The principles are the same, but states are able to have their own internal finite-state machine, making them more flexible and scalable.

Getting ready

This recipe is based on top of the previous recipe, so it is important that we grasp and understand how the finite-state machine recipe works.

How to do it...

We will create a state that is capable of holding internal states, in order to develop multi-level hierarchical state machines:

1. Create the `StateHighLevel` class deriving from `State`:

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

public class StateHighLevel : State
{
}
```

2. Add the new member variables to control the internal states:

```
public List<State> states;
public State stateInitial;
protected State stateCurrent;
```

3. Override the initialization function:

```
public override void OnEnable()
{
    if (stateCurrent == null)
        stateCurrent = stateInitial;
    stateCurrent.enabled = true;
}
```

4. Override the finalization function:

```
public override void OnDisable()
{
    base.OnDisable();
    stateCurrent.enabled = false;
    foreach (State s in states)
    {
        s.enabled = false;
    }
}
```

How it works...

The high-level state class lets us activate the internal FSMs when it is enabled and recursively disables its internal states when disabled. The working principle stays the same thanks to the list of states and the way the parent class resolves the transitioning process.

See also

Kindly refer to the recipe, *Working a finite-state machine.*

Combining FSMs and decision trees

Given the previous recipes' ease of implementation and learning, we can combine them to develop a powerful decision-making system with benefits from both worlds, making it a very powerful technique in many different scenarios.

Getting ready

We will learn how to make modifications and develop child classes in order to create a finite-state machine that is capable of creating complex transitions based on decision trees.

How to do it...

This recipe relies on creating a couple of child classes from the one we already know and making a little modification:

1. Create a new action class that holds a reference to a state:

```
using UnityEngine;
using System.Collections;

public class ActionState : DecisionTreeNode
{
    public State state;

    public override DecisionTreeNode MakeDecision()
    {
        return this;
    }
}
```

2. Implement a transition class that is able to hold a decision tree:

```
using UnityEngine;
using System.Collections;

public class TransitionDecision : Transition
{
    public DecisionTreeNode root;

    public State GetState()
```

```
        {
            ActionState action;
            action = root.MakeDecision() as ActionState;
            return action.state;
        }
    }
```

3. Modify the `LateUpdate` function in the `State` class to support both transition types:

```
public void LateUpdate()
{
    foreach (Transition t in transitions)
    {
        if (t.condition.Test())
        {
                  State target;
            if (t.GetType().Equals(typeof(TransitionDecision)))

                TransitionDecision td = t as TransitionDecision;
                target = td.GetState();
            }
            else
                target = t.target;
            target.enabled = true;
            this.enabled = false;
            return;
        }
    }
}
```

How it works...

The modification on the `State` class lets us deal with the new type of transition. The new child classes are specific types created to trick both systems and obtain the desired result of having an action node that doesn't do anything itself, but returns a new state to be activated after choosing with a decision tree.

See also

Refer to the following recipes:

- *Choosing through a decision tree*
- *Working a finite-state machine*

Implementing behavior trees

Behavior trees can be seen as a synthesis of a number of other artificial intelligence techniques, such as finite-state machines, planning, and decision trees. In fact, they share some resemblance to FSMs, but instead of states, we think in terms of actions spanned across a tree structure.

Getting ready

This recipe requires us to understand Coroutines.

How to do it...

Just like decisions trees, we will create three pseudo-abstract classes for handling the process:

1. Create the base class, `Task`:

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

public class Task : MonoBehaviour
{
    public List<Task> children;
    protected bool result = false;
    protected bool isFinished = false;
}
```

2. Implement the finalization function:

```
public virtual void SetResult(bool r)
{
    result = r;
    isFinished = true;
}
```

3. Implement the function for creating behaviors:

```
public virtual IEnumerator Run()
{
    SetResult(true);
    yield break;
}
```

4. Implement the general function for starting behaviors:

```
public virtual IEnumerator RunTask()
{
    yield return StartCoroutine(Run());
}
```

5. Create the `ConditionBT` class:

```
using UnityEngine;
using System.Collections;

public class ConditionBT : Task
{
    public override IEnumerator Run()
    {
        isFinished = false;
        bool r = false;
        // implement your behaviour here
        // define result (r) whether true or false
        //---------
        SetResult(r);
        yield break;
    }
}
```

6. Create the base class for actions:

```
using UnityEngine;
using System.Collections;

public class ActionBT : Task
{
    public override IEnumerator Run()
    {
        isFinished = false;
        // implement your behaviour here
        //---------
        return base.Run();
    }
}
```

7. Implement the `Selector` class:

```
using UnityEngine;
using System.Collections;

public class Selector : Task
```

```
{
    public override void SetResult(bool r)
    {
        if (r == true)
            isFinished = true;
    }

    public override IEnumerator RunTask()
    {
        foreach (Task t in children)
            yield return StartCoroutine(t.RunTask());
    }
}
```

8. Implement also the `Sequence` class:

```
using UnityEngine;
using System.Collections;

public class Sequence : Task
{
    public override void SetResult(bool r)
    {
        if (r == true)
            isFinished = true;
    }

    public override IEnumerator RunTask()
    {
        foreach (Task t in children)
            yield return StartCoroutine(t.RunTask());
    }
}
```

How it works...

Behavior trees work in a similar fashion to decision trees. However, the leaf nodes are called tasks and there are some branch nodes that are not conditions, but run a set of tasks in one of two ways; Selector and Sequence. Selectors run a set of tasks and return true when one of their tasks return true, it can be seen as an OR node. Sequences run a set of tasks and return true when all of their tasks return true, it can be seen as an AND node.

See also

For more theoretical insights, refer to Ian Millington's book, *Artificial Intelligence for Games.*

Working with fuzzy logic

There are times when we have to deal with gray areas, instead of binary-based values, to make decisions, and fuzzy logic is a set of mathematical techniques that help us with this task.

Imagine that we're developing an automated driver. A couple of available actions are steering and speed control, both of which have a range of degrees. Deciding how to take a turn, and at which speed, is what will make our driver different and possibly smarter. That's the type of gray area that fuzzy logic helps represent and handle.

Getting ready

This recipe requires a set of states indexed by continuous integer numbers. As this representation varies from game to game, we handle the raw input from such states, along with their *fuzzification*, in order to have a good general-purpose fuzzy decision maker. Finally, the decision maker returns a set of fuzzy values representing the degree of membership of each state.

How to do it...

We will create two base classes and our fuzzy decision maker:

1. Create the parent class, `MembershipFunction`:

```
using UnityEngine;
using System.Collections;

public class MembershipFunction : MonoBehaviour
{
    public int stateId;
    public virtual float GetDOM(object input)
    {
        return 0f;
    }
}
```

2. Implement the `FuzzyRule` class:

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

public class FuzzyRule
{
    public List<int> stateIds;
    public int conclusionStateId;
}
```

3. Create the `FuzzyDecisionMaker` class:

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

public class FuzzyDecisionMaker : MonoBehaviour
{
}
```

4. Define the decision-making function signature and its member variables:

```
public Dictionary<int,float> MakeDecision(object[] inputs,
MembershipFunction[][] mfList, FuzzyRule[] rules)
{
    Dictionary<int, float> inputDOM = new Dictionary<int,
float>();
    Dictionary<int, float> outputDOM = new Dictionary<int,
float>();
    MembershipFunction memberFunc;
    // next steps
}
```

5. Implement the loops for traversing the inputs and populate the initial degree of membership (DOM) for each state:

```
foreach (object input in inputs)
{
    int r, c;
    for (r = 0; r < mfList.Length; r++)
    {
        for (c = 0; c < mfList[r].Length; c++)
        {
            // next step
        }
    }
}
// steps after next
```

6. Define the body of the innermost loop, which makes use of the proper membership functions to set (or update) the degrees of membership:

```
memberFunc = mfList[r][c];
int mfId = memberFunc.stateId;
float dom = memberFunc.GetDOM(input);
if (!inputDOM.ContainsKey(mfId))
{
    inputDOM.Add(mfId, dom);
    outputDOM.Add(mfId, 0f);
```

```
}
else
    inputDOM[mfId] = dom;
```

7. Traverse the rules for setting the output degrees of membership:

```
foreach (FuzzyRule rule in rules)
{
    int outputId = rule.conclusionStateId;
    float best = outputDOM[outputId];
    float min = 1f;
    foreach (int state in rule.stateIds)
    {
        float dom = inputDOM[state];
        if (dom < best)
            continue;
        if (dom < min)
            min = dom;
    }
    outputDOM[outputId] = min;
}
```

8. Finally, return the set of degrees of membership:

```
return outputDOM;
```

How it works...

We make use of the boxing/unboxing technique for handling any input via the object data type. The *fuzzification* process is done with the help of our own membership functions, derived from the base class that we created in the beginning. Then, we take the minimum degree of membership for the input state for each rule and calculate the final degree of membership for each output state given the maximum output from any of the applicable rules.

There's more...

We can create an example membership function to define whether an enemy is in enraged mode, knowing that its life points (ranging from 0 to 100) are equal to or less than 30.

The following is the code for the example `MFEnraged` class:

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

public class MFEnraged : MembershipFunction
```

```
{
    public override float GetDOM(object input)
    {
        if ((int)input <= 30)
            return 1f;
        return 0f;
    }
}
```

It's worth noting that it is a common requirement to have a complete set of rules; one for each combination of states from each input. This makes the recipe lack in scalability, but it works well for a smaller number of input variables and a small number of states per variable.

See also

For more theoretical insights regarding (de)*fuzzification* and scalability weaknesses, please refer to Ian Millington's book, *Artificial Intelligence for Games*.

Representing states with numerical values: Markov system

Having learned about fuzzy logic, it may do us well to mix some approaches and probably extend the functionality with finite-state machines. However, fuzzy logic doesn't work directly with values—they have to be *defuzzified* before they have a meaning within its scope. A Markov chain is a mathematical system that allows us to develop a decision-making system that can be seen as a fuzzy state machine.

Getting ready

This recipe uses the matrix and vector classes that come with Unity to illustrate the theoretical approach and make a working example, but it can be improved with our own matrix and vector classes with the proper implementation of the required member functions, such as vector-matrix multiplication.

How to do it...

1. Create the parent class for handling transitions:

```
using UnityEngine;
using System.Collections;

public class MarkovTransition : MonoBehaviour
{
```

```
    public Matrix4x4 matrix;
    public MonoBehaviour action;
}
```

2. Implement the `IsTriggered` member function:

```
public virtual bool IsTriggered()
{
    // implementation details here
    return false;
}
```

3. Define the Markov state machine with its member variables:

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

public class MarkovStateMachine : MonoBehaviour
{
    public Vector4 state;
    public Matrix4x4 defaultMatrix;
    public float timeReset;
    public float timeCurrent;
    public List<MarkovTransition> transitions;
    private MonoBehaviour action;
}
```

4. Define the `Start` function for initialization:

```
void Start()
{
    timeCurrent = timeReset;
}
```

5. Implement the `Update` function:

```
void Update()
{
    if (action != null)
        action.enabled = false;

    MarkovTransition triggeredTransition;
    triggeredTransition = null;
    // next steps
}
```

6. Look for a triggered transition:

```
foreach (MarkovTransition mt in transitions)
{
    if (mt.IsTriggered())
    {
        triggeredTransition = mt;
        break;
    }
}
```

7. If found, compute its matrix into the game state:

```
if (triggeredTransition != null)
{
    timeCurrent = timeReset;
    Matrix4x4 matrix = triggeredTransition.matrix;
    state = matrix * state;
    action = triggeredTransition.action;
}
```

8. Otherwise, update the countdown timer and compute the default matrix into the game state, if necessary:

```
else
{
    timeCurrent -= Time.deltaTime;
    if (timeCurrent <= 0f)
    {
        state = defaultMatrix * state;
        timeCurrent = timeReset;
        action = null;
    }
}
```

How it works...

We define a game state based on the numerical value of the vector 4 member variable, with each position corresponding to a single state. The values in the game state change according to the matrix attached to each transition. When transitions are triggered, the game state changes, but we also have a countdown timer to handle a default transition and change the game accordingly. This is useful when we need to reset the game state after a period of time or just apply a regular transformation.

See also

For more theoretical insights regarding the Markov process' application to game AI, please refer to Ian Millington's book, *Artificial Intelligence for Games*.

Making decisions with goal-oriented behaviors

Goal-oriented behaviors are a set of techniques aimed at giving agents not only a sense of intelligence, but also a sense of free will, once a goal is defined, and given a set of rules to choose from.

Imagine that we're developing a trooper agent that needs to only reach the endpoint of capturing the flag (the main goal), while taking care of its life and ammo (the inner goals for reaching the first). One way of implementing it is by using a general-purpose algorithm for handling goals, so the agent develops something similar to free will.

Getting ready

We will learn how to create a goal-based action selector that chooses an action considering the main goal, avoids unintentional actions with disrupting effects, and takes an action's duration into account. Just like the previous recipe, this requires the modeling of goals in terms of numerical values.

How to do it...

Along with the action chooser, we will create base classes for actions and goals:

1. Create the base class for modeling actions:

```
using UnityEngine;
using System.Collections;

public class ActionGOB : MonoBehaviour
{
    public virtual float GetGoalChange(GoalGOB goal)
    {
        return 0f;
    }

    public virtual float GetDuration()
    {
        return 0f;
    }
}
```

2. Create the `GoalGOB` parent class with member functions:

```
using UnityEngine;
using System.Collections;

public class GoalGOB
{
    public string name;
    public float value;
    public float change;
}
```

3. Define the proper functions to handle discontentment and change over time:

```
public virtual float GetDiscontentment(float newValue)
{
    return newValue * newValue;
}

public virtual float GetChange()
{
    return 0f;
}
```

4. Define the `ActionChooser` class:

```
using UnityEngine;
using System.Collections;

public class ActionChooser : MonoBehaviour
{
}
```

5. Implement the function for handling unintentional actions:

```
public float CalculateDiscontentment(ActionGOB action, GoalGOB[]
goals)
{
    float discontentment = 0;
    foreach (GoalGOB goal in goals)
    {
        float newValue = goal.value + action.GetGoalChange(goal);
        newValue += action.GetDuration() * goal.GetChange();
        discontentment += goal.GetDiscontentment(newValue);
    }
    return discontentment;
}
```

6. Implement the function for choosing an action:

```
public ActionGOB Choose(ActionGOB[] actions, GoalGOB[] goals)
{
    ActionGOB bestAction;
    bestAction = actions[0];
    float bestValue = CalculateDiscontentment(actions[0], goals);
    float value;
    // next steps
}
```

7. Pick the best action based on which one is least compromising:

```
foreach (ActionGOB action in actions)
{
    value = CalculateDiscontentment(action, goals);
    if (value < bestValue)
    {
        bestValue = value;
        bestAction = action;
    }
}
```

8. Return the best action:

```
return bestAction;
```

How it works...

The discontentment functions help avoid unintended actions, depending on how much a goal's value changes, in terms of an action and the time it takes to be executed. Then, the function for choosing an action is taken care of by computing the most promising one in terms of the minimum impact (discontentment).