# Utilizando `Behaviors` Para o Gerenciamento da Máquina de Estados em Jogos Desenvolvidos com Java 3D

Silvano Maneck Malfatti
Luciane Machado Fraga

LNCC - Laboratório Nacional de Computação Científica

Laboratório de Ambientes Colaborativos e Multimídia Aplicada.

## Resumo

O presente trabalho descreve formas de utilização da classe *`Behavior`* e suas descendentes, oferecidas na API de Programação Java 3D, para o controle da máquina de estados em jogos desenvolvidos com Java 3D.

**Palavras-Chave**: `Behavior`, máquina de estados, jogos, Java 3D.

**Contato**:
malfatti@lncc.br
lmfraga@lncc.br

## 1. Introdução

À medida que a complexidade dos jogos aumenta, (tamanho, animações, efeitos), maior é a dificuldade de realizar o gerenciamento dos seus recursos e estados. Este gerenciamento consiste em definir cada situação que pode ocorrer em um jogo, como por exemplo, a troca de contextos, a execução de uma animação, a utilização do sistema de partículas, entre outras.

Para que esta tarefa seja bem sucedida, o programador precisa definir mecanismos que facilitem este controle, principalmente durante o processo de depuração, a qual pode se tornar uma etapa bastante custosa se o jogo não estiver bem estruturado.

Uma das técnicas mais utilizadas pelos programadores para controlar todas as etapas de um jogo é a utilização do conceito de máquina de estados finitos [Perucia et al. 2005]. As máquinas de estados finitos ou *Finish State Machines* (FSM) permitem ao programador modelar todas as etapas de um jogo antes mesmo de codificá-lo. Em um jogo, a máquina de estados descreve todas as mudanças de estado que podem ocorrer durante a sua execução, como por exemplo, a troca do contexto "*Carregando"* para o "*Menu Principal"*, o disparo de efeitos que dependem de temporizadores, a animação do personagem de acordo com a situação, entre outros.

Esta tarefa não é trivial, especialmente porque muitas das trocas de estados de um jogo dependem de fatores como o tempo e também dos eventos disparados pelo jogador (através do mouse e teclado), e cabe ao gerenciamento do jogo ficar verificando constantemente se houve troca de estados ou não, e caso ocorra tomar as decisões adequadas para aquele momento.

Felizmente, o Java 3D oferece um conjunto de classes utilitárias, descendentes da classe abstrata `Behavior` (comportamento), que facilitam este trabalho para o programador [Sun Microsystens 2006]. O presente trabalho tem por objetivo demonstrar algumas formas de como utilizar este importante recurso para facilitar o gerenciamento da máquina de estados em jogos implementados com Java 3D.

Para isto, o trabalho está organizado da seguinte forma: a seção 2 apresenta o gerenciamento dos estados de um jogo a partir de sua máquina de estados finitos; a seção 3 descreve o funcionamento da classe `Behavior` e suas classes descendentes; a seção 4 apresenta o estudo de caso de um jogo do tipo FPS, onde sua máquina de estados é controlada por `Behaviors`; e por fim, a seção 5 apresenta as conclusões.

## 2. Aplicando o Conceito de Máquina de Estados Finitos para Jogos

Devido à complexidade dos jogos atualmente, um dos principais recursos usados pelos desenvolvedores para o controle dos estados de um jogo é a definição de um diagrama de estados finitos para o jogo Perucia et al. 2005].

O objetivo desta técnica é definir o fluxo de funcionamento do jogo antes mesmo de ele começar a ser codificado. A primeira etapa, portanto, consiste na definição de um diagrama de estados finitos. Esta etapa é feita no papel antes mesmo do início da programação e deve definir todos os estados de um jogo, bem como as condições necessárias para a troca de estados. A figura 1 exemplifica um diagrama de estados simples para um jogo:

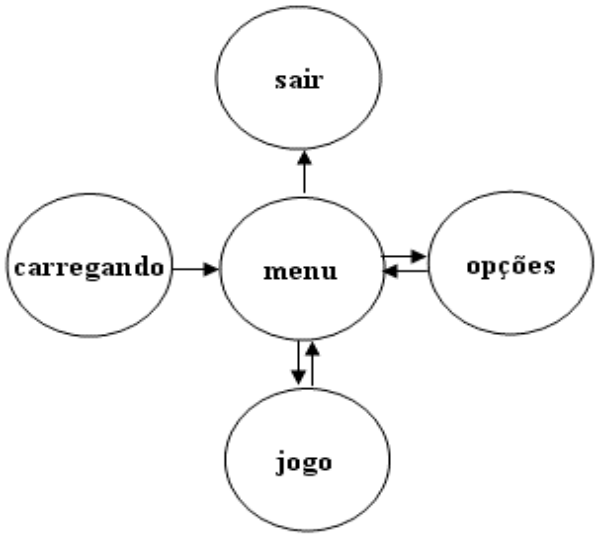


Figura 1 – Exemplo de um diagrama de estados finitos para jogos

Um jogo geralmente possui várias máquinas de estados e sub-estados. A figura 1, apresenta um exemplo de máquina de estados principal, que representa basicamente todos os contextos de um jogo. No entanto, a máquina de estados principal pode ser quebrada em várias máquinas de estados, sendo cada uma responsável pelo gerenciamento dos recursos de cada contexto.

Como é possível observar na figura 1, uma transição de estado (representado por um círculo) somente ocorre se um evento (representado por uma seta) específico for disparado.

## 3. Entendendo os *Behaviors* do Java 3D

A classe `Behavior` oferecida pelo Java 3D é uma classe abstrata que oferece recursos para que comportamentos, como animações e interações, sejam facilmente realizadas sobre os objetos 3D no mundo virtual. Esta classe conta com várias classes descendentes que permitem alterar fatores como a aparência, posição, rotação, escala dos objetos 3D de acordo com eventos disparados pelo usuário ou um outro fator, como por exemplo, o tempo [Sun Microsystens 2006].

Todo o objeto do tipo `Behavior` possui uma região de influência, que nada mais é do que a área ou o volume espacial onde o comportamento estará ativo no mundo virtual. Este recurso é muito interessante, pois permite que objetos animados sejam executados somente quando a posição da câmera estiver relativamente perto do objeto, pois não faz sentido realizar o processamento de animações quando estas ainda nem estão sendo mostradas na tela.

Por padrão, sempre que um `Behavior` for criado, a sua região de influência não estará configurada. Este é um problema muito comum enfrentado por quem começa a utilizar `Behavior`, pois mesmo que o código esteja definido corretamente, a sua região de influência ainda não estará configurada, e por este motivo o recurso não irá funcionar.

Para definir uma região de influência para um objeto `Behavior`, é preciso utilizar objetos descendentes da classe `Bounds`. A classe `Bounds` é uma classe abstrata que define o volume espacial onde o comportamento estará ativo no mundo virtual. A região de influência é representada por um envoltório imaginário, sendo que o comportamento somente passará a executar se a posição da câmera estiver dentro dos seus limites. As principais formas de definir estes limites imaginários são através das classes `BoundingSphere`, que define uma esfera imaginária, e `BoundingBox`, que define uma caixa imaginária [Selman 2002].

Além de oferecer uma série de recursos previamente definidos através da classe `Behavior`, como por exemplo, a possibilidade de movimentar ou rotacionar um objeto 3D de acordo com eventos disparados pelo mouse ou teclado, o Java 3D permite ainda que sejam criados comportamentos personalizados. Para estes comportamentos, além da área de influência, o programador precisa definir ainda uma condição que precisará ser satisfeita para que ele possa executar ou "acordar". Esta condição é definida através da classe `WakeupCondition` que referencia um ou mais critérios para que o comportamento passe a executar [Selman 2002]. A tabela 1 apresenta alguns critérios que podem ser usados para disparo de um comportamento:

| Critério | Descrição |
|---|---|
| `WakeupOnElapsedFrames` | Dispara o comportamento após o número de *frames* especificado ter sido renderizado. Se o valor zero for especificado, este comportamento será chamado a cada *frame*. |
| `WakeupOnElapsedTime` | Dispara o comportamento após um intervalo de tempo especificado. |
| `WakeupOnAWTEvent` | Dispara o comportamento de acordo com um evento lançado pelo mouse ou teclado. |
| `WakeupOnActivation` | Dispara o comportamento quando a posição da câmera entra na região de influência do comportamento. |
| `WakeupOnDesactivation` | Dispara o comportamento quando a posição da câmera sai da região de influência do comportamento. |

Tabela 1 – Critérios para ativação dos comportamentos

A definição de um comportamento personalizado em Java 3D é uma tarefa relativamente fácil, basta que primeiramente seja criada uma classe que estenda a classe abstrata `Behavior`. Feito isto, o programador é obrigado a implementar dois métodos nesta classe. O primeiro deles é o método `initialize()`, onde deverão ser configurados os critérios de ativação para o comportamento que se está definindo. O segundo método a ser implementado é `processStimulus(Enumeration criteria)` que será o método chamado quando qualquer um dos critérios especificados para o comportamento for satisfeito.

O fragmento de código da figura 2 mostra a criação de um comportamento que é executado a cada vez que uma tecla é pressionada:

```
class MyBehavior extends Behavior
{
   MyBehavior()
   {
      …
   }

   void initialize()
   {
      this.wakeupOn(new wakeupOnAWTEvent(
                    keyEvent.KEY_PRESSED));
   }

   void processStimulus(Enumeration events)
   {
      …
   }
}
```

Figura 2 - Fragmento de código para a definição de um comportamento personalizado

Para que um comportamento seja executado em um mundo virtual Java 3D é necessário ainda que ele seja inserido em um objeto do tipo `TransformGroup` e que sua região de influência tenha sido configurada.

No Java 3D, a classe TransformGroup é responsável por qualquer transformação que ocorra sobre determinado objeto no mundo virtual [Manssour 2006]. O fragmento de código da figura 3 mostra a utilização do comportamento definido na figura 2:

```
…
TransformGroup       object3D       =       new
TransformGroup();
object3D.addChild(new ColorCube(0.5));

MyBehavior myBehavior = new MyBehavior();
myBehavior.setSchedulingBounds(new
BoundingSphere());

object3D.addChild(myBehavior);
…
```

Figura 3 - Fragmento de código demonstrando a utilização do `Behavior`

O fragmento de código mostrado na figura 3, fará com que o código definido no método `processStimulus` do `Behavior`, seja executado sobre o objeto 3D contido no `TransformGroup`.

Os Behaviors fazem com que o loop principal do jogo esteja intrínsico à propria classe que realiza o gerenciamento de estados, não sendo necessário portanto criar os tradicionais loops infinitos **while(true)** ou **for ( ; ; )**, nem mesmo a implementação de Threads que precisam ser controladas. Na próxima seção veremos como aplicar estes conceitos no gerenciamento de estados de um jogo.

## 4. Aplicando `Behaviors` no Gerenciamento de um Jogo de Estilo FPS

Nesta seção é apresentado um estudo de caso da utilização da classe `Behavior` para o gerenciamento da maquina de estados de um jogo em primeira pessoa, também conhecido como FPS (*First Person Shooter*). Jogos deste estilo sempre foram muito populares, sendo que dentre os mais famosos desta categoria estão: *Doom, Quake, Half-Life, Counter Strike, Battlefield*.

Jogos do tipo FPS fazem com que o jogador se sinta muito mais imerso no ambiente, visto que é realizado em primeira pessoa sendo a sua representação, portanto, apenas um braço ou mão segurando uma arma utilizada para disparar contra os inimigos.

Em termos de implementação, este tipo de jogo oferece algumas facilidades, principalmente no que se refere à implementação da câmera, que não requer recursos muito complexos.

O desenvolvimento do FPS a ser demonstrado foi dividido em duas etapas: a primeira etapa consiste na utilização dos `Behaviors` para a definição dos estados gerais do jogo (ou máquina de estados principal), que compreendem basicamente as trocas de contexto, como por exemplo tela de *loading* para menu principal, menu principal para o jogo, etc. A segunda etapa consiste em definir os sub-estados de cada contexto.

### 4.1 Definindo um Behavior para a Máquina de Estados Principal

A máquina de estados finitos principal representa basicamente todos os contextos (telas) que o jogo pode apresentar e o seu fluxo de execução. Isto é, olhando para a máquina de estados é possível determinar quais os possíveis contextos que poderão ser acionados a partir de um determinado estado.

Para o protótipo deste FPS definiu-se seis estados principais, como mostra a figura 4::

- **Estado carregando**: representa o primeiro contexto a ser executado no jogo, que realiza o carregamento dos recursos como sons, imagens, modelos etc. Após o término do carregamento, ocorre a primeira troca de contexto para o menu principal;

- **Estado menu principal**: representa o menu principal, que apresenta algumas opções para o jogador antes do início do jogo. Sendo assim, a troca de estados deste contexto para o próximo depende de uma ação do jogador. Os possíveis estados a partir do menu são: créditos, opções e pré-game;

- **Estado créditos**: estado que apresenta os créditos do jogo e ao término volta automaticamente para o contexto menu principal;

- **Estado opções:** este estado apresenta uma tela de opções, onde o jogador pode configurar volume da música, resolução, controles, etc. Depende de uma ação do usuário para retornar ao menu principal;

- **Estado pré-jogo:** é o estado que antecede o início do jogo, e serve para apresentar os objetivos ou a missão do jogador. Este estado é baseado em um intervalo de tempo que ao se esgotar fará com que o estado jogo seja chamado.

- **Estado jogo:** este estado refere-se ao jogo propriamente dito, o qual é iniciado após o término do estado pré-jogo.

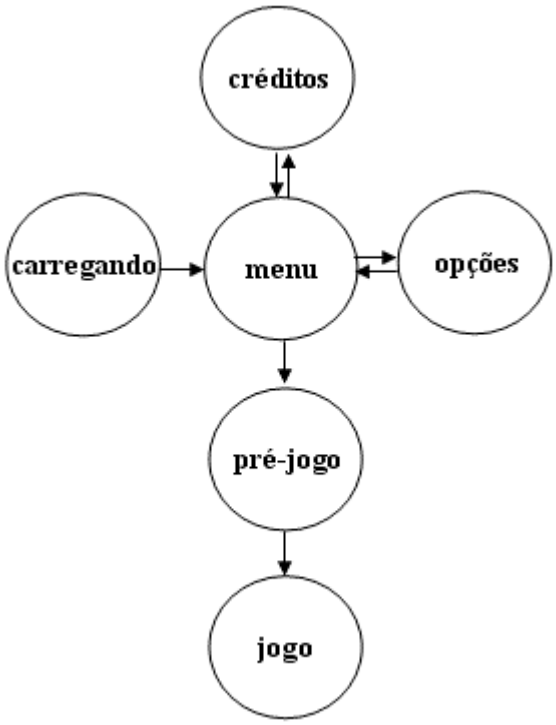


Figura 4 - Máquina de estados principal

Cada um destes estados foi implementado em uma classe que estende `BranchGroup`. A classe `BranchGroup` do Java 3D permite que os seus objetos possam ser removidos ou inseridos no mundo virtual em tempo de execução [Manssour 2006].

Cada uma destas classes possui ainda dois métodos fundamentais: `initialize` e `execute`. O primeiro é chamado quando ocorre a troca de contexto, e serve para inicializar os objetos e variáveis do contexto que será exibido.

Já o método `execute` é chamado constantemente enquanto o estado do contexto estiver ativo, e serve para atualizar os objetos do contexto na tela, como animações, posições, etc.

Após cada contexto ter sido implementado, desenvolveu-se a classe responsável pelo gerenciamento da máquina de estados principal, a classe `GameManagerBehavior` que estende `Behavior`. Para que esta classe possa realizar o controle do jogo, ela possui uma referência para cada um dos contextos apresentados na máquina de estados principal.

Para a classe `GameManagerBehavior`, os métodos `initialize` e `processStimulus` foram implementados como mostra a figura 5:

```
public void initialize()
{
   wakeupOn(new WakeupOnElapsedTime(30));
}

public void processStimulus(Enumeration enum)
{
   handleGame();
   wakeupOn(new WakeupOnElapsedTime(30));
}
```

Figura 5 - Implementação dos métodos `initialize` e `processStimulus`

O que o código apresentado na figura 5 faz é criar um `Behavior` que é chamado em um intervalo de tempo de 30 milisegundos. Portanto, o método `processStimulus` será chamado 33 vezes por segundo e que por sua vez chamará o método `handleGame` que será apresentado a seguir.

Para que fosse possível controlar o fluxo de estados, a classe `GameManagerBehavior` possui um atributo do tipo inteiro chamado i`GameState`, que armazena o estado atual do jogo. Para cada um dos contextos apresentados anteriormente foi definida uma constante exclusiva para representar o seu estado, como definido na figura 6:

```
…
/*Estados do jogo*/
public final int WGS_LOADING     =  0,
                 WGS_MAINMENU    =  1,
                 WGS_PREGAME     =  2,
```

```
                WGS_GAME        =  3,
                WGS_CREDITS     =  4,
                WGS_OPTIONS     =  5,
                WGS_END         =  6;
/*Variável que guarda o estado atual do jogo*/
public int iGameState = WGS_LOADING;
…
```

Figura 6 - Constantes que representam os estados do jogo

Após definir as constantes para cada estado, a variável iGameState é inicializada com o valor WGS_LOADING, representando o contexto de carregamento, o qual que será o estado inicial a ser chamado pela aplicação. Para completar o processo de gerenciamento de estados, a classe GameManagerBehavior possui ainda com dois métodos importantes: changeState e handleGame.

O método changeState(int), apresentado pelo fragmento de código da figura 7, é chamado somente quando há uma troca de estados, por exemplo, a passagem do estado carregando para o estado menu principal. Portanto, o que este método faz é retirar o contexto anterior, inserir o novo contexto e chamar o seu método initialize.

```
/*Método que faz a troca de estado do jogo*/
public void changeState(int iNewState)
{
   //Passa do estado Loading para menu
   if(gameManager.iGameState==gameManager.
                                   WGS_LOADING)
   {
      if(iNewState==gameManager.WGS_MAINMENU)
      {
         ((GameMenu)gameManager.bgMainMenu).
                                  initialize();
          gameManager.bgLoading.detach();
          gameManager.bgControlRender.
             addChild(gameManager.bgMainMenu);
          ((GameSound)gameManager.soundList.
                elementAt(0)).playSound(true);
        }
      }

      //Passa do estado MainMenu para Pre_Game
     (Mission)
      if(gameManager.iGameState==gameManager.
                                   WGS_MAINMENU)
       {
       if(iNewState==gameManager.
                                   WGS_PREGAME)
          {
              gameManager.bgMainMenu.detach();

              ((GameMission)gameManager.
                      bgMission).initialize();
             gameManager.bgControlRender.
             addChild(gameManager.bgMission);

            ((GameSound)gameManager.soundList.
               elementAt(0)).playSound(false);
            ((GameSound)gameManager.soundList.
               elementAt(3)).playSound(true);
          }
       }
       …
    }
}
```

Figura 7 - Método que realiza a troca de estados

Assim, sempre que um determinado estado em execução acabar, ele chamará o método changeState passando como parâmetro o valor do novo estado a ser chamado. O último método do gerente de estados a ser considerado chama-se handleGame.

Este método é chamado toda a vez que processStimulus receber um comunicado, que no caso deste comportamento será a cada 33 milisegundos. O método handleGame tem a função de realizar as atualizações em um determinado contexto quando ele está sendo executado, ou seja, chamar o método execute do contexto que estiver sendo referenciado pela variável iGameState. A figura 8 apresenta um fragmento de código do método handleGame:

```
private void handleGame()
{
   //Estado Loading
   if(gameManager.iGameState==gameManager.
                                   WGS_LOADING)
   {
      /*Verifica se o loading acabou ou nao*/
     if(!((GameLoading)gameManager.bgLoading).
                                   isLoading())
      {
        changeState(gameManager.WGS_MAINMENU);
         return;
      }

      /*Executa o loading*/
      ((GameLoading)gameManager.bgLoading).
                                     execute();

   }//Estado Main Menu
   else if (gameManager.iGameState==
                     gameManager.WGS_MAINMENU)
   {
      ((GameMenu)gameManager.bgMainMenu).
                                     execute();
   }//Estado Pregame
   …
}
```

Figura 8 – Método handleGame

Desta forma, através da variável de estado iGameState, o *loop* realizado pelo comportamento do gerenciador executará somente as atualizações para o contexto atual, que será exibido até que o método chanteState com o valor de um novo contexto seja chamado. A figura 9 mostra o diagrama geral de classes definidos no protótipo:

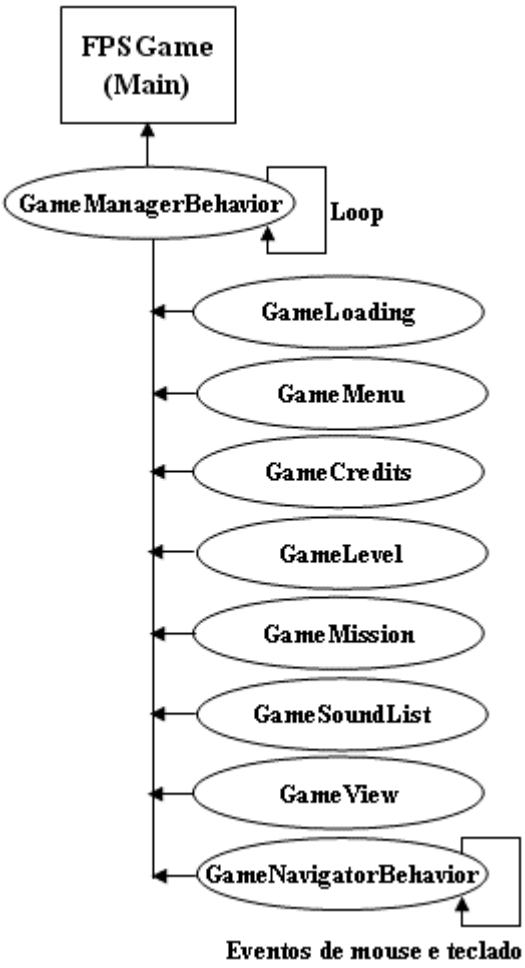


Figura 9 – Diagrama de classes do protótipo

Através da figura 9 é possível observar que o controle total do jogo é realizado pela classe `GameManagerBehavior`, onde somente um dos contextos estará sendo apresentado e atualizado.

Além do `Behavior` utilizado para a classe que gerencia a maquina de estados principal, um segundo `Behavior` é utilizado para controlar os eventos de mouse e teclado na classe que realiza a movimentação e rotação de câmera pelo cenário. Este comportamento somente estará ativado quando o seu contexto estiver sendo exibido pela classe `GameManagerBehavior`.

A figura 10 mostra alguns *screenshots* do protótipo implementado, e suas trocas de contexto realizadas pela classe `GameManagerBehavior`, sendo a figura 10a) a troca do estado carregando para menu, a figura 10b) a troca do estado menu para créditos e a figura 10c) a troca do estado pré-jogo para jogo.

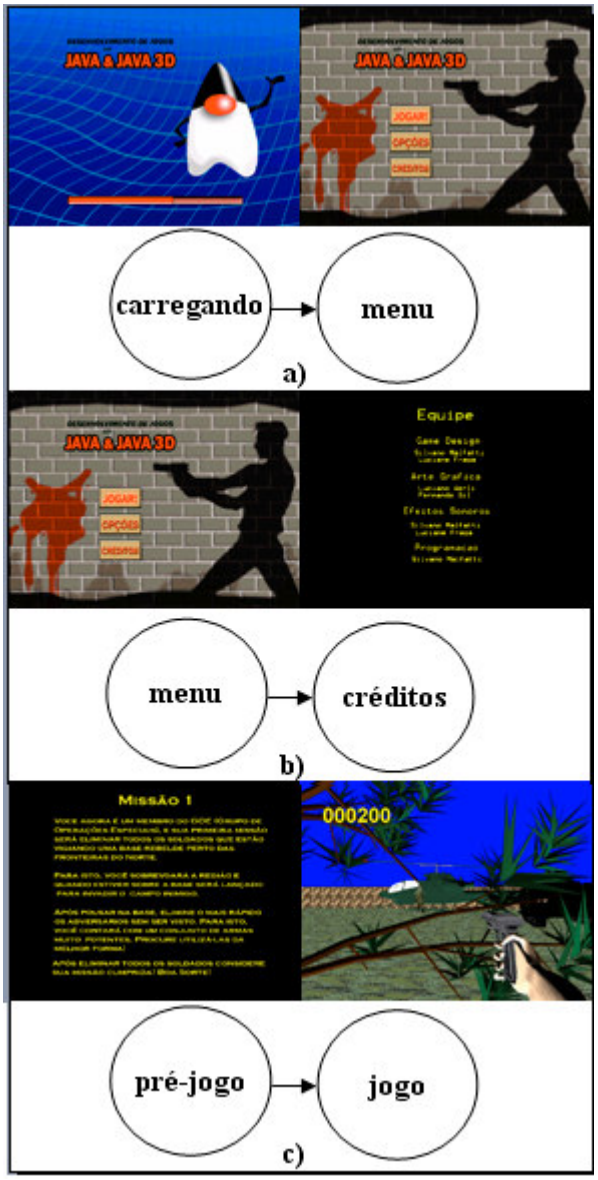


Figura 10 – Transição de Estados

### 4.2 Definindo um Behaviors para Sub-Estados

Além de controlar o fluxo de execução da máquina de estados principal, os `Behaviors` podem ser usados ainda para controlar sub-estados. No caso do FPS proposto, outro `Behavior` foi criado para controlar a interação do usuário com teclado/mouse para realizar a movimentação do personagem pelo cenário.

Para isto, implementou-se uma classe chamada `GameNavigatorBehavior`, demonstrada na `figura 9`, e que também estende `Behavior`. Os critérios de ativação neste caso são o deslocamento do mouse na tela ou o pressionamento de uma tecla.

O código abaixo mostra a implementaçao do método initialize() classe `GameNavigatorBehavior`:

```
public void initialize()
{

   wakeup1 = new
   WakeupOnAWTEvent(KeyEvent.KEY_PRESSED);

   wakeup2 = new
   WakeupOnAWTEvent(MouseEvent.MOUSE_MOVED);

   wakeup3 = new
   WakeupOnAWTEvent(MouseEvent.MOUSE_PRESSED);

   wakeupArray[0] = wakeup1;
   wakeupArray[1] = wakeup2;
   wakeupArray[2] = wakeup3;


   wakeupCondition = new WakeupOr( wakeupArray
);
}
```

Figura 11 – Implementação da classe GameNavigatorBehavior

Desta forma, implementou-se rapidamente um recurso de movimentação conhecido como "*move around*", onde a rotação do personagem é realizada com o movimento do mouse, o disparo da arma com o botão do mouse e a movimentação (avanço ou recuo) é realizada com o teclado.

A principal diferença deste comportamento para aquele definido no gerenciador de estados, é que este somente será chamado se algum evento de mouse ou teclado for gerado, e neste quando o método `processStimulus` for chamado haverá uma verificação para saber que tipo de evento foi gerado e com base nisto realizar a modificação adequada.

## 5 Conclusões

O uso de `Behaviors` no gerenciamento da máquina de estados para jogos mostrou ser um recurso muito adequado tendo em vista que facilita e agiliza o trabalho de estruturação e programação. Através da utilização de comportamentos é possível definir *loops* seqüenciais baseados em tempo sem que haja a necessidade de implementar o recurso para o controle do tempo tradicional, o qual exige que a hora do sistema seja obtida e gerenciada para controlar pequenos intervalos de tempo usados no jogo.

Além disso, os `Behaviors` fazem com que os estados de um jogo, principalmente aqueles baseados em tempo, se comportem da mesma forma independentemente da capacidade de processamento da máquina, ou seja, um determinado efeito ou troca de contexto baseado em tempo, executará da mesma forma em máquinas com velocidades diferentes.

## Referencias


PERUCIA, A. S.,Berthêm, A. C., Bertschinger, G. L. Menezes, R. R. C, 2005. Desenvolvimento de Jogos Eletrônicos, Teoria e Prática. Editora novatec. ISBN 85-7522-068-3.

SUN MICROSYSTENS, 2006. Java 3D API Tutorial. *Disponível em: http://java.sun.com/developer/onlineTraining/java3d* [Acessado 25 Agosto 2006].

SELMAN, D., 2002. Java 3D Programming. Manning Publications. ISBN 1-930110-35-9

MANSSOUR, I. H. 2006. Introdução a Java 3D. Disponível em: http://www.inf.pucrs.br/~manssour/Publicacoes/TutorialSib2003.pdf#search=%22%22Java%203D%22%2Btutorial%22. [Acessado 27 Agosto 2006]