

**UNIFEOB – CENTRO UNIVERSITÁRIO DA FUNDAÇÃO
DE ENSINO OCTÁVIO BASTOS**

Curso de Sistemas de Informação

**UTILIZAÇÃO DA PLATAFORMA JAVA PARA O
DESENVOLVIMENTO DE JOGOS 2D**

DAVID BUZATTO

SÃO JOÃO DA BOA VISTA – SP

2007

**UNIFEOB – CENTRO UNIVERSITÁRIO DA FUNDAÇÃO
DE ENSINO OCTÁVIO BASTOS**

Curso de Sistemas de Informação

**UTILIZAÇÃO DA PLATAFORMA JAVA PARA O
DESENVOLVIMENTO DE JOGOS 2D**

David Buzatto

Orientadora – Prof^ª. M.Sc. Luciana José Garcia
Ribeiro. Monografia apresentada como parte dos re-
quisitos para a obtenção do Grau de Bacharel em Sis-
temas de Informação no Curso de Sistemas de In-
formação.

SÃO JOÃO DA BOA VISTA – SP

2007

BUZATTO, David. Utilização da Plataforma Java para o Desenvolvimento de Jogos 2D. São João da Boa Vista, SP. (Trabalho de Conclusão de Curso de Sistemas de Informação da UniFEOB – Centro Universitário da Fundação de Ensino Octávio Bastos).

Orientadora: Prof^ª. M.Sc. Luciana José Garcia Ribeiro.

Declaração de Aprovação

Declaro para fins de avaliação de desempenho escolar, que o aluno David Buzatto, apresentou o Trabalho de Conclusão de Curso intitulado Utilização da Plataforma Java para o Desenvolvimento de Jogos 2D para atender às exigências da disciplina de Trabalhos de Conclusão de Curso perante banca examinadora na UniFEOB – Centro Universitário da Fundação de Ensino Octávio Bastos, composta pelos seguintes membros:

Prof^ª. M.Sc. Luciana José Garcia Ribeiro – Orientadora

Prof. M.Sc. Frederico Fagnoli Ribeiro

Prof. M.Sc. Luis Marcelo Bortolotti

Diante disto declaro que o referido aluno foi aprovado com a nota 10,0 (Dez), tendo apresentado este relatório e foi aprovado.

São João da Boa Vista, 19 de novembro de 2007.

Professora Orientadora: M.Sc. Luciana José
Garcia Ribeiro

Professor: M.Sc. Frederico Fagnoli Ribeiro

Professor: M.Sc. Luis Marcelo Bortolotti

Dedicatória

Dedico este trabalho a minha namorada Fernanda, aos meus amigos Joel, Lucas, Pablo, Marco Coelho e Vinícius, a minha família e a todos aqueles que me apoiaram e me incentivaram durante a pesquisa e implementação desse trabalho.

Agradecimentos

A primeiramente a minha namorada Fernanda, por sempre me ouvir e ficar ao meu lado durante o desenvolvimento do trabalho e ao longo desses quase 5 anos de namoro.

A minha família, principalmente minha mãe, que sempre me apoiou nas minhas decisões e contribuiu para que eu sempre seguisse o caminho correto.

Aos meus amigos Joel, Lucas, Pablo e Marco Coelho por sempre me ouvirem e me apoiarem a buscar o que eu queria.

A professora Luciana, que esteve disposta a me ajudar no desenvolvimento deste trabalho e incentivou minha participação na maratona de programação.

Aos professores, que sempre me ajudaram quando precisei, apoiando e sugerindo soluções para os problemas que apareciam.

A todos os colegas de classe que ajudaram a moldar ainda mais a minha personalidade e também ter uma melhor visão sobre o mundo.

A todos os integrantes do setor de informática da UniFEOB.

A toda a comunidade da plataforma Java, principalmente aos membros do GUVJ (Grupo de Usuários Java), que na maioria das vezes puderam me ajudar com minhas dúvidas.

Ao Vinícius, lá de Curitiba, que me auxiliou com material, sugestões sobre o tema do trabalho e por ajudar a consertar as classes *SoundManager* e *ThreadPool*.

Por fim, agradeço a todas as pessoas que estiveram ao meu lado me apoiando e me aconselhando.

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand”.

Resumo

Atualmente a plataforma Java vem sendo utilizada em vários tipos de arquiteturas e também para muitos propósitos diferentes. Entre um desses propósitos está o desenvolvimento de jogos, o qual a plataforma vem sendo aprimorada cada vez mais, seja por recursos implementados na mesma ou por bibliotecas que a estendem para fornecer funcionalidades sofisticadas no processamento gráfico e sonoro.

Sendo os jogos uma ferramenta de aprendizagem e entretenimento, foi utilizada uma literatura atual e renomada para a pesquisa e aplicação de conceitos sobre o desenvolvimento dos mesmos, tendo como base a utilização da plataforma Java.

Como resultado da pesquisa realizada e da aplicação dos conceitos aprendidos, foi produzido um jogo 2D baseado no famoso jogo *Super Mario World*, da produtora japonesa *Nintendo*.

Sendo a implementação de um jogo algo que requer grande disciplina e vontade de aprender, a conclusão obtida com o desenvolvimento deste trabalho é que a criação de jogos é uma tarefa extremamente prazerosa, que necessita de uma boa parcela de conhecimento e pesquisa devido a complexidade existente em tal tipo de aplicação e por se tratar da criação de uma ferramenta de entretenimento.

Palavras-chave: Java, Jogo, *Design* de jogos

Sumário

Lista de Abreviaturas e Siglas	p. xi
Lista de Figuras	p. xii
1 Introdução	p. 1
1.1 Considerações Iniciais	p. 1
1.2 Definição do Problema e Justificativa	p. 3
1.3 Motivação	p. 3
1.4 Objetivos	p. 3
1.4.1 Objetivo Geral	p. 3
1.4.2 Objetivos Específicos	p. 4
1.5 Metodologia	p. 4
1.6 Estrutura do Trabalho	p. 5
2 Revisão da Literatura	p. 6
2.1 Jogos	p. 6
2.2 Classes de Jogos	p. 7
2.2.1 Jogos de Tabuleiro	p. 7
2.2.2 Jogos de Cartas	p. 7

2.2.3	Jogos Atléticos	p. 8
2.2.4	Jogos Infantis	p. 8
2.2.5	Jogos de Computador	p. 9
2.3	Características Comuns aos Jogos	p. 9
2.3.1	Representação	p. 9
2.3.2	Interação	p. 10
2.3.3	Conflito	p. 12
2.3.4	Segurança	p. 13
2.4	Motivação Para Jogar	p. 13
2.5	<i>Design</i> de Jogos	p. 15
2.5.1	Documentação do <i>Design</i> do Jogo	p. 17
2.5.2	Características do <i>Designer</i> de Jogos	p. 18
2.6	Gêneros de Jogos de Computador	p. 19
2.7	Engenharia de <i>Software</i>	p. 28
2.8	Java	p. 30
2.8.1	Características	p. 30
2.8.2	Fases de uma Aplicação	p. 33
2.8.3	Java 2D	p. 34
2.8.4	Java para Desenvolvimento de Jogos	p. 36
3	Aplicação de Conceitos	p. 40
3.1	Definição do Jogo Criado	p. 40
3.2	Implementação do <i>Framework</i>	p. 41

3.2.1	Processamento dos Desenhos	p. 41
3.2.2	Interação com o Usuário	p. 44
3.2.3	Efeitos Sonoros	p. 45
3.2.4	<i>Game Loop</i>	p. 45
3.3	Implementação do Jogo	p. 46
4	Resultados	p. 50
5	Conclusões	p. 52
	Referências	p. 54
	Anexo A – Exemplo de uma aplicação simples que usa o Java 2D	p. 55
	Anexo B – Trecho da classe ScreenManager que fornece a implementação principal para o gerenciamento da tela do jogo	p. 58
	Anexo C – Trecho da classe InputManager que fornece a implementação para o registro de eventos para a aplicação	p. 61
	Anexo D – Trecho da classe GameAction que fornece a implementação para o registro de um tipo de ouvinte de evento para a aplicação	p. 63
	Anexo E – Trecho da classe MidiPlayer responsável em executar seqüências MIDI	p. 66
	Anexo F – Classe abstrata GameCore	p. 68
	Anexo G – Um exemplo de mapa	p. 73

Anexo H – Método draw da classe TileMapRenderer

p. 74

**Anexo I – Métodos da classe GameManager que tratam a detecção de
colisão**

p. 77

Lista de Abreviaturas e Siglas

2D	Duas Dimensões,	p. 2
3D	Três Dimensões,	p. 2
API	<i>Application Programming Interface,</i>	p. 34
AWT	<i>Abstract Window Toolkit,</i>	p. 44
FSEM	<i>Full-Screen Exclusive Mode,</i>	p. 37
Java EE	<i>Java Enterprise Edition,</i>	p. 1
Java ME	<i>Java Mobile Edition,</i>	p. 1
Java SE	<i>Java Standard Edition,</i>	p. 1
JDK	<i>Java SE Development Kit,</i>	p. 1
JIT	<i>Just-In-Time,</i>	p. 32
JRE	<i>Java SE Runtime Environment,</i>	p. 1
JVM	<i>Java Virtual Machine,</i>	p. 30
MIDI	<i>Musical Instrument Digital Interface,</i>	p. 45
RPG	<i>Role-Playing Game,</i>	p. 9
WAV	<i>Waveform Audio Format,</i>	p. 45

Lista de Figuras

1.1	Estrutura da plataforma Java	p. 2
1.2	Diagrama da estrutura do <i>Java Development Kit 6</i>	p. 3
2.1	Taxonomia das expressões criativas [1]	p. 6
2.2	Comparação entre o fluxo de uma história e o fluxo de um jogo	p. 11
2.3	Principais áreas do <i>design</i> de jogos.	p. 16
2.4	Cena do Jogo <i>Day of the Tentacle</i>	p. 20
2.5	Cena do Jogo <i>Myst V: End of Ages</i>	p. 21
2.6	Cena do Jogo <i>Call Of Duty 3</i>	p. 21
2.7	Cena do Jogo <i>Resident Evil 4</i>	p. 22
2.8	Cena do Jogo <i>Final Fantasy XII</i>	p. 23
2.9	Cena do Jogo <i>Starcraft II</i>	p. 23
2.10	Cena do Jogo <i>Flight Simulator 2004: A Century of Flight</i>	p. 24
2.11	Cena do Jogo <i>Winning Eleven: Pro Evolution Soccer 2007</i>	p. 24
2.12	Cena do Jogo <i>Tekken 5</i>	p. 25
2.13	Cena do Jogo <i>Chessmaster 10th Edition</i>	p. 25
2.14	Cena do Jogo <i>Columns</i>	p. 26
2.15	Cena do Jogo <i>The Sims 2</i>	p. 26
2.16	Cena do Jogo <i>Black & White</i>	p. 27

2.17	Cena do Jogo Coelho Sabido 1ª Série	p. 27
2.18	Cena do Jogo <i>The Incredible Machine</i>	p. 28
2.19	Cena do Jogo <i>World of Warcraft</i>	p. 28
2.20	A engenharia de <i>software</i> envolve a dinâmica de melhoras entre o produto e o processo.	p. 30
2.21	Fases de um programa Java SE	p. 33
2.22	Exemplo do uso do Java 2D	p. 36
3.1	Mecânica do <i>side-scroll</i>	p. 40
3.2	O <i>double buffer</i> [2].	p. 42
3.3	Ponteiro apontando para o <i>buffer</i> 1 [2].	p. 42
3.4	Depois da paginação, o ponteiro aponta para o <i>buffer</i> 2 [2].	p. 43
3.5	A linha imaginária mostra a localização do <i>tear</i> (lágrima) [2].	p. 43
3.6	Grade de um mapa.	p. 46
3.7	Alguns <i>tiles</i> utilizados no jogo.	p. 46
3.8	Algumas <i>sprites</i> utilizadas no jogo.	p. 47
3.9	Uma <i>sprite</i> colide com uma parede e a atravessa.	p. 48
3.10	Movimentação na horizontal, detecção de colisão horizontal detectada.	p. 48
3.11	Corrige o posicionamento da <i>sprite</i> para que ela não colida com nada horizontalmente.	p. 49
3.12	Move a <i>sprite</i> verticalmente, como não tem colisão, está pronto.	p. 49

1 *Introdução*

1.1 *Considerações Iniciais*

Desde que a plataforma Java foi formalmente anunciada em maio de 1995 ela vem se tornando cada vez mais poderosa e difundida em todos os tipos de ambientes operacionais, indo de pequenos sistemas móveis até computadores de grande porte, onde é requerido um alto grau de processamento e confiabilidade. Esta variedade de funcionalidades fez com que a plataforma fosse dividida em três grandes partes principais, sendo que o alicerce de tudo é o denominado Java SE (*Java Standard Edition*) onde são definidas as principais funcionalidades da linguagem e da plataforma. O Java SE, por sua vez, é dividido em duas outras partes, sendo a primeira delas, o JRE (*Java SE Runtime Environment*) o coração da plataforma, pois é em cima dele que tudo é executado. O Java RE contém as bibliotecas básicas da linguagem e a máquina virtual. A outra parte, denominada JDK (*Java SE Development Kit*), constitui todas as ferramentas de desenvolvimento do ambiente Java, como por exemplo, os utilitários de compilação e depuração e também o Java RE. As outras duas partes da plataforma são chamadas de Java ME (*Java Mobile Edition*), destinada aos dispositivos móveis e a Java EE (*Java Enterprise Edition*), utilizada em aplicações corporativas. A estrutura da plataforma é apresentada na figura 1.1 na página 2.



Figura 1.1: Estrutura da plataforma Java

Devido ao alto grau de difusão, portabilidade e funcionalidades disponíveis, a plataforma é atualmente utilizada na maioria dos tipos de aplicações possíveis, e um desses tipos são os jogos eletrônicos, que podem ser implementados utilizando todos os recursos da mesma. A variedade de funcionalidades disponíveis no ambiente Java faz com que a programação de jogos se torne mais fácil e que o programador tenha um controle maior sobre a execução do jogo, podendo utilizar uma grande quantidade de recursos avançados, inclusive 3D (Três Dimensões).

Segundo Crawford [3], os jogos têm papel importante na sociedade, pois permitem que sejam utilizados desde a educação de crianças e adultos, por meio de jogos educativos, até na simulação de situações reais.

Este trabalho tem como objetivo focar a utilização do ambiente Java na implementação de um jogo 2D (Duas Dimensões), utilizando recursos nativos da plataforma para implementar as funcionalidades requeridas para este tipo de aplicação, exemplificando as técnicas utilizadas para criar desenhos, tratar efeitos sonoros, detectar colisões entre objetos gráficos e qual a forma de se verificar a entrada fornecida por um usuário. Na figura 1.2 na página 3, tem-se a estrutura do JDK 6.

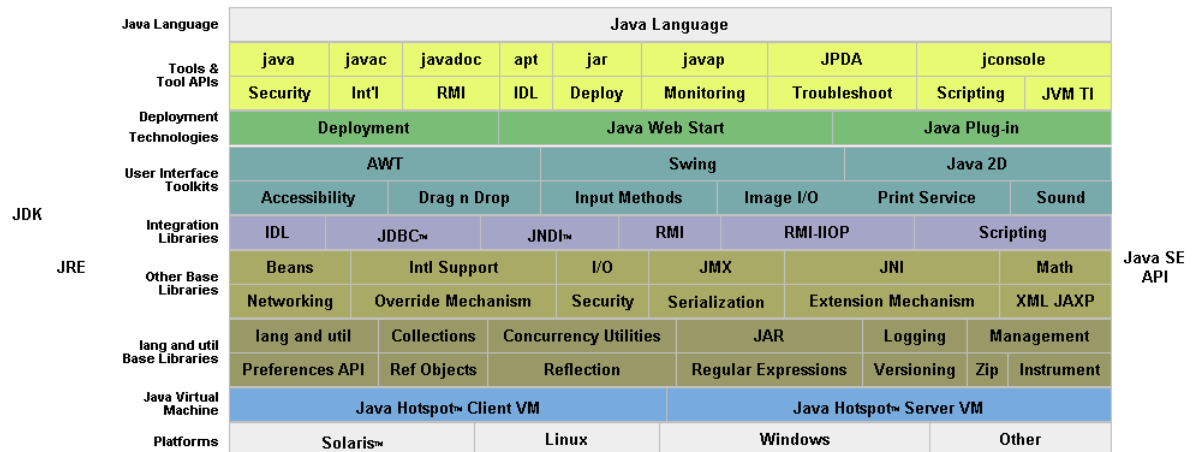


Figura 1.2: Diagrama da estrutura do *Java Development Kit 6*

Fonte: <http://java.sun.com/javase/6/docs/>

1.2 Definição do Problema e Justificativa

Para o desenvolvimento de jogos 2D na plataforma Java são necessários conhecimentos sobre a forma que os recursos requeridos para esta funcionalidade são implementados e como estes podem ser acessados e utilizados. Decidiu-se então realizar um estudo aprofundado sobre o que são jogos e como podem ser implementados no ambiente Java.

1.3 Motivação

A motivação para o tema escolhido baseou-se na vontade de mostrar como um jogo funciona nos bastidores, deixando assim uma contribuição para a comunidade acadêmica nesta área, que vem sendo cada vez mais explorada devido a sua popularidade.

1.4 Objetivos

1.4.1 Objetivo Geral

Desenvolver um jogo 2D utilizando a tecnologia Java como base.

1.4.2 Objetivos Específicos

- Definir o que é um jogo;
- Justificar a utilização de técnicas 2D para a construção de jogos;
- Apresentar a plataforma Java;
- Analisar os pontos fortes e fracos do desenvolvimento de jogos na plataforma Java, assim como os locais onde ele pode ser melhor aplicado;
- Mostrar os recursos disponíveis e utilizados em Java para a manipulação gráfica, sonora e de tratamento de entradas do usuário;
- Mostrar e implementar técnicas básicas para o desenvolvimento de jogos no estilo 2D;
- Implementar um estudo de caso para o desenvolvimento de um jogo;
- Levantar os requisitos para o estudo de caso, e;
- Aplicar todas as técnicas apresentadas na pesquisa no desenvolvimento de um jogo 2D.

1.5 Metodologia

A metodologia utilizada neste trabalho baseou-se nas atividades do ciclo de vida de um *software*, indo da concepção à implantação, onde estas atividades são:

- Realização de uma pesquisa bibliográfica com o objetivo de fornecer uma base teórica sobre o assunto;
- Estudo sobre a forma que os recursos necessários para o desenvolvimento de jogos 2D podem ser implementados e manipulados;

- Implementação de classes básicas que têm a função de prover acesso às funcionalidades gráficas e sonoras da plataforma;
- Aplicação de todos os recursos implementados durante a pesquisa na construção de um jogo 2D, e;
- Testes de como as entidades da aplicação estão se comportando e se estes comportamentos estão em conformidade com as necessidades requeridas pelo jogo a ser criado.

1.6 Estrutura do Trabalho

Este trabalho está estruturado em quatro capítulos além desta introdução:

- Capítulo 2 – Revisão da Literatura: Neste capítulo são apresentados conceitos sobre a importância que os jogos têm na sociedade e a forma com que estes podem ser implementados na plataforma Java, mostrando as técnicas empregadas no desenvolvimento de jogos 2D;
- Capítulo 3 – Aplicação de Conceitos: São implementadas as técnicas pesquisadas para a implementação de um jogo de plataforma 2D, no estilo *sides-croll*¹, bem como a forma com que elas se comportam no decorrer da execução da aplicação;
- Capítulo 4 – Resultados: Capítulo dedicado a apresentar o projeto resultante da aplicação de conceitos;
- Capítulo 5 – Conclusões: Neste capítulo são apresentadas as considerações finais deste trabalho.

¹O estilo *sides-croll* consiste em exibir o jogo lateralmente, ou seja, o personagem inicia de um lado, normalmente o esquerdo, e termina do lado oposto, fazendo com que o cenário seja “desenrolado”. Um exemplo de jogo dessa categoria é o título *Sonic*, da *Sega*.

2 *Revisão da Literatura*

2.1 Jogos

Segundo Crawford [3], os jogos constituem parte fundamental da existência humana e devido a esta relação intrínseca com o homem são gerados obstáculos ao se tentar entendê-los, principalmente por serem tratados como coisas simples, mas que na verdade não o são. Em Crawford [1], observa-se um diagrama, reproduzido na figura 2.1 abaixo, que mostra a classificação das expressões criativas existentes e quais dessas expressões são necessárias para se ter um jogo.



Figura 2.1: Taxonomia das expressões criativas [1]

2.2 Classes de Jogos

Crawford [3] divide os jogos em cinco classes principais: jogos de tabuleiro, jogos de cartas, jogos atléticos, jogos infantis e jogos de computador, as quais são caracterizadas a seguir.

2.2.1 Jogos de Tabuleiro

Os jogos de tabuleiro são caracterizados por serem jogados em uma superfície dividida em setores onde estes são populados por um conjunto de peças móveis. Normalmente as peças são relacionadas ao jogador, enquanto a superfície de jogo representa o ambiente que está sob controle dos jogadores, os quais movem suas peças pelo tabuleiro na tentativa de capturar peças de outros participantes, alcançarem um objetivo, obterem controle sobre um território ou adquirirem algum recurso presente nas regras do jogo.

2.2.2 Jogos de Cartas

Nos jogos de cartas, especificamente no baralho, é utilizado um conjunto de cinqüenta e dois símbolos obtidos através de dois fatores: nível (treze valores) e naipe (quatro valores). Os jogos desenvolvem-se utilizando combinações criadas por esses dois fatores. Os jogadores podem ganhar ou perder a posse de tais símbolos de forma aleatória ou então realizando alguma combinação permitida pelas regras do jogo. Cada combinação permitida é associada a um valor positivo para a apuração final dos resultados do jogo.

Os jogadores precisam reconhecer tanto as combinações existentes quanto às combinações potenciais e estimar a probabilidade de obter as cartas necessárias para completar a combinação. Para essa probabilidade, deve-se atribuir um peso relacionado ao valor positivo que o jogador irá obter. Sendo o número de combinações possíveis muito grandes, o cálculo preciso desta probabilidade excede o poder mental da maioria dos jogadores, fazendo com que o jogo se torne um exercício primariamente intuitivo. Além do baralho, existem outras formas de jogos com cartas, sendo o “Super-trunfo” e “*Magic*”,

alguns exemplos.

2.2.3 Jogos Atléticos

Os jogos atléticos enfatizam principalmente a capacidade física do que mental dos jogadores. As regras do jogo especificam de forma rígida um conjunto preciso das ações permitidas ao jogador ou então ações que devem ser realizadas por ele. A habilidade corporal do jogador é o principal atributo neste tipo de jogo. Além dessas características, existe a necessidade de se distinguir jogos atléticos de competições atléticas. Essa diferenciação é feita tomando-se como base o grau de interação o qual os competidores possuem. Em um jogo atlético, a característica principal é a de que cada jogador obrigatoriamente tem que interagir com outros jogadores.

Por exemplo, uma corrida de atletismo é caracterizada como uma competição, pois apesar de várias pessoas estarem sujeitas as regras, elas não interagem entre si, estando elas competindo apenas contra o relógio, ou seja, o indivíduo que demorar menos para realizar o trajeto ganha a competição. Em uma competição nenhum competidor influi ou se relaciona diretamente com outro. Um jogo atlético é uma competição que permite interação entre os integrantes, por exemplo um jogo de futebol.

2.2.4 Jogos Infantis

Os jogos infantis são formados de um grupo de atividades que enfatizam a interação física de forma simples, por exemplo, esconde-esconde, pega-pega, pé na lata, entre outros. Por esses jogos serem constituídos de componentes físicos e mentais simples, sua função não é desafiar a criança a ultrapassar seus limites nesses aspectos. Ao invés disso, o objetivo primário nesses tipos de jogos é o uso de habilidades sociais que motivem a criança a aprender a viver em comunidade, assimilando valores e aprendendo a superar perdas e dificuldades.

2.2.5 Jogos de Computador

Por fim, os jogos de computador – objeto principal deste trabalho – são feitos de forma que o computador haja como oponente e juiz, provendo na maioria das vezes, recursos gráficos para que o jogador entenda o que está acontecendo. Crawford [3] mostra que os jogos de computador são divididos em diversas outras áreas, sendo os jogos de aventura, ação, RPG (*Role-Playing Game*) e estratégia algumas delas.

2.3 Características Comuns aos Jogos

Crawford [3] diz que todas as classes de jogos possuem quatro características em comum: representação, interação, conflito e segurança. Essas similaridades são explicadas a seguir.

2.3.1 Representação

Um jogo é definido como um sistema formal fechado, que subjetivamente representa um subconjunto da realidade. Para se entender essa afirmação, faz-se necessária a análise da mesma.

- Sistema – Uma coleção de módulos que interagem entre si, normalmente de forma complexa;
- Formal – Utilização de regras claras e rígidas;
- Fechado – Um jogo é completo e auto-suficiente quanto a sua estrutura. Jogos que falham ao atender essa característica fazem com que os jogadores criem meios de burlar regras, pois se as regras do jogo não ditam todas as situações e possibilidades existentes, os jogadores, em certo momento, podem se encontrar em um estado o qual não conseguem realizar algo sem ter que inventar uma regra própria para aquela situação. Um jogo que atende a essa característica consegue informar de

forma correta tudo o que o jogador pode fazer, não sendo necessária a criação de regras externas ao ambiente;

- **Representação Subjetiva** – A representação é constituída de duas faces, uma objetiva e outra subjetiva. Nos jogos, essas faces são interligadas, com ênfase na subjetiva. Em um jogo onde o jogador destrói centenas de inimigos, essa ação normalmente não é vista como uma referência ao mundo real, mas para o jogador esta pode representar uma metáfora para a percepção do seu mundo. Algo além da percepção da destruição dos inimigos ocorre na mente do jogador, que tem o jogo como uma representação de algo presente em seu mundo privado de fantasia. Portanto, um jogo é uma representação da realidade subjetiva e não objetiva.

Os jogos são objetivamente irreais pois eles não recriam fisicamente a realidade a qual eles representam, mas são subjetivamente reais para o jogador. O agente que transforma a não realidade objetiva em uma realidade subjetiva é a fantasia humana a qual têm responsabilidade vital em qualquer situação de um jogo. Um jogo cria uma representação fantástica, não um modelo científico, e;

- **Subconjunto da Realidade** – Se um jogo representasse toda a realidade, ele seria na verdade a realidade em si, portanto um jogo deve ser um subconjunto da realidade. Esse subconjunto é responsável por prover o foco do jogo. Um jogo que possui um subconjunto muito grande da realidade, acaba tendo um foco muito vago, fazendo com que o jogo se torne incompreensível para o jogador.

2.3.2 Interação

A interação é parte essencial de um jogo, pois como no mundo real, é ela que faz com que o meio se transforme.

Uma forma de se entender a natureza do elemento interativo dos jogos é fazer uma comparação entre os jogos e os quebra-cabeças ou qualquer outra forma de desafio não interativo. Por exemplo, brincar com um cubo mágico e jogar jogo da velha e, o esporte

de salto em altura com o jogo de basquete. A diferença chave entre uma atividade que é um jogo e outra que não o é se dá pelo elemento interativo. Um cubo mágico não responde ativamente ao movimento do jogador, bem como o bastão do salto em altura. Já no jogo da velha e no basquete, existem oponentes que respondem as ações do jogador.

Outra forma de ilustrar a importância da interação é comparando jogos com histórias. Uma história é uma coleção de fatos, ordenados cronologicamente, que sugere um relacionamento de causa e efeito. A diferença entre os jogos e as histórias, apresentada na figura 2.2 abaixo, é que nas histórias os fatos são apresentados numa seqüência imutável, enquanto nos jogos os fatos possuem inúmeras ramificações, permitindo que o jogador crie sua própria história, fazendo escolhas a cada alternativa encontrada. Nos jogos os jogadores são encorajados a explorar outras alternativas, podendo ter uma visão de vários ângulos.

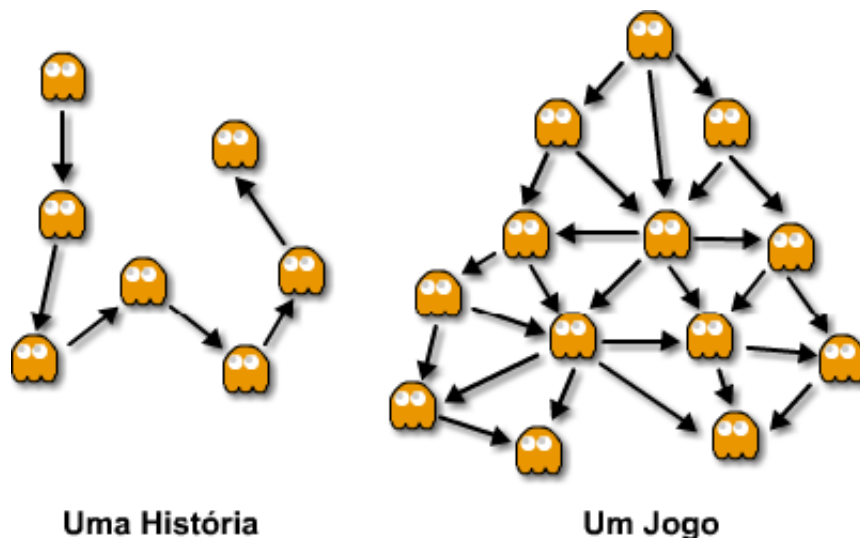


Figura 2.2: Comparação entre o fluxo de uma história e o fluxo de um jogo
 Fonte: <http://www.vancouver.wsu.edu/fac/peabody/game-book/story-game.JPG>

A importância de interação é devida principalmente a dois motivos.

- Adiciona ao evento o elemento social ou interpessoal, transformando o desafio técnico do jogo em um desafio interpessoal. Resolver um cubo mágico é uma operação estritamente técnica, pois o desafio é lidar com a lógica da situação, enquanto jogar xadrez é uma operação interpessoal, onde a lógica é utilizada para jogar contra o

oponente, e;

- Transforma a natureza passiva de um desafio em uma natureza ativa. Em um quebra-cabeça o problema é sempre apresentado da mesma forma, enquanto que em um jogo há a necessidade de o jogador reagir de acordo com as ações do oponente.

2.3.3 Conflito

O conflito aflora naturalmente durante a interação com o jogo. Normalmente o jogador está em busca de algum objetivo. Os obstáculos fazem com que a busca a tais objetivos se torne mais árdua. Se os obstáculos são passivos ou estáticos, o desafio é um quebra-cabeça ou um desafio atlético. Se eles forem ativos ou dinâmicos, respondendo propositalmente ao jogador, o desafio é um jogo. A partir do momento que tais obstáculos dificultam a passagem do jogador o conflito é gerado, portanto o conflito é essencial para todos os jogos.

Houveram tentativas de se criar jogos que não apresentassem nenhum tipo de conflito, enfatizando assim atividades cooperativas, mas na maioria das vezes esses jogos não foram comercialmente bem aceitos, sugerindo então que são poucas as pessoas que apreciam esse tipo de jogo. Para que o conflito seja retirado, faz-se necessário que sejam eliminadas as respostas dadas ao jogador, quebrando com isso a interação, fazendo com que o jogo seja destruído.

Não há possibilidade de se eliminar todo o conflito sem destruir um jogo, entretanto é possível inserir elementos cooperativos que se intercalam com os elementos conflituosos. Membros de um mesmo time cooperam entre si ao mesmo tempo em que entram em conflito com membros de outro time, com apenas um jogador ou mesmo contra o computador.

Sendo os jogos representações subjetivas do mundo real, eles focam nossa atenção em um aspecto particular do mundo, acentuando esse aspecto. Na maioria das vezes, o conflito nos jogos tende a ser acentuado na sua forma mais direta e intensa, a violência,

sendo que esta não é essencial ou fundamental para os jogos. O uso da violência é comum nos jogos pois esta é a expressão mais óbvia e natural do conflito.

Crawford [3] conclui que o conflito é um elemento intrínseco aos jogos, podendo ser direto ou indireto, violento ou não violento, mas ele está sempre presente nos jogos.

2.3.4 Segurança

O jogo é um artifício que provê experiências psicológicas de conflito e perigo, enquanto exclui essas experiências de serem realizadas realmente. Um jogo é uma forma segura de se experimentar a realidade. Enquanto assassinar um ser humano no mundo real implica em punições na vida real, em um jogo pode-se realizar tal ato sem que as conseqüências reais desta ação recaiam sobre o jogador. O jogo é um ambiente seguro de se experimentar situações que nunca seriam possíveis na vida real.

2.4 Motivação Para Jogar

Crawford [3] diz que para se entender o propósito dos jogos é necessário que sejam investigados as primeiras formas de jogos, as quais se originaram antes mesmo da origem do ser humano, fazendo então com que os jogos não sejam criação da humanidade. Por exemplo, quando dois filhotes de leão são observados no seu habitat, pode-se notar que eles agem de forma parecida como que estivessem brincando, aprendendo a rugir, dando patadas e mordendo um ao outro. Enquanto a brincadeira se desenrola, um dos dois filhotes pode notar um inseto voando, então ele abaixa na grama e se aproxima vagarosamente até conseguir capturar sua presa.

Realmente, os filhotes parecem estar brincando em um tipo de jogo, onde os quatro atributos fundamentais dos jogos podem ser observados: representação, interação, conflito e segurança. Talvez eles estejam se divertindo, pois apesar de ser possível notar seus comportamentos é impossível saber o que os leões estão sentindo naquele momento.

Os filhotes não se envolvem nesses tipos de jogos sem um motivo importante. Eles

estão apurando suas habilidades de caça e sobrevivência, aprendendo como se aproximar de uma presa sem serem vistos. Os filhotes aprendem fazendo, mas de uma forma segura, pois é melhor cometer erros caçando um inseto ou brincando com um irmão ao invés de se arriscar a se ferir com os chifres de um animal.

Os jogos são as formas mais antigas e bem sucedidas de educação. O ato de se jogar constitui uma função educacional vital para qualquer criatura capaz de aprender e já foi observado em mamíferos e pássaros, enquanto em peixes, anfíbios e répteis esse comportamento não foi encontrado.

A aprendizagem é a motivação fundamental ao se jogar, apesar dessa motivação normalmente não ser consciente. Além do aprendizado, existem vários outros motivos que fazem com que uma pessoa queira jogar: fantasia/exploração, *nose-thumbng*¹, auto provação, socialização, exercício e necessidade de reconhecimento.

Segundo Crawford [3], esses motivos são justificados da seguinte forma:

- Fantasia/Exploração – A sensação de fantasia é uma motivação importante para se jogar. Um jogo transporta o jogador do mundo real para um mundo de fantasia, o qual faz com que a pessoa esqueça seus problemas;
- *Nose-thumbng* – Uma função comum nos jogos é fornecer meios de superar restrições sociais, ao menos na fantasia;
- Auto provação – A auto provação é uma função dos jogos que fornece meios do jogador demonstrar suas habilidades;
- Socialização – Os jogos são utilizados, especialmente por adultos, como meios de socialização;
- Exercício – O exercício é mais uma forma de motivação, podendo ser físico, mental ou uma combinação de ambos, e;

¹*Nose-thumbng*, no contexto do texto, pode ser entendido como uma ação realizada por uma pessoa com o intuito de caçar de outra.

- Necessidade de reconhecimento – Alguns jogadores têm no jogo uma forma de serem reconhecidos por objetivos alcançados ou então algo que faz com que outros jogadores os respeite devido a habilidades obtidas.

2.5 *Design* de Jogos

Segundo Rollings e Adams [4], o design de jogos é um processo constituído de quatro tarefas essenciais:

- Imaginar o jogo;
- Definir a forma de seu funcionamento;
- Descrever seus elementos (conceituais, funcionais, artísticos, entre outros), e;
- Transmitir essa informação ao time que vai construí-lo.

São essas tarefas que constituem o trabalho de um *designer* de jogos, ou seja, é ele que define todos os aspectos que o jogo deve ter.

O objetivo de um jogo é entreter as pessoas, fazendo com que o *design* do jogo requeira tanto criatividade quanto um planejamento cuidadoso. Como na criação de outras formas de entretenimento, o *design* de jogos não pode ser resumido em um conjunto de instruções e processos rígidos e exatos, pois não existe uma fórmula a ser seguida para criar um jogo de sucesso. Entretanto existem alguns princípios que se aplicam a todos os jogos que tiveram boa aceitação.

O *design* de jogos pode ser dividido em três grandes áreas distintas – mecânica, história e narrativa e interatividade – as quais se completam. O diagrama da representação dessas áreas do *design* de jogos pode ser visto na figura 2.3 na página 16.

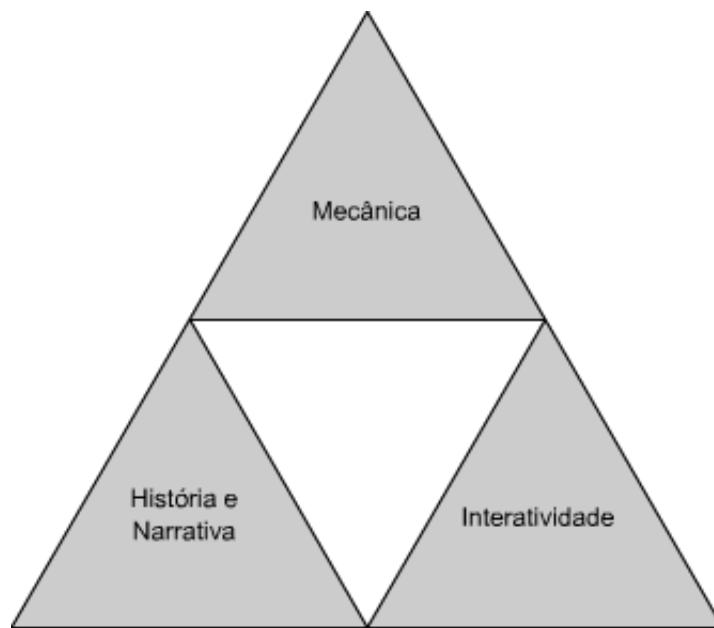


Figura 2.3: Principais áreas do *design* de jogos.

Essas áreas são definidas por Rollings e Adams [4] da seguinte forma:

- Mecânica – A mecânica constitui a base do jogo, definindo as regras que regem o mundo e a operação do jogo. A mecânica é a conversão da visão do *designer* de jogo em regras consistentes que podem ser interpretadas pelo computador, ou mais precisamente, as regras que são interpretadas pelas pessoas que irão construir o *software* que será executado pelo computador;
- História e Narrativa – Todo jogo possui uma história, a qual varia em profundidade, indo do extremo em que o jogo propriamente dito é a história ao outro extremo onde o ato de jogar cria a história. A narrativa é a parte da história que é contada ao jogador pelo autor ou *designer* do jogo. A narrativa é a parte não interativa da história, podendo existir ou não, dependendo do jogo, e;
- Interatividade – A interatividade é a forma com que o jogador vê, ouve e age dentro do mundo do jogo, ou seja, a forma com que o jogador joga. A interatividade contempla todos os elementos que são usados para criar a experiência do jogo, como por exemplo os gráficos, sons e a interface gráfica com o usuário.

2.5.1 Documentação do *Design* do Jogo

Segundo Rollings e Adams [4], os designers de jogos produzem uma série de documentos com o intuito de mostrar para outras pessoas a forma e o funcionamento do jogo. A confecção de tais documentos é de extrema importância, pois serve como um referencial durante o projeto e desenvolvimento do produto. Antes início da construção do jogo, esses documentos são analisados verificando-se viabilidade, mercado, plataforma para a qual o jogo será desenvolvido, falhas presentes no design, entre outros.

Os tipos de documentos, segundo Rollings e Adams [4], são:

- *High Concept* – O documento *high concept* (conceitual) é criado na forma de um resumo, ou mesmo de um currículo, onde os principais conceitos e características do jogo são mostrados. Ele é usado normalmente para apresentar o jogo para os executivos de produção ou publicação, que irão decidir se aquele jogo é viável e se a empresa de desenvolvimento tem interesse em desenvolver aquele produto.

O documento conceitual precisa conter a descrição de como o jogador irá interagir como o jogo, quais os tipos de cenários ele irá encontrar e como o fluxo da história deverá seguir. Recomenda-se utilizar de duas a quatro folhas na confecção desse documento. Outros conceitos que devem ser apresentados do documento conceitual são:

- Premissa do jogo;
- Audiência pretendida;
- Gênero (se o jogo se enquadrar em um);
- Questões sobre comercialização;
- Plataforma(s) alvo, e;
- Resumo da história.

- *Treatment* – O documento *treatment* (tratamento) é utilizado para apresentar o jogo de forma profunda à pessoa que se interessou por ele e que quer saber mais sobre seus detalhes. É nesse documento que se deve explicar de forma detalhada todos os aspectos do jogo, inserindo figuras explicativas, características principais dos personagens e uma breve descrição da história. Para a confecção desse documento, recomenda-se utilizar de dez a vinte páginas.
- *Script* – O documento *script* (roteiro), chamado também de bíblia do jogo, é o maior dos três documentos. O roteiro é criado para ser a referência definitiva do jogo, mostrando e explicando todos os detalhes da estrutura, organização e jogabilidade. Todas as características da história, personagens, interface com o usuário e as regras do jogo também devem ser totalmente explicadas nesse documento. O script deve conter de cinquenta a duzentas páginas.

2.5.2 Características do *Designer* de Jogos

O *designer* de jogos deve possuir algumas características, sendo que um bom *designer* não precisa possuir necessariamente todas elas.

Segundo Rollings e Adams [4], essas características são:

- Imaginação – O jogo existe em um mundo artificial, com regras e comportamentos artificiais. Um bom *designer* deve ser capaz de criar esse mundo e fazer com que o jogador seja envolvido pelo mesmo;
- Conhecimento Técnico – É necessário que o *designer* tenha conhecimento técnico sobre como os programas de computador são feitos e como funcionam;
- Competência Analítica – O *designer* deve ser capaz de analisar seu trabalho de forma crítica, sabendo quando algo deve ser alterado e aceitando a opinião de terceiros;
- Competência Matemática – A matemática se faz importante no desenvolvimento dos jogos e o *designer* deve ter conhecimentos básicos sobre ela, particularmente em

estatística;

- Competência Estética – Não há necessidade de o *designer* ser um artista, mas ele deve ter senso estético e estilístico para que seu jogo não utilize personagens e ambientes estereotipados;
- Conhecimento Geral – O conhecimento geral é muito importante para um *designer* de jogos, pois irá ajudar no processo de criação. É importante que o *designer* tenha a habilidade de buscar e aprender sozinho novos assuntos;
- Habilidades de Escrita – Essa é uma habilidade que o *designer* deve ter obrigatoriamente, pois toda a sua documentação deve ser clara, concisa, correta, sem ambigüidades e acima de tudo, fácil de ler;
- Habilidades de Desenho – Habilidades básicas de desenho e rascunho são extremamente importantes para que o *designer* consiga passar a idéia que ele teve para o profissional que irá cuidar da parte artística do desenvolvimento, e;
- Habilidade de Integrar o Time – Essa é provavelmente a habilidade mais importante que um *designer* de jogos deve ter, pois é ele que deverá conduzir o time, integrando opiniões diferentes da equipe com os interesses da empresa que está investindo no jogo, fazendo com que a integridade e o foco do projeto sejam preservados.

2.6 Gêneros de Jogos de Computador

Segundo Bates [5], os jogos de computador, bem como os desenvolvidos para consoles², atualmente são híbridos, combinando vários tipos de elementos, os quais são divididos em alguns gêneros. Sendo os jogos constituídos de gêneros híbridos, sua classificação é dada observando-se as características que se sobressaem. A seguir são caracterizados os gêneros

²Um console é um computador especializado em executar jogos eletrônicos, fornecendo interfaces de entrada e saída além da plataforma de execução. São exemplos de consoles atuais: *Sony Playstation 2*, *Sony Playstation 3*, *Nintendo GameCube*, *Nintendo Wii*, *Microsoft XBox 360*, entre outros.

principais existentes nos dias de hoje segundo Bates [5], bem como apresentada uma figura para exemplificar o conceito.

- Aventura – Os jogos de aventura são baseados em histórias e geralmente utilizam a solução de enigmas como mecanismo para o jogador prosseguir no curso da história. Normalmente os jogos de aventura não são em tempo real, sendo assim, o jogador pode gastar o tempo que for necessário para realizar alguma ação e o mundo do jogo só é alterado quando o jogador interage com esse mundo. Uma característica importante é que o ambiente explorado pelo jogador é sempre constituído de um mundo muito extenso. Como exemplos de jogos de aventura podem-se citar *Day of the Tentacle* e *Myst V: End of Ages*. Na figura 2.4 abaixo é mostrada uma imagem do jogo *Day of the Tentacle* e na figura 2.5 na página 21 pode-se ver uma cena de *Myst V: End of Ages*;



Figura 2.4: Cena do Jogo *Day of the Tentacle*

Fonte: <http://www.abandonia.com/games/en/17/images/games/Day%20of%20the%20Tentacle4.png>



Figura 2.5: Cena do Jogo *Myst V: End of Ages*

Fonte: <http://j.i.uol.com.br/galerias/pc/mystvendofages37.jpg>

- Ação – Os jogos de ação são jogos em tempo real nos quais os jogadores precisam reagir rapidamente a o que está acontecendo na tela. Essa categoria é dominada por jogos de tiro em primeira pessoa. Na figura 2.6 abaixo pode-se ver uma cena do jogo *Call Of Duty 3*. Nesse gênero, pode-se destacar também os jogos híbridos de ação e aventura, onde o a mecânica do jogo além de constituída de muita ação possui elementos que utilizam enigmas para a resolução de problemas. Como exemplo de um jogo de ação e aventura pode-se citar *Resident Evil 4*, mostrado na figura 2.7 na página 22;



Figura 2.6: Cena do Jogo *Call Of Duty 3*

Fonte: <http://j.i.uol.com.br/galerias/playstation2/callofduty312.jpg>



Figura 2.7: Cena do Jogo *Resident Evil 4*

Fonte: <http://j.i.uol.com.br/galerias/gamecube/re4180.jpg>

- RPG – Nos jogos de RPG, o jogador normalmente guia um grupo de heróis em várias aventuras, onde gradualmente esses personagens aumentam suas habilidades, evoluindo-as. Como em um jogo de aventura, um RPG possui um mundo enorme para ser explorado. Os jogadores desse estilo esperam poder gerenciar seus personagens, decidindo quais armas eles usarão bem como armaduras e outros apetrechos. Outro elemento importante nesse gênero é o combate, que permite que os personagens apurem suas técnicas evoluindo e ganhando novas habilidades. Em RPGs de fantasia, além dos elementos citados, existe normalmente um complexo sistema de magias bem como vários tipos de raças³ de personagens que compõem o grupo controlado pelo jogador. Na figura 2.8 na página 23 é vista uma cena do jogo *Final Fantasy XII*, um dos mais famosos RPGs.

³As raças em um jogo de RPG são formas de vida diferentes, sendo elas inteligentes ou não. No jogo *Final Fantasy XII* por exemplo, podem-se notar diversas raças, entre elas os *Hume* (humanos) e os *Vieras* que são seres especializados em ações furtivas e batalhas.



Figura 2.8: Cena do Jogo *Final Fantasy XII*

Fonte: <http://j.i.uol.com.br/galerias/playstation2/ffxii194.jpg>

- Estratégia – Os jogos de estratégia requerem que o jogador gere um conjunto limitado de recursos para que possa alcançar um objetivo predeterminado. Este gerenciamento de recursos frequentemente envolve decisões de que tipos de unidades⁴ serão criadas e colocadas em ação. Na figura 2.9 abaixo pode-se ver uma cena do jogo de estratégia *Starcraft II*;



Figura 2.9: Cena do Jogo *Starcraft II*

Fonte: <http://j.i.uol.com.br/galerias/pc/starcraft221.jpg>

- Simuladores – Os simuladores, ou “sims”, são jogos que procuram emular a operação de máquinas complexas, como aviões a jato e helicópteros. Os jogadores esperam gastar horas aprendendo como operar os mecanismos complexos da máquina além de terem um grande manual para os guiarem. Um simulador famoso que pode ser citado é o *Flight Simulator*, onde uma de suas versões é mostrada em execução na

⁴As unidades dos jogos de estratégia são os componentes do jogo, como por exemplo, unidades de batalha que são utilizadas para confrontos.

figura 2.10 abaixo;



Figura 2.10: Cena do Jogo *Flight Simulator 2004: A Century of Flight*
 Fonte: <http://j.i.uol.com.br/galerias/pc/centuryofflight54.jpg>

- Esportes – Os jogos de esporte permitem ao jogador participar de seu esporte favorito, sendo um jogador ou um treinador. Nesse estilo de jogo, as regras e estratégias dos esportes são rigorosamente reproduzidas. Na figura 2.11 abaixo pode-se ver a cena de uma das versões do jogo de futebol *Winning Eleven*;



Figura 2.11: Cena do Jogo *Winning Eleven: Pro Evolution Soccer 2007*
 Fonte: <http://j.i.uol.com.br/galerias/playstation2/winningelevenproevolutionsoccer200718.jpg>

- Luta – Os jogos de luta são jogos de duas pessoas, onde cada jogador controla um personagem, usando uma combinação de movimentos para atacar seu oponente ou se defender. Normalmente esses jogos são vistos sob uma perspectiva lateral, e cada batalha dura alguns minutos. Como exemplo de um jogo de luta pode-se citar *Tekken 5*, mostrado na figura 2.12 na página 25;



Figura 2.12: Cena do Jogo *Tekken 5*

Fonte: <http://j.i.uol.com.br/galerias/playstation2/tekken5155.jpg>

- Casuais – Os jogos casuais constituem adaptações de jogos tradicionais, como por exemplo xadrez e paciência. Jogos de televisão também são incluídos nessa categoria. Na figura 2.13 abaixo vê-se uma cena do jogo *Chessmaster 10th Edition*, o qual é uma reprodução de um jogo de xadrez. O gênero de jogos casuais também contempla jogos que apesar de não serem adaptações, são jogos de regras simples, onde a característica é um jogo rápido, jogado apenas para passar o tempo. Como exemplo desse tipo de jogo casual, pode-se citar *Columns* visto na figura 2.14 na página 26;



Figura 2.13: Cena do Jogo *Chessmaster 10th Edition*

Fonte: <http://j.i.uol.com.br/galerias/pc/chessmaster1007.jpg>

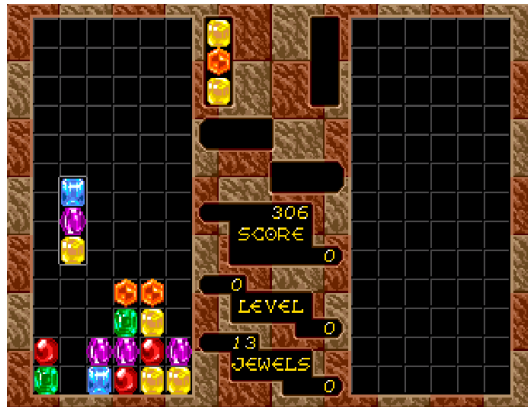


Figura 2.14: Cena do Jogo *Columns*

- *God Games* – Nesse tipo de jogo, o jogador age como se fosse o espectador, ou então uma espécie de Deus, que controla ou assiste o que acontece. Um exemplo desse tipo de jogo é o *The Sims 2*, mostrado na figura 2.15 abaixo. Outro exemplo importante deste gênero é *Black & White*, jogo pioneiro na utilização de técnicas de inteligência artificial avançadas, como aprendizado e cognição. Uma cena de *Black & White* pode ser vista na figura 2.16 na página 27;



Figura 2.15: Cena do Jogo *The Sims 2*

Fonte: <http://j.i.uol.com.br/galerias/pc/sims2183.jpg>



Figura 2.16: Cena do Jogo *Black & White*

Fonte: <http://j.i.uol.com.br/galerias/pc/blackandwhite10.jpg>

- Educacionais – Os jogos educacionais são aqueles que ensinam enquanto entretêm. Normalmente esses jogos são orientados a uma audiência infantil. Os desenvolvedores trabalham normalmente com especialistas em educação para que o jogo realmente sirva como uma ferramenta de apoio à aprendizagem. O jogo brasileiro Coelho Sabido é um exemplo de jogo educacional. Na figura 2.17 abaixo é mostrada uma cena do jogo;



Figura 2.17: Cena do Jogo Coelho Sabido 1ª Série

Fonte: <http://www.softmarket.com.br/images/403T01.jpg>

- Quebra-cabeça – Os jogos de quebra-cabeça existem essencialmente para servirem como desafio intelectual para resolução de problemas. Os quebra-cabeças não são integrados a histórias como um jogo de aventura. *The Incredible Machine* é um exemplo de um jogo quebra-cabeça sendo que uma cena pode ser vista na figura 2.18 na página 28, e;



Figura 2.18: Cena do Jogo *The Incredible Machine*

Fonte:

<http://www.worldvillage.com/wv/gamezone/images/screenshot/machine2.gif>

- *Online* – Os jogos *online* podem incluir qualquer um dos gêneros citados, sendo que sua única diferença é a de que estes podem ser jogados pela Internet. *World of Warcraft* é um exemplo de um jogo online e uma cena deste pode ser vista na figura 2.19 abaixo.



Figura 2.19: Cena do Jogo *World of Warcraft*

Fonte: <http://j.i.uol.com.br/galerias/pc/wow249.jpg>

2.7 Engenharia de *Software*

Segundo Flynt e Salem [6], a engenharia de *software* consiste em um conjunto de técnicas que guiam os desenvolvedores durante o processo de criação do *software*. Os desenvolvedores devem seguir algumas recomendações para a construção de *software* de qualidade. Entre essas recomendações, pode-se destacar:

- Obtenção e análise de todos os requisitos necessários para a construção do sistema que lhes foi incumbido de desenvolver;
- Criar o projeto do *software* de forma que este siga rigorosamente os detalhes obtidos durante o levantamento dos requisitos, e;
- Testar o sistema desenvolvido de forma a validar se seu projeto e implementação estão de acordo com os requisitos.

Seguindo essas recomendações básicas, os desenvolvedores conseguem atender apropriadamente os requisitos solicitados pelo *designer* de jogos e produzindo *software* sem redundância.

A engenharia de *software* tem como principal objetivo o aumento da qualidade dos processos de desenvolvimento de *software* e dos produtos resultantes destes processos.

Quando um produto de *software* possui qualidade, ele se torna:

- Confiável – executa o que se propõe a fazer com o mínimo de falhas;
- Extensível – fácil de inserir novas funcionalidades, e;
- Fácil de manter – quando encontrado um problema, sua correção é feita de forma fácil.

Quando o processo de desenvolvimento de *software* possui qualidade, ele se torna um processo de engenharia entendido de forma consciente. Da mesma forma que um produto de *software* com qualidade, um processo de desenvolvimento de *software* com qualidade pode ser corrigido caso possua algum problema e estendido adicionando novas funcionalidades.

A melhora contínua dos processos é a base para se melhorar continuamente os produtos. Da mesma forma, a busca para a melhora dos produtos, motiva a criação de novas formas para se melhorar os processos. A engenharia de *software* mescla essas duas atividades em um ciclo contínuo, visto no diagrama da figura 2.20 na página 30.

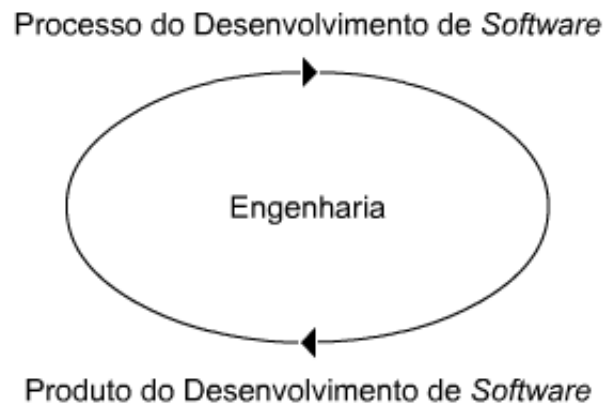


Figura 2.20: A engenharia de *software* envolve a dinâmica de melhoras entre o produto e o processo.

Sendo os jogos de computadores constituídos de produtos de *software*, pode-se então utilizar as técnicas da engenharia de *software* para a construção dos mesmos.

2.8 Java

2.8.1 Características

Segundo Horstmann e Cornell [7], Java na verdade é uma plataforma e não apenas uma linguagem de programação, como muitos acreditam. A plataforma Java é constituída de uma biblioteca que contém grande quantidade de código reutilizável e um ambiente de execução denominado máquina virtual Java, ou simplesmente JVM (*Java Virtual Machine*) que fornece vários serviços importantes como segurança, portabilidade entre plataformas e coleta de lixo. A plataforma Java possui tudo que um programador precisa para desenvolver suas aplicações: uma boa linguagem, um ambiente de execução de alta qualidade e uma vasta biblioteca.

A plataforma Java é definida por seus autores por algumas palavras-chave, sendo que estas definições podem ser encontradas em Gosling e McGilton [8] e um resumo destas em [9].

Estas palavras-chave são, segundo Horstmann e Cornell [7]:

- Simples – De fato, a linguagem Java é uma versão simplificada do C++, não havendo

necessidade de arquivos de cabeçalho, aritmética de ponteiros, estruturas, entre outros recursos. A programação visual em Java requer uma grande quantidade de codificação, enquanto que em ambientes visuais como o *Visual Basic* essa codificação é transparente aos olhos do desenvolvedor. Atualmente este problema vem sendo sanado gradualmente pelas ferramentas de programação visual em Java que vem se tornando mais poderosas e amigáveis.

- Orientada a Objetos – Os recursos para implementação da orientação a objetos em Java são comparáveis a aqueles fornecidos pelo C++. A maior diferença nesse aspecto entre Java e C++ é o que diz respeito à herança múltipla, a qual não existe no Java, mas que pode ser implementado obtendo resultados parecidos usando-se o conceito de interfaces e o modelo de metaclasses.
- Distribuída – No Java os recursos para implementação de comunicação via rede é algo simples de se fazer e que possui grande poder. Enquanto em outras linguagens são necessários vários passos para se criar uma conexão via *socket*, em Java o mesmo objetivo é alcançado de forma extremamente simples. Atualmente a arquitetura do Java EE permite que sejam construídas aplicações distribuídas de grande porte.
- Robusta – O compilador Java detecta na fase de compilação vários problemas que podem acontecer durante a execução de um programa, algo que em outras linguagens não acontece. Em Java não há perigo de corromper a memória, devido à ausência de ponteiros como os do C/C++.
- Segura – Desde o início o Java foi criado para fazer com que certos tipos de ataques fossem impossíveis de serem executados. Entre esses tipos de ataques, pode-se citar a corrupção de memória externa ao processo e leitura e escrita de arquivos sem permissão. Vários recursos de segurança foram inseridos no Java durante o tempo. Desde a versão 1.1 é suportada a idéia de classes assinadas digitalmente, que fornecem aos usuários dessas classes a certeza de quem as desenvolveu.
- Neutralidade de Arquitetura – A mais de vinte anos atrás, a implementação original

da linguagem Pascal e do sistema UCSD Pascal, ambos criados por Niklaus Wirth, já utilizavam a idéia da utilização de um código intermediário que fosse interpretado por uma máquina virtual. Naturalmente, a interpretação dos *bytecodes*⁵ é um processo mais lento do que a execução de código nativo, entretanto, as máquinas virtuais atuais conseguem identificar trechos de código que são freqüentemente executados e então traduzi-los para código nativo para que sejam executados de forma mais rápida. Esse processo de geração de código nativo durante a execução é denominado compilação JIT (*Just-In-Time*), e o responsável por este processo é o compilador *Java HotSpot*.

- Portável – Em Java, um **int** possui sempre 32 bits, enquanto em C/C++ um **int** pode ser um inteiro de 16 bits ou um inteiro de 32 bits ou mesmo ter o tamanho escolhido pelo desenvolvedor do compilador. *Strings*⁶ sempre são armazenadas utilizando o padrão *Unicode*⁷. No aspecto referente à interface gráfica com o usuário, as versões atuais do Java possuem uma vasta biblioteca para gerenciamento de janelas e componentes gráficos, sendo que estes são independentes de plataforma, sendo exibidos de forma totalmente igual, seja em um ambiente Windows, Macintosh ou UNIX.
- Interpretada – Os *bytecodes* são interpretados pela JVM;
- Alta Performance – Apesar a utilização de um interpretador para a execução dos *bytecodes*, as JVMs atuais utilizam compiladores JIT para melhorar a performance da execução da aplicação;
- *Multithreaded* - A utilização de *threads*⁸ em Java é algo realmente simples comparado a outras linguagens. A facilidade do uso de *threads* em Java é uma das principais

⁵O *bytecode* é o código intermediário gerado pelo processo de compilação no Java.

⁶Uma *String* em Java é um dos tipos responsáveis pelo armazenamento e manipulação de cadeias de caracteres.

⁷O *Unicode* é um padrão que permite aos computadores representarem e manipularem texto de virtualmente qualquer sistema de escrita.

⁸Em Java, uma *thread* é um fluxo de um processo.

razões que fazem com que a linguagem tenha um alto apelo no desenvolvimento *server-side*, e;

- Dinâmica – A dinamicidade é um recurso importante em situações onde há necessidade de se adicionar código em um programa durante sua execução. Um exemplo seria o código que é baixado da Internet para ser executado em um navegador. As versões atuais do Java fornecem recursos ao programador para total introspecção, tanto na estrutura quanto no comportamento, dos seus objetos.

2.8.2 Fases de uma Aplicação

Segundo Deitel e Deitel [10], em geral, os programas Java SE passam por cinco fases: edição, compilação, carga, verificação e execução, as quais são descritas na figura 2.21 abaixo e explicadas logo depois.

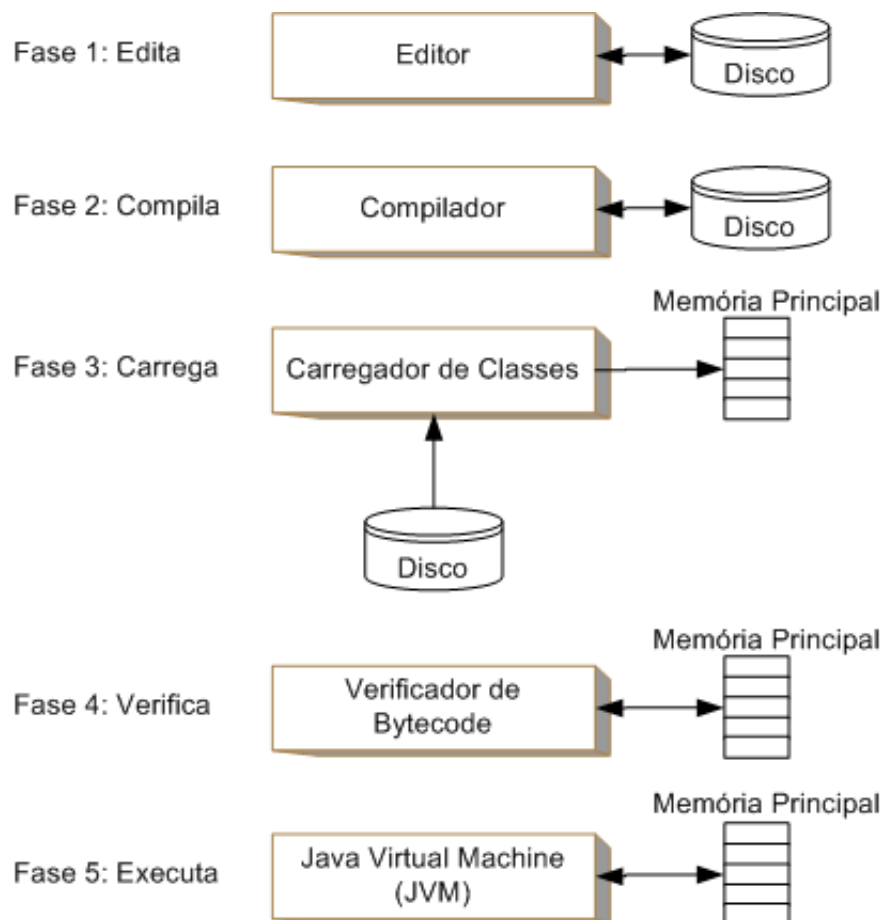


Figura 2.21: Fases de um programa Java SE

- Fase 1 – Utilizando um editor de textos, o programador escreve o código-fonte do seu programa armazenando o mesmo em qualquer dispositivo de armazenamento secundário, por exemplo o disco rígido. Os arquivos de código-fonte da linguagem Java terminam com a extensão `.java`.
- Fase 2 – O programador utiliza o utilitário de compilação **javac** para compilar o código fonte criado.

Se a compilação for bem sucedida, o compilador gerará um novo arquivo com o mesmo nome do arquivo fonte, mas com a extensão `.class`. Este novo arquivo conterá os *bytecodes* que representam a versão compilada do arquivo de código-fonte.

- Fase 3 – Para ser executado, é necessário que um programa em Java seja colocado primeiramente na memória. Este processo é conhecido como carga. O responsável por realizar tal processo é o carregador de classes, que faz a transferência dos arquivos `.class` contendo os *bytecodes* de um programa para a memória principal. O carregador de classe também irá carregar na memória principal todos os arquivos `.class` fornecidos pelo Java que são utilizados pelo programa.
- Fase 4 – Durante o processo de carga, o verificador de *bytecodes* examina os *bytecodes* para garantir sua validade e que estes não violam nenhuma restrição de segurança do Java. Devido a essa forte segurança, o Java assegura que os programas a serem executados não danificam os arquivos ou o sistema subjacente.
- Fase 5 – A JVM executa os *bytecodes* do programa.

2.8.3 Java 2D

Segundo Zhang e Liang [11], desde que as APIs (*Application Programming Interface*) Swing e Java 2D foram inseridas na plataforma Java, esta obteve um melhoramento significativo nas capacidades gráficas oferecidas. Por terem um projeto muito bem feito, essas APIs fornecem fácil suporte para várias tarefas em computação gráfica, tornando-se um grande atrativo para a programação gráfica no Java.

O suporte gráfico nas primeiras versões do Java era muito primitivo e limitado. O Java 2D oferece um conjunto completo de funcionalidades para manipular e renderizar gráficos 2D. As melhorias oferecidas pelo Java 2D incluem:

- Uma hierarquia de classes dedicadas para objetos geométricos;
- Processo de renderização refinado;
- Recursos para processamento de imagens, e;
- Modelos sofisticados de cores, fontes, impressão e outros recursos relacionados a gráficos.

A classe *Graphics2D*, subclasse da classe *Graphics*, constitui o motor de renderização do Java 2D. Esta oferece métodos para renderizar formas geométricas, imagens e textos. O processo de renderização pode ser controlado utilizando transformações, objetos de pintura, propriedades para linhas, composições, entre outras propriedades.

A listagem contida no anexo A mostra um programa que utiliza alguns recursos gráficos do Java 2D, incluindo recursos avançados como transparência e pintura em gradiente. O programa pode ser visto em execução na figura 2.22 na página 36.



Figura 2.22: Exemplo do uso do Java 2D

O Java 2D constitui uma parte padrão do núcleo da plataforma Java a partir da versão 1.2.

2.8.4 Java para Desenvolvimento de Jogos

Segundo Davison [12], o Java é uma ótima linguagem para o desenvolvimento de jogos, pois esta possui muitas funcionalidades importantes, tais como o uso do paradigma orientado a objetos, suporte multiplataforma, reutilização de código, facilidade de desenvolvimento, variedade de ferramentas, confiabilidade e estabilidade, boa documentação, suporte da *Sun Microsystems*, baixo custo de desenvolvimento, habilidade de usar código legado (como C e C++) e aumento da produtividade do programador.

Apesar de todas essas vantagens, existem algumas críticas na utilização do Java como plataforma para desenvolvimento de jogos.

As críticas mais comuns são, segundo Davison [12]:

- “Java é muito lento para programação de jogos” – Isso depende de como o desenvolvedor utiliza a linguagem. Se o programador conhece bem a plataforma e os seus

recursos, há maneiras de se programar de forma que os programas em Java sejam executados quase que na mesma velocidade de uma aplicação compilada em código nativo;

- “Java tem vazamentos de memória” – Essa afirmação é feita por programadores C/C++ que não entendem como o Java realmente funciona. O problema apontado por esses programadores é que os objetos não mais utilizados pelo programa não são liberados da memória. Isso realmente pode acontecer caso o programa crie objetos indefinidamente até que eventualmente a quantidade máxima de memória alocada seja ultrapassada;
- “Java é muito ‘alto-nível’ ” - Essa afirmação é relacionada a duas questões:
 - “A utilização de classes, objetos e herança pelo Java adicionam muita sobrecarga a linguagem, sem fornecer benefícios que justifiquem seu uso” – Essa afirmação ignora os benefícios da utilização das APIs do Java, que fornecem vários recursos fáceis de se utilizar, bem como a utilização da orientação a objetos que atualmente é o padrão de desenvolvimento.
 - “A independência de plataforma do Java implica que operações rápidas de baixo nível, como manipulação da memória de vídeo, sejam impossíveis” – Essa afirmação influencia os jogos quando é considerado o uso de recursos gráficos de alta velocidade, mas a partir da versão 1.4 do Java SE foi adicionado à plataforma o FSEM (*Full-Screen Exclusive Mode*) – modo exclusivo em tela cheia – o qual suspende o ambiente de janelamento padrão e permite que a aplicação acesse diretamente o hardware gráfico subjacente. Esse modo permite a utilização de técnicas como *page flipping*⁹, controle da resolução da tela e controle da profundidade da imagem. O principal objetivo do FSEM é

⁹O *page flipping* é uma técnica que consiste na utilização de dois *buffers* de desenho. Enquanto um buffer está sendo exibido na tela, o outro passa pelo processo de desenho, a seguir, os *buffers* são paginados, ou seja, o que estava sendo exibido passa pelo processo de desenho enquanto o que estava sendo desenhado passa a ser exibido.

aumentar a velocidade de aplicações que utilizam gráficos de forma intensiva, como os jogos;

- “A instalação de aplicações Java é complicada” – Este problema pode ser contornado utilizando um bom software de instalação, que irá fornecer uma forma fácil de se instalar todas as dependências utilizadas pelo programa, fazendo com que o processo de instalação da máquina virtual seja feito de forma transparente para o usuário. Existem vários softwares que ajudam a criar pacotes de instalação de programas Java, sendo que um exemplo é o install4J (<http://www.ejtechnologies.com>);
- “Java não é suportada em consoles” – Essa afirmação é problemática pelo fato dos consoles dominarem o mercado de vídeo games. Na conferência *JavaOne* de 2001 a Sony, dona do console Playstation, e a *Sun Microsystems* anunciaram que tinham intenção de portar a JVM para o Playstation 2, mas nada foi oficialmente lançado. Atualmente a linguagem mais usada para o desenvolvimento de jogos é o C/C++, pois existem diversas ferramentas capazes de portar jogos de uma plataforma de console para outra de forma fácil, fazendo com que a utilização do Java como plataforma de desenvolvimento seja de difícil aceitação. O mercado de jogos o qual o Java vem crescendo é o de jogos para dispositivos móveis, pois existem milhões de dispositivos com Java instalado;
- “Java não é usado para escrever jogos reais” – “Jogo real” no contexto dos jogos significa um jogo comercial. O número de jogos comerciais escritos em Java é muito pequeno quando comparado a jogos desenvolvidos em C/C++, mas esse número vem crescendo a cada dia. A utilização do Java é feita principalmente no mercado de jogos casuais, os quais são normalmente menos complexos e com tempo de desenvolvimento menor, e;
- “A *Sun Microsystems* não se interessa em suportar o desenvolvimento de jogos na plataforma Java” – O mercado de jogos não é algo que tenha tradição na *Sun Microsystems* e provavelmente nunca terá a profundidade do conhecimento detido

pela Sony ou Nintendo, entretanto nos últimos anos a Sun Microsystems tem se interessado cada vez mais pelo mercado de jogos. Através da evolução do Java SE, foram sendo implementados recursos que possibilitam o desenvolvimento de jogos cada vez mais sofisticados.

3 Aplicação de Conceitos

Para a aplicação de conceitos deste trabalho, foi implementado um jogo 2D no estilo *side-scroll*, sendo que as classes que compõem a infraestrutura do jogo foram criadas usando-se como base a implementação contida nos cinco primeiros capítulos do livro “*Developing Games In Java*” [2].

No estilo *side-scroll*, o jogador inicia o jogo de um lado da tela e finaliza do outro lado. Na figura 3.1 abaixo é mostrado como funciona a mecânica do *side-scroll*.

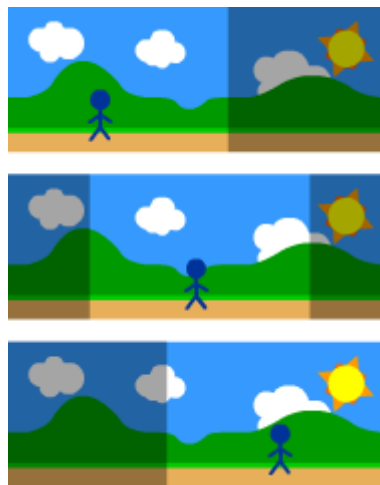


Figura 3.1: Mecânica do *side-scroll*

3.1 Definição do Jogo Criado

O jogo criado foi baseado no *Super Mario World* da *Nintendo*. Neste jogo, o personagem principal se chama *Mario* e a história gira em torno do conflito existente entre este com o vilão *Bowser*, uma espécie de tartaruga. O nome dado ao jogo foi *JMario*, sendo que a letra “J” faz referência a plataforma Java enquanto *Mario* faz referência ao personagem

principal do jogo acima citado. Durante o desenrolar do jogo são apresentadas várias fases com diversos tipos de desafios. Na implementação, foram criadas apenas três fases, onde o jogador irá encontrar desafios simples, parecidos com os dos primeiros estágios do jogo original.

3.2 Implementação do *Framework*

Toda a infraestrutura do jogo foi baseada em um *framework*¹ para jogos 2D implementado em [2]. Serão comentadas as classes principais que foram utilizadas bem como as técnicas empregadas na sua implementação. Houve a necessidade de se adaptar algumas classes desse *framework* para que as mesmas atendessem aos requisitos da aplicação e para que alguns erros fossem solucionados.

3.2.1 Processamento dos Desenhos

Para o desenho do jogo na tela, foi utilizado o recurso do modo exclusivo em tela cheia (FSEM), onde o Java consegue ter um maior controle sobre o *hardware* gráfico do computador, podendo realizar operações de desenho de forma mais rápida. Além da utilização do FSEM, é necessário se utilizar outras técnicas para que a renderização dos desenhos não sofram distorções.

A primeira técnica, chamada *double buffer* consiste em manter um *buffer* de desenho, que irá receber as instruções de desenho ao invés de desenhar o que se precisa diretamente na tela. Após as instruções terem sido processadas, elas são renderizadas na tela enquanto o *buffer* passa novamente pelo processo de desenho. O diagrama que ilustra o *double buffer* pode ser visto na figura 3.2 da página 42.

¹*Framework* é a designação dada a uma API ou conjunto de APIs que tem como objetivo fornecer serviços específicos para o desenvolvedor.

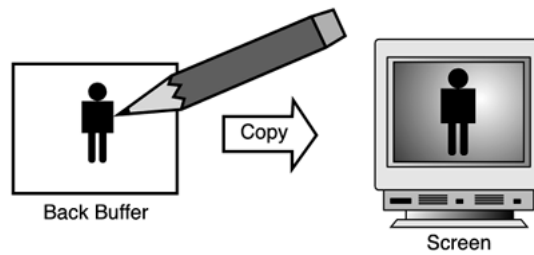


Figura 3.2: O *double buffer* [2].

A segunda técnica, chamada *page flipping*, consiste em manter dois *buffers* de renderização. Enquanto a aplicação está desenhando em um dos *buffers* (*back buffer*), o outro *buffer* (*display buffer*) é utilizado para ser renderizado na tela. Quando o desenho do *display buffer* termina, ocorre a paginação (*page flipping*) dos *buffers*, fazendo com que o *back buffer* seja renderizado na tela e o *display buffer* passe a ser desenhado. Com essa técnica, consegue-se evitar um problema chamado *flickering*, que é diagnosticado quando um desenho que está sendo desenhado parece piscar na tela. Os diagramas explicativos do *page flipping* podem ser visto na figura 3.3 abaixo e na figura 3.4 na página 43.

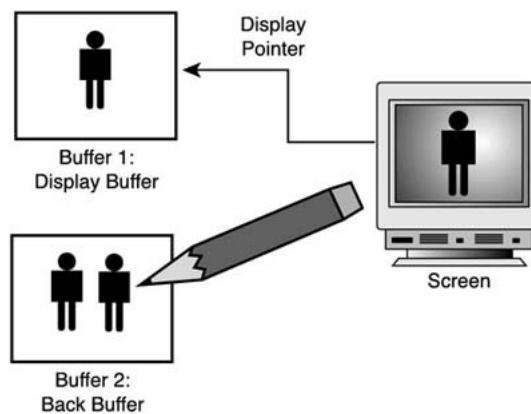


Figura 3.3: Ponteiro apontando para o *buffer* 1 [2].

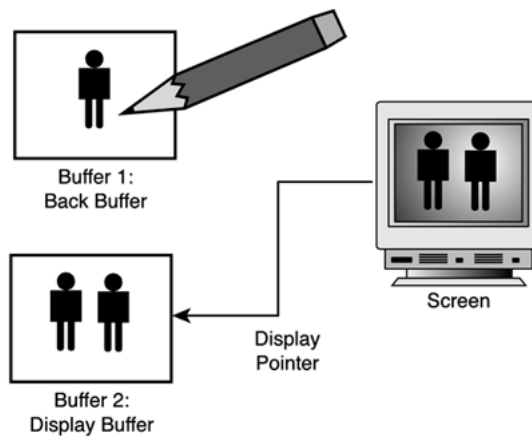


Figura 3.4: Depois da paginação, o ponteiro aponta para o *buffer 2* [2].

A terceira técnica é utilizada para sincronizar a taxa de atualização do monitor com a taxa de desenho dos *buffers*, para que não ocorra um problema chamado *tearing* (lacrimajamento), que é justamente o que acontece por causa de uma falta de sincronismo entre a atualização de tela e quando a imagem é exibida pelo monitor. A aplicação começa a atualizar a imagem da tela e, quando está no meio, o monitor exibe uma imagem incompleta e parcialmente inválida. Como isso ocorre em apenas alguns *frames*, o que é percebido pelo usuário são pequenos pontos na tela, pequena lágrimas. Daí o termo *tearing*, que significa, lacrimajamento em inglês. O problema do *tearing* pode ser visto na figura 3.5 abaixo.



Figura 3.5: A linha imaginária mostra a localização do *tear* (lágrima) [2].

O *double buffer*, *page flipping* e a sincronização com a taxa de atualização do monitor são implementados utilizando-se a classe *BufferStrategy* que é capaz de escolher a melhor forma de bufferização baseada nas capacidades do sistema. Essas funcionalidades são implementadas pela classe *ScreenManager* do *framework* do jogo. Um trecho da implementação dessa classe pode ser vista no anexo B, estando o código totalmente comentado.

3.2.2 Interação com o Usuário

Para que um usuário interaja com a aplicação é usado o modelo de eventos da API AWT (*Abstract Window Toolkit*) do Java. Esse modelo fornece uma *thread* de despacho de eventos que é utilizada para processar eventos provenientes de ações que um usuário realiza em um programa, bem como eventos disparados por componentes do próprio pacote de janelamento do Java.

Para cada tipo de evento que um usuário pode disparar em um programa, o AWT fornece uma interface de ouvinte apropriada para tratar esse evento. Por exemplo, a interface *KeyListener* é utilizada para implementar ouvintes de teclas. Com ela pode-se processar eventos de quando uma tecla é pressionada ou quando a mesma é solta.

Da mesma forma que existe uma interface para a implementação de eventos de teclado, também existe uma interface para processamento dos cliques do mouse. A diferença entre os eventos do mouse e do teclado, é que além dos eventos de clique, o mouse também pode disparar eventos de movimentação.

A interface *MouseListener* é responsável em fornecer o contrato a ser implementado para a criação de eventos de clique, enquanto a interface *MouseMotionListener* é responsável em fornecer meios para que se possa processar os eventos de movimentação do mouse. Além da implementação dos ouvintes é necessário registrar tais ouvintes nos componentes que irão gerar tais eventos, como por exemplo um botão.

No *framework* do jogo foram implementadas duas classes, sendo a *InputManager* responsável pelo gerenciamento dos eventos, registrando os mesmos no componente apropriado, enquanto a classe *GameAction* é uma classe que serve para se definir que tipo de evento será criado e como o mesmo deve ser processado.

Um trecho da classe *InputManager* pode ser visto no anexo C, e um trecho da classe *GameAction* pode ser visto no anexo D.

3.2.3 Efeitos Sonoros

Para que um jogo ganhe vida, é necessário que exista uma maneira de se reproduzir sons enquanto o jogador interage com o jogo. Por exemplo, se o personagem principal recolhe algum item, um efeito sonoro deve ser reproduzido para que aquela ação pareça ser real. A música também é algo importante, pois ambienta o jogador, auxiliando no processo de imersão do mesmo. Por exemplo, em uma situação de perigo, a música pode se tornar mais acelerada, enquanto em situações de calma, a música pode ser mais relaxante.

Para a implementação dos efeitos sonoros no jogo, foram implementadas algumas classes no *framework* que tem a capacidade de reproduzir sons no formato WAV (*Waveform Audio Format*) e MIDI (*Musical Instrument Digital Interface*), usando como base implementações já existentes no Java.

Um trecho da classe *MidiPlayer* – responsável pela reprodução de MIDI's – pode ser visto no anexo E.

3.2.4 *Game Loop*

As classes consideradas o núcleo do jogo, são as classes *GameCore* e *GameManager* que são responsáveis respectivamente pela criação do laço do jogo (*game loop*) e por implementar como os módulos do jogo serão desenhados e seus comportamentos. Como a classe abstrata *GameCore* é extremamente importante, esta é apresentada integralmente no anexo F, onde seu código é totalmente comentado, enquanto a classe *GameManager* (subclasse de *GameCore*) será melhor explicada na seção Implementação do Jogo.

O método mais importante de *GameCore* é o método *gameLoop()* que é responsável em fazer que o jogo execute propriamente dito. O método *gameLoop()* executa três tarefas:

1. Calcula o tempo que foi gasto entre a iteração anterior e atual do laço;
2. Atualiza o estado do jogo baseado no tempo que foi calculado, e;

3. Atualiza o desenho do jogo baseado nas atualizações realizadas.

3.3 Implementação do Jogo

Para a construção do jogo propriamente dito, ainda faltam alguns detalhes e algumas classes a se implementar. Para a criação do mapa de cada fase é necessário construir uma classe que irá gerenciar um mapa baseado em ladrilhos (*tile-based map*). A utilização desse mapa tem como objetivo facilitar a criação das fases, pois assim o mapa pode ser criado utilizando blocos de construção. Na figura 3.6 abaixo pode-se ver como um mapa desse tipo funciona.

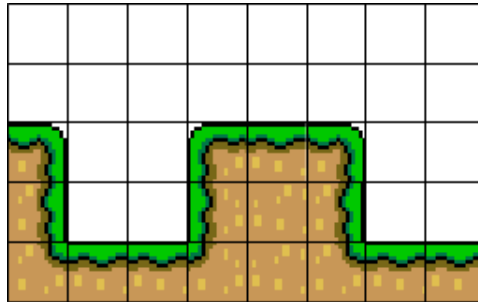


Figura 3.6: Grade de um mapa.

Esse mapa contém referências que indicam qual imagem deve estar em cada célula da grade, fazendo com que o consumo de memória seja menor. Na figura 3.7 abaixo pode-se ver alguns dos *tiles* (ladrilhos) utilizados no jogo.

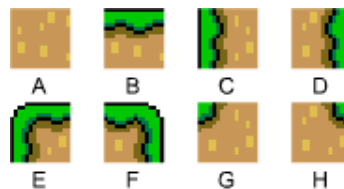


Figura 3.7: Alguns *tiles* utilizados no jogo.

A construção de tal tipo de mapa necessita de duas classes. A primeira, *TileMap* é a classe que define o tamanho do mapa (largura x altura) e fornece métodos para a inclusão e exclusão das *sprites*² do jogo. Algumas das *sprites* utilizadas no jogo podem ser vistas

²*Sprite* é a definição usada para caracterizar um personagem do jogo. No caso do jogo *JMario*, uma *sprite* é o jogador, os inimigos e os itens existentes (as moedas por exemplo).

na figura 3.8 abaixo.



Figura 3.8: Algumas *sprites* utilizadas no jogo.

A segunda classe que deve ser implementada é a *TileMapRenderer*, que usa um objeto do tipo *TileMap* para renderizar o mapa na tela.

Para criar os mapas, são utilizados arquivos de texto que são analisados e para cada caracter encontrado é carregado o *tile* correspondente. Esse serviço é implementado no método *loadMap()* na classe *ResourceManager*. No anexo G pode-se ver o exemplo de um arquivo texto que contém as informações necessárias para construir o mapa apresentado na figura 3.6 da página 46.

Quando um mapa está completo, ele provavelmente será muito maior do que a tela, sendo assim há a necessidade de se renderizar apenas a parte visível do mapa. Para isso é necessário manter o jogador no centro da tela e também verificar se o início e o fim do mapa foram alcançados. Após o desenho dos *tiles*, é a vez das *sprites* serem desenhadas, sendo assim há a necessidade de verificar se a *sprite* está em uma posição visível e manter as *sprites* em uma lista ordenada onde a posição na lista é referente a posição da *sprite* no mapa.

Por fim há a necessidade do desenho do fundo da tela, que é feito utilizando um efeito chamado *parallax*, que consiste em fazer com que a imagem do fundo deslize em uma taxa menor que o mapa que é desenhado acima dela, dando assim o efeito de profundidade

Todos esses cálculos são exibidos no trecho de código da listagem contida no anexo H.

O último detalhe a ser implementado no jogo é referente à detecção de colisão. Isso se

faz necessário para que se possa verificar se uma *sprite* “tocou” em outra ou em um *tile*, e se isso for verdadeiro executar alguma ação. Para essa operação são necessários vários cálculos para verificar onde a *sprite* tocou e como foi o movimento antes desse toque.

Na figura 3.9 abaixo, é mostrado um exemplo de quando uma *sprite* está se movendo diagonalmente e toca em dois *tiles* ao mesmo tempo.

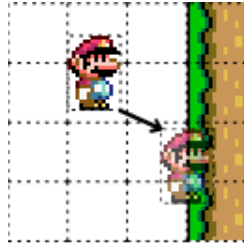


Figura 3.9: Uma *sprite* colide com uma parede e a atravessa.

Para corrigir o posicionamento da *sprite*, é necessário dividir o movimento em duas partes, uma horizontal e outra vertical e assim corrigir o posicionamento. Na figura 3.10 abaixo vê-se a primeira fase da correção, onde o posicionamento horizontal errado é detectado.

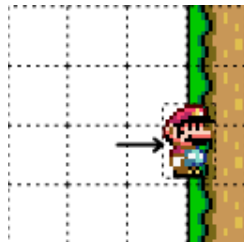


Figura 3.10: Movimentação na horizontal, detecção de colisão horizontal detectada.

Com o posicionamento errado detectado, corrige-se o posicionamento horizontal. A figura 3.11 na página 49 ilustra essa correção.

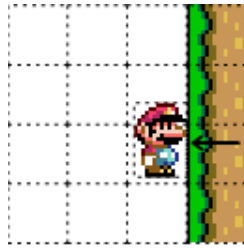


Figura 3.11: Corrige o posicionamento da *sprite* para que ela não colida com nada horizontalmente.

Por fim, corrige-se o posicionamento vertical. Como nesse exemplo a figura não colide com nada na vertical, a correção está pronta. A última correção é vista na figura 3.12 abaixo.

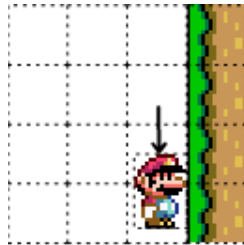


Figura 3.12: Move a *sprite* verticalmente, como não tem colisão, está pronto.

Na listagem contida no anexo I estão os trechos de código responsáveis em se fazer a detecção de colisão. Esses métodos de detecção fazem parte da classe *GameManager* que estende *GameCore*. A classe *GameManager* é a classe que contém o método *main* do jogo, e é nela em que todas as operações e ações são interligadas.

4 *Resultados*

Com o resultado da pesquisa e da implementação do jogo, foi submetido um pedido para criação de um projeto – com o nome de *JMario* – no site <http://www.sourceforge.net> que é um repositório de projetos de software livre. O envio de pedido para abertura do projeto foi feito no dia 22 de setembro de 2007 e a aprovação para a criação do projeto foi expedida no dia 24 de setembro de 2007.

Como resultado da aprovação, foi criado um espaço dedicado para a hospedagem do projeto no *SourceForge*, sendo que o mesmo pode ser acessado pelos seguintes endereços:

- Página do projeto
 - Descrição: Nessa página pode-se ver a descrição do projeto bem como acessar os pacotes de arquivos binários e de código fonte do mesmo.
 - Endereço: <http://www.sourceforge.net/projects/jmario>
- *Site* do projeto
 - Descrição: Nessa página é encontrado o *site* do projeto, onde pode-se ver o resumo do que foi feito bem como a pesquisa para o mesmo.
 - Endereço: <http://jmario.sourceforge.net>
- Repositório *Subversion*
 - Descrição: Endereço para *checkout* do código fonte do projeto.
 - Servidor: <https://jmario.svn.sourceforge.net>

– Repositório: `/svnroot/jmario`

O jogo é totalmente funcional, possuindo três fases, uma interface que mostra a pontuação, telas de final de fase e fim de jogo e transição de telas.

Todo o código apresentado neste trabalho se refere à versão 0.8.1 do *JMario*.

5 *Conclusões*

A partir da pesquisa realizada e da aplicação dos conceitos estudados, foi possível realizar a implementação de um jogo 2D completo que possui todas as características que um jogo deve ter:

- Representação – Presente na criação de um mundo que modela a realidade não de forma completa, mas de maneira suficiente para que o jogo possa ser representado da forma desejada, ou seja, modelagem de personagens, animações, efeito de gravidade, detecção de colisão, música e utilização de efeitos sonoros;
- Interação – O jogador deve interagir primeiramente com a máquina que irá fazer a interface entre o mesmo e o jogo em si. Pelo sistema do jogo o personagem interage com os inimigos, os itens e o ambiente;
- Conflito – O desafio de vencer os obstáculos da fase e de driblar inimigos, e;
- Segurança – Experimentar todo o mundo criado de forma segura, experimentando emoções e ações que não poderiam ser feitas de forma segura no mundo real, como pular de uma grande altura ou morrer e voltar a vida em alguns segundos.

A criação de um jogo, mesmo que esse seja simples, é algo que requer grande dedicação e acima de tudo muita paciência para desenvolver, testar e aprimorar o mesmo. É importante também que o desenvolvedor tenha conhecimento técnico apropriado para que a plataforma de desenvolvimento – Java no caso deste trabalho – seja usada como uma ferramenta e não como algo a aprender.

A conclusão obtida com o desenvolvimento deste trabalho é que a criação de jogos é uma tarefa extremamente prazerosa, que necessita de uma boa parcela de conhecimento e pesquisa devido a complexidade existente em tal tipo de aplicação e por se tratar da criação de uma ferramenta de entretenimento.

Referências

- [1] CRAWFORD, C. *Chris Crawford on Game Design*. 1. ed. Indiana: New Riders Publishing, 2003. 496 p. ISBN 0131460994.
- [2] BRACKEEN, D.; BARKER, B.; VANHEL SUWÉ, L. *Developing Games in Java*. 1. ed. Indiana: New Riders Publishing, 2003. 969 p. ISBN 1592730051.
- [3] CRAWFORD, C. *The art of computer game design*. Washington State University, Vancouver, 1982. Disponível em: <<http://www.vancouver.wsu.edu/fac/peabody/gamebook/Coverpage.html>>. Acesso em: 05 de maio de 2007.
- [4] ROLLINGS, A.; ADAMS, E. *Andrew Rollings and Ernest Adams on Game Design*. 1. ed. Indiana: New Riders Publishing, 2003. 648 p. ISBN 1592730019.
- [5] BATES, B. *Game Design*. 2. ed. Massachusetts: Thomson Course Technology PTR, 2004. 376 p. ISBN 1592004938.
- [6] FLYNT, J. P.; SALEM, O. *Software Engineering for Game Developers*. 1. ed. Massachusetts: Thomson Course Technology PTR, 2004. 904 p. ISBN 1592001556.
- [7] HORSTMANN, C. S.; CORNELL, G. *Core Java, Volume I – Fundamentals*. 8. ed. New Jersey: Prentice Hall PTR, 2007. 880 p. ISBN 0132354764.
- [8] GOSLING, J.; MCGILTON, H. *The java language environment*. 1996. Disponível em: <<http://java.sun.com/docs/white/langenv/>>. Acesso em: 10 de julho de 2007.
- [9] THE Java Language: An Overview. Disponível em: <<http://java.sun.com/docs/overviews/java/java-overview-1.html>>. Acesso em: 10 de julho de 2007.
- [10] DEITEL, H. M.; DEITEL, P. J. *Java How To Program*. 7. ed. New Jersey: Prentice Hall, 2006. 1500 p. ISBN 0132222205.
- [11] ZHANG, H.; LIANG, Y. D. *Computer Graphics Using Java 2D and 3D*. 1. ed. New Jersey: Pearson Prentice Hall, 2006. 632 p. ISBN 0130351180.
- [12] DAVISON, A. *The myths (and truths) of java games programming*. Prince of Songkla University, Hat Yai, 2006. Disponível em: <<http://fivedots.coe.psu.ac.th/~ad/jg/ch00/myths.pdf>>. Acesso em: 26 de abril de 2007.

ANEXO A – Exemplo de uma aplicação simples que usa o Java 2D

```
1  /**
2   * Classe que implementa um exemplo do Java2D.
3   *
4   * @author David Buzatto
5   */
6  public class Java2D extends JFrame {
7
8      public static void main( String[] args ) {
9          Java2D j2d = new Java2D();
10         j2d.setTitle( "Exemplo do Java2D" );
11         j2d.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
12         j2d.add( new Painel2D() );
13         j2d.setSize( 300, 300 );
14         j2d.setVisible( true );
15     }
16
17 }
18
19 class Painel2D extends JPanel {
20
21     public void paintComponent( Graphics g ) {
22         super.paintComponent( g );
23
24         Graphics2D g2d = ( Graphics2D ) g;
25
```

```
26     // desenha uma elipse
27     Shape ellipse = new Ellipse2D.Double( 50, 40, 180, 180 );
28
29     // cria um gradiente para pintar a elipse
30     GradientPaint paint = new GradientPaint( 50, 50, Color.WHITE,
31         250, 250, Color.GRAY );
32
33     // seta o gradiente como o paint para o graphics 2D
34     g2d.setPaint( paint );
35
36     // desenha com preenchimento a elipse no graphics 2D
37     g2d.fill( ellipse );
38
39     // configura a transparência
40     AlphaComposite ac = AlphaComposite.getInstance(
41         AlphaComposite.SRC_OVER, 0.4f );
42
43     g2d.setComposite( ac );
44
45     // configura a cor para verde
46     g2d.setColor( Color.RED );
47
48     // configura a fonte
49     Font font = new Font( "Serif" , Font.BOLD, 40 );
50     g2d.setFont( font );
51
52     // desenha o texto transparente
53     g2d.drawString( "Olá Java 2D!", 30, 140 );
54 }
55
56 }
```

A classe *Painel2D* estende a classe *JPanel* e sobrescreve o método *paintComponent*. Com o parâmetro *Graphics* do método, é realizado um *cast*¹ para *Graphics2D* tornando possível a utilização dos recursos presentes no Java 2D. Uma elipse é desenhada com uma pintura em gradiente. É utilizada uma regra de composição para configurar o grau de transparência utilizado na pintura do texto.

¹O *cast* consiste na conversão explícita entre tipos.

*ANEXO B – Trecho da classe
ScreenManager que fornece a
implementação principal para
o gerenciamento da tela do
jogo*

```
1 /**
2  * A classe ScreenManager gerencia a inicialização e visualização de
3  * modos de tela cheia.
4  *
5  * @author David Buzatto
6  */
7 public class ScreenManager {
8
9     private GraphicsDevice device;
10
11     /**
12     * Cria um novo ScreenManager.
13     */
14     public ScreenManager() {
15         // obtém o ambiente gráfico
16         GraphicsEnvironment environment =
17             GraphicsEnvironment.getLocalGraphicsEnvironment();
18         // obtém o dispositivo gráfico padrão
19         device = environment.getDefaultScreenDevice();
20     }
21
```

```
22  /**
23   * Entra no modo de tela cheia o muda o modo de visualização.
24   * Se o modo de visualização especificado é null ou não compatível
25   * com este dispositivo, ou o modo de visualização não puder ser
26   * alterado nesse sistema, o modo de visualização atual é
27   * utilizado.
28   * <p>
29   * A visualização usa um BufferStrategy com 2 buffers.
30   */
31  public void setFullScreen( DisplayMode displayMode ) {
32
33      final JFrame frame = new JFrame();
34
35      frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
36      frame.setUndecorated( true );
37      frame.setIgnoreRepaint( true );
38      frame.setResizable( false );
39      device.setFullScreenWindow( frame );
40
41      if ( displayMode != null &&
42          device.isDisplayChangeSupported() ) {
43          try {
44              device.setDisplayMode( displayMode );
45          } catch ( IllegalArgumentException ex ) { }
46          // fix para o Mac OS
47          frame.setSize( displayMode.getWidth(),
48                        displayMode.getHeight() );
49      }
50
51      // evita deadlock no Java 1.4
52      try {
53          EventQueue.invokeAndWait(
54              new Runnable() {
55                  public void run() {
```

```
56             frame.createBufferStrategy( 2 );
57         }
58     }
59 );
60 } catch ( InterruptedException ex ) {
61     // ignora
62 } catch ( InvocationTargetException ex ) {
63     // ignora
64 }
65 }
66
67 }
```

*ANEXO C – Trecho da classe
InputManager que fornece a
implementação para o
registro de eventos para a
aplicação*

```
1  /**
2   * O InputManager gerencia a entrada de teclas e eventos do mouse.
3   * Os eventos são mapeados para GameActions.
4   *
5   * @author David Buzatto
6   */
7  public class InputManager implements KeyListener, MouseListener,
8     MouseMotionListener, MouseWheelListener {
9
10     /**
11      * Cria um novo InputManager que ouve as entradas de um componente
12      * específico.
13      */
14     public InputManager( Component comp ) {
15         this.comp = comp;
16         mouseLocation = new Point();
17         centerLocation = new Point();
18
19         // registra os ouvintes de tecla e do mouse
20         comp.addKeyListener( this );
21         comp.addMouseListener( this );
```

```
22     comp.addMouseMotionListener( this );
23     comp.addMouseWheelListener( this );
24
25     /*
26      * permite a entrada da tecla TAB e outras teclas normalmente
27      * usadas pelo focus traversal.
28      */
29     comp.setFocusTraversalKeysEnabled( false );
30 }
31
32 }
```

ANEXO D – Trecho da classe `GameAction` que fornece a implementação para o registro de um tipo de ouvinte de evento para a aplicação

```
1 /**
2  * A classe GameAction é uma abstração para uma ação iniciada pelo
3  * usuário, como pular ou mover. As GameActions podem ser mapeadas
4  * para teclas ou mouse usando o InputManager.
5  *
6  * @author David Buzatto
7  */
8 public class GameAction {
9
10     /**
11      * Comportamento normal. O método isPressed() retorna true quando a
12      * tecla é mantida pressionada.
13      */
14     public static final int NORMAL = 0;
15
16     /**
17      * Comportamento de pressionamento inicial. O método isPressed()
18      * retorna true somente depois que a tecla é pressionada pela
19      * primeira vez, e não novamente até que a tecla seja solta e
20      * pressionada novamente.
21      */
```

```
22     public static final int DETECT_INITIAL_PRESS_ONLY = 1;
23
24     private static final int STATE_RELEASED = 0;
25     private static final int STATE_PRESSED = 1;
26     private static final int STATE_WAITING_FOR_RELEASE = 2;
27
28     private String name;
29     private int behavior;
30     private int amount;
31     private int state;
32
33     /**
34      * Cria uma nova GameAction com o comportamento especificado.
35      */
36     public GameAction( String name, int behavior ) {
37         this.name = name;
38         this.behavior = behavior;
39         reset();
40     }
41
42     /**
43      * Reseta esta GameAction, fazendo parecer que esta não foi
44      * pressionada.
45      */
46     public void reset() {
47         state = STATE_RELEASED;
48         amount = 0;
49     }
50
51     /**
52      * Sinaliza que a tecla foi pressionada.
53      */
54     public synchronized void press() {
55         press( 1 );
```

```
56     }
57
58     /**
59      * Sinaliza que a tecla foi solta.
60      */
61     public synchronized void release() {
62         state = STATE_RELEASED;
63     }
64
65     /**
66      * Retorna se a tecla foi pressionada ou não desde a última checagem
67      *
68      */
69     public synchronized boolean isPressed() {
70         return ( getAmount() != 0 );
71     }
72 }
```

ANEXO E – Trecho da classe MidiPlayer responsável em executar seqüências MIDI

```
1  /*
2  * A classe MidiPlayer é responsável por reproduzir arquivos midi.
3  *
4  * @author David Buzatto
5  */
6  public class MidiPlayer implements MetaEventListener {
7
8      // Evento meta Midi
9      public static final int END_OF_TRACK_MESSAGE = 47;
10
11     private Sequencer sequencer;
12     private boolean loop;
13     private boolean paused;
14
15     /**
16     * Cria um novo MidiPlayer.
17     */
18     public MidiPlayer() {
19         try {
20             sequencer = MidiSystem.getSequencer();
21             sequencer.open();
22             sequencer.addMetaEventListener( this );
23         } catch ( MidiUnavailableException ex ) {
```

```
24         sequencer = null;
25     }
26 }
27
28 /**
29  * Executa uma seqüência, realizando um loop opcionalmente.
30  * Esse método retorna imediatamente.
31  * A seqüência não é executada se não for válida.
32  */
33 public void play( Sequence sequence, boolean loop ) {
34     if ( sequencer != null && sequence != null ) {
35         try {
36             sequencer.setSequence( sequence );
37             sequencer.start();
38             this.loop = loop;
39
40         } catch ( InvalidMidiDataException ex ) {
41             ex.printStackTrace();
42         }
43     }
44 }
45 }
```

ANEXO F – Classe abstrata GameCore

```
1  /**
2   * Classe abstrata utilizada como base do jogo. As subclasses devem
3   * implementar o método draw().
4   *
5   * @author David Buzatto
6   */
7  public abstract class GameCore {
8
9      protected static final int FONT_SIZE = 24;
10
11     // modos de visualização
12     private static final DisplayMode POSSIBLE_MODES [] = {
13         new DisplayMode( 800, 600, 16, 0 ),
14         new DisplayMode( 800, 600, 32, 0 ),
15         new DisplayMode( 800, 600, 24, 0 ),
16         new DisplayMode( 640, 480, 16, 0 ),
17         new DisplayMode( 640, 480, 32, 0 ),
18         new DisplayMode( 640, 480, 24, 0 ),
19         new DisplayMode( 1024, 768, 16, 0 ),
20         new DisplayMode( 1024, 768, 32, 0 ),
21         new DisplayMode( 1024, 768, 24, 0 ),
22     };
23
24     // indica se o jogo está sendo executado
25     private boolean isRunning;
26
```

```
27     // tela do jogo
28     protected ScreenManager screen;
29
30     /**
31      * Sinaliza ao loop do jogo que é hora de terminar.
32      */
33     public void stop() {
34         isRunning = false;
35     }
36
37
38     /**
39      * Chama init() e gameLoop().
40      */
41     public void run() {
42         try {
43             init();
44             gameLoop();
45         } finally {
46             screen.restoreScreen();
47             lazilyExit();
48         }
49     }
50
51
52     /**
53      * Finaliza a máquina virtual usando uma thread daemon.
54      * A thread daemon aguarda 2 segundos então chama System.exit(0).
55      * Como a máquina virtual deve finalizar apenas quando o daemon
56      * estiver rodando, isso dá certeza que System.exit(0) é chamado
57      * somente quanto necessário. Isso se faz necessário para quando
58      * o sistema de som do Java estiver sendo executado.
59      */
60     public void lazilyExit() {
```

```
61     Thread thread = new Thread() {
62         public void run() {
63             // primeiro aguarda que a máquina virtual finaliza
64             // por si própria
65             try {
66                 Thread.sleep( 2000 );
67             } catch ( InterruptedException ex ) { }
68             // o sistema ainda está rodando, então força a
69             // finalização
70             System.exit( 0 );
71         }
72     };
73     thread.setDaemon( true );
74     thread.start();
75 }
76
77
78 /**
79  * Configura o modo de tela cheia, inicializa e cria os objetos.
80  */
81 public void init() {
82     screen = new ScreenManager();
83     DisplayMode displayMode =
84         screen.findFirstCompatibleMode( POSSIBLE_MODES );
85     screen.setFullScreen(displayMode);
86
87     Window window = screen.getFullScreenWindow();
88     window.setFont( new Font( "Dialog", Font.PLAIN, FONT_SIZE ) );
89     window.setBackground( Color.BLUE );
90     window.setForeground( Color.WHITE );
91
92     isRunning = true;
93 }
94
```

```
95
96     /**
97     * Carrega uma imagem.
98     */
99     public Image loadImage( String name ) {
100         return new ImageIcon( getClass().getResource(
101             "/recursos/imagens/" + name ) ).getImage();
102     }
103
104
105     /**
106     * Executa o game loop até que stop() seja chamado.
107     */
108     public void gameLoop() {
109
110         // obtém a hora atual do sistema
111         long startTime = System.currentTimeMillis();
112         long currTime = startTime;
113
114         // enquanto está executando...
115         while ( isRunning ) {
116
117             // calcula o tempo que passou
118             long elapsedTime =
119                 System.currentTimeMillis() - currTime;
120             currTime += elapsedTime;
121
122             // atualiza
123             update( elapsedTime );
124
125             // desenha
126             Graphics2D g = screen.getGraphics();
127             draw( g );
128             g.dispose();
```

```
129         screen.update();
130
131         // não forme, executando da forma mais rápida possível
132         /*try {
133             Thread.sleep(20);
134         }
135         catch (InterruptedException ex) { }*/
136     }
137 }
138
139 /**
140  * Atualiza o estado do jogo/animação baseado da quantidade
141  * de tempo que passou entre a iteração atual e a anterior.
142  */
143 public void update( long elapsedTime ) {
144     // não faz nada
145 }
146
147 /**
148  * Desenha na tela. As subclasses devem sobrescrever esse método.
149  */
150 public abstract void draw( Graphics2D g );
151
152 }
```

ANEXO G - Um exemplo de mapa

```
1 # essa linha é um comentário e não será processada em loadMap()  
2  
3  
4 F EBF  
5 D CAD  
6 HBBGAHBB
```

ANEXO H – Método draw da classe TileMapRenderer

```
1 public class TileMapRenderer {
2
3     /**
4      * Desenha o TileMap especificado.
5      */
6     public void draw( Graphics2D g, TileMap map,
7         int screenWidth, int screenHeight ) {
8
9         Sprite player = map.getPlayer();
10        int mapWidth = tilesToPixels( map.getWidth() );
11
12        // obtém a posição de scrolling do mapa, baseado
13        // na posição do jogador
14        int offsetX = screenWidth / 2 -
15            Math.round( player.getX() ) - TILE_SIZE;
16        offsetX = Math.min( offsetX, 0 );
17        offsetX = Math.max( offsetX, screenWidth - mapWidth );
18
19        // obtém o offset de y para desenhar todas as sprites
20        // e tiles
21        int offsetY = screenHeight -
22            tilesToPixels( map.getHeight() );
23
24        // desenha um fundo preto se necessário
25        if ( background == null ||
```



```
60         Math.round( player.getY() ) + offsetY,
61         null );
62
63     // desenha as sprites
64     Iterator i = map.getSprites();
65     while ( i.hasNext() ) {
66         Sprite sprite = ( Sprite ) i.next();
67         int x = Math.round( sprite.getX() ) + offsetX;
68         int y = Math.round( sprite.getY() ) + offsetY;
69         g.drawImage( sprite.getImage(), x, y, null );
70
71         // acorda a criatura quando a mesma estiver na tela
72         if ( sprite instanceof Creature &&
73             x >= 0 && x < screenWidth ) {
74             ( ( Creature ) sprite ).wakeUp();
75         }
76     }
77 }
78 }
```

ANEXO I – Métodos da classe GameManager que tratam a detecção de colisão

```
1 public class GameManager extends GameCore {
2
3     /**
4      * Obtém o tile que a Sprite colide. Somente X ou Y da Sprite
5      * deve ser mudado não ambos. Retorna null se nenhuma colisão
6      * for detectada.
7      */
8     public Point getTileCollision( Sprite sprite,
9         float newX, float newY ) {
10
11         float fromX = Math.min( sprite.getX(), newX );
12         float fromY = Math.min( sprite.getY(), newY );
13         float toX = Math.max( sprite.getX(), newX );
14         float toY = Math.max( sprite.getY(), newY );
15
16         // obtem a localização do tile
17         int fromTileX = TileMapRenderrer.pixelsToTiles( fromX );
18         int fromTileY = TileMapRenderrer.pixelsToTiles( fromY );
19         int toTileX = TileMapRenderrer.pixelsToTiles(
20             toX + sprite.getWidth() - 1 );
21         int toTileY = TileMapRenderrer.pixelsToTiles(
22             toY + sprite.getHeight() - 1);
23
```

```

24     // checa cada tile para verificar a colisão
25     for ( int x = fromTileX; x <= toTileX; x++ ) {
26         for ( int y = fromTileY; y <= toTileY; y++ ) {
27             if ( x < 0 || x >= map.getWidth() ||
28                 map.getTile( x, y ) != null ) {
29                 // colisão achada, retorna o tile
30                 pointCache.setLocation( x, y );
31                 return pointCache;
32             }
33         }
34     }
35     // nenhuma colisão achada
36     return null;
37 }
38
39 /**
40  * Verifica se duas sprites colidiram entre si. Retorna false
41  * caso duas Sprites sejam a mesma. Retorna false se um uma
42  * das Sprites não estiver viva.
43  */
44 public boolean isCollision( Sprite s1, Sprite s2 ) {
45
46     // se as sprites são a mesma, retorn false
47     if ( s1 == s2 )
48         return false;
49
50     // se uma das sprites é uma criatura morta, retorna false
51     if ( s1 instanceof Creature && ! ( ( Creature ) s1 ).isAlive() )
52         return false;
53     if ( s2 instanceof Creature && ! ( ( Creature ) s2 ).isAlive() )
54         return false;
55
56     // obtem a localização em pixel das sprites
57     int s1x = Math.round( s1.getX() );

```

```

58     int s1y = Math.round( s1.getY() );
59     int s2x = Math.round( s2.getX() );
60     int s2y = Math.round( s2.getY() );
61
62     // verifica se as bordas das sprites se interceptam
63     return ( s1x < s2x + s2.getWidth() &&
64             s2x < s1x + s1.getWidth() &&
65             s1y < s2y + s2.getHeight() &&
66             s2y < s1y + s1.getHeight() );
67 }
68
69 /**
70  * Obtém a Sprite que colide com uma Sprite específica,
71  * ou null se nenhum Sprite colide com a Sprite especificada.
72  */
73 public Sprite getSpriteCollision(Sprite sprite) {
74
75     // itera pela lista de sprites
76     Iterator i = map.getSprites();
77
78     while ( i.hasNext() ) {
79         Sprite otherSprite = ( Sprite ) i.next();
80         if ( isCollision( sprite, otherSprite ) )
81             // colisão encontrada, retorna a sprite
82             return otherSprite;
83     }
84     // sem colisão
85     return null;
86 }
87
88 /**
89  * Atualiza as criaturas, usando gravidade para as criaturas
90  * que não estão voando e verifica colisão.
91  */

```

```
92     private void updateCreature( Creature creature, long elapsedTime ) {
93
94         // usa gravidade
95         if ( !creature.isFlying() )
96             creature.setVelocityY( creature.getVelocityY() +
97                 GRAVITY * elapsedTime );
98
99         // altera x
100        float dx = creature.getVelocityX();
101        float oldX = creature.getX();
102        float newX = oldX + dx * elapsedTime;
103        Point tile = getTileCollision( creature, newX, creature.getY() )
104
105        ;
106
107        if ( tile == null ) {
108            creature.setX( newX );
109        } else {
110            // alinha com a borda do tile
111            if ( dx > 0 ) {
112                creature.setX(
113                    TileMapRenderer.tilesToPixels( tile.x ) -
114                    creature.getWidth() );
115            } else if ( dx < 0 ) {
116                creature.setX(
117                    TileMapRenderer.tilesToPixels( tile.x + 1 ) );
118            }
119            creature.collideHorizontal();
120        }
121
122        if ( creature instanceof Player )
123            checkPlayerCollision( ( Player ) creature, false );
124
125        // troca y
126        float dy = creature.getVelocityY();
```

```
125     float oldY = creature.getY();
126     float newY = oldY + dy * elapsedTime;
127     tile = getTileCollision( creature, creature.getX(), newY );
128
129     if ( tile == null ) {
130         creature.setY( newY );
131     } else {
132         // alinha com a borda do tile
133         if ( dy > 0 ) {
134             creature.setY(
135                 TileMapRenderer.tilesToPixels( tile.y ) -
136                 creature.getHeight() );
137         } else if ( dy < 0 ) {
138             creature.setY(
139                 TileMapRenderer.tilesToPixels( tile.y + 1 ) );
140         }
141         creature.collideVertical();
142     }
143     if ( creature instanceof Player ) {
144         boolean canKill = ( oldY < creature.getY() );
145         checkPlayerCollision( ( Player ) creature, canKill );
146     }
147
148     // se o jogador cai (y muito alto), tira vida e reinicia
149     if ( creature instanceof Player ) {
150         // se o jogador está além do pixel 2000 de altura, morre
151         if ( creature.getY() > 2000 ) {
152             creature.setState( creature.STATE_DEAD );
153             quantidadeVidas--;
154         }
155     }
156 }
157
158 }
```

```
159
160  /**
161   * Verifica colisão entre o jogador e outras sprites. Se canKill
162   * é true a colisão com as criaturas irá matá-las.
163   */
164  public void checkPlayerCollision( Player player,
165                                   boolean canKill ) {
166
167      if ( !player.isAlive() )
168          return;
169
170      // verifica a colisão do jogador com outras Sprites
171      Sprite collisionSprite = getSpriteCollision( player );
172
173      if ( collisionSprite instanceof PowerUp ) {
174          acquirePowerUp( ( PowerUp ) collisionSprite );
175      } else if ( collisionSprite instanceof Creature ) {
176
177          Creature badguy = ( Creature ) collisionSprite;
178
179          if ( canKill ) {
180              // mata o inimigo a faz o jogador oscilar
181              soundManager.play( boopSound );
182              badguy.setState( Creature.STATE_DYING );
183              quantidadePontos += 100;
184              player.setY( badguy.getY() - player.getHeight() );
185              player.jump( true );
186          } else {
187              // jogador morre
188              player.setState( Creature.STATE_DYING );
189              // decrementa quantidade de vidas
190              quantidadeVidas--;
191          }
192      }
```

193 }

194 }
