4
Sistema Baseado em Macros para Máquinas de Estados Hierárquicas

Os capítulos anteriores procuraram introduzir a utilização de Máquinas de Estados no controle do comportamento de agentes inteligentes em jogos eletrônicos, estendendo o funcionamento tradicional dessas estruturas para atender às necessidades de aplicações em tempo real. Este capítulo trata da concretização dessa discussão, apresentando um sistema que implementa Máquinas de Estados conforme descritas até aqui de maneira apropriada para ser agregado a sistemas de jogos eletrônicos.

O Sistema Baseado em Macros apresentado nas seções a seguir é uma extensão da Linguagem de Máquinas de Estados concebida por Steve Rabin (2002a), permitindo a utilização dessa linguagem no desenvolvimento de Máquinas de Estados *Fuzzy* e Hierárquicas.

4.1.
Máquinas de Estados na Prática em Jogos Eletrônicos

Máquinas de Estados, em sua concepção tradicional, são uma estrutura rígida e formal, cuja principal aplicação compreende o reconhecimento de linguagens. Utilizadas dessa forma, Máquinas de Estados oferecem pouca utilidade a sistemas de inteligência para jogos eletrônicos. O desenvolvimento de agentes inteligentes de jogos requer modificações e extensões do funcionamento das Máquinas de Estados concebido originalmente, de forma que essas estruturas se tornem úteis no controle de comportamento dos agentes (Rabin, 2002a, Houlette e Fu, 2003).

Essas modificações e extensões ao funcionamento das Máquinas de Estados foram apresentadas de maneira textual e didática nos capítulos anteriores. Neste capítulo, esse relaxamento do conceito original das Máquinas de Estados é colocado em prática, através da exibição de um sistema que implementa Máquinas de Estados de maneira útil para sua aplicação em agentes inteligentes de jogos.

Houlette e Fu (2003) apontam algumas características das Máquinas de Estados utilizadas em jogos que representam modificações ao funcionamento original dessas estruturas, como as seguintes:

- *estados representam ações*: cada estado denota uma situação diferente, que implica em uma mudança de comportamento do agente para lidar com essas novas constatações. Desse modo, cada estado representa um conjunto de instruções para implementar o comportamento correspondente, que deve ser executado quando esse estado se torna corrente;
- *estados sabem de suas transições*: as instruções que implementam condições de transição estão localizadas no estado origem da transição, de modo que cada estado implementa suas transições de saída;
- *estados finais são irrelevantes*: as Máquinas de Estados que controlam o comportamento de um agente tipicamente executam por tempo indeterminado, até que esse agente não exista mais no mundo do jogo;
- *dados de entrada são "infinitos"*: enquanto o agente está ativo, ou até que a partida termine, a Máquina de Estados permanece recebendo dados de entrada. Não acontece o mesmo com a utilização de Máquinas de Estados no reconhecimento de linguagens, onde o marcador de entrada alcança "o final da fita".

4.1.1. Máquinas de *Mealy* e Máquinas de *Moore*

A necessidade de que ações sejam tomadas a partir do estado corrente remete à definição das *Máquinas de Moore*, que são Máquinas de Estados onde a saída depende apenas do estado atual da Máquina (Hopcroft e Ullman, 1979, Rocha, 1999). Esse modelo contrasta com as *Máquinas de Mealy*, nas quais a saída depende do estado atual e do símbolo lido da entrada (Hopcroft e Ullman, 1979, Rocha, 1999).

Em termos práticos, Máquinas de *Moore* têm suas ações executadas nos estados, enquanto Máquinas de *Mealy* têm suas ações executadas nas transições (Houlette e Fu, 2003). Em geral, implementações de Máquinas de Estados em

jogos eletrônicos tendem a se parecer com as Máquinas de *Moore*, pois, como discutido na seção anterior, as ações necessárias ao controle de comportamento dos agentes inteligentes são realizadas nos estados.

Entretanto, é muito comum que seja utilizada uma abordagem híbrida entre Máquinas de *Moore* e Máquinas de *Mealy*. Em muitos casos, é desejável que algumas ações sejam tomadas no momento de uma transição. Além disso, o momento em que ocorre uma transição de entrada em um estado pode ser utilizado para a alocação de recursos e inicialização de estruturas necessárias às instruções que serão executadas enquanto esse estado for o estado corrente; analogamente, o momento em que ocorre uma transição de saída de um estado pode ser utilizado para a liberação desses recursos.

4.2. Uma Linguagem de Máquinas de Estados Baseada em Macros

Steve Rabin apresenta uma linguagem para a definição de Máquinas de Estados em seu artigo *Implementing a State Machine Language* (Rabin, 2002a). Essa abordagem emprega macros da linguagem C para definir algumas palavras-chave que facilitam a implementação de Máquinas de Estados em jogos eletrônicos, enquanto mantêm a flexibilidade do uso de C e C++ para implementar ações.

A Linguagem apresentada por Rabin se resume às macros exibidas na **Listagem 4.1**. Vistas isoladamente, elas mais parecem um conjunto de cláusulas *if-then-else* desmembradas, mas o exemplo da **Listagem 4.2** e a expansão das macros exibida na **Listagem 4.3** mostram o real poder e flexibilidade dessa técnica.

```
#define BeginStateMachine  if(state<0){if(0){
#define EndStateMachine    return(true);}}else{ASSERT(0); \
                           return(false);}return(false);
#define State(a)           return(true);}}else if(state==a){if(0){
#define OnEvent(a)         return(true);}else if(event==a){
#define OnEnter            OnEvent(EVENT_ENTER)
#define OnUpdate           OnEvent(EVENT_UPDATE)
#define OnExit             OnEvent(EVENT_EXIT)
```

Listagem 4.1 Macros da Linguagem de Máquinas de Estados, por Rabin (2002a).

```
bool runMachine(int event,int state){
    BeginStateMachine
        State(STATE_0)
            OnEnter
                // Código C/C++ para entrada no estado STATE_0
            OnUpdate
                // Código C/C++ para permanência no estado STATE_0
            OnExit
                // Código C/C++ para saída do estado STATE_0
        State(STATE_1)
            OnUpdate
                // Código C/C++ para permanência no estado STATE_1
    EndStateMachine
}
```

Listagem 4.2 Exemplo de Máquina de Estados escrita com a Linguagem.

```
bool CFSM::runMachine(int event,int state){
    if(state<0){                                // BeginStateMachine
        if(0){                                  // BeginStateMachine
            return(true);                       // State(STATE_0)
        }                                       // State(STATE_0)
    }                                           // State(STATE_0)
    else if(state==STATE_0){                    // State(STATE_0)
        if(0){                                  // State(STATE_0)
            return(true);                       // OnEnter
        }                                       // OnEnter
        else if(event==EVENT_ENTER){            // OnEnter
            // Código C/C++ para entrada no estado STATE_0
            return(true);                       // OnUpdate
        }                                       // OnUpdate
        else if(event==EVENT_UPDATE){           // OnUpdate
            // Código C/C++ para permanência no estado STATE_0
            return(true);                       // OnExit
        }                                       // OnExit
        else if(event==EVENT_EXIT){             // OnExit
            // Código C/C++ para saída do estado STATE_0
            return(true);                       // State(STATE_1)
        }                                       // State(STATE_1)
    }                                           // State(STATE_1)
    else if(state==STATE_1){                    // State(STATE_1)
        if(0){                                  // State(STATE_1)
            return(true);                       // OnUpdate
        }                                       // OnUpdate
        else if(event==EVENT_UPDATE){           // OnUpdate
            // Código C/C++ para permanência no estado STATE_1
            return(true);                       // EndStateMachine
        }                                       // EndStateMachine
    }                                           // EndStateMachine
    else{                                       // EndStateMachine
        ASSERT(0);                              // EndStateMachine
        return(false);                          // EndStateMachine
    }                                           // EndStateMachine
    return(false);                              // EndStateMachine
}
```

Listagem 4.3 Exemplo de Máquina de Estados com as macros expandidas.

A partir dos exemplos, é possível perceber o caráter conciso e prático da especificação de Máquinas de Estados através da Linguagem. A estrutura bem elaborada introduzida pelo uso das macros melhora muito o entendimento e facilita a gerência das Máquinas de Estados conforme elas vão sendo modificadas durante o processo de desenvolvimento do jogo. As cláusulas *if(0)* são necessárias para a manutenção dessa estrutura, e serão facilmente excluídas do código-objeto gerado por qualquer compilador com uma mínima atenção à otimização.

A Linguagem de Máquinas de Estados oferece três *eventos* aos quais podem ser associados blocos de instruções que executam as ações do agente. O evento *OnUpdate* é acionado a cada iteração em que o estado correspondente permanece como estado corrente; esse evento é a oportunidade para que ações associadas ao estado sejam executadas, como em uma Máquina de *Moore*. Os eventos *OnEnter* e *OnExit* são acionados quando ocorrem transições de entrada e de saída do estado, respectivamente; esses eventos representam a oportunidade de execução de ações associadas a transições, como em uma Máquina de *Mealy*.

Ainda resta apresentar alguns métodos necessários ao suporte de uma Máquina de Estados definida através da linguagem. Esses métodos se referem ao controle de execução das iterações sucessivas da Máquina de Estados (método *process*) e à sinalização de uma transição (método *setState*). Esses métodos são mostrados na **Listagem 4.4**.

```
void CFSM::process(){
      runMachine(EVENT_UPDATE,m_currentState);

      int safetyCount=10;
      while(m_stateChange && (--safetyCount>=0)){

            ASSERT(safetyCount>0 && "States are flip-flopping!");

            m_stateChange=false;
            runMachine(EVENT_EXIT,m_currentState);

            m_currentState=m_nextState;
            runMachine(EVENT_ENTER,m_currentState);
      }
}

void CFSM::setState(int newState){
      m_stateChange=true;
      m_nextState=newState;
}
```

Listagem 4.4 Métodos de suporte à Linguagem de Máquinas de Estados (Rabin 2002a).

O método *setState* configura uma transição para ser efetuada na próxima iteração, e deve ser chamado pelo método *runMachine* nos blocos que implementam as ações (como discutido na **Seção 4.1**, cada estado deve implementar as condições de suas transições de saída). O método *process*, chamado pelo sistema do jogo sempre que a próxima iteração tiver que ser executada, executa o evento *OnUpdate* do estado corrente, sinalizando a permanência do controle nesse estado durante a iteração atual; procede então ao tratamento de possíveis transições; a variável *m_stateChange* funciona como *flag* para sinalizar que existe uma transição configurada; caso exista transição a ser realizada, o evento *OnExit* do último estado corrente é acionado, a variável *m_currentState* recebe o valor do novo estado corrente, e o evento *OnEnter* desse estado é acionado. O laço *while* garante que transições sucessivas ocorram na mesma iteração. Se a Máquina de Estados tiver sido mal especificada e permaneça efetuando transições seguidamente, o laço detectará esse *bug* após algumas ocorrências (variável *safetyCount*) e exibirá uma mensagem assertiva.

Em geral, o método *process* é chamado a cada *frame* da simulação do jogo, mas em alguns casos é conveniente que seja definido um intervalo fixo de tempo (ou um número fixo de *frames*) entre cada iteração. Essa definição é crucial para a execução de Máquinas de Estados *Fuzzy*, como será discutido na seção a seguir.

4.3. Máquinas de Estados *Fuzzy* no Sistema de Macros

Da maneira apresentada na seção anterior, a Linguagem de Máquinas de Estados pode ser diretamente utilizada na implementação de Máquinas de Estados Finitos. Para Máquinas de Estados *Fuzzy*, entretanto, a Linguagem oferece recursos para a definição dos estados (de maneira idêntica à definição de Máquinas de Estados Finitos), mas os métodos de suporte e a implementação de transições devem ser modificados para atender ao caráter *Fuzzy* do estado corrente e das condições de transição.

Em primeiro lugar, o método *process* precisa lidar com a multiplicidade de estados correntes. O conjunto estado corrente é determinado pelos valores dos graus de pertinência de cada estado; estados com valor não-nulo pertencem ao conjunto.

Além disso, o método *setState* não deve mais configurar uma transição descontínua, e sim computar a transferência de quantidade de grau de pertinência entre estados, proporcionalmente ao valor de verdade da condição de transição avaliada. A avaliação das condições de transição é efetuada por cada estado nos blocos que implementam suas ações (como nas Máquinas de Estados Finitos), e a chamada de *setState* deve conter o estado origem, o estado destino e a proporção de grau de pertinência que será transferida.

O estado cujas ações serão executadas é chamado de *estado ativo*, e deve ser sorteado a cada iteração, através de um sorteio ponderado que utiliza os valores dos graus de pertinência dos estados como probabilidades. Devido ao sorteio, o intervalo entre iterações deve ser grande o suficiente para que o agente não pareça confuso, sorteando um novo estado ativo a cada fração de segundo. Esse problema foi discutido na **Seção 2.2.3**.

A **Listagem 4.5** exibe a implementação completa do exemplo de Máquina de Estados *Fuzzy* da **Figura 2.4** utilizando o Sistema Baseado em Macros. As declarações das variáveis-membro (aquelas cujo nome começa com "*m_*") foram omitidas; deve-se considerar que estão declaradas na classe que engloba os métodos (*CFuSM*). Alguns métodos triviais também foram omitidos por concisão.

```
void CFuSM::process(){
    // Aciona evento OnUpdate dos estados com
    // grau de pertinência não-nulo
    for(int i=0;i<m_numberOfStates;i++)
        if(m_degreeOfMembership[i]>0.0f)
            runMachine(EVENT_UPDATE,i);

    // Sorteia próximo estado que será o estado ativo
    // utilizando o grau de pertinência como probabilidade
    int newActiveState=getRandomStateBasedOnDegreeOfMembership();

    if(newActiveState!=m_activeState){
        runMachine(EVENT_EXIT,m_activeState);
        m_activeState=newActiveState;
        runMachine(EVENT_ENTER,m_activeState);
    }
}

float CFuSM::fuzzyEval(float number,float target){
    float factor;
    if(number<target)
        factor=number/target;
    else
        factor=target/number;
    if(factor<0.6f){    // cropa valores para evitar transições
        return 0.0f;    // com valores muito pequenos
    return factor;
```

```
}

bool CFuSM::runMachine(int event,int state){
    BeginStateMachine
        State(STATE_1)    // Ocioso
            OnUpdate
                // transição ao estado 2
                float val=fuzzyEval(m_enemyDist,90);
                if(val>0.0f){
                    setState(STATE_1,STATE_2,val);
                }

        State(STATE_2)    // Perseguindo um Inimigo
            OnEnter
                // Guarda ponteiro do inimigo
                m_enemy=getClosestEnemy();

            OnUpdate
                // se ativo, executa ações
                if(m_activeState==STATE_2)
                    runToEnemy(m_enemy);

                // transição ao estado 1
                float val=fuzzyEval(m_enemyDist,90);
                if(val>0.0f){
                    setState(STATE_2,STATE_1,val);
                }

                // transição ao estado 3
                val=fuzzyEval(m_enemyDist,25);
                if(val>0.0f){
                    setState(STATE_2,STATE_3,val);
                }

                // transição ao estado 4
                val=fuzzyEval(m_health,25);
                if(val>0.0f){
                    setState(STATE_2,STATE_4,val);
                }
            OnExit
                m_enemy=NULL;

        State(STATE_3)    // Atacando um Inimigo
            OnEnter
                // Guarda ponteiro do inimigo
                m_enemy=getClosestEnemy();

            OnUpdate
                // se ativo, executa ações
                if(m_activeState==STATE_3)
                    attackEnemy(m_enemy);

                // transição ao estado 2
                val=fuzzyEval(m_enemyDist,35);
                if(val>0.0f){
                    setState(STATE_3,STATE_2,val);
                }

                // transição ao estado 4
                val=fuzzyEval(m_health,25);
                if(val>0.0f){
```

```
                    setState(STATE_3,STATE_4,val);
                }
            OnExit
                m_enemy=NULL;

        State(STATE_3)    // Descansando
            OnEnter
                findSafeRestingPosition();

            OnUpdate
                // se ativo, executa ações
                if(m_activeState==STATE_4)
                    rest();

                // transição ao estado 1
                val=fuzzyEval(m_health,100);
                if(val>0.0f){
                    setState(STATE_4,STATE_1,val);
                }
}

void CFuSM::setState(int sourceState,int destState,float factor){
    float transf=m_degreeOfMembership[sourceState]*factor;
    m_degreeOfMembership[sourceState]-=transf;
    m_degreeOfMembership[destState]+=transf;
}
```

Listagem 4.5 Exemplo de Máquinas de Estados *Fuzzy* implementada com o Sistema Baseado em Macros.

4.4. Máquinas de Estados Hierárquicas no Sistema de Macros

A implementação de Máquinas de Estados Finitos Hierárquicas e Máquinas de Estados *Fuzzy* Hierárquicas através do Sistema Baseado em Macros necessita que seja possível executar iterações de uma sub-Máquina a partir de um bloco de instruções contido no estado associado a ela na super-Máquina. Isso é realizado através de uma chamada a um método global que retorne um ponteiro para a sub-Máquina especificada.

O exemplo da **Listagem 4.6** é bastante auto-explicativo, e apresenta uma implementação da Máquina de Estados Hierárquica mostrada na **Figura 3.2** como uma Máquina de Estados Finitos Hierárquica. Os métodos de suporte são idênticos aos mostrados na **Listagem 4.4**.

```
bool CSuperFSM::runMachine(int event,int state){
    BeginStateMachine
        State(STATE_1)    // Ocioso
            OnUpdate
                // transição ao estado 2
                if(m_enemyDist<=100)
                    setState(STATE_2);

        State(STATE_2)    // Perseguindo um Inimigo
            OnEnter
                // Guarda ponteiro do inimigo
                m_enemy=getClosestEnemy();

            OnUpdate
                // executa ações
                runToEnemy(m_enemy);

                // transição ao estado 1
                if(m_enemyDist>100)
                    setState(STATE_1);

                // transição ao estado 3
                if(m_enemyDist<=30)
                    setState(STATE_3);

                // transição ao estado 4
                if(m_health<=25)
                    setState(STATE_4);
            OnExit
                m_enemy=NULL;

        State(STATE_3)    // Atacando um Inimigo
            OnEnter
                // Pega ponteiro da sub-Máquina
                m_subMachine=getMachinePtr(SUBMACHINE_OF_STATE_3);

                // Re-inicializa sub-Máquina para estado inicial
                m_subMachine->reset();

            OnUpdate
                // executa sub-Máquina
                m_subMachine->process();

                // transição ao estado 2
                if(m_enemyDist>30)
                    setState(STATE_2,);

                // transição ao estado 4
                if(m_health<=25)
                    setState(STATE_4);
            OnExit
                m_subMachine=NULL;

        State(STATE_4)    // Descansando
            OnEnter
                findSafeRestingPosition();

            OnUpdate
                // executa ações
                rest();
```

```
                // transição ao estado 1
                if(m_health==100)
                    setState(STATE_1);
}
```

Listagem 4.6 Exemplo de Máquinas de Estados Finitos Hierárquica implementada com o Sistema Baseado em Macros.

A implementação de Máquinas de Estados *Fuzzy* Hierárquicas decorre da mistura dos exemplos da **Listagem 4.5** e da **Listagem 4.6**. O único método fundamentalmente diferente é o que realiza o sorteio do estado ativo nas Máquinas *Fuzzy*: no caso de Máquinas de Estados *Fuzzy* Hierárquicas, esse método deve considerar as probabilidades dos estados das sub-Máquinas multiplicadas pelo grau de pertinência do estado associado na super-Máquina.